Parallel Single-Source Shortest Paths

Kevin Kelley
Tao B. Schardl

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

We designed and implemented a parallel algorithm for solving the single-source shortest paths (SSSP) problem for graphs with nonnegative edge weights, based on Gabow's scaling algorithm for SSSP. This parallel Gabow algorithm attains a theoretical parallelism of $\Omega(E/(V \lg \Delta \lg E/D))$ in the worst case and $\Omega(E/(D \lg V/D \lg E/D \lg Delta)$ with high probability on a random graph. In practice this algorithm decent parallelism on random graphs and outperforms a simple Dijkstra implementation on six or more processors.

1. INTRODUCTION

The single-source shortest path (SSSP) problem on graphs with nonnegative edge weights is a common problem in graph theory that has been studied since the 1950s. This problem arises in a host of real-world applications, particularly in routing.

Given a weighted, directed graph $G = (V, E)$ with vertex set $V = V(G)$ and edge set $E = E(G)$, a weight function $w : E \to \mathbf{R}^+$, and a starting vertex v_0, the SSSP problem is to compute the "shortest path weight" from v_0 to all $u \in V$. The ***weight*** of a path is the sum of the weights of edges along that path, and the ***shortest path weight*** from $u \in V$ to $v \in V$ is the minimum weight for any path from u to v, or ∞ if no path exists from u to v. The ***distance*** to a vertex $v \in V$ is the shortest path weight from v_0 to v.

The canonical serial algorithm for solving the SSSP problem on graphs with nonnegative edge weights is Dijkstra's algorithm, whose pseudocode is presented in Figure 1. This algorithm stores the vertices of G in a minimum priority queue Q keyed on the best known distance to each vertex. A common implementation for this priority queue is a k-ary heap. A vertex v has been ***evaluated*** when all outgoing edges from v . Dijkstra's algorithm loops over the contents of Q until all vertices have been evaluated. the repeatedly extracts from the priority queue the unevaluated vertex u with the minimum known distance from the priority queue in line 6, examines u's adjacency list in line 7, and recomputes minimum known distances for each neighbor of u in lines 8–10. One can prove that greedily evaluating the unevaluated vertex with minimum distance yields the correct shortest path distances for graphs with nonnegative edge weights.

```
DIJKSTRA($G, w, v_0$)
 1  for each vertex $u \in V(G)$
 2      $u.dist = \infty$
 3  $v_0.dist = 0$
 4  $Q = V(G)$
 5  while $Q \neq \emptyset$
 6      $u$ = EXTRACT-MIN($Q$)
 7      for each vertex $v \in V(G)$ such that $(u, v) \in E(G)$
 8          if $v.dist > u.dist + w(u, v)$
 9              $v.dist = u.dist + w(u, v)$
10              DECREASE-KEY($Q, v, v.dist$)
```

Figure 1: Pseudocode for Dijkstra's SSSP algorithm for graphs with nonnegative edge weights. At all times, the distance of a vertex u, $u.dist$, is the shortest path weight known from v_0 to u. We use a minimum priority queue Q keyed on the distance to each vertex.

Dijkstra's algorithm does not readily lend itself to parallelization however, because it relies on a priority queue. Because evaluating the vertex at the head of the priority queue may change the order of elements remaining within the priority queue, it is not necessarily correct or efficient to evaluate more than the head of the priority queue on each iteration. Substantial research has been done into parallelizing Dijkstra's algorithm, but these strategies often rely on known qualities of the graph being searched.

In this paper we present a novel parallel solution to the SSSP problem, based on Gabow's scaling algorithm for SSSP [4]. This algorithm makes use of a simpler data structure for its priority queue, which naturally exposes available much of the parallelism in the problem. Furthermore, the only assumption this ***parallel Gabow*** algorithm uses is that all edge weights in the graph are integral, which is a reasonable assumption for SSSP problems being solved on modern computers.

The remainder of this paper is organized as follows. Section 2 presents Gabow's original scaling algorithm, while Section 3 presents the strategy for parallelizing this SSSP algorithm. Section 4 describes several optimizations we used in implementing parallel Gabow. Section 5 presents some theoretical analysis for the parallelism in parallel Gabow, while Sections 6 and 7 present our empirical findings for the available parallelism and real-world performance of parallel Gabow, respectively. We conclude in Section 9.

2. GABOW'S SCALING ALGORITHM FOR SSSP

Gabow's algorithm is a classic scaling approach to the single-source shortest paths problem. In effect, we determine shortest paths to each vertex in a graph by iteratively refining approximations of those shortest paths.

Initially, we have the trivial zero-weight path to the source and no path to each other vertex. We begin each iteration by exposing an additional bit of the weight of each edge starting from the highest-order bit. Then, we calculate new *temporary weights* for each edge by using the shortest-path weights from the previous iteration as a cost function. Finally, we perform a single-source shortest path computation on this reweighted graph. The final shortest-path weight to each vertex is found by accumulating the shortest-path weights from each iteration. The paths corresponding to those weights are the same as those from the final iteration.

In the first iteration, no bits of the edge weights are exposed and we have no existing paths, so the temporary weight of each edge is simply zero; we are, in effect, simply finding the connected component of the graph containing the source vertex. In the second iteration, only the highest-order bit of each path weight is exposed, and so on.

```
GABOW($G, w, v_0$)
for each vertex $u \in V(G)$
    $u.dist = \infty$
$i = \lg(\max\{range(w)\})$
$v_0.dist = 0$
$Q = \emptyset$
INSERT($Q, v_0, 0$)
while $Q \neq \emptyset$
    $u$ = EXTRACT-MIN($Q$)
    for each vertex $v \in V(G)$ such that $(u,v) \in E(G)$
        if $v.dist > u.dist + w_i(u,v)$
            $v.dist = u.dist + w_i(u,v)$
            INSERT($Q, v, v.dist$)
while $i > 0$
    $i--$
    for each vertex $u \in V(G)$
        $u.extra\text{-}dist = \infty$
    $v_0.extra\text{-}dist = 0$
    $Q = \emptyset$
    INSERT($Q, v_0, 0$)
    while $Q \neq \emptyset$
        $u$ = EXTRACT-MIN($Q$)
        for each vertex $v \in V(G)$ such that $(u,v) \in E(G)$
            $l_{(u,v)} = w_i(u,v) + 2(u.dist - v.dist)$
            if $v.extra\text{-}dist > u.extra\text{-}dist + l_{(u,v)}$
                $v.extra\text{-}dist = u.extra\text{-}dist + l_{(u,v)}$
                INSERT($Q, v, v.extra\text{-}dist$)
    for each vertex $u \in V(G)$
        if $u.dist < \infty$
            $u.dist = 2 \times u.dist + u.extra\text{-}dist$
```

Figure 2: Pseudocode for Gabow's scaling algorithm for SSSP. The distance of a vertex u, $u.dist$, is the shortest path weight from v_0 to u. In this algorithm $w_i(u,v) = w(u,v) >> i$.

On first examination, this might appear to be far worse than a much simpler algorithm such as Dijkstra; after all, we have to perform $\lg W$ iterations, and each one requires that we solve an SSSP subproblem. However, there are several observations we can make regarding the structure of these subproblems that both will allow us to make them less expensive to solve and will expose parallelism which, as previously discussed, is absent from Dijkstra.

Bounding the size of the priority queue

We assert that the temporary weight of any edge which was on the previous iteration's shortest path is at most one. Consider the edge (u,v) which is on the shortest path to v. Since the shortest path to v necessarily includes the shortest path to u, the weight of the shortest path to v is the sum of the weight of the shortest path to u and the weight of (u,v).

When, in the next iteration, we use these shortest-path weights as a cost function, the temporary weight of (u,v) is therefore clearly zero. We proceed to expose an additional bit of the cost of (u,v), which becomes the low-order bit of that cost. This is effectively added to the temporary weight; if the bit is zero, the temporary weight remains zero, and if it is one, the temporary weight becomes one.

Another fundamental observation is that the length of any shortest path in a graph with nonnegative edge weights is at most V. Gabow [4] presents only the bound E; in truth, the bound is the minimum of these two, but since the algorithm effectively only operates on a connected subgraph, V is almost always the tighter bound. In any case, the intuition is nearly the same. Since edge weights are nonnegative, a shortest path cannot have cycles and thus each vertex must be visited at most once.

These two pieces of information together give us a bound of V on the maximum length of any shortest path in each iteration of the scaling algorithm. This bound is clearly not present in Dijkstra. It is rather important to the performance of our implementation in that it allows any insertion in to the priority queue with a path weight greater than V to be instantly discarded.

Further pruning the queue

We present a second bound on the weight of any shortest path which is significantly tighter, although not asymptotically so. Imagine that we are aware of d, the maximum length of the shortest path from the previous iteration. With the same reasoning about the maximum weight of any edge on the previous iteration's shortest paths, we can conclude that we may safely discard any candidate path whose weight exceeds d.

We present statistics demonstrating that d is often a fraction of a percent of V.

3. PARALLELIZING THE ITERATIVE STEP

Some portions of the iterative step are trivial to parallelize. For example, bits can be exposed in parallel; temporary weights can be computed in parallel; and the cost function can be applied in parallel. Each of these operations is performed on a single vertex in isolation. However, by far the most expensive part of each step is the process of finding shortest paths once the edges have been reweighted.

Consider the priority queue around which these SSSP subproblems are based. In the queue, candidate paths with equal weights can be arbitrarily reordered without any ill effect. We assert that, with some caveats, they can also be processed in parallel.

The parallel, bounded priority queue

We present a variant of the priority queue which we call a ***parallel, bounded priority queue***. This queue has a bounded size n, and consists of bins numbered from 0 to $n-1$. It supports only two operations; INSERT(Q, x, k) places an item x with key k into the queue Q, and EXTRACT-MIN(Q) returns the contents of the lowest-indexed nonempty bin.

Since edge weights are nonnegative, processing the contents of a bin i can only generate insertions into bins $j \geq i$. Consequently, to evaluate the entire queue, each iteration of Gabow's algorithm only needs to consider monotonically increasing bin indexes. This monotonicity allows Gabow's algorithm to use a simple data structure for the priority queue, which exposes parallelism in the algorithm.

The priority queue's bins must support parallel insertions. Initially we implemented this in a simple fashion with locks and STL vectors. However, we replaced this with a lock-free data structure, the `TLSSet`, which is described in more detail in subsequent sections.

Zero-weight edges and chains

One wrinkle in this approach is the existence of zero-weight edges, which appear even if none exist in the original graph. (For example, any time we expose a zero bit on an edge which was part of a shortest path, the result is a temporary weight of zero.) When we explore one of these edges, it is not immediately obvious how best to proceed. Making matters worse, we possibly encounter ***chains*** — successive zero-weight edges forming a zero-weight subpath. Chains are unavoidably explored serially.

We could immediately explore zero-weight edges (for example, by recursion); however, as with any serializing approach, this reduces parallelism by creating an imbalance in the division of labor amongst workers. Instead, we add them to a temporary bin; when we finish processing the initial bin, if the temporary bin is nonempty, we process it before extracting the next bin from the priority queue. This allows us to preserve as much parallelism as is possible.

Making an inconvenient race benign

In processing the contents of a bin in parallel, we create one race condition which is particularly challenging to resolve. We explain the race, present a fix, and argue that the fix makes the race benign.

Recall that, as we solve SSSP at each step, we maintain an array of the best path weights we have found so far. Whenever we find a new candidate path and perform an insertion into the queue, we must also update this shared state.

Imagine that, before examining a bin, the shortest path to vertex v is of weight i, and that upon examining the bin, we find two shorter paths of lengths j and k such that $i > j > k$. It is possible that paths j and k are discovered simultaneously by two different workers.

Each worker sees the existing shortest path of weight i and believes that it has discovered a new shortest path. Each worker performs a queue insertion, and then each attempts to update the shared state. This potentially produces two undesirable effects. First, we have a worthless candidate path of weight j in the queue; and second, if the worker which discovered the weight-j path won the race, the shared state will describe the best known path to v as being of weight j. If, subsequently, additional paths to v whose weight is less than j but greater than k are discovered, those will be added to the queue as well despite their being equally useless.

However, the shortest candidate path to v will be the first one explored because the bins are examined in order. Therefore, if we update the shortest path array again upon discovering v in bin k, when we find v in later bins we can immediately know to ignore it. Vertex v may appear multiple times in a single bin, but since each appearance will cause the same value to be written to shared state, this race is also benign.

Thus this race, suitably fixed, will not threaten correctness. We have empirical evidence which suggests, based on the number of queue items discarded by this test, that it also has little impact on performance.

4. ADDITIONAL IMPLEMENTATION OPTIMIZATIONS

The first iterative step

In our implementation, we special-case the first iterative step. This step is unique because every edge has a temporary weight of zero. Therefore, the SSSP problem in this step is effectively reduced to a traversal of the graph, which we can do without involving a priority queue. This makes sense–if all edge weights are zero, then every candidate path would wind up in the 0th queue anyway.

We can also find d, the maximal shortest path length, which we need to size the queue for the next iteration; it is simply the longest path to any vertex in the graph, which is the same as the number of times we have to iterate before running out of zero-weight edges to explore (the maximal chain length).

Lazy computation of temporary edge weights

One simple change we were able to make that had a profound performance impact was to postpone the computation of temporary edge weights until they were actually needed. Since we explore each edge at most once, this allows us to avoid more computation that is strictly necessary. This also reduces the size of the edge data structure by a third.

Making queue bins lock-free

The bins in our priority queue are implemented as a special type of multiset which we call a *staging set*. A staging set supports three operations. When it is first created, it supports only INSERT(x), which inserts an item into the set. After we FINISH a set, we are no longer allowed to add new elements to it, but we can iterate over the contents of the set using STL-compatible BEGIN and END operators. No ordering guarantees are provided.

We found that this data structure was repeatedly useful. We can have a group of worker threads contribute to building a set and then divide-and-conquer the contents of the set, such as with a `cilk_for` loop.

Our initial implementation of this data structure, the `LockingSet`, is very simple; it is simply an STL vector whose INSERT operator is synchronized with a mutual exclusion lock. The FINISH operator does nothing.

We replaced this with a lock-free implementation, the `TLSSet`. As its name might suggest, this data structure utilizes thread-local storage to allow lock-free parallel insertions. The FINISH operator computes offsets that allow the combined contents of the set to be uniformly distributed.

5. THEORETICAL ANALYSIS OF PARALLEL GABOW

This section presents a theoretical analysis of the performance of our parallel Gabow algorithm. For simplicity of analysis, we assume that each vertex in the graph appears at most once within each bin in the priority queue during each iteration of parallel Gabow.

First we bound the work of parallel Gabow.

LEMMA 1. *Let $G = (V, E)$ be a connected, directed graph. Let $w : E \to \mathbf{Z}^+$ be a weight function on $E = E(G)$, and let W be the maximum weight in the range of w. The total work of parallel Gabow is $\Theta(E \lg W)$.*

PROOF. The ith iteration of parallel Gabow computes the shortest paths in G when only the top i bits of every edge weight are used. The number of iterations of parallel Gabow that must be run to compute the exact shortest path distances equals $\lg W$, the maximum number of bits to represent any weight in the range of w.

Each iteration of parallel Gabow evaluates every vertex exactly once, which requires $\Theta(E)$ total work per phase. Since each edge adds at most one vertex to the queue to either evaluate or ignore, $O(E)$ items must be examined per iteration of Gabow. Finally, $O(E)$ total work is required to spawn and sync all parallel computation in an iteration of parallel Gabow. The total work of parallel Gabow is therefore $O(E \lg W)$.

□

Second, we will prove a worst-case bound on the span of parallel Gabow.

LEMMA 2. *Let $G = (V,E)$ be a connected, directed graph; let $w : E \to \mathbf{Z}^+$ be a weight function on E; let W be the maximum weight in the range of w; let Δ be the maximum degree of any vertex $v \in V$; and let D be the length of the longest minimum-weight path in any iteration of Gabow's algorithm. The worst-case span of parallel Gabow is $O(V \lg \Delta \lg E/D \lg W)$.*

PROOF. The iterations of parallel Gabow must be executed serially, contributing $\Theta(\lg W)$ to the span of the algorithm. Within a single iteration of parallel Gabow, the D bins in the priority queue are examined serially. For each distance $d < D$, the bin representing distance d may be evaluated k_d times, where k_d is the length of the longest zero-weight edge chain at distance d. At each of these $\sum_{d=0}^{D} k_d$ serial steps, the contents in each bin is evaluated with a logarithmic span to divide the computation and a $O(\lg \Delta)$ span to evaluate the adjacency list of each evaluated vertex in parallel.

Since each vertex is evaluated at most once, each vertex may appear at most once in any bin's zero-weight edge chain. Consequently, the number of serial steps $\sum_{d=0}^{D} k_d \le V$, and the worst-case span of parallel Gabow is $O(V \lg E/D \lg \Delta \lg W)$. □

Lemmas 1 and 2 implies a lower bound for the parallelism of $\Omega(E/(V \lg \Delta \lg E/D))$. If we assume a random graph, we can prove larger parallelism lower bound with high probability.

LEMMA 3. *Let $G = (V,E)$ be a random connected, directed graph; let $w : E \to \mathbf{Z}^+$ be a random weight function on E; let W be the maximum weight in the range of w; let Δ be the maximum degree of any vertex $v \in V$; and let D be the length of the longest minimum-weight path in any iteration of Gabow's algorithm. The span of parallel Gabow is $O(D \lg V/D \lg E/D \lg \Delta \lg W)$.*

PROOF. The iterations of parallel Gabow must be executed serially, contributing $\Theta(\lg W)$ to the span of the algorithm. Within a single iteration of parallel Gabow, the D bins in the priority queue are examined serially. For each distance $d < D$, the bin representing distance d may be evaluated k_d times, where k_d is the length of the longest zero-weight edge chain at distance d. At each of these $\sum_{d=0}^{D} k_d$ serial steps, the contents in each bin is evaluated with a logarithmic span to divide the computation and a $O(\lg \Delta)$ span to evaluate the adjacency list of each evaluated vertex in parallel.

With high probability, the longest zero-weight chain within any particular bin distance d is logarithmic in the number of vertices evaluated at distance d. Consequently, the maximum value of $\sum_{d=0}^{D} k_d = O(D \lg V/D)$, and with high probability the span of parallel Gabow on a random graph is $O(D \lg V/D \lg E/D \lg \Delta \lg W)$. □

Lemmas 1 and 3 imply a lower parallelism bound of $\Omega(E/(D \lg V/D \lg E/D \lg Delta)$.

Graph type	Min	Median	Mean	Max
Random	25	36	50	226
Road Network	1109	2551	2684	4973

Figure 3: Minimum, median, mean, and maximum queue sizes during parallel Gabow's execution on a random graph and on a road network. The random graph used contains 1.5 million vertices and 4 million edges. The road network graph used is models the road network for the northeastern part of the U.S.

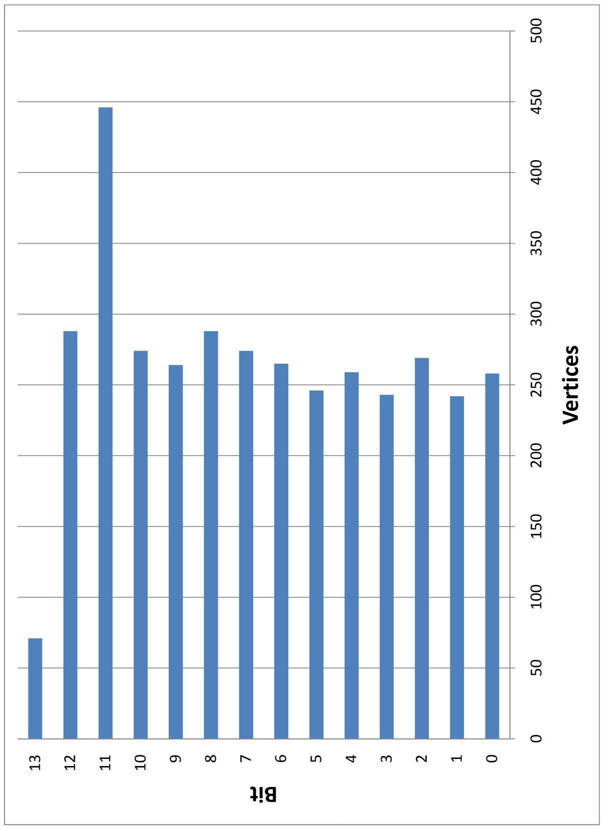

Figure 4: Total number of vertices in all longest zero-weight chains for each iteration of parallel Gabow on a random graph.

6. STATISTICS FOR PARALLEL GABOW

To verify the feasibility of parallelizing Gabow's shortest paths algorithm, we measured several statistics concerning the serial execution of parallel Gabow on random graphs and on road networks. First, we measured the necessary queue size for each iteration of Gabow. Second, we measured the total length of all zero-weight chains on each iteration. Finally, we measured the number of vertices within each bin that were evaluated and that were evicted.

Random graphs

We collected statistics from running parallel Gabow serially on a random graph with 1.5 million vertices and 4 million edges.

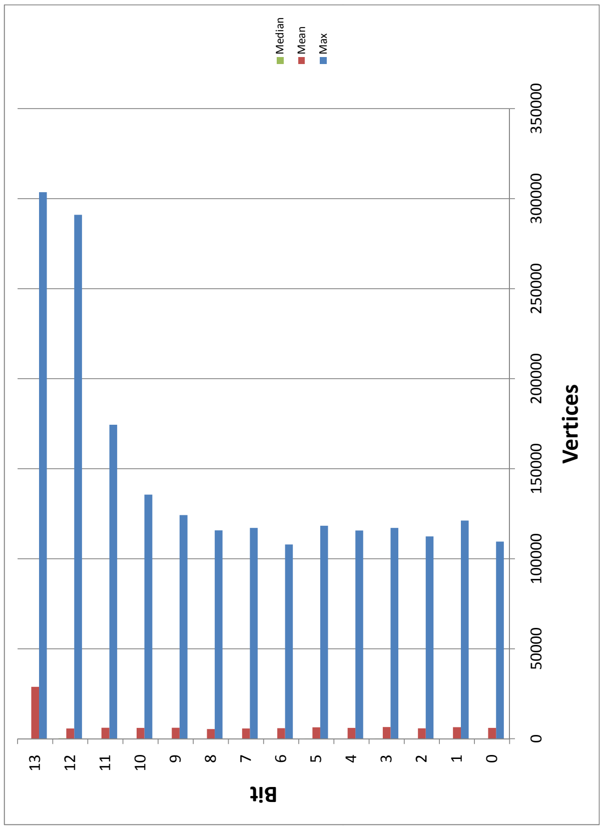

Figure 5: Max, mean, and median of the number of vertices evaluated in each bin during each iteration of parallel Gabow on a random graph.

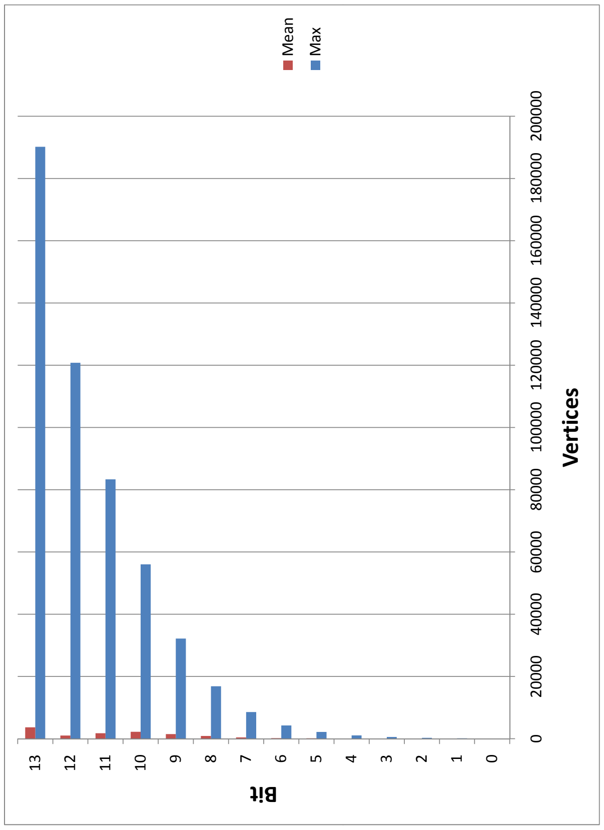

Figure 6: Max and mean of the number of vertices ignored in each bin during each iteration of parallel Gabow on a random graph.

First we examined the number of bins needed in the queue, and found the results presented in Figure 3. We found that over all iterations of Gabow's algorithm on this graph, the number of bins needed in the priority queue for this graph is only 50 on average, and never more than 250. This suggests that a large number of vertices may end up within each bin, all of which could be processed in parallel.

We also measured the total length of all zero-weight chains in each iteration of parallel Gabow on a random graph in order to bound the span of the parallel Gabow's execution. As shown in Figure 4, the total length of zero-weight chains for this is usually around 275 vertices and never above 450 vertices. This further supports the intuition that each step of parallel Gabow may process a large number of vertices in parallel.

We then measured the number of vertices evaluated within each bin for each iteration of Gabow, and we found the results presented in Figure 5 for a random graph. The average number of vertices evaluated within a bin is consistently around 10000, while the median number of evaluated vertices is even smaller. The maximum number of vertices evaluated in any bin is initially large — over 250000 — and drops to around 125000 for later iterations of the algorithm, suggesting that the distribution of shortest path weights becomes more even in later iterations. All together these results suggest that a small number of bins processed on each iteration contain most of the vertices to evaluate, all of which may be processed in parallel.

Finally we examined the number of vertices within any bin that were evicted for each iteration. These vertices represent additional work within the queue that parallel Gabow must deal with due to the queue's lack of a parallel DECREASE-KEY operation. As shown in Figure 6, for random graphs the maximum number of evicted vertices is large in early iterations of Gabow's algorithm, but tapers off in subsequent iterations. Compared to the number of vertices evaluated, however, the ignored vertices total at most 37%, of V, and are typically less than 10%. We therefore do not expect these vertices to hurt the performance of parallel Gabow too much for random graphs.

Road networks

We collected the same set of statistics for parallel Gabow on a road network of the northeastern U.S., which also contained approximately 1.5 million vertices and 4 million edges. The statistics follow a similar trend to those for random graphs, but their values are scaled compared to random graphs, demonstrating that road network exhibit less parallelsim for Gabow to exploit. As shown in Figure 3, the queue sizes needed for a road network graph are typically around 2500: 10 times larger than for a comparably sized random graph, but small compared to V nonetheless. The total length of zero-weight chains for this road network is also larger — typically around 10% of V. As expected from the larger queue size, the number of vertices evaluated within a bin on each iteration is smaller compared to Gabow's performance on a random graph, suggesting that less parallelism is available for Gabow on this graph. Finally, the number of ignored vertices in the queue is never more than 16% of V at max, and on average is much smaller.

7. PERFORMANCE RESULTS FOR PARALLEL GABOW

We implemented the parallel Gabow algorithm in Cilk++ and

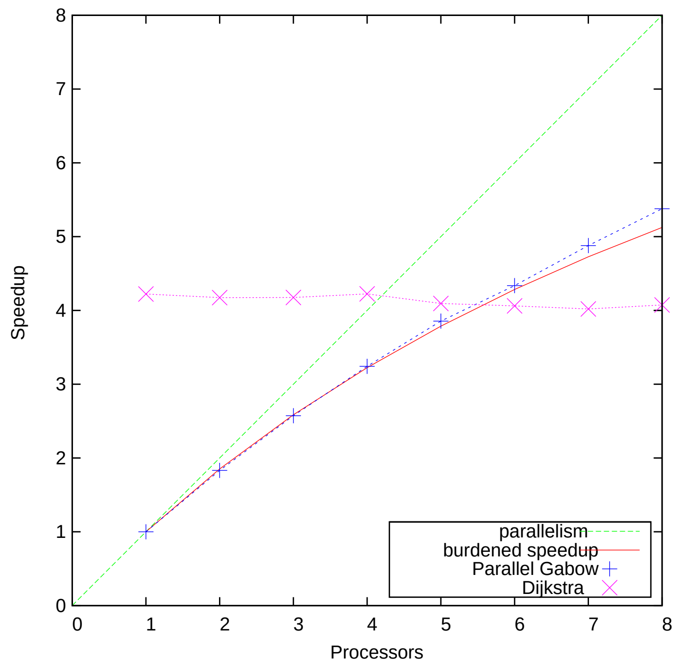

Figure 7: Example speedup for parallel Gabow on a random graph. All speedups are normalized to parallel Gabow's serial performance.

compared its performance to a simple serial Dijkstra implementation in C++. This section compares their performance on random graphs and graphs of parts of the U.S. road network.

Our serial Dijkstra implementation used a priority queue provided by the C++ Standard Template Library [1]. This priority queue did not support a DECREASE-KEY operation. Consequently, our implementation of Dijkstra only performs INSERTs and ignores previously evaluated vertices when they appear more than once from EXTRACT-MIN.

We ran our tests on an Intel Core i7 quad-core machine with a total of eight 2.53-GHz processing cores (hyperthreading disabled), 12 GB of DRAM, two 8-MB L3-caches each shared between 4 cores, and private L2- and L1-caches with 256 KB and 32 KB, respectively.

Random graphs

Figure 8 shows the performance results form running Dijkstra and parallel Gabow on four different random graphs. These random graphs were generated using the random graph generator in the shortest paths library by Cherkassky et al. [3].

As can be see in Figure 8, parallel Gabow executed serially takes over 4 times longer to run than Dijkstra. On eight cores, however, parallel Gabow attains a speedup of approximately 5, and manages to run faster than Dijkstra. With a measured parallelism of 20 for these random graphs, parallel Gabow's performance may scale on several more processors.

Figure 7 shows an example speedup characteristic for parallel Gabow on a random graph, along with the burdened speedup estimate according to Cilkview. The graph used for this example is a random graph of 1.5 million vertices and 4 million edges generated by the same random graph generator in the shortest paths library by Cherkassky et al.

In Figure 7 the speedup characteristic for parallel Gabow closely fits the burdened estimate from Cilkview, which suggests that the burdening involved in spawning computations in parallel Gabow has a large effect on its real-world performance.

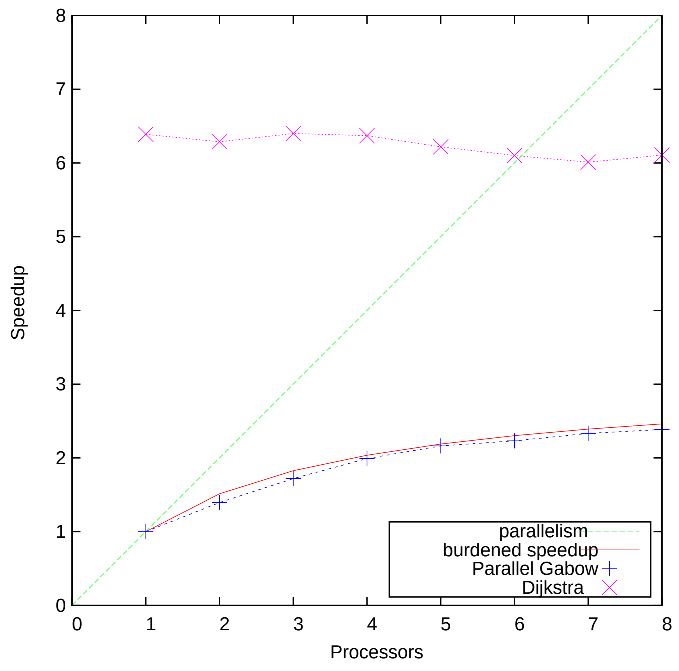

Figure 9: Characteristic performance on a U.S. road network. Speedup values are normalized to parallel Gabow's serial performance.

U.S. road networks

Figure 10 shows the performance results form running Dijkstra and parallel Gabow on four different U.S. road networks. These road network graphs were obtained from the proceedings of the 9th DIMACS Implementation Challenge on Shortest Paths [2].

From Figure 10 parallel Gabow executed serially is 6× to 8× slower than Dijkstra on road networks. Unlike random graphs, however, parallel Gabow attains a speedup of only 1.6 to 2.4 on road networks. Consequently, this suggests that Dijkstra outperforms parallel Gabow on road network graphs for any number of processors.

Figure 9 shows an example speedup characteristic for parallel Gabow on a U.S. road network, along with the burdened speedup estimate according to Cilkview. The graph used for this example is the road network for the northeastern portion of the U.S.

As with random graphs, parallel Gabow's performance on road networks closely follows the burdened speedup curve, suggesting that the burden of spawning parallel computation in parallel Gabow has a substantial effect on its real-world performance. Furthermore, parallel Gabow's performance often falls slightly below the burdened speedup estimate, suggesting that system-related factors, such as the memory bandwidth or sharing, may be restricting parallel Gabow's performance.

8. FUTURE WORK

Variable radix sizes

First, because in practice the queue size on each iteration of Gabow is reasonably small relative to V, it may be feasible for a single iteration of Gabow to examine multiple bits of each weight at a time, speeding up Gabow's performance in practice.

$\lvert V\rvert$	$\lvert E\rvert$	Parallelism	Dijkstra T_1	Parallel Gabow T_1	Parallel Gabow T_8	Parallel Gabow T_1/T_8
0.45M	1M	21.04	1.55	6.75	1.43	4.73
1M	2.7M	21.15	4.11	17.91	3.36	5.34
1.2M	2.8M	20.99	4.68	19.91	3.79	5.26
1.5M	4M	22.76	6.37	26.87	5.00	5.38

Figure 8: Performance results of parallel Gabow and a standard Dijkstra implementation on random graphs. All runtimes are measured in seconds.

Graph	$\lvert V\rvert$	$\lvert E\rvert$	Parallelism	Dijkstra T_1	Parallel Gabow T_1	Parallel Gabow T_8	Parallel Gabow T_1/T_8
COL	0.44M	1.1M	10.99	1.02	7.81	4.82	1.62
FLA	1.1M	2.7M	11.30	2.52	19.73	11.52	1.71
NW	1.2M	2.8M	11.80	2.83	20.09	11.36	1.77
NE	1.5M	3.9M	14.73	3.89	24.90	10.44	2.38

Figure 10: Performance results of parallel Gabow and a standard Dijkstra implementation on US road networks. All runtimes are measured in seconds. The graph COL is the road network for Colorado, and FLA is the road network for Florida. The graphs NW and NE are the road networks for the northwestern and northeastern parts of the U.S. respectively.

Discarding edges

It is possible that, during an iteration of Gabow, if some edge is found to have a potential difference across it that is too large for it to be used in a shortest path, parallel Gabow can remove that edge from the graph. We suspect that these optimizations may improve the empirical performance of Gabow substantially, and we plan to investigate if these optimizations make the serial performance of Gabow's algorithm comparable to that of Dijkstra.

9. CONCLUSION

We designed and implemented a parallel algorithm for solving the single-source shortest paths (SSSP) problem for graphs with nonnegative edge weights, based on Gabow's scaling algorithm for SSSP. We were able to expose parallelism on all but the least advantageous graphs.

Our parallel Gabow algorithm performs well in practice on random graphs, outperforming a simple Dijkstra implementation on six or more cores.

There are two additional optimizations to parallel Gabow we would like to try in future research. First, because in practice the queue size on each iteration of Gabow is reasonably small relative to V, it may be feasible for a single iteration of Gabow to examine multiple bits of each weight at a time, speeding up Gabow's performance in practice. Second, during an iteration of Gabow, if some edge is found to have a potential difference across it that is too large for it to be used in a shortest path, parallel Gabow can remove that edge from the graph. We suspect that these optimizations may improve the empirical performance of Gabow substantially, and we plan to investigate if these optimizations make the serial performance of Gabow's algorithm comparable to that of Dijkstra.

10. REFERENCES

[1] *Standard Template Library Programmer's Guide*, 1994. Available from `http://www.sgi.com/tech/stl/`.

[2] AMS. *9th DIMACS Implementation Challenge — Shortest Paths*, Providence, RI, 2006.

[3] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Math. Program.*, 73:129–174, 1996.

[4] Harold N. Gabow. Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168, 1985.