A Parallel Shortest Path Algorithm Based on Graph-Partitioning and Iterative Correcting

Yuxin Tang[1,2,3], Yunquan Zhang[1,2], Hu Chen[1,2,3]
{tangyx, zyq, chenhu}@mail.rdcps.ac.cn
1) Laboratory of Parallel Computing, Institute of Software, Chinese Academy of Sciences
2) State Key Laboratory of Computer Science, Chinese Academy of Sciences
3) Graduate University of Chinese Academy of Sciences

Abstract

In this paper, we focus on satisfying the actual demands of quickly finding the shortest paths over real-road networks in an intelligent transportation system. A parallel shortest path algorithm based on graph partitioning and iterative correcting is proposed. After evaluating the algorithm using three real road networks, we conclude that our graph-partitioning and iterative correcting based parallel algorithm has good performance. In addition, it achieves more than a 15-fold speedup on 16 processors in an IBM cluster over these real road networks.

Key words: *parallel shortest path algorithm, intelligent transportation, parallel computing, graph partitioning*

1. Introduction

With the development of intelligent transportation and its corresponding applications, the increasingly intensive demands of dynamically solving the shortest paths problem in real time has coincided with the increasing growth in real-road network scale. The classic problem of finding single source shortest paths (SSSP) over a network plays a significant role in many transportation related analyses.

SSSP algorithms are highly sensitive to the type of graph processed. In addition, some graph types, such as road networks, do not offer enough parallelism for these algorithms to work well[1]. In this paper, we focus on satisfying the actual demands of quickly locating the most effective shortest paths over real-road networks in an intelligent transportation system. We propose and implement a parallel shortest path algorithm schema based on graph partitioning and iterative correcting. We also evaluate our parallel algorithm's performance using three real-road networks in this paper. And from the experiments, it can be concluded that the parallel algorithm based on graph-partitioning and iterative correcting has good speedup ratio and efficiency.

The rest of this paper proceeds as follows. Section 2 provides some background propaedeutics of SSSP and related work. Section 3 details our parallel algorithm. In section 4 we evaluate the algorithm's performance using real road networks. Finally, section 5 presents our conclusions and future work.

2. Background and related works

Let us first formally introduce some notations and definitions about a network or a graph. A network is a graph $G = (N, E)$ consisting of an indexed set of nodes N with $n = |N|$ and a spanning set of directed edges E with $m = |E|$. Each edge is represented as an ordered pair of nodes, in the form from node i to node j, denoted by (i, j). Each edge (i, j) has an associated numerical value l_{ij}, which represents the distance or cost incurred by traversing the edge. In this paper, we assume that bidirectional travel between a pair of nodes i and j is represented by two distinct directed edges $(i, j)$and (j, i). Given a directed network $G = (N, E)$ with known edges length l_{ij} for each edge $(i, j) \in E$, the single source shortest path (SSSP) problem is to find the shortest distance (least cost) path from a source node s to every other node in the node set N. These one-to-all shortest paths can be represented as a directed out-tree rooted at the source node s. This directed tree is referred to as a shortest path tree.

Many sequential algorithms about shortest paths have been proposed and researched by scientists in related fields. The labeling method is a central procedure in most of shortest path algorithms. In a graph $G = (N, E)$, each node i maintains three pieces of information while constructing a shortest path tree: the distance label, $d(i)$, the parent node, $p(i)$, and the node status, $S(i)$. The distance label, $d(i)$, stores the shortest path distance from s to i during iteration. Upon the end of an algorithm, $d(i)$ represents the unique shortest path from s to i. The parent node $p(i)$ records the node that immediately precedes node i in the out-tree. The node status, $S(i)$, can be one of the following: *unreached*, *temporarily labeled* and *permanently labeled*. When a node is not scanned, it is unreached. Normally the distance label of an unreached node is set to positive

978-0-7695-3352-0/08 $25.00 © 2008 IEEE
DOI 10.1109/HPCC.2008.113

infinite. When it is known that the currently known shortest path of getting to node *i* is also the absolute shortest path we will ever be able to attain, the node is called permanently labeled. When further improvement is expected for finding the shortest path to node *i*, node *i* is considered only temporarily labeled. It follows that *d(i)* is an upper bound on the shortest path distance to node *i* if the node is temporarily labeled; and *d(i)* represents the final and optimal shortest path distance to node *i* if the node is permanently labeled.

The performance of a particular shortest path algorithm partly depends on how the basic operations in the labeling method are implemented. Two aspects are particularly important to the performance of a SSSP algorithm: 1) the strategies used to select the next temporarily labeled node to be scanned, and 2) the data structures utilized to maintain the set of labeled nodes. Many algorithms belong to a labeling method such as the Dijkstra algorithm[2] and use effective data structures or certain strategies to optimize the above two aspects.

Considerable empirical studies on the practical performance of sequential shortest paths algorithms have been reported numerously in literature. Included is the study by Cherkassky et al. [3], which is the most comprehensive since they evaluated the 17 algorithms on a number of randomly generated networks with different characteristics.

To choose suitable candidates from the existing algorithms for computing shortest paths on real road networks, Zhan and Noon[4] tested a set of 15 shortest paths algorithms using real road networks. Three algorithms that run fastest on the real road networks have been identified. These three algorithms are: 1) the graph growth algorithm implemented with two queue; 2) the Dijkstra algorithm implemented with approximate buckets and 3) the Dijkstra algorithm implemented with double buckets. As a follow-up to that study, Zhan [5] demonstrated the data structures and implementation strategies related to these algorithms.

For parallel SSSP, the algorithm based on Dijkstra is inherently serial [1]. Most parallel algorithms attempt to find and exploit parallelism by identifying nodes that might be settled simultaneously. Meyer and Sanders[6,7] reviewed parallel algorithms for SSSP, and also gave the Δ *-stepping* SSSP algorithm and divided the Dijkstra algorithm into a number of phases which executes in parallel. Several theoretical improvements and implementations are given for Δ*-stepping*. Perhaps the most notable of these came from Madduri and Bader et al.[8,9], who used the experimental study of SSSP in parallel. This resulted in good performance on the RMAT graph on MTA-2, which is a multithreaded parallel machine.

In this paper, we propose a new parallel algorithm on cluster architecture, unlike Madduri and Bader's work on MTA-2. Our parallel algorithm does not use Δ *-stepping* to divide the sequential algorithm into different phases. Instead, our algorithm iteratively corrects the shortest paths on the whole sequential algorithm. In addition, we parallelize the Dijkstra algorithm with two queues.

3. Our Parallel SSSP Algorithm

3.1 Review of sequential algorithms

In this paper, we parallelize the Dijkstra algorithm with two queues (TQQ) [10, 3]. The sequential TQQ is reviewed here. The two queues data structure is presented in Figure 1. Both Q' and Q" are queues in the two-queue data structure, and nodes can be inserted at the end of Q' and Q".

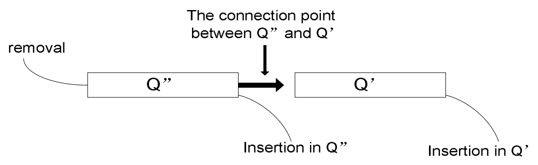

Figure 1 Two-Q data structure (Q) consisting of Q" and Q'[5]

```
Algorithm 1: Sequential Dijkstra Algorithm with Two-Queue
    Input  : A graph $G = (N, E)$ with non-negative weights and a source
             node $s \in N$
    Output: Single source shortest path
    begin
        TwoQueueInit (Q);
        for $i \leftarrow 0$ to $n - 1$ do
            $d(i) = +infinite$;
            $d(s) = 0$;
        while $Q! = NULL$ do
            TwoQueueRemove $(Q, i)$;
            for each successor node $j$ of node $i$ do
                if $d(j) > d(i) + l(i, j)$ then
                    $d(j) = d(i) + l(i, j)$;
                    TwoQueueInsert $(Q, j)$);
    end
    TwoQueueInit $(Q)$ : initialize two-queue Q;
    TwoQueueInsert $(Q, j)$ : for any node $j$ that is not already in $Q$,
    insert $j$ at the end of $Q'$ if $j$ is unreached or insert $j$ at the end of $Q"$
    if $j$ is temporarily labled;
    TwoQueueRemove $(Q, i)$ : remove node $i$ from the beginning of
    two-queue $Q$(the top of queue $Q"$);
```

Figure 2 Sequential Dijkstra Algorithm with Two-Queue

The sequential algorithm of TQQ is described in Figure 2. Although the complexity is $O(n^2m)$[10] , TQQ is proved to be one of the most efficient algorithms in practice mainly on sparse graphs[5].

3.2 Graph partitioning

Since our parallel algorithm is based on graph-partitioning, before continuing, we give a general introduction to graph partitioning here.

The graph partitioning problem can be defined as follows: Given a graph $G = (N, E)$, where N is the set of nodes, E is the set of edges e_{ij} connecting node i and node j, with $|N| = n$ and $|E| = m$, partitioning G means dividing N into the union of p disjoint subsets, N_1, N_2,. . . , N_p, such that $N_i \cap N_j = \phi$,for $i \neq j$, $\cup_i N_i = N$. Once the graph is partitioned, the dependencies associated with the edges connecting nodes in separate sub-graphs, called edge-cutes, represent the amount of inter-processor communication. And the graph partitioning is done subject to the certain optimality constrains below. First, the numbers of the nodes N_i in each sub-graph is approximately equal, that is $|N_i| \approx n/p$, which means the load is approximately balanced across processors. Secondly, the number of edges of E whose incident vertices across different sub-graphs $\sum_{i,j} e_{ij} = \{(v,w)|v \in N_i,\ \ w \in N_j\}$ (edge-cut) should be minimized. A partition of N is commonly represented by a partition vector P of length n, such that for every node $v \in N$, $P[v]$ is an integer between 0 and $p-1$, indicating the sub-graph which node v belongs to.

Since choosing optimal graph partitioning is NP hard [11,12], obtaining exact solutions is computationally intractable, so heuristics giving suboptimal solutions are often proposed to solve the problem. Kernighan-Lin(1970)[13] is one of the earliest algorithms proposed for partitioning graphs and it's variants are still in use today. This algorithm begins with an initial partition into two sets, and then iteratively finds improvements by exchanging sets of nodes to perform hill-climbing and partition a graph into more than two sub-graphs by recursively applying the bipartition algorithm. Fiduccia-Mattheyses [14] uses the similar idea and introduces a faster implementation of Kernighan-Lin algorithm for graph partitioning with effective data structures. Spectral methods [15] can produce good partitions for a wide range of problems, and they are used very extensively. But these methods are expensive since they need a large amount of computation of the eigenvector corresponding to the second small eigenvalue (Fiedler vector). Computation of the Fiedler vector by using a multilevel spectral bisection algorithm can speed up the execution of the spectral methods. However, it still takes a large amount of time. Another class of graph partitioning algorithms called multilevel graph partitioning schemes are proposed to moderate computational complexity and produce good partitions [16]. They reduce the size of the graph (i.e., coarsen the graph) by collapsing nodes and edges, partition the smaller graph, and then un-coarsen it to construct a partition for the original graph. Importantly, these methods tend to provide better partitions at reasonable cost. Karypis and Kumar[17] did a comprehensively theoretical analysis to various multilevel algorithms on the quality of partition, and proposed a new faster refinement scheme during the un-coarsening. The results of the experiment showed their scheme was good. Always, the k-way partitioning problem is solved by recursive bisection. That is, you first obtain 2-way partitioning, and then recursively obtain 2-way partitioning of each resulting partition. Karypis and Kumar[18] also present the multilevel paradigm that can be used to construct a k-way partitioning directly. The coarsest graph is now directly, which can potentially produce much better partitioning. The k-way METIS [19], which is also developed in the Karypis Lab, is a good software package for partitioning graphs based on a multilevel paradigm, so we use METIS to perform graph partitioning.

3.3 Parallel SSSP Algorithm

Figure 3 shows the brief pseudo codes of our parallel SSSP Algorithm which can be divided into two phases: Graph Partitioning Phase and Iterative Correcting Phase.

```
Algorithm 2: Parallel SSSP Algorithm for Real-road Network
Input  : A graph $G = (N, E)$ with non-negative weights and a source
         node $s \in N$
Output: Single source shortest path
for each processes parallel do
    if $processrank == 0$ then
        Graph Partitioning
    Broadcast graph partition information to other processes
    TwoQueueInit ($Q$)
    while $true$ do
        while $Q! = NULL$ do
            TwoQueueRemove ($Q, i$)
            for each successor node $j$ of node $i$ in current subgraph do
                if $d(j) > d(i) + l(i, j)$ then
                    $d(j) = d(i) + l(i, j)$
                    TwoQueueInsert ($Q, j$))
            $sendTag = 0$
            for each adjacent subgraph $k$ do
                for each successor node $r$(in $k$)of node $i$ do
                    if $d(r) > d(i) + l(i, r)$ then
                        Add information of node $r$ to MessageArray[$k$]
                        $sendTag = 1$
        $exitFlag \leftarrow$ the sum of $sendTag$ in all processes
        if $exitFlag! = 0$ then
            for each adjacent subgraph $k$ do
                Send $MessageArray[k]$ to every adjacent subgraph $k$
                Receive the MessageArray from subgraph $k$
            for each received node $m$ whose successor in current
            process is $n$ do
                if $d(n) > d(n) + l(m, n)$ then
                    $d(n) = d(n) + l(m, n)$
                    TwoQueueInsert ($Q, n$))
        else
            break form While(true)
    if $processrank == 0$ then
        Gather the results from other processes
    ALGORITHM END
```

Figure 3 Parallel SSSP Algorithm for Real-road Network

Phase 1: Graph Partitioning Phase

The corresponding pseudo codes in Figure 3 are line 2~ line 4. The transportation network is partitioned into *p* disjoint sub-graphs in the phase, and then each sub-graph is assigned to one of *p* processors. Each sub-graph contains a roughly equal number of nodes *(n/p)* and the number of edge-cuts is minimized to balance the load and minimize communications among processors. Let $T_{net\text{-}part}$ represent the time-consuming of this phase, and we call this step *Graph Partitioning Step*.

Phase 2: Iterative Correcting Phase

This phase (corresponding pseudo codes in Figure 3 are line 5~ line 29) is to compute the temporary shortest paths in each sub-graph locally, and keep correcting the shortest paths during iterations. This phase can be divided into two steps: *Computation Step* and *Communication Step*.

1) Computation Step

In this step, each process first finds local optimal shortest paths in each sub-graph using the sequential shortest path algorithm (pseudocodes line 8~line 12), which is just the same as Figure 2. Then boundary information is collected for exchanging in the Communication Step (pseudocodes line 13~line 19). If there is exchanging information, we store this information in *MessageArray* and set a tag named *sendTag* to 1(pseudo codes line 16~ line 18). At the end of this step, the sum of *sendTag* across *p* processes are assigned to *exitFlag*(pseudo codes line 19). Let $T_{compute}(n/p)$ represent the time-consuming of this step. From 3.1, we know the computational complexity of this step is $O((n/p)^2(m/p))=O(n^2m/p^3)$, so if there is no communication cost, our algorithm has the potential of getting hyper-linear speedups if the number of iterative steps is small.

2) Communication Step

If the *exitFlag* is not *0*, which means the boundary nodes have update information, the algorithm exchanges the boundary information among the adjacent sub-graphs (as pseudo codes line 21~ line 23). Then the shortest path distance of corresponding nodes is updated (as pseudo codes line 24~line 27). If the *exitFlag* is 0, which means there are no boundary nodes needed to update shortest path distances and the final optimal shortest paths have been obtained, the algorithm ends. Let T_{comm} represent the time-consuming of this step.

Assume the algorithm needs *k* iterations from start to end, so the general time of the parallel algorithms can be formally described as:

$$T_{sum} = T_{net\text{-}part} + k\times T_{compute}(n/p) + k\times T_{comm} \quad \ldots (1)$$

Let's take a sample network (Figure 4) to illustrate our parallel algorithm.

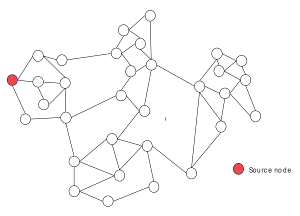

Figure 4 A Sample Network

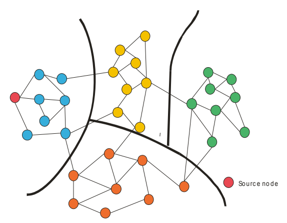

Figure 5 Graph Partitioning

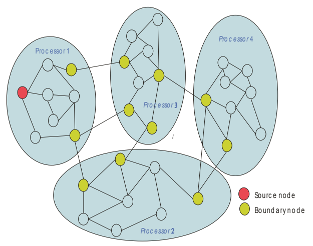

Figure 6 Graph Partitioning

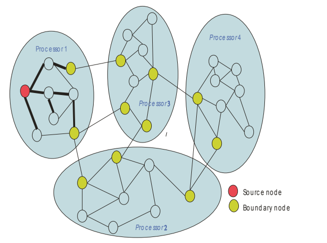

Figure 7 Locally compute shortest paths (SP)

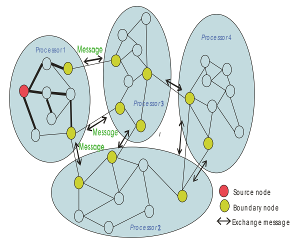

Figure 8 1st time exchange messages

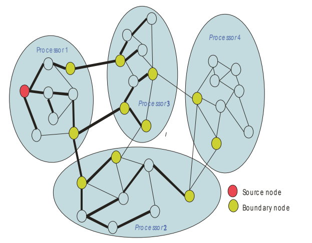

Figure 9 Locally compute SP (procs 1, 2, 3)

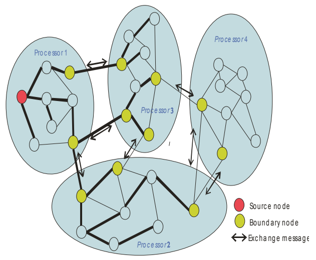

Figure 10 2nd time exchange messages

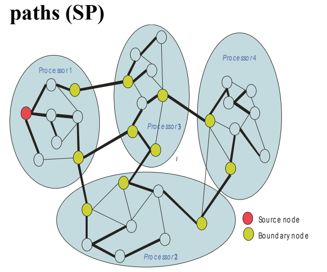

Figure11 Locally compute SPs (procs 1, 2, 3, 4)

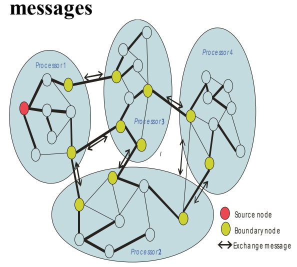

Figure12k-th time exchange messages

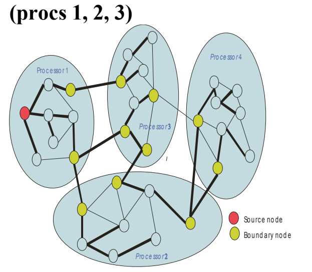

Figure 13 no messages exchanged and end

Given the sample transportation network, in our algorithm, first the network needs to be partitioned into disjoint sub-graphs using a certain graph-partitioning strategy, as Figure 5. Each sub-graph has roughly the same number of nodes, and the number of edges crossing sub-graphs is minimized. In Figure 6, after graph partition, each sub-graph is assigned to one process. We assume these processes are process 1 to process 4. Figure 7- Figure 13 illustrate the process of iterative correcting in detail.

In Figure 7, each process starts to find temporary shortest paths in each sub-graph locally. But in the first iteration, only process 1 actually computes temporary shortest paths (the bold lines) in sub-graph 1 because the other processes do not have the information of the source node. Then, the boundary information is exchanged among the adjacent sub-graphs in Figure 8. Figure 9~Figure 11 show that with the information exchanging, the number of processes which have got the information of the source node are increasing. After that, the algorithm continues iterating and the temporary shortest paths in each process keep correcting and updating with the boundary information exchanged in Figure 12. The algorithm continues until there are no messages exchanged among the adjacent sub-graphs (Figure 13), and we obtain the final shortest path tree.

4. Experiment and Evaluation

In this section, we evaluate the performance of our parallel algorithm using three large real road networks. The data sets are high-detail real-road networks (four levels of roads) from three US states (AL, MO, GA). The characteristics of the networks are given in Table I. Our parallel algorithm was implemented in C++ programming language with MPI. The testing environment is based on a cluster consisting of 4 blades of IBM JS21 servers interconnected with Gigabit Ethernet. Each IBM JS21 blade has two dual-core 2.5GHz PowerPC 970 processors with 4G memory. And we use MPICH2, gcc compiler on Red Hat Enterprise Linux 4.0.

Table I. Characteristics of the Networks [2]

Network Name	Number of Nodes	Number of Arcs	Arc/Node Ratio	Arc Length		
				Maximum	Mean	Stnd.Dev
ALNET	66082	185986	2.82	0.298232	0.011383	0.0124
MONET	67899	204144	3.00	0.21247	0.015542	0.0133
GANET	92792	264392	2.84	0.174245	0.010511	0.0001

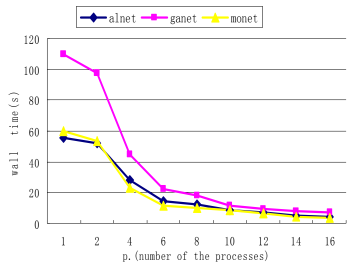

Figure 14 Execution time of our parallel SSSP algorithm on the three real-road networks

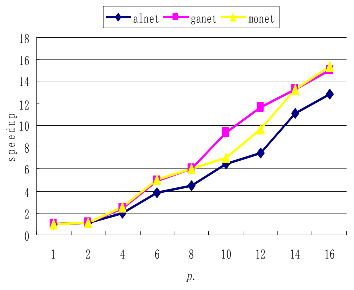

Figure 15 Speedup of our parallel SSSP algorithm

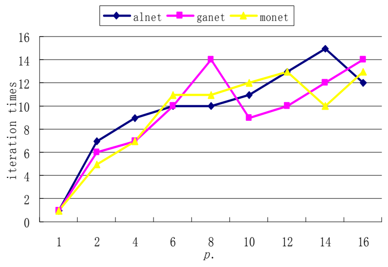

Figure 16 The number of Iterations slowly zigzag up

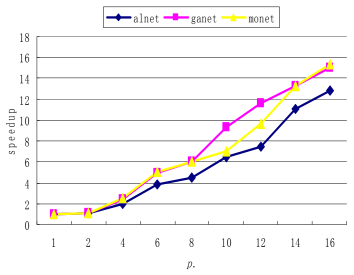

Figure 17 Edge-cuts of the graph partitioning

Figure 14 shows the wall time of our parallel algorithm over above three networks, and Figure 15 shows the speedups. The results demonstrate that the wall time reduces drastically as the number of processes increases, and the speedup can reach 15.02 on 16 processors over network GANET, 15.39 on 16 processors over network MONET, and 12.79 on 16 processors over network ALNET. The results reveal that our parallel algorithm has good scalability.

Figure 16 demonstrates the number of iterations (the parameter *k* in formula (1)) slowly zigzag up as the number of processes increases. Figure 17 illustrates the growth of edge-cuts with the growth of the number of sub-graphs.

Next, we will analyze the performance of our algorithm in detail. According to the discussion in 3.3, we know that the total execution time includes three parts:

- ♦ $T_{net\text{-}part}$ is the graph partitioning time. Figure 18 shows the execution time of the Graph Partitioning Step, and we can see that this part of time is very short compared with the other two parts, although it grows with the increase of the number of processes.
- ♦ $k \times T_{compute}(n/p)$ is the execution time of Computation Step. Figure 19 illustrates the execution time of this step which reduces drastically.
- ♦ $k \times T_{comm}$ is the execution time of Communication Step. The performance of this step partly depends on the bandwidth and latency of the network equipment. Figure 20 demonstrates the wall time of this step. The wall time reduces quickly during 2~16 processes. When there is only one process, no communication is needed, since there is only one sub-graph after the partition, so the wall time is nearly equally to zero.

It is necessary to explain why the communication wall time decreases in Figure 20 while the total edge-cuts increase in Figure 17. The reason is that it is the edge-cuts per processor (total edge-cuts/p), not the total edge-cuts, represent inter-processors communication. In fact, the edge-cuts per processor reduce when the number of processes increases.

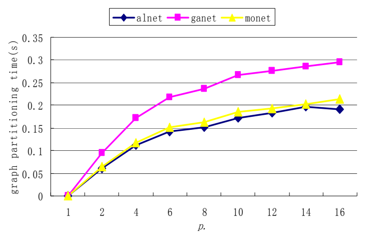

Figure 18 Execution time of Graph Partitioning Step

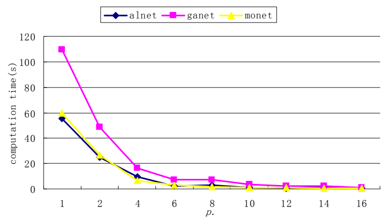

Figure 19 Execution times of Computation Step

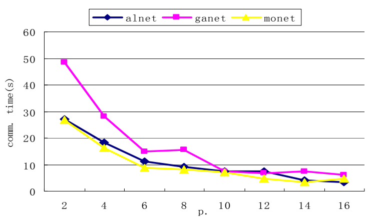

Figure 20 Execution times of Communication Step

Figure 21 shows the percentages of the execution time for the three steps in the whole algorithm. From the result, it is obvious that the proportion of graph-partitioning has trends of increases, the proportion of the computation step has trends of decreases, and the communication step has trends of increases. It reveals that the algorithm requires considerable effort to enhance the communication performance or use better network equipment such as InfiniBand and so on.

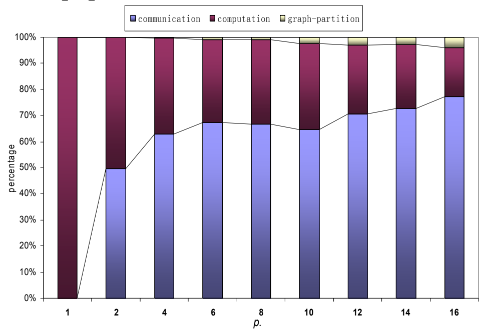

Figure 21 The proportion of the three parts

5. Conclusion and Future Works

The results lead us to several conclusions. First, our parallel algorithm based on graph partitioning and iterative correcting has good speedup ratio and efficiency, and achieved more than a 15-fold speedup on 16 processes over the real road networks. Second, the total execution time includes three parts: graph-partitioning, computing local shortest paths and communication. According to the analysis in section 4, we know that the computation time and communication time reduce as the number of the processes increases. It also reveals that the algorithm requires considerable effort to enhance the communication performance and needs a better bandwidth and lower latency of network equipment. We are going to run our test on a larger cluster experiment environment and make a comparison with other parallel algorithms in the future. We also plan to test our parallel algorithm on better network equipment such as InfiniBand to exploit more benefits of communication.

Acknowledgements

We thank the anonymous reviewers for their very invaluable feedback. We thank Professor F. Benjamin Zhan at the Department of Geography, Texas State University-San Marcos, for his insightful discussion and providing the real-road test data. We also thank Suxiang Li at Department of Computer Science and Technology, Beihang University, for her useful comments. We thank Professor Weisong Shi from Wayne State University for his kindly help with revising this paper.

This work was partially supported by the National Nature Science Foundation of China (No.60303032), Key Program of National Nature Foundation of China (No. 60533020), the National 863 Plan of China (No.2006AA01A102 and No.2006AA01A125), and Opening Foundation (2005-05) of State Key Lab of Networking and Switching Technology in Beijing University of Posts and Telecommunications.

References

[1] B. Hendrickson, and J.W. Berry, "Graph Analysis with High-Performance Computing", *Computing in Science and Engineering*, vol.10, no. 2, pp.14-19, Mar/Apr, 2008

[2] E.W. Dijkstra, "A note on two problems in connexion with graphs", *Numer. Math.* 1:269-71, 1959

[3] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. "Shortest Path Algorithms: Theory and Experimental Evaluation", *Mathematical Programming,* Vol. 73, pp. 129-174, 1996

[4] F.B. Zhan, and C. E. Noon. "Shortest Path Algorithms: An Evaluation Using Real Road Networks", *Transportation Science* 32(1): 65-73. 1998.

[5] F.B. Zhan, "Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures," *Journal of Geographic Information and Decision Analysis* 1(1): 69-82. 1998.

[6] U. Meyer, "Design and Analysis of Sequential and Parallel Single-Source Shortest-Paths Algorithms". PhD thesis, Universit¨at Saarlandes, Saarbr¨ucken, Germany,October 2002

[7] U. Meyer and P. Sanders, " Δ *-stepping*: a parallelizable shortest path algorithm." *J. Algs.*, 49(1):114–152, 2003

[8] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak, "Parallel Shortest Path Algorithms for Solving Large-Scale Instances", *9th DIMACS Implementation Challenge -- The Shortest Path Problem,* DIMACS Center, Rutgers University, Piscataway, NJ, November 13-14, 2006

[9] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak, "An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances", *Workshop on Algorithm Engineering and Experiments (ALENEX)*, New Orleans, LA, January 6, 2007

[10] S. Pallottino, "Shortest path methods: Complexity, interrelations and new propositions", *Networks*, vol. 14, pp. 257-267, 1984

[11] T.N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is NP-hard", *Inf. Process. Lett.*, 42 (1992), pp. 153–159.

[12] M.R. Geary and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of Incompleteness.* W.H. Freeman and Company, New York, first edition, 1979

[13] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Technical Journal*, 49(2), pp. 291-308, 1970.

[14] C.M. Fiduccia, and R.M. Mattheyses, "A linear heuristic for improving network partitions," *Proc. Design Automat. Conf.*, pp. 175-181, 1982

[15] A. Pothen, H. D. Simon, and K. P. Liou, "Partitioning sparse matrices with eigenvectors of graphs", *SIAM J. Matrix Anal. Appl.*, vol. 11, n. 3, pp. 430-452, 1990

[16] G. Karypis and V. Kumar. "Multilevel graph partition and sparse matrix ordering." *In Intl. Conf. on Parallel Processing*, 1995

[17] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs", *Journal of Parallel and Distributed Computing,* Volume 48, Number 1, January 1998 , pp. 96-129(34)

[18] G. Karypis and V. Kumar, "Multilevel k-way Hypergraph Partitioning", *36th Design Automation Conference*, pp. 343 - 348, 1999

[19] METIS, www.cs.umn.edu/~metis

[20] H. Chen, Y. Tang, and Y. Zhang, "Implementation and Evaluation of Graph-partitioning based Parallel Shortest Paths Algorithms", to be appeared in *Journal of Computer Research and Development(Chinese)*, 2008 *(Supp.)*

[21] J.Q. Michael. *Parallel Programming in C with MPI and OpenMP*. The McGraw-Hill Companies, Inc.(Chinese Edition, Translated by W.G. Chen et al),2004.

[22] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to algorithms*, MIT Press, Cambridge, MA, 2001