A new synchronization in parallel shortest paths analysis for massive road networks

HUANG Yuefeng\textsuperscript{a,b,*} and ZHONG Ershun\textsuperscript{b}

\textsuperscript{a}Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, 11A Datun Road, Chaoyang District, Beijing 100101, China; \textsuperscript{b}Graduate University of the Chinese Academy of Sciences, 19A Yuquan Road, Beijing 100034, China

(Received 28 March 2011; final version received 16 September 2011)

To effectively solve the single-source shortest path (SSSP) problem for massive road networks in geographical information systems, a new synchronization method is proposed in the implementations of parallel SSSP algorithm. It applies spinlock by inline assembly language for the sake of small overheads of controlling the interaction of multiple threads. The performance of our method is compared with widely used Pthreads application programming interfaces and the powerful sequential solution given by DIMACS. The experimental platform is a shared address space workstation with two processors (i.e. eight cores) at a clock speed of 3 GHz. Problem instances for experiments contain a directed road networks of the USA with more than 23 million vertices and 57 million edges, and its 11 subnetworks of variant sizes. This method answers the SSSP of the USA road network in 1231 ms, while Pthreads costs 1808 ms and DIMACS sequential solution takes 4856 ms. It achieves a speedup of 3.95, which is 47% faster than Pthreads with the speedup of 2.69. When the size of instance is larger, our method achieves a better performance.

Keywords: parallel shortest path; Pthreads; multiple threads; synchronization

1. Introduction

A road network is a kind of spatial data usually found at a massive scale, with millions of vertices and edges, nearly constant degree, and nonnegative edge weights. A shortest paths query is a typical kind of spatial analysis in geographical information systems. It is used in many real-world applications like route planning in road networks, timetable information for railways, or scheduling for airplanes. Theoretically, it is defined as single-source shortest path (SSSP) problem in computer science, finding shortest paths from a given source to all other vertices in graphs, where the length of a path is the sum of the weights of its edges. Numerous sequential SSSP algorithms have been developed, achieving less and less asymptotic runtime. Unfortunately, the algorithms are still too slow to be used on huge data-sets. For this reason, parallel SSSP algorithms are playing a more important role in solving large-scale problems since the flourishing of multiple-core processors.

Threads help to execute programs in parallel on platforms with shared address memory space and multiple processors. While communication is implicit in shared-address-space programming, much of the effort associated with writing correct threaded programs is spent on synchronizing concurrent threads with respect to their data accessing or scheduling. The operating systems offer application programming interfaces (APIs) for creation, termination, parallelization, and synchronization of threads. Pthreads is one of the most important paradigms which is emerged by POSIX as the standard threads API and supported by most vendors. It provides support for concurrency, synchronization, and data handling by explicitly setting up mutexes, condition variables, and so on. As a system function, it deals much more work for threads which are transparent to programmer. In some sense, it is functional but not effective enough for extreme need of performance.

SSSP is a primary problem and well studied both in sequence and parallel processing research. For a graph with \( n \) vertices and \( m \) edges, Thorup (2) solved SSSP for undirected graphs with nonnegative integer weights in \( O(m) \) time. Hagerup (3) extended Thorup’s approach to digraphs in \( O(m + n \cdot \log \log n) \) time. Both provide the best results for general graphs. Road networks are often seemed as planar graphs by removing a few crossing edges for a better time bound. Klein et al. (4, 5) gave an \( O(n \cdot \log^2 n) \) algorithm for planar digraphs with real edge weights and an \( O(n) \) algorithm for planar digraphs with nonnegative edge weights.

No parallel PRAM algorithm is known that runs in sublinear time and \( O(m + n \cdot \log n) \) work for arbitrary digraphs with nonnegative edge weights. Driscoll et al. (6) presented an \( O(n \cdot \log n) \) time algorithm requiring \( O(m + n \cdot \log n) \) work. The algorithm of Brodal et al. (7)
completes $O(m + n \cdot \log n)$ work in $O(n)$ time. These algorithms find vertices in the order of Dijkstra’s algorithm and only perform edge relaxations in parallel priority queues: relaxed heaps. They fail to break the worst-case time bound of $\Omega(n)$.

Not only improvements in theory but also in experiments have been proposed in recent years. There are two types of platforms for parallel graph algorithms. One platform is computers that support massive threads, such as Cray MTA-2 with 40 processors and shared memory. Madduri et al. (8) did an experimental study of SSSP on MTA-2 and got a relative speedup of close to 30. Bader and Madduri (9) studied breadth first search on a MTA-2. Edmonds et al. (10) described the parallel implementation of SSSP in the Parallel Boost Graph Library. Crobak et al. (11) applied the MultiThreaded Graph Library to the SSSP algorithm. The other platform is Graphics Processing Units (GPU) offering dedicated data-parallel hardware, such as the GTX 580 with 512 CUDA cores. Harish et al. (12) computed SSSP on a graph with 10 million vertices in about 1.5 s using a GTX8800 GPU.

Meyer and Sanders (15) presented $\Delta$-stepping algorithm that divides Dijkstra’s algorithm into a number of phases, each one can be executed in parallel. For random graphs with uniformly distributed edge weights, it runs in sublinear time with linear average case work. On a random $\Delta$-regular graph instance ($2^{10}$ vertices and $\Delta=3$), they reported a speedup of 9.2 on the 16 processors of an Intel Paragon machine. In this paper, we regenerate this algorithm with a new work distribution method. In our implementation, synchronizations are needed, so we present a new synchronization method and compare it with Pthreads.

In this paper, we propose an effective synchronization method and compare it with Pthreads and DIMACS sequential implementation, for evaluation of performance and report of parallel speedup.

2. Main results

Parallelization and synchronization contribute much to the performance of the parallel shortest path algorithm. Shortest path algorithms have an irregular structure for parallelization, to support poor grained parallelization. The sequential steps and concurrent steps are mixed together. Moreover, there are as many synchronizations as vertices in our implementation. Consequently, the parallel shortest path algorithm is not as effective as prefix sums or list ranking in parallel, so our new method makes good sense. The following main results are described in this paper.

Our new synchronization effectively solves SSSP for massive road networks in the order of 23 million vertices and 57 million edges. Our implementation performs about 1 s to answer an SSSP query. It achieves a speedup of almost 3.95, 47% faster than Pthreads with a speedup of 2.69, in comparison with the sequential implementation by DIMACS. It has a competitive performance for application.

Our experiments also show a conservation of performance when the size of problem varies. Especially, when the size is bigger, our method gains better speedup.

The rest of paper is organized as follows. In Section 2, we introduce sequential $\Delta$-stepping algorithm to show a general idea about how the algorithm works. In Section 3, we give the details of Pthreads synchronization and our new method. Sections 4 and 5 describe our experimental setup and performance results and analysis. Eventually, our experimental study is concluded and future plans are overlooked in Section 6.

3. Review of the $\Delta$-stepping algorithm

3.1. Preliminaries

Let $G=(V,E)$ be a graph with $n$ vertices and $m$ edges. Let $s \in V$ denotes the source vertex. Each edge $e \in E$ is assigned a nonnegative real weight by the length function $l:E \rightarrow R$. Define the weight of a path as the sum of the weights of its edges. The SSSP problem computes $\Delta(v)$ of every vertex, the weight of the shortest path is from $s$ to $v$. We set $\Delta(s)=0$ and $\Delta(v)=\infty$, if $v$ is unreachable from $s$.

Most shortest path algorithms maintain a tentative distance value for each vertex, which are updated by edge relaxations. Let $d(v)$ denotes the tentative distance of a vertex $v$ and is initially set to $\infty$. Shortest path algorithms are classified into two types: label-setting or label-correcting. Label-setting algorithms (such as Dijkstra’s algorithm) perform edge relaxations only from vertices with shortest path settled ($d(v)=\Delta(v)$) to vertices with shortest path unsettled ($d(v)\neq\Delta(v)$). Label-setting algorithms solve SSSP after $m$ edge relaxations at most. Label-correcting algorithms (such as Bellman–Ford algorithm) perform the same edge relaxations with label-setting algorithms in addition to edge relaxations between vertices with shortest paths unsettled. In this way, label-correcting algorithms solve SSSP after $mn$ edge relaxations. $\Delta$-stepping belongs to the label-correcting type of shortest path algorithms, and degenerates to Bellman–Ford algorithm under some parameter settings.

3.2. $\Delta$-stepping algorithm

The sequential $\Delta$-stepping algorithm shown in Figure 1 resembles the “approximate bucket implementation of Dijkstra’s algorithm” (16). It maintains a one-dimension array as $B$ buckets such that $B[i]$ contains vertices set $\{v \in V \text{ and } l(d(v)) \in [i \cdot \Delta, (i+1) \cdot \Delta]\}$. The parameter $\Delta$ is a positive real number that is called “step width” or “bucket width.” For maximum shortest path weight $L$, the array $B$ must contain $\lceil L/\Delta \rceil$ buckets. By cyclically reusing empty buckets, only $b=\max_{e \in E} l(e)/\Delta + 1$
buckets are sufficient. $B[i]$ is in charge of all tentative distances in $[(j \cdot b + i) \cdot \Delta, (j \cdot b + i + 1) \cdot \Delta]$ for all $j \geq 0$.

Edges are relaxed phase by phase. A phase is an iteration of the inner while loop signed by (c) schemes of sequential $\Delta$-stepping algorithm. In each phase, all vertices from the first nonempty bucket are removed and light edges out of these vertices are relaxed. Light edge relaxations may deserve new vertices inserted into the current bucket, including previously deleted vertices if their tentative distances have been decreased by these relaxations. After light edge relaxations, heavy edges are relaxed to empty the current bucket. Once the current bucket becomes empty, all vertices in its distance have been assigned their shortest paths.

The algorithm sequentially relaxes edges from bucket to bucket until there is no nonempty buckets left, terminating with all shortest paths from the source.

### 3.3. Complexity

Edge relaxations and phases contribute to the complexity of $\Delta$-stepping algorithm. The overhead compared to Dijkstra’s algorithm is due to reinsertions and relaxations, i.e. insertions of vertices which have previously been deleted and relaxations of their outgoing edges.

We define $n_\Delta$ as the total number of reinsertions and $m_\Delta$ as the total number of relaxations. If $L$ is the maximum weight of any shortest path in $G$, i.e. $L = \max \{d(v) : d(v) \leq \infty\}$, Meyer and Sanders gave the following theorem. We give a short proof, more details of which are referred in Meyer’s (15).

**Theorem 1.** Sequential $\Delta$-stepping algorithm can be implemented to run in time $O(n + m + L/\Delta + n_\Delta + m_\Delta)$.

**Proof.** The loop control traverses $L/\Delta$ buckets in all, and each of them is searched only once and in constant time from lines (a) to (c). There are $n + m + n_\Delta + m_\Delta$ edge relaxations, and each can be requested and completed in constant time from lines (d) to (h).

Edge relaxations in the same phase can be done in an arbitrary order, as long as an individual relaxation is atomic. For a better performance, edge relaxations of each phase are done in parallel. Assuming processors number is $p$, we get the following theorem briefly.

**Theorem 2.** Parallel $\Delta$-stepping algorithm can be implemented to run in time $O(L/\Delta + (n + m + n_\Delta + m_\Delta) / p)$.

### 4. Parallel implementation

We implement $\Delta$-stepping algorithm in computers offering concurrent-read and exclusive-write by multithreading. In the version of Meyer and Sanders’s implementation, edge relaxations are distributed to some processor randomly by preprocessing. Their method necessitates a big workload imbalance when the thread number is small. To improve on this, we distribute edge relaxations to threads at the beginning of every phase search. This means distinct relaxations may decrease the tentative distance of the same vertex by distinct threads. In order to prevent incoherent results occurring due to changes in the same data item in parallel, synchronization is considered in our implementation.
For a graph with \( n \) vertices and \( m \) edges, the overhead of synchronizations by our implementation is in the order of \( O((n + m + n_\Delta + m_\Delta)/p) \). Synchronization is significantly heavier than other general operations. It contributes to excessive time cost for the parallel shortest path algorithm. We first describe the synchronization primitives by Pthreads in Subsection 3.1 and then our new synchronization as a counterpart in Subsection 3.2. The details of implementation of \( \Delta \)-stepping algorithm are ignored, which are referred in Meyer’s (15).

### 4.1. Synchronization by Pthreads

Edge relaxation is a test-and-update operation of shared memory. There are two steps in sequence, i.e. test and update. They must be executed by only one thread without any interruption; else a nondeterministic value is written to the memory. Consequently, edge relaxation is demanded to be atomic. Threaded APIs provide support for implementing an atomic operation using mutual exclusion locks (mutex-locks). There are two states for a mutex-lock: locked and unlocked. At any time, only one thread can lock a mutex lock. An atomic operation is bracketed as a critical section by a mutex-lock. To do the operation, a thread must first try to acquire the mutex-lock. If the mutex-lock has already been locked, some other thread is doing the operation. The thread trying to acquire the lock is blocked. After the operation is completed, the mutex-lock is unlocked; the hanging thread is invoked to do the operation. The Pthreads provide a number of functions for handling mutex-locks, we take the pthread_mutex_lock as an example.

The scheme of the implementation by Pthreads is shown in Figure 2(a). Every tentative distance of vertices is initialized with a mutex-lock by preprocessing. Before any edge relaxation, the thread tries to acquire the mutex-lock assigned with the tentative distance stored in shared memory in (a1). After that, the tentative distance is compared with the new one in (a2). If it is bigger, it is decreased to be smaller in (a3). In the end, the mutex-lock is unlocked in (a4). Threads cooperate in this manner to do edge relaxations in each phase. Pthreads are effective APIs for the synchronization of multiple threads.

### 4.2. Our synchronization

For a parallel shortest path algorithm, synchronizations charge a big time cost for their frequent occurrence. We design a new synchronization method with a spin-lock

![Figure 2. Schemes of a phase by Pthreads API (a) and our synchronization (b).](image-url)
pattern. This synchronization uses a busy loop to check an additional flag. When the flag shows there is not any other thread is in the control of the critical section, the checking thread sets the flag and takes charge of the critical section. After its runtime, it sets the flag back again so that other waiting thread can enter the critical section. The key point is that the check and switch of the flag must be an atomic operation. For good performance, we give a function in inline assembly language for checking and switching the flag atomically, as shown in Figure 2 (b).

Function “CmpXch” in (b5) compares the pdst with cmp, if they are equal, pdst is replaced by src and the former value of pdst is returned, else the value of cmp is returned. “CmpXch” is an atomic function. The “while” loop in (b1) is our spin lock. A thread in the spin lock checks whether the tentative value is negative. If it is negative, there is another thread working with the tentative value and the checking thread goes to another loop to check once again. If it is positive, it sets the tentative value to negative and begins to operate on the tentative value. The tentative value is considered to be decreased by (b2). If the tentative value can be decreased, it is set to the new positive value by (b3), else it is set to original positive value by (b4).

Our new synchronization method involves only inline assembly instead of Pthreads APIs. It cuts down on complex operations by operating system for controlling the interaction of multiple threads. It is expected to yield better performance in experiments.

5. Experimental setup
We evaluate performance of our synchronization method on a dual Intel Xeon workstation, running a Linux 2.6.32-24-generic. Every processor has a clock speed of 3.0 GHz with four cores. There are in total eight cores of the whole system supporting eight hardware threads concurrently. The codes are written in GNU C and compiled by gcc with –O2 optimization flag. Two versions are almost the same except different synchronization methods. We ignore the time cost of loading the network data into internal memory, allocating memory for the structure of the algorithm, and so on. These are reused in application, so our disregard is reasonable.

Experiment 1 compares our synchronization method with Pthreads while the thread number varies. We implement them on the road network of the USA from the DIMACS shortest path implementation challenge (1). This massive road network contains 23,947,347 vertices and 58,333,344 edges with distance weights. We run separately two versions of the algorithm on this network, with the thread number stepping from 1 to 12. To get an average result, we choose sources by uniformly spreading them in the map and averaging their running time. One hundred and nineteen grids are spread uniformly and the vertex (if there is) nearest to the center of each grid is chosen as a source, shown in Figure 3.

Experiment 2 examines the performance of two versions while the instance size varies. Subnetworks of the USA road network are also available from the DIMACS shortest path implementation challenge. There are 11 subnetworks with vertex number from 264,346 to 14,081,816 and edge number from 730,100 to 33,866,826. The sources of each subnetwork are chosen randomly by the benchmark toolkit in DIMACS. More details about these problem instances are shown in Table 1. The “City” column is the abbreviated names of cities, for example New York is abbreviated as NY. The “Vertex” and “Edge” columns are the vertex and edge number, respectively. The “Min weight” and “Max weight” columns are the minimum and maximum edge weight, which determine the size of bucket structure of the parallel shortest path algorithm. The “Case” column is the source vertex number.

6. Results and analysis
The results of Experiment 1 are shown in Figures 4 and 5. Figure 4 shows the runtime of implementations by Pthreads, our method, and solution of the DIMACS benchmark. DIMACS applies a sequential SSSP algorithm, so its runtime is constant via thread number, i.e.
4856 ms. Our synchronization method performs much faster than Pthreads, shrinking from 500 to 2000 ms on account of thread number. Most impressive is that when the thread number is eight, two parallel implementations occur both achieve best performance. Our method is 1231 ms, while Pthreads are 1808 ms. After that, they all become worse when thread number become bigger. The reason for this is that our platform contains eight cores and supports eight threads running concurrently. If there are more than eight threads, the context switching of threads charges a lot, leading to greater time cost.

Figure 5 shows the comparison of speedup of two counterparts. We formally define the speedup as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem. In this experiment, we set DIMACS sequential algorithm as the best sequential algorithm for comparison, i.e. 4856 ms. The implementation of our method gains a speedup of nearly 3.95 at most, while the implementation of Pthreads is 2.69 at most. Our method runs 47% faster than Pthreads.

The results of Experiment 2 are shown in Figures 6 and 7. This experiment shows that while the size of instances change from small to big, our method maintains a good speedup. Figure 6 shows that the runtime of implementations by DIMACS, Pthreads, and our method grows with the size of instance. The axis of runtime is in the log pattern. Figure 7 shows the speedup of runtime in Figure 6. They illustrate that our method runs 50% faster than Pthreads in shortest paths algorithm when instance size varies. When instance size is small, our method has a small speedup because of the control overhead for parallelism. When the instance is big, our method achieves splendid speedup of more than three. With eight hardware threads, the biggest speedup is near four. Shortest path

Table 1. Description of problem instances.

| City | Vertex | Edge   | Min. weight | Max. weight | Case |
|------|--------|--------|-------------|-------------|------|
| NY   | 264,346| 730,100| 1           | 36,946      | 532  |
| BAY  | 321,270| 794,830| 1           | 94,305      | 438  |
| COL  | 435,666| 1,042,400| 1          | 137,384     | 323  |
| FLA  | 1,070,376| 2,687,902| 1         | 214,013     | 131  |
| NW   | 1,207,945| 2,820,774| 1         | 128,569     | 116  |
| NE   | 1,524,453| 3,868,020| 1         | 63,247      | 92   |
| CAL  | 1,890,815| 4,630,444| 1         | 215,354     | 74   |
| LKS  | 2,758,119| 6,794,808| 1         | 138,911     | 51   |
| E    | 3,598,623| 8,708,058| 1         | 200,760     | 39   |
| W    | 6,262,104| 15,119,284| 1         | 368,855     | 22   |
| CTR  | 14,081,816| 33,866,826| 1        | 214,013     | 10   |
analyst is a kind of algorithm whose bottleneck is not calculation by processors, but transportation of data by memory. Our platform has a quad memory tunnel, so the speedup of four at most is reasonable.

According to our experiment results, we find the following corollary.

**Corollary 3.** Our synchronization performs better than Pthreads in the implementation of parallel shortest path algorithm. It runs near 50% faster than Pthreads on big problem instances.

7. Conclusion and future work

In this paper, we propose and evaluate a new synchronization method for a parallel shortest path algorithm. We study the algorithm performance for massive road networks on a general multiple cores workstation. We compare the performance of our method and Pthreads. We observe that our method is more effective than Pthreads in the implementation of parallel shortest path algorithm. We also study the dependence of performance on the size of problem instance. Our method performs better on bigger instances.

There are many other paradigms for parallel programming, such as MPI in message-passing paradigms with OpenMP as an open standard. GPU provides high computation power in parallel. Parallel programming models such as CUDA allows up to millions of threads to run in parallel. An implementation based on CUDA will be considered in the future.

**Notes on contributors**

Huang Yuefeng, PhD, interested in theoretical and engineering graph algorithms.

Zhong Ershun, PhD, interested in theory and technology of geographical information system.

**References**

(1) DIMACS. 9th DIMACS Implementation Challenge: Shortest Paths. 2006, http://www.dis.uniroma1.it/~challenge9/download.shtml

(2) Thorup, M. Undirected Single-source Shortest Paths with Positive Integer Weights in Linear Time. J. ACM 1999, 46, 362–394.

(3) Hagerup, T. In Improved Shortest Paths on the Word RAM. Proceedings of the 27th International Colloquium on Automata, Languages and Programming, Geneva, Switzerland, 2000.

(4) Klein, P.N.; Mozes, S.; Weimann, O. Shortest Paths in Directed Planar Graphs with Negative Lengths: A Linear-Space $O(n \log n)$-Time Algorithm. Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms, New York, NY, USA, 2009.

(5) Klein, P.N.; Rao, S.; Rauch, M.; Subramanian, S. Faster Shortest-Path Algorithms for Planar Graphs. Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Quebec, Canada, 1994.

(6) Driscoll, J.R.; Gabow, H.N.; Shrairman, R.; Tarjan, R.E. Relaxed Heaps: An alternative to Fibonacci Heaps with Applications to Parallel Computation. Commun. ACM 1988, 31 (11), 1343–1354.

(7) Brodal, R.E.; Träff, J.L.; Zaroliagis, C.D. A Parallel Priority Queue with Constant Time Operations. J. Parallel Dist. Computing 1998, 49 (1), 4–21.

(8) Madduri, K.; Bader, D.A.; Berry, J.W.; Crobak, J.R. Parallel Shortest Path Algorithms for Solving Large-Scale Instances. The 9th DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ, 2006.

(9) Bader, D.A.; Madduri, K. Designing Multithreaded Algorithms for Breadth-First Search and ST-Connectivity on the Cray MTA-2. Proceedings of the 2006 International Conference on Parallel Processing, Columbus, OH, USA, 2006.

(10) Edmonds, N.; Breuer, A.; Gregor, D.; Lumsdaine, A. Single-Source Shortest Paths with the Parallel Boost Graph Library. The 9th DIMACS Implementation Challenge: The Shortest Path Problem, Piscataway, NJ, 2006.

(11) Crobak, J.R.; Berry, J.W.; Madduri, K.; Bader, D.A. Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture. Proceedings of 2007 International Parallel and Distributed Processing Symposium, Long Beach, CA, USA, 2007.

(12) Harish, P.; Narayanan, P.J. Accelerating Large Graph Algorithms on the GPU Using CUDA. Proceedings of the 14th International Conference on High Performance Computing, Goa, India, 2007.

(13) Delling, D.; Goldberg, A.V.; Nowatzky, A.; Werneck, R.F. PHAST: Hardware-Accelerated Shortest Path Trees. Technical Report MSR-TR-2010-125, Microsoft Research, 2010.

(14) Harish, P.; Vineet, V.; Narayanan, P.J. Large Graph Algorithms for Massively Multithreaded Architecture. Technical Report, IIIT/TR/2009/74, Center for Visual Information Technology, International Institute of Information Technology, Hyderabad, 2009.

(15) Meyer, U.; Sanders, P. Δ-stepping: A Parallelizable Shortest Path Algorithm. J. Algorithms 2003, 49, 114–152.

(16) Dijkstra, E.W. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1959, 1, 269–271.