Master's Project Report: Static and Time-Dependent Shortest Paths on Parallel Platforms[*]

Lijie Ren
Department of Computer Science
University of California, Santa Barbara
Email: lijie.cs@gmail.com

Committee members:

John R. Gilbert (Chair)
Subhash Suri

June 7, 2012

[*]This work was partially supported by NSF grant CNS- 0709385, by DOE grant DE-AC02-05CH11231, by a contract from Intel Corporation, by a gift from Microsoft Corporation and by the Center for Scientific Computing at UCSB under NSF Grant CNS-0960316.

Abstract

We implement the Bellman-Ford algorithm for graphs with constant or time-varying edge costs on a large-scale parallel computing platform. We use the parallel computing library Combinatorial BLAS to achieve high performance with minimal coding effort. We compare our results with reported results using Google's Pregel system to implement a similar algorithm. We show that our implementation runs faster and scales well on synthetic large graphs. We also show that our parallel implementation has better scalability than the serial implementation in the Boost Graph Library.

1 Introduction

Shortest path problems, time-dependent graphs and parallel computing are three vital subjects in today's computer science. Shortest path algorithms are fundamental combinatorial algorithms which have a wide range of applications in scientific research and practice. Time-dependent graphs model extensive real world phenomena and are emerging more frequently due in part to the success of social networks. In addition, parallel computing is gaining unprecedented popularity thanks to the booming of cloud computing and the trend of big data processing.

In this paper, we aim to combine shortest path problems, time-dependent graphs and parallel computing as a whole and give a general solution with high performance. The benchmark we achieved can be used as a baseline for comparing with further optimizations and new implementations in the future.

In the past, each of the three subjects has been extensively investigated. Among all shortest path algorithms, one of the most general is the Bellman-Ford algorithm [1], which only requires that no negative loop exists in the graph. A less general but more efficient algorithm, Dijkstra's algorithm [2], requires no negative edge costs. There exist many other special purpose shortest path algorithms such as A* [3], planar graph shortest path algorithms [4], and Contraction Hierarchy [5], etc. Each of them has its own merits and limitations.

One challenge is to efficiently implement shortest path algorithms on parallel platforms. It is a known fact that certain sequential data structures are hard to parallelize. One of them is the priority queue [6], which is used by most shortest path algorithms. The parallel implementation of such algorithms is possible, such as $\Delta - \textit{Stepping}$ [10]. However, it requires heavy tuning and the performance is not guaranteed in theory.

Time-dependent graphs apply to a large space of computing problems, be it the interactive graphs on social networks, or congestions in traffic networks in heavily populated metropolises. Many classic graph problems are more difficult to solve on time-dependent graphs than on static graphs since the exact cost of each edge is not known without knowing either the departure time or arrival time at the end points. However, when solving the shortest path problem on time-dependent graphs [7], as we will show later, Bellman-Ford still works efficiently, given the departure time at the source vertex in the graph.

In the following section we present a detailed formulation of the problem and a theoretical analysis of the time-dependent Bellman-Ford, which we are going to implement. In Section III we first implement Bellman-Ford on static graphs using Combinatorial BLAS [9], a highly scalable parallel computing library. Then we extend the implementation to solve shortest path problems on time-dependent graphs as well. In Section IV we evaluate the performance of our implementation on different types of metrics.

2 Shortest Path Problem on parallel platforms

2.1 Shortest Paths on Static Graphs

Given a directed graph G with N vertices $v_1, v_2, \ldots, v_N$ and M edges, $e_{i,j}$ from v_i to v_j, each associated with a cost to traverse, $c_{i,j}$, one can compute the cost of traversing any path $(e_{i_1,i_2}, e_{i_2,i_3}, \ldots, e_{i_{k-1},k})$ as $c_{i_1,i_2} + c_{i_2,i_3} + \ldots + c_{i_{k-1},k}$.

The shortest path from a source vertex v_s to any vertex v_i can then be defined as the path with the minimum cost to traverse, among all paths from v_s to v_i one can find in G, if it exists.

```
input  : $G$: directed graph, $v_s$: source vertex
output: $d = \{d_i\}$: distance from $v_s$ to each vertex,
        $p = \{p_i\}$: each vertex's parent.
//Initialization;
$V = \{v_i\}$ = all vertices in $G$;
$E = \{e_{i,j}\}$ = all edges in $G$;
$C = \{c_{i,j}\}$ = costs of all edges;
foreach vertex $v_i$ in $V$ do
    if $v_i == v_s$ then
        $d_i = 0$;
    else
        $d_i = \inf$;
    end
    $p_i = null$;
end
//Edge relaxation;
for $i \leftarrow 1$ to $|V| - 1$ do
    foreach edge $e_{i,j}$ in $E$ do
        if $d_i + c_{i,j} < d_j$ then
            $d_j = d_i + c_{i,j}$;
            $p_j = i$;
        end
    end
end
//Validation;
foreach edge $e_{i,j}$ in $E$ do
    if $d_i + c_{i,j} < d_j$ then
        print "$G$ contains negative loops";
        break;
    end
end
```

Algorithm 1: The Bellman-Ford algorithm

Among all algorithms to solve this problem, the most general one is the Bellman-Ford algorithm, which computes the shortest path tree (the shortest path from v_s to all vertices in G). Bellman-Ford is guaranteed to find the shortest path tree unless there are loops with negative costs in G. If negative loops exist, the shortest path does not exist since a cheaper path can always be constructed by traversing a negative loop one more time. The algorithm is described in Algorithm 1.

It first associates a distance d_i with each vertex v_i and initializes all d_is to $+\infty$ except for the source vertex v_s, whose distance is 0. It then iterates until all t_i converges. In each iteration, it updates the distance of each v_i independently, according to the update rule $d_i = min_{e_{j,i}}\{c_{j,i} + d_j\}$.

It is proven [1] that Bellman-Ford terminates in $N - 1$ iterations if the shortest paths do exist. Furthermore, the time complexity of the algorithm is $O(MN)$. There exist a large amount of less general algorithms that trade generality for efficiency. As a consequence, most of them use a more sophisticated data structure — priority queue [6]. Priority queues do not have a clean and efficient parallel implementation so far, which makes those more efficient algorithms less ideal for solving shortest path problems on parallel platforms. While in the Bellman-Ford algorithm, each d_i is updated independently in each iteration. Therefore, it is inherently a parallel operation.

For the sake of maintaining generality, ease of complexity analysis and the actual implementation, we choose the Bellman-Ford algorithm to implement in parallel and focus our attention on pursuing high performance on large parallel computing platforms.

2.2 Shortest Paths on Time-Dependent Graphs

We may think of cost $c_{i,j}$ as the time required to traverse edge $e_{i,j}$ and the distance d_i is the total time needed to go to v_i from v_s. Then the shortest path problem becomes a real world analogy which is, to find the path from v_s to a destination vertex v_d such that v_d can be reached the earliest. Furthermore, instead of a cost value c_i, we can associate a function $c_{i,j}(t)$ of time t with each edge $e_{i,j}$, denoting the *shortest* traverse time of $e_{i,j}$ when leaving from v_i at time *no earlier than* t. Then we effectively turn the graph into a time-dependent graph with a dynamic version of the shortest path problem defined as the following:

Given a directed graph G with time-dependent edge costs $c_{i,j}(t)$ for each edge $e_{i,j}$, a source vertex v_s, and a departure time t_s, compute the earliest possible arrival time at any vertex v_i if one leaves v_s at t_s.

The reason that we choose not to interpret $c_{i,j}(t)$ as the traverse time of $e_{i,j}$ when leaving from v_i at *exact* time t, is to take non first-in-first-out (FIFO) edges into consideration. That is, for some edge $e_{i,j}$, it may be possible to leave from v_i later, but arrive at v_j earlier. Therefore one gains from simply waiting at v_i until a better departure time. For the simplicity of the following analysis, we incorporate such a wait time into $c_{i,j}(t)$ to make all edges "look FIFO", so that we do not need to consider the "waiting time" when implementing Bellman-

Ford, which has no concept of "waiting" built-in since such a scenario does not exist on static graphs.

For the ease of the following analysis, we derive an arrival-time function $a_{i,j}(t) = c_{i,j}(t) + t$ for each edge $e_{i,j}$, meaning the arrival time at v_j if leaving from v_i no earlier than t. For any path $(e_{i_1,i_2}, e_{i_2,i_3}, \ldots, e_{i_{k-1},k})$, the arrival time at v_{i_k}, when leaving from v_{i_1} no earlier than t, is $a_{i_{k-1},i_k}(\cdots a_{i_2,i_3}(a_{i_1,i_2}(t))\cdots)$. Notice that for static graphs, since the cost to traverse each edge does not depend on the departure time, the arrival time at v_j along the same path, $(e_{i_1,i_2}, e_{i_2,i_3}, \ldots, e_{i_{k-1},k})$, is simply $a_{i_{k-1},i_k}(\cdots a_{i_2,i_3}(a_{i_1,i_2}(t))\cdots) = c_{i_1,i_2} + c_{i_2,i_3} + \ldots + c_{i_{k-1},i_k} + t$.

The key difference between time-dependent graphs and static graphs lies in the capability to know an edge's cost without knowing the departure time. For instance, bidirectional shortest path algorithms will not work for solving time-dependent shortest path problems in that the reverse search started from the destination v_d needs that the edge costs of all in-edges of v_d to be available. But they remain unknown until the forward search from v_s reaches those edges. Fortunately, single directional shortest path algorithms can easily be generalized to solve time-dependent shortest path problems [7]. They start from v_s and whenever an edge $e_{i,j}$ is encountered, the departure time at v_i, which is d_i, would already be computed from the previous iteration. Therefore $c_{i,j}$ can be obtained by a simple lookup, just like in the static scenarios.

However, there is one restriction for the time-dependent graphs, in order for Bellman-Ford to work properly and efficiently. The restriction is that no negative loops should ever exist in G. That is, for any v_i in G, it should not be possible to depart from v_i at t_1 and return to v_i at t_2 through some loop, with $t_2 < t_1$. It is worth noting that unlike static graphs, time-dependent graphs containing negative loops *occasionally* may still have shortest paths defined, given the loop does not remain negative for unlimited times of traversing, i.e., after some times of traversing the loop, all further traversing the loop will result in non-negative costs. Even with this relaxation, it is no longer guaranteed that Bellman-Ford would stop within $N - 1$ iterations, as proven on static graphs. Therefore, the total running time would not be $O(MN)$ any more. In fact, one can easily construct time-dependent graphs which have negative loops but allow Bellman-Ford to finish in arbitrarily many iterations.

With no negative loops in the time-decadent graphs, we now prove that Bellman-Ford is guaranteed to finished within $N - 1$ iterations. It is easy to argue that the shortest path from v_s to any vertex v_i is composed of at most $N - 1$ edges. Therefore we only need to prove that after the nth iteration, all shortest paths with length no greater than n will be found. Again, this can be easily proven by induction.

Given that in each iteration at most M edges could be processed, the worst-case time complexity of time-dependent Bellman-Ford is $O(MN)$, the same as that of the static version.

3 Parallel Bellman-Ford Implementation

We aim to implement Bellman-Ford in parallel efficiently so large graphs can be processed quickly using this general algorithm. We choose to use a C++ library: Combinatorial BLAS (CombBLAS) [9] to facilitate our implementation. The Combinatorial BLAS is an extensible distributed-memory parallel graph library offering a small but powerful set of linear algebra primitives specifically targeting graph analytics. It uses MPI for distributed memory communications. Using CombBLAS allows us to achieve the high performance and scalability we pursue with light coding effort.

```
input  : $G$: directed graph, $v_s$: source vertex
output: $x$: distance and parent of all vertices.
//Initialization;
$V = \{v_i\}$ = all vertices in $G$;
finished = false;
foreach vertex $v_i$ in $V$ do
    if $v_i == v_s$ then
        $x_i.d = 0$;
    else
        $x_i.d = \text{inf}$;
    end
    $x_i.p = null$;
end
//Semi-ring operations;
for $i \leftarrow 1$ to $|V| - 1$ do
    $x = G \odot x$;
    if $x$ is not changed then
        finished = true;
        break;
    end
end
//Validation;
if finished == false then
    print "$G$ contains negative loops";
end
```

Algorithm 2: Parallel Bellman-Ford, not efficient.

CombBLAS has several data structures to support graph computations internally. Graphs can be represented as adjacency matrices, which are stored as sparse matrices in distributed memory. A large number of operations interacting with matrices and sparse/dense vectors have been implemented in highly scalable ways in CombBLAS to facilitate massive graph operations. A very useful type of operation involves semi-rings. They enable customized matrix-vector-multiplication-type operations $G \odot x$ with user defined element-wise '+' and '$\cdot$' operations. For instance, if '+' is defined as arithmetic addition and '$\cdot$' arithmetic multiplication, then $G \odot x$ becomes traditional matrix-vector multiplication Gx. As it turns out, each iteration in static Bellman-Ford, which is $d_i' = \min_j (d_i + c_{j,i})$, can be viewed as a semi-ring operation with '+' defined as $min()$ and '$\cdot$' defined as $(d_i + c_{j,i})$. Similarly, each iteration in time-dependent Bellman-Ford, $d_i' = \min_j (d_j + c_{j,i}(d_j))$, can be viewed as a semi-ring operation with '+' defined as $min()$ and '$\cdot$' defined as $d_j + c_{j,i}(d_j)$.

Given the transformations above, we implement each iteration in Bellman-Ford based on a semi-ring operation $G \odot x$, where $x = \{d, p\}$ is a dense vector containing the information about the distance d_i and parent p_i of each v_i in G. Our first implementation of parallel Bellman-Ford is shown in Algorithm 2.

The implementation works, yet not efficiently. As one can see, all elements in x are processed in each iteration. Since x is a dense vector, the distance and parent values of *all* vertices are going to be updated in every iteration, regardless of whether or not those values of their neighbors have been changed in the previous iteration or not. Therefore, much unnecessary processing is being done.

Our next implementation eliminates these unnecessary updates by incorporating a intermediate result holder called the frontier f. f is a sparse vector containing only the information of the vertices which has been updated in the last iteration. The new implementation is shown in Algorithm 3.

```
input  : $G$: directed graph, $v_s$: source vertex
output: $x$: distance and parent of all vertices.
//Initialization;
$V = \{v_i\}$ all vertices in $G$;
finished = false;
$f = []$;
foreach vertex $v_i$ in $V$ do
    if $v_i == v_s$ then
        $x_i.d = 0$;
        $f_i.d = 0$;
    else
        $x_i.d = \inf$;
    end
    $x_i.p = null$;
end
//Semi-ring operations;
for $i \leftarrow 1$ to $sizeof(V) - 1$ do
    $f = G \odot f$;
    trim $f$;
    if $f$ is empty then
        finished = true;
        break;
    end
    merge $f$ into $x$;
end
//Validation;
if finished == false then
    print "$G$ contains negative loops";
else
    print "done";
end
```
Algorithm 3: Parallel Bellman-Ford, efficient.

After the semi-ring operation in each iteration, we trim f so that it only contains vertices with smaller distance than in x. If no vertex remains, x has converged so we terminate the iterations, otherwise we merge the updated information to x and start the next iteration. Experiments show the optimized implementation outperforms the naive implementation significantly.

The static Bellman-Ford and the time-dependent Bellman-Ford differ only in the '$\cdot$' operation defined in the semi-ring. For static graphs, computing $(d_i + c_{j,i})$ takes constant time. If the time-dependent variant, $d_j + c_{j,i}(d_j)$, can be carried out in the similar amount of time as well, then both of the implementations would have almost the same running time.

4 Experiments

To check whether our implementation has good performance and scalability, we conduct extensive experiments on two types of graphs — binary trees and random graphs. We first compare the performance of our static implementation of Bellman-Ford with that of the time-dependent implementation. Then for each type of the graphs, we first fix the number of cores and test the scalability against the size of the graph. Then we fix the size of the graph and test the scalability against the number of cores. Finally we compare the performance of our static implementation on a single core with the performance of the Bellman-Ford implementation in the Boost Graph Library.

We try to compare our implementation with previous serial/parallel implementations of Bellman-Ford. As far as we know there are no time-dependent implementations available. So we focus on comparing with the static implementations. There are two main implementations that we are aware of, which are in Google's Pregel system (parallel) and Boost Graph Library (serial). Since the exact implementation and architecture of Pregel is not available publicly, we only use their reported performance and compare it with ours, assuming linear scalability. For BGL, we do the comparison on the same machine using one core.

The parallel computing platform that we have access to in this project is a 4 processor Westmere-EX system with 40 physical cores, 80 logical cores (SMT), and 256 GiB of memory. The system runs RHEL 6.1 with gcc version 4.4.6 and Open MPI 1.5.3.

To achieve load balancing across multiple cores, the vertices in the graph are randomly permuted before the Bellman-Ford iterations. The results show our implementation has very high performance and scalability.

4.1 Static Bellman-Ford and time-dependent Bellman-Ford

We artificially generate some random graphs by uniformly creating x out-going edges from each vertex in the graph, where the random variable d follows a log-normal distribution $p(d) = \frac{1}{\sqrt{2\pi}\sigma d} e^{-(\ln d - \mu)^2/\sigma^2}$ with $\mu = 4, \sigma = 1.3$ [8]. The mean out-degree is about 127. The static graphs have uniform edge weight 1 and the time-dependent graphs have edge weights $c_{i,j}(t) = 2t$. Figures 1 and 2 show the runtime of the static and time-dependent implementation with respect to the size of the graphs and the number of cores we use. As we concluded in the previous section, both implementations have nearly identical performance due to the fact that the time to fetch $c_{i,j}$ in the time-dependent implementation is similar to the time for a lookup in the static implementation. Therefore, in the following experiments, we only show the results obtained from the static implementation for simplicity.

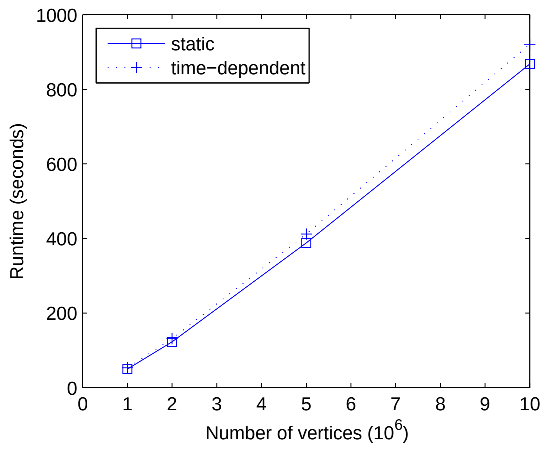

Figure 1: Comparison between static and time-dependent implementations. (Random graphs, single core.)

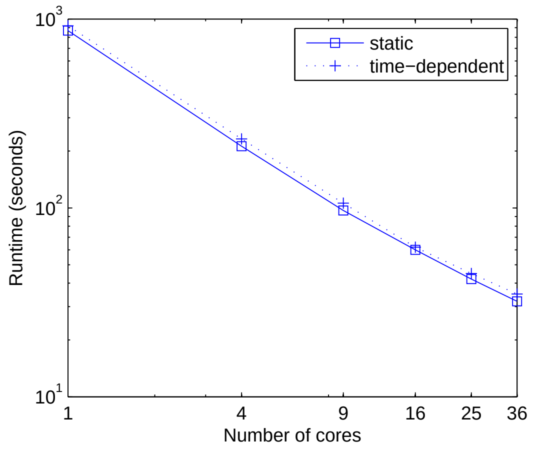

Figure 2: Comparison between static and time-dependent implementations. (Random graphs, 10M vertices.)

4.2 Shortest Paths on Binary Trees

We generate binary trees of different sizes where each non-root vertex has exactly one incoming edge, with all edges' costs set to one.

First we fix the size of the graph to be 100 million vertices and vary the number of cores to use. The results are shown in Figure 3. We obtain a speedup of 18 when increasing the number of cores from 1 to 36. We also depict the number of traversed edges per second (TEPS) per core in Figure 4. The results show that our implementation scales reasonably well with the the number of

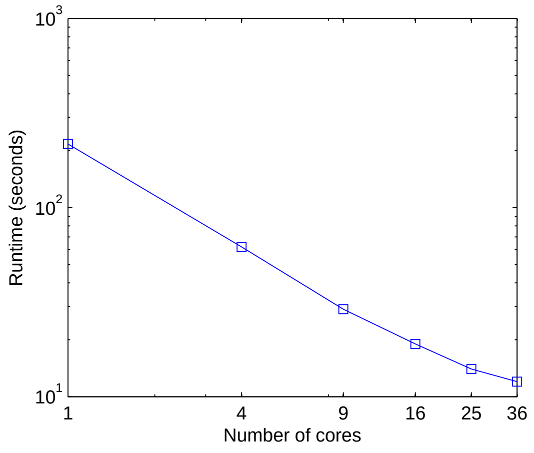

Figure 3: Runtime on binary trees with 100M vertices.

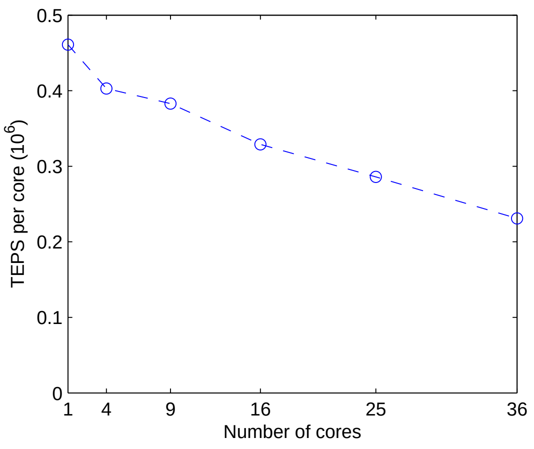

Figure 4: TEPS/core on Binary trees with 100M vertices.

cores on binary graphs.

Then, by fixing the number of cores at 36, we increase the size of the graph gradually from 100 million to 1 billion vertices. As shown in Figure 5, the running time increased by about 10 times when the size of the graph grows by 10 times, demonstrating a very good scalability. TEPS/core results are shown in Figure 6.

We compare our results with reported performance of Google's Pregel system. Using 50 cores, their Bellman-Ford-flavored implementation takes 174 seconds on the binary tree with 1 billion vertices, while ours takes 147 seconds using 36 cores. Both results show a linear increase of runtime against the graph size.

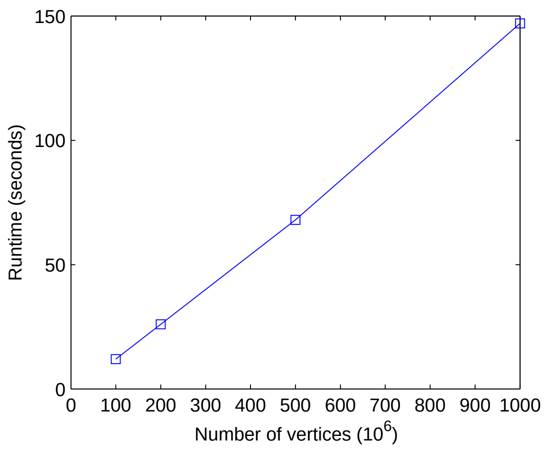

Figure 5: Runtime on binary trees using 36 cores.

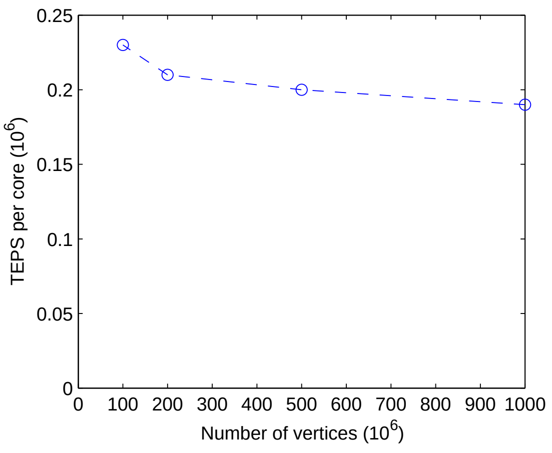

Figure 6: TEPS/core on binary trees using 36 cores.

4.3 Shortest Paths on Random Graphs

As was done for Google's Pregel [8], we also conducted experiments on random graphs introduced in Section 4.1. This kind of graph bears more similarity with social networks and is more parallelism-friendly than binary trees due to the large number of out-degrees. Fixing the number of vertices to be 10 million (hence approximately 1.27 billion edges), the runtime and TEPS/core against the number of cores are shown in Figure 7 and 8. We obtain a speedup of 27 when the number of cores is increased from 1 to 36.

Figures 9 and 10 show the runtime and TEPS/core against the size the graph using 36 cores. From 10 million vertices (1.27 billion edges) to 100 million

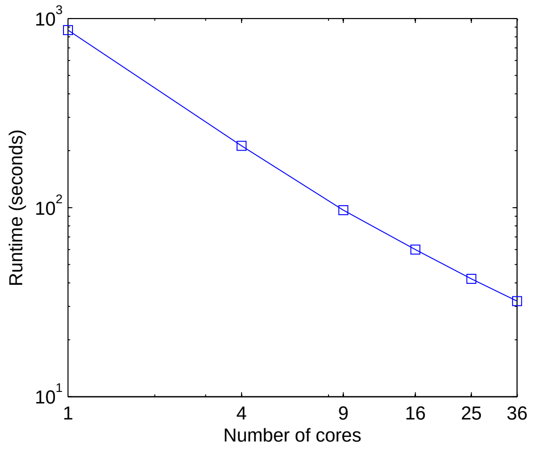

Figure 7: Runtime on random graphs with 10M vertices.

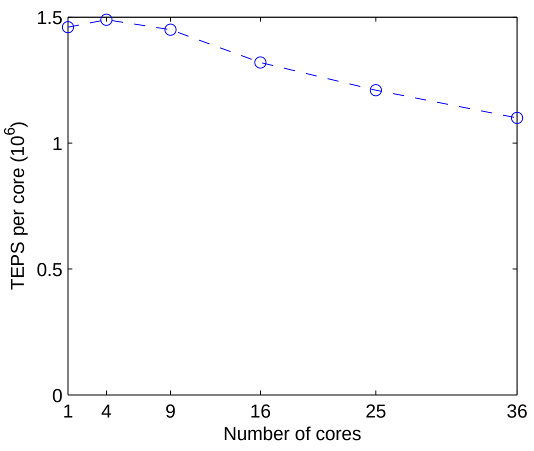

Figure 8: TEPS/core on random graphs with 10M vertices.

vertices (12.7 billion edges), the running time also increased 10 times.

Again, we compare our result with Pregel's. The closest comparison we obtain is that, when the graph size is 100 million vertices, our runtime is 325 seconds on 36 cores, while Pregel's runtime is about 80 seconds on 800 cores. That is, we use 1/22 of their resources and achieve performance 4 times worse than theirs. We speculate a better performance than theirs when we increase the number of cores to 800.

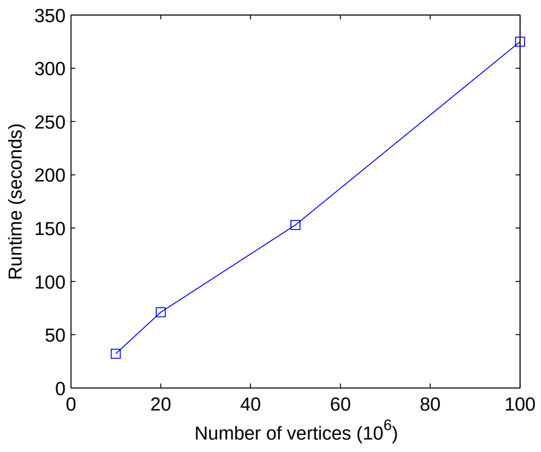

Figure 9: Runtime on random graphs using 36 cores.

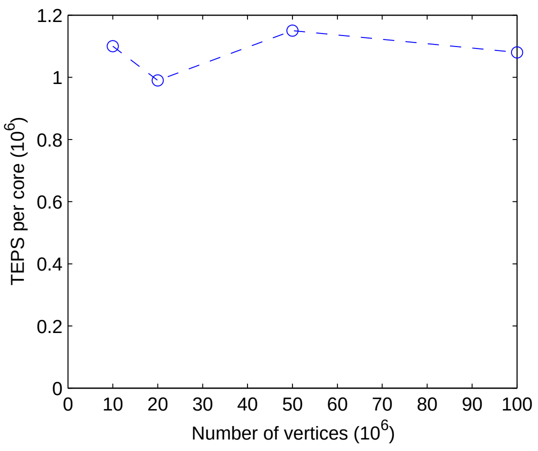

Figure 10: TEPS/core on random graphs using 36 cores.

4.4 Scalability on a Single-Core Machine

We compare our implementation with the one in the Boost Graph Library (BGL) on a single core. Figures 11 and 12 show the results on binary trees and random graphs, respectively. As one can see, our implementation typically has better performance and scalability on binary trees than the BFL version.

However, on random graphs, BGL outperforms ours by about a factor of about 5 in the beginning. This factor gets smaller when the graph gets larger, which shows that our implementation has a better scalability than BGL's. Since our implementation is built upon Combinatorial BLAS, which is specifically optimized for parallel efficiency. The slow down by 5 times when the graph is

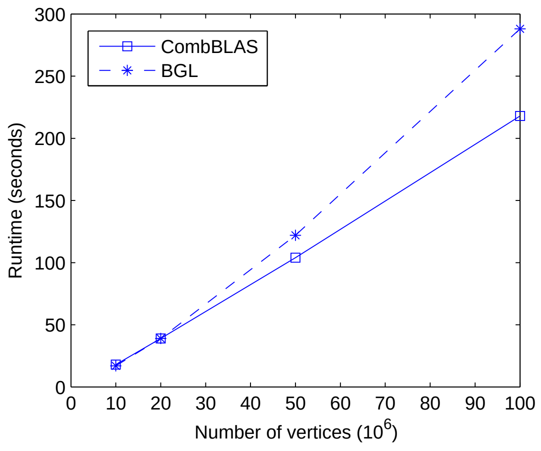

Figure 11: Comparison between our implementations and BGL's. (Binary trees, single core.)

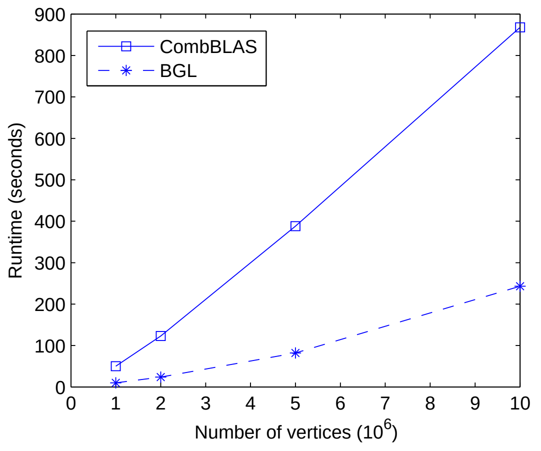

Figure 12: Comparison between our implementations and BGL's. (Random graphs, single core.)

small is a penalty we consider reasonable brought by the parallelization.

The correctness of our implementation is ensured by the fact that it does the same number of Bellman-Ford iterations with BGL's implementation. And for any vertex, its distance converges to the same value in both of the implementations.

5 Conclusion

Despite much work that has been done in the past, the parallelization of shortest path algorithms remains a difficult problem. In this project we present a parallel implementation of the Bellman-Ford algorithm using a C++ distributed-memory parallel graph library, Combinatorial BLAS. Experiments show that the our implementation scales up to 36 cores with graphs which have up to more than 10 billion edges.

Given that CombBLAS is still a work in progress, we expect better performance from optimizing the library and our implementation in the future. We hope our general and efficient solution of solving shortest path problems in parallel can be exploited by real-world problems, such as highway routing and social network analysis, on large-scale distributed systems.

References

[1] Richard Bellman, *On a routing problem.* Quarterly of Applied Mathematics 16.

[2] E.W. Dijkstra, *A note on two problems in connexion with graphs.* Numerische Mathematik 1.

[3] P. E. Hart, N. J. Nilsson and B. Raphael, *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*. SIGART Newsletter 37.

[4] Sairam Subramanian, Philip Klein, Satish Rao and Monika Rauch, *Faster Shortest-Path Algorithms for Planar Graphs.* Journal of Computer and System Sciences 1994.

[5] Robert Geisberger, *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.* Diploma Thesis.

[6] P. van Emde Boas, *Preserving order in a forest in less than logarithmic time.* In Proceedings of the 16th Annual Symposium on Foundations of Computer Science, pages 75-84. IEEE Computer Society, 1975.

[7] Luca Foschini, John Hershberger and Subhash Suri, *On the Complexity of Time-Dependent Shortest Paths.* ACM SODA 2011.

[8] Malewicz G, et al, *Pregel: a system for large-scale graph processing.* PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing.2009.

[9] Aydin Buluc and John R. Gilbert, *The Combinatorial BLAS: Design, implementation, and applications.* The International Journal of High Performance Computing Applications, 2011.

[10] Ulrich Meyer and Peter Sanders, Δ-*stepping: A parallel single source shortest path algorithm*. Proceedings of the 6th Annual European Symposium on Algorithms.