Shortest Path Computation over Disk-Resident Large Graphs Based on Extended Bulk Synchronous Parallel Methods

Zhigang Wang[1], Yu Gu[1], Roger Zimmermann[2], and Ge Yu[1]

[1] Northeastern University, China
wangzhigang_mail@yahoo.cn, {guyu,yuge}@ise.neu.edu.cn
[2] National University of Singapore, Singapore
rogerz@comp.nus.edu.sg

Abstract. The Single Source Shortest Path (SSSP) computation over large graphs has raised significant challenges to the memory capacity and processing efficiency. Utilizing disk-based parallel iterative computing is an economic solution. However, costs of disk I/O and communication affect the performance heavily. This paper proposes a state-transition model for SSSP and then designs two optimization strategies based on it. First, we introduce a tunable hash index to reduce the scale of *wasteful data* loaded from the disk. Second, we propose a new iterative mechanism and design an Across-step Message Pruning (ASMP) policy to deal with the communication bottleneck. The experimental results illustrate that our SSSP computation is 2 times faster than a basic Giraph (a memory-resident parallel framework) implementation. Compared with Hadoop and Hama (disk-resident parallel frameworks), the speedup is 21 to 43.

1 Introduction

The Single Source Shortest Path (SSSP) computation is a classical problem with numerous applications and has been well-studied over the past decades. However, new challenges have been raised by the rapid growth of graph data. For instance, up to March 2012, Facebook has owned about 900 million vertices (i.e., users) and over 100 billion edges. Such large graphs have exceeded the memory capacity of a single machine [1]. Even for memory-resident parallel frameworks [2,3], the data processing capacity of a given cluster is also limited [4]. This problem can be relieved by enlarging the cluster scale, but the consumption will also increase. It is an economic solution if we extend memory-resident parallel frameworks by spilling data on the disk [5]. In this case, how to reduce costs of disk I/O and message communication becomes challenging especially for the iterative computation tasks, such as SSSP.

For *in-memory* algorithms on SSSP, some are difficult to be executed in parallel due to the inherent priority order of relaxation and others perform poorly if data are organized as their sophisticated structures on the disk [6,7]. *External-memory* algorithms with the polynomial I/O complexity have also been proposed [8]. However, the practical performance is unsatisfactory [9] considering

W. Meng et al. (Eds.): DASFAA 2013, Part II, LNCS 7826, pp. 1–15, 2013.

the impact of *wasteful data* (load a block of data from the disk but only use a portion). In addition, they are all centralized algorithms and take no account of the communication cost. Recently, G. Malewicz et al. propose a new parallel iterative implementation for SSSP (P-SSSP) and evaluate its performance on Pregel, a memory-resident parallel framework [2] based on the Bulk Synchronous Parallel (BSP) model [10]. Although its outstanding performance is impressive, the runtime will increase rapidly if it is implemented on disk-based frameworks, such as Hama and Hadoop [5,11]. I/O costs incurred by reading *wasteful data* may offset the parallel gains. Furthermore, the large scale of messages will also exacerbate costs of disk-accesses and communication. In this paper, we aim to crack the nut for these two problems of disk-resident P-SSSP over large graphs.

Based on the theoretical and experimental analysis on P-SSSP, we divide iterations into three stages: divergent → steady → convergent, and then propose a state-transition model. It adopts a *bottom-up* method to evaluate which stage the current iteration belongs to. Afterwards, two optimization policies are designed by analyzing features of the three states.

For divergent and convergent states, the scale of processed data will shade as the iteration progresses, which leads to huge costs of reading *wasteful data*. A tunable hash index is designed to skip *wasteful data* to the utmost extent by adjusting the bucketing granularity dynamically. The time of adjusting depends on the processed data scale instead of inserting or deleting elements, which is different from existing mechanisms [12,13]. In addition, for different adjusting operations (i.e., bucketing granularity), we adopt a Markov chain to estimate their cumulative impacts for iterations and then execute the optimal plan. Another optimization is an Across-step Message Pruning (ASMP) policy. The large scale of messages during the steady state incurs expensive costs of disk I/O and communication. The further analysis shows that a considerable portion of messages are redundant (i.e., the value of a message is not the real shortest distance). By extending BSP, we propose a new iterative mechanism and design the ASMP policy to prune invalid messages which have received. Then a large portion of new redundant messages will not be generated.

Experiments illustrate the runtime of our tunable hash index is 2 times as fast as that of a static one because roughly 80% of *wasteful data* are skipped. The ASMP policy can reduce the message scale by 56% during the peak of communication, which improves the performance by 23%. The overall speedup of our P-SSSP computation compared to a basic implementation of Giraph [3], an open-source clone of Pregel, is a factor of 2. For Hadoop and Hama, the speedup is 21 to 43. In summary, this paper makes the following contributions:

- **State-Transition Model:** We propose a state-transition model which divides iterations of P-SSSP into three states. Then we analyze characteristics of the three states, which is the theoretical basis for optimization policies.
- **Tunable Hash Index:** It can reduce costs of reading *wasteful data* dynamically as the iteration progresses, especially for divergent and convergent states. A Markov chain is used to choose the optimal bucketing granularity.

– **Across-step Message Pruning:** By extending BSP, this policy can prune invalid received messages and avoid the generation of redundant messages. Consequently, the message scale is reduced, especially for the steady state.

The remaining sections are structured as follows. Section 2 reviews the related work. Section 3 gives the state-transition model. Section 4 describes the tunable hash index. Section 5 proposes the Across-step Message Pruning policy. Section 6 presents our performance results. Section 7 concludes and offers an outlook.

2 Related Work

Many algorithms have been proposed for the SSSP computation. However, centralized in-memory algorithms can not process increasingly massive graph data. Advanced parallel algorithms perform poorly if data are spilled on the disk [6,7]. For example, the Δ-stepping algorithm must adjust elements among different buckets frequently [6] and Thorup's method depends on a complex in-memory data structure [7], which is I/O-inefficient. Existing *external-memory* algorithms are dedicated to designing centralized I/O-efficient data structure [8]. Although they have optimized the I/O complexity, the effect is limited for reducing the scale of loaded $wasteful\ data$ because their static mechanisms can not be adjusted dynamically during the computing.

Nowadays, most of existing indexes are in-memory or designed for the *s-t* shortest path [14,15], which is not suitable for SSSP over large graphs. As a precomputed index, VC-index is proposed to solve the disk-resident SSSP problem [9]. However, this centralized index is still static and requires nearly the same storage space with the initial graph or more. Also, dynamic hash methods for general applications have been proposed, but they are concerned on adjusting the bucketing granularity with changes in the scale of elements [12,13]. While, in our case, the element number (vertices and edges) in a graph is constant.

Implementing iterative computations on parallel frameworks has been a trend. The representative platform is Pregel [2] based on BSP and its open-source implementations, Giraph and Hama [3,5]. Pregel and Giraph are memory-resident, which limits the data processing capacity of a given cluster. This problem also exists for Trinity [16], another well-known distributed graph engine. Although Hama supports disk operations, it ignores the impact of $wasteful\ data$. For other disk-based platforms based on MapReduce, such as Hadoop, HaLoop and Twister [11,17,18], restricted by HDFS and MapReduce, it is also difficult to design optimization policies to eliminate the impact.

The parallel computing of SSSP can be implemented by a synchronous mechanism, such as BSP [10], or an asynchronous strategy [4]. Compared with the former, although the latter accelerates the spread of messages and improves the speed of convergence, a large scale of redundant messages will be generated, which increases the communication cost. The overall performance of them depends on a graph's density when data are memory-resident [4]. However, for the asynchronous implementation of disk-based SSSP, the frequent update of vertex values will lead to fatal I/O costs, so BSP is more reasonable in this case.

3 State-Transition Model

3.1 Preliminaries

Let $G = (V, E, \omega)$ be a weighted directed graph with $|V|$ vertices and $|E|$ edges, where $\omega : E \to \mathbb{N}^*$ is a weight function. For vertex v, the set of its outgoing neighbors is $adj(v) = \{u|(v,u) \in E\}$. Given a source vertex v_s, $\delta(u)$, the length of a *path* from v_s to u, is defined as $\sum \omega(e)$, $e \in path$. δ is initialized as $+\infty$. The SSSP problem is to find the minimal $\delta(u)$, $\forall u \in V$. By convention, $\delta(u) = +\infty$ if u is unreachable from v_s. We assume that a graph is organized with the adjacency list and each vertex is assigned a unique ID which is numbered consecutively.

The P-SSSP computation proposed by Pregel is composed of a sequence of SuperSteps (i.e., iterations). At the first SuperStep t_1, only v_s sets its $\delta(v_s) = 0$. Then $\forall u \in adj(v_s)$, a message (i.e., candidate shortest distance) is generated: $msg(u) = \langle u, m \rangle$, $m = \delta(v_s) + \omega(v_s, u)$, and sent to u. At t_2, vertex u with a list of $msg(u)$, namely $lmsg(u)$, sets its $\delta(u) = min\{\delta(u), min\{lmsg(u)\}\}$. Here, if $msg_i(u) < msg_j(u)$, that means $m_i < m_j$. If $\delta(u)$ is updated, new messages will be generated and sent to neighbors of u. The remaining iterations will repeat these operations until $\forall v \in V$, its $\delta(v)$ is not be updated. Operations of one SuperStep are executed by several tasks in parallel.

If a large graph exceeds the memory capacity of a given cluster, the topology of the graph is firstly spilled on the disk. Furthermore, the overflowing messages will also be spilled. Data are divided into three parts and respectively stored: message data, vertex data and outgoing edge data. For example, an initial record $\{u, \delta(u), adj(u)\&\omega, lmsg(u)\}$ will be partitioned into three parts: $\{lmsg(u)\}$, $\{u, \delta(u)\}$ and $\{adj(u)\&\omega\}$. $\{u, \delta(u)\}$ and $\{adj(u)\&\omega\}$ are stored in two files respectively but located on the same line, which avoids the cost of restructuring when sending messages. By this mechanism, the topology of a graph will not be accessed when receiving messages. In addition, we only need to rewrite $\{u, \delta(u)\}$ on the disk and ignore $\{adj(u)\&\omega\}$ when updating δ.

Now, we give two notations used throughout this paper. We define *load ratio* as $LR = |V_l|/|V|$, where $|V_l|$ denotes the scale of vertices loaded from the disk. Another notation is *load efficiency*, which is defined as $LE = |V_p|/|V_l|$, where $|V_p|$ denotes the scale of vertices with received messages $lmsg$.

3.2 Three Stages of Iterations and State-Transition

The iterative process of P-SSSP is a wavefront update from v_s [2]. At the early stage, the message scale of the SuperStep t_i is small because only a few vertices update their δ. However, most of these messages will lead to updating δ at t_{i+1} because a majority of vertices still keep $\delta = +\infty$. Then more new messages will be generated since $|adj(u)| > 1$ generally. Consequently, the message scale will increase continuously. If most of vertices have updated their δ, the speedup of the message scale will decrease. As more and more vertices have found the shortest distance, their δ will be updated no longer. Then the number of messages will reduce until iterations terminate. We have run the P-SSSP implementation on

our prototype system over real graphs, S-LJ and USA-RN (described in Section 6). As illustrated in Fig 1(a), the message scale can be simulated as a parabola opening downwards. This curve can also express the trend of processed vertices, since only vertices with received messages will be processed. Furthermore, we divide the process into three stages: divergent → steady → convergent.

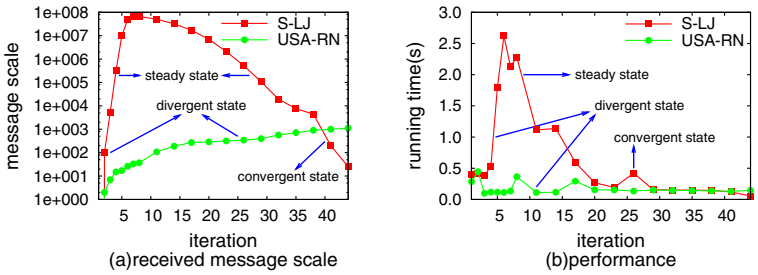


Fig. 1. Three processing stages of P-SSSP

Considering the message scale of divergent and convergent states, we only need to process a small portion of vertices. However, graph data must be loaded in blocks since they are spilled on the disk, which leads to a low LE and incurs considerable costs of reading *wasteful data*. In addition, the running time at the steady state is more than that of other two states obviously (Fig 1(b)). The reason is that massive messages lead to expensive costs of communication and disk-accesses (LR is high because many vertices need to process received messages). To improve the performance, we expect a low LR but a high LE.

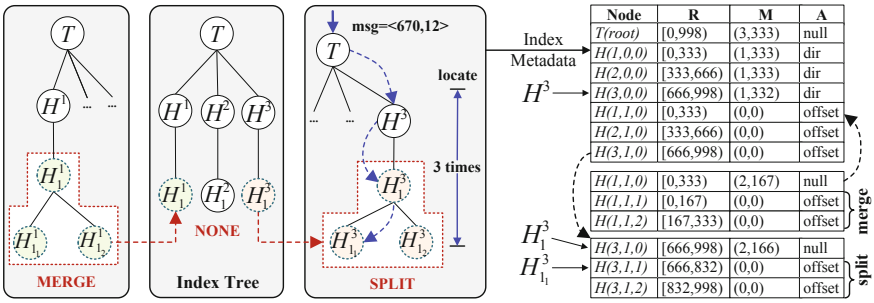
We notice that there is no general standard to separate the three states because real graphs have different topology features. For example, S-LJ spreads rapidly at the divergent state. However, it is the opposite for USA-RN, which is a sparser graph. In section 4.2, we will introduce a *bottom-up* method to separate different states according to dynamical statistics.

4 A Tunable Hash Index

4.1 Hash Index Strategy

It is essential to match $\{lmsg(u)\}$ with $\{u, \delta(u)\}$ when updating δ. By a static hash index, we can load messages of one bucket into memory. Then $\{u, \delta(u)\}$ and $\{adj(u)\&\omega\}$ in the same bucket are read from the local disk one by one to complete matching operations. The static hash index can avoid random disk-accesses. Data in buckets without received messages will be skipped, which improves LE, but the effect is limited because $|V_p|$ is changing continuously among the three states. Therefore, we propose a tunable hash index to maximize the scale of skipped graph data by adjusting the bucketing granularity dynamically.

Three parts of data (described in Section 3.1) will be partitioned by the same hash function. Illustrated in Fig 2, index metadatas of buckets are organized as a tree which includes three kinds of nodes: Root Node (T), Message Node (e.g., H^1) and Data Node (e.g., H_1^1). The Message Node is a basic unit for receiving and combining (i.e., only save the minimal $msg(u)$ for vertex u) messages. Initially, every Message Node has one child node, Data Node, which is the basic unit for loading $\{u, \delta(u)\}$ and $\{adj(u)\&\omega\}$. The metadata is a three-tuple $\{R, M, A\}$. R denotes the range of vertex IDs. M is a *key-value* pair, where *key* is the number of direct successor nodes and $value = \lceil R.length/key \rceil$. A is a location flag. For a Message Node, it means the location of memory-overflow message files (dir). For a leaf node, it includes the starting offset of $\{u, \delta(u)\}$ and $\{adj(u)\&\omega\}$. For anyone of parallel tasks, we deduce that the number of its Message Nodes depends on B_s, which is the cache size of sending messages. In order to simplify the calculation, we estimate the cache size in the number of messages instead of bytes. B_s is defined by disk I/O costs and communication costs of the real cluster. Limited by the manuscript length, the details are not illustrated here.



Node	R	M	A
T(root)	[0,998)	(3,333)	null
H(1,0,0)	[0,333)	(1,333)	dir
H(2,0,0)	[333,666)	(1,333)	dir
H(3,0,0)	[666,998)	(1,332)	dir
H(1,1,0)	[0,333)	(0,0)	offset
H(2,1,0)	[333,666)	(0,0)	offset
H(3,1,0)	[666,998)	(0,0)	offset
H(1,1,0)	[0,333)	(2,167)	null
H(1,1,1)	[0,167)	(0,0)	offset
H(1,1,2)	[167,333)	(0,0)	offset
H(3,1,0)	[666,998)	(2,166)	null
H(3,1,1)	[666,832)	(0,0)	offset
H(3,1,2)	[832,998)	(0,0)	offset

Fig. 2. The tunable hash index

The number of Message Nodes is fixed, but Data Nodes may be split or merged recursively during iterations. The former is only for leaf nodes. If one bucket H_j^i is split into N_j^i child buckets $H_{j_k}^i$, $1 \leq k \leq N_j^i$, that means vertex and outgoing edge data are divided equally in consecutive order. Then, the metadata of H_j^i needs to be updated (e.g., H_1^3). The merging operation is only for a direct predecessor node of leaf nodes. All child nodes will be merged and their parent node becomes a new leaf node (e.g., H_1^1).

$$k = \frac{getID(msg(v)) - \theta}{M.value} + 1 \tag{1}$$

To skip buckets without messages, we must locate the leaf node every message belongs to. Given a message $msg(v)$, we can locate $H_{j_k}^i$ it belongs to by Formula 1, where θ is the minimal vertex ID in R of H_j^i. The time complexity is $\Omega((h-1) \cdot |E|/N_t)$, where h is the height of the tree and N_t is the number of parallel tasks. In Fig 2, for $\langle 670, 12 \rangle$, we can find the leaf node $H_{1_1}^3$ by locating 3 times.

4.2 Adjust Hash Index Dynamically

Although splitting a leaf node can improve its *load efficiency*, the time of splitting and the number of child nodes are two critical problems.

Algorithm 1. Global Adjusting Type for *Leaf Nodes*

Input : Statistics of the current SuperStep t_i: S; slope of t_{i-1}: K
Output: Global adjusting type of t_{i+1}: AT; slope of t_i: $K^{'}$

1 **Job Master**
2 wait until all tasks report the $vector^k$ and $active^k$
3 $vector \leftarrow \sum_{k=1}^{N_t} vector^k$ /* N_t: the number of parallel tasks */
4 $active \leftarrow \sum_{k=1}^{N_t} active^k$
5 put $active$ into $HistoryQueue$ and estimate $K^{'}$ by the last K_Δ values
6 $AT = \max\{vector(i) | 0 \leq i \leq 2\}$
7 send $\{AT, K^{'}\}$ to each task
8 **Task** k
9 $vector^k \leftarrow \langle 0, 0, 0\rangle$ /* count the number of every adjusting type */
10 **while** $S^k \neq \phi$ **do**
11 $S_i^k \leftarrow$ remove one from S^k /* S_i^k: the statistics of the ith leaf node */
12 $type = getAdjustType(K, LE_i^k)$ /* $type$: $Split(0)$, $Merge(1)$, $None(2)$ */
13 $vector^k[type]$++
14 $active^k = active^k + getActive(S_i^k)$ /* the number of processed vertices */
15 send $vector^k$ and $active^k$ to **Job Master**
16 wait until **Job Master** returns $\{AT, K^{'}\}$ and then set $K = K^{'}$

Algorithm 1 is used to obtain a global adjusting type (AT) of the SuperStep t_{i+1}, which solves the first problem. It is also a *bottom-up* method to separate the three states. AT includes $Split$, $Merge$ and $None$. Algorithm 1 runs in a master-slave mode between two consecutive SuperSteps. First, task k judges the expected adjustment type for every leaf node by LE_i^k and K, then records statistics (Steps 10-14). LE_i is *load efficiency* of the ith leaf node at t_i. K is the slope of a fitting curve about *active*'s changing. Second, Job Master sums for all reports (Steps 3-4). $K^{'}$ (i.e., K of t_i) and AT are computed, and then sent to every task (Steps 5-7). Generally, $K_\Delta = 5$ by considering the robustness and veracity. The three states can be separated by AT and $K^{'}$ as follows: the divergent state, $AT \in \{Split, None\} \& K^{'} > 0$; the steady state, $AT \in \{Merge\}$; the convergent state, $AT \in \{Split, None\} \& K^{'} \leq 0$.

In the function $getAdjustType(K, LE_i^k)$, we first try to estimate the effect of $Split$. If it is positive, $type$ is $Split$, else $None$. If $type$ of all child nodes of the same parent node is $None$, we consider merging child nodes. Similarly, if the estimated result is positive, $type$ of them will be changed to $Merge$.

The effect of $Split$ depends on the number of child nodes. We use a Markov chain to find the optimal value, which solves the second problem. For a leaf node

H_j^i which is split into N_j^i child nodes, let $V_{j_k}^i$ be the set of vertices in $H_{j_k}^i$ and ${}^tV_p^{ij}$ be the set of processed vertices at the SuperStep t, then ${}^tV_p^{ij} \subseteq \bigcup V_{j_k}^i = V_j^i$, where V_j^i is the vertex set of H_j^i. ${}^t\Lambda$ denotes the set of child nodes with received messages at t, then ${}^t\Lambda = \{k|{}^tV_p^{ij} \cap V_{j_k}^i \neq \phi, 1 \leq k \leq N_j^i\}$.

Theorem 1. *For H_j^i, let the random variable $X(t)$ be $|{}^t\Lambda|$ at the SuperStep t, then the stochastic process $\{X(t), t \in T\}$ is a homogeneous Markov chain, where $T = \{0, 1, 2, ..., t_{up}\}$ and t_{up} is an upper bound of the process.*

Proof. In our case, the time set can be viewed as the set of SuperStep counters and the state-space set is $I = \{a_i|0 \leq a_i \leq N_j^i\}$. In fact, ${}^t\Lambda$ denotes the distribution of messages among child nodes. At the SuperStep t, vertices send new messages based on their current δ and received messages from t-1. Therefore, ${}^{t+1}\Lambda$, ${}^{t+2}\Lambda$, ..., ${}^{t+n}\Lambda$ only depend on ${}^t\Lambda$. The transition probability from ${}^t\Lambda$ to ${}^{t+1}\Lambda$ is decided by ${}^t\Lambda$ and δ. So $X(t)$ has the Markov property. Considering I and T are discrete, then $\{X(t), t \in T\}$ is a Markov chain. Furthermore, $P_{xy}(t, t + \Delta t) = P_{xy}(\Delta t)$ in the transition matrix P, so it is also homogeneous.

The original probability can be estimated by a sampling distribution. At t_m, we can get a random sample from ${}^{t_m}V_p^{ij}$, then the distribution of vertices among N_j^i child buckets can be calculated. Optimistically, we think the probability distribution of going from the state a_x to the state a_y is an arithmetic progression. Its common difference $d = (LE_i) \cdot K$ and the minimal value is $(2y)/x(x+1)$. Then, p_{xy}, the 1-step transition probability, can be calculated. The Δm-step transition probability satisfies the Chapman-Kolmogorov equation. Therefore, $P\{X(t_{m+\Delta m}) = a_y|X(t_m) = a_x\} = p_x(t_m)P_{xy}(\Delta m)$. We can calculate the mathematical expectation about the number of skipped buckets at $t_{m+\Delta m}$:

$$\Phi(N_j^i, t_{m+\Delta m}) = \sum_{x=1}^{N_j^i} \sum_{y=1}^{N_j^i} (N_j^i - x) p_x(t_m) P_{xy}(\Delta m) \tag{2}$$

Considering the time complexity described in Section 4.1, we can infer the splitting cost $\Psi(N_j^i, \Delta t) = \sum_{k=1}^{\Delta t}(T_{cpu}\Omega(\Delta h|E|/N_t))$, where Δh is the change of height for the index tree after splitting. Specially, $\Delta h = 0$ if $N_j^i = 1$. T_{cpu} is the cost of executing one instruction. $\Delta t = t_{up} - t_m$. If $K < 0$, t_{up} is the max number of iterations defined by the programmer, else $\Delta t = K_\Delta$. The benefit of splitting H_j^i is that data in some child buckets will not be accessed from the disk. Then the saving cost is:

$$\Psi'(N_j^i, \Delta t) = (\frac{\mathbb{V}_j^i + \mathbb{E}_j^i}{s_d N_j^i}) \sum_{\Delta m=1}^{\Delta t} \Phi(N_j^i, t_{m+\Delta m}) \tag{3}$$

where $\mathbb{V}_j^i$ is the vertex data scale of H_j^i in bytes, $\mathbb{E}_j^i$ is the outgoing edge data scale and s_d is the speed of disk-accesses. The candidate values of N_j^i are $C = \{\langle n, \rho\rangle|1 \leq n \leq \varepsilon\}$, where $\rho = \Psi'(n, \Delta t) - \Psi(n, \Delta t)$. ε is a parameter which insures the size of our index will not be more than the given memory capacity.

For *Split*, we find the optimal splitting value γ as follows: first, compute a subset $C^{'}$ of C by choosing the maximal ρ in $\langle n, \rho\rangle$; then, $\forall\langle n, \rho\rangle \in C^{'}$, γ is the minimal n. If $\gamma = 1$, $type = None$, otherwise, $type = Split$. For $Merge$, we view the parent node as a leaf node and assume γ be the real number of its child nodes. Then, if $\rho < 0$, *types* of its child nodes will be changed to $Merge$.

5 Message Pruning Optimization

In this section, we propose a new iterative pattern, namely EBSP, by extending BSP. EBSP updates δ synchronously but processes messages across-step. By integrating the Across-step Message Pruning (ASMP) policy, the scale of redundant messages can be reduced effectively.

5.1 Analyze Messages of P-SSSP

Definition 1. *Multipath-Vertex*

Given a directed graph, let the collection of vertices be $V_{mul} = \{v | v \in V_i \bigwedge v \in V_j \bigwedge ... \bigwedge v \in V_k, i \neq j \neq k\}$*, where* V_i *is the collection of vertices located* i*-hop away from the source vertex. Every vertex in* V_{mul} *is a Multipath-Vertex.*

As shown in Fig 3, we assume s is the source vertex, then the 1-hop collection is $V_1 = \{a, b, c, d, e\}$ and the 2-hop collection is $V_2 = \{e, f, g\}$. Obviously, $e \in V_1 \bigcap V_2$, is a Multipath-Vertex. As its successor vertices, f, g, h, i are also Multipath-Vertices. For P-SSSP, the synchronous implementation based on BSP can reduce the number of redundant messages [2]. For example, during the ith SuperStep, vertex u receives two messages $msg_i^{t_1}$ and $msg_i^{t_2}$ at the time of t_1 and t_2, where $t_1 < t_2$. According to the synchronous updating mechanism, if $\delta(u) > msg_i^{t_1} > msg_i^{t_2}$, $msg_i^{t_1}$ is invalid and will be eliminated. Consequently, redundant messages motivated by $msg_i^{t_1}$ will not be generated. However, our in-depth analysis finds that the similar phenomenon will occur again and can not be eliminated by BSP due to the existence of Multipath-Vertices. Considering the following scenario, u receives msg_j at the jth SuperStep, $j = i + 1$. If $msg_i^{t_2} > msg_j$, all messages generated by u at i are still redundant. Furthermore, redundant messages will be spread out continuously along outgoing edges until the max-HOP vertex is affected in the worst case. That incurs extra costs of disk-accesses and communication. In addition, some buckets may not be skipped because they have messages to process, even though the messages are redundant.

Fig 3 illustrates the phenomena in a directed graph. At the 1th SuperStep, $\delta(e)$ is updated to 4, then e sends messages to f and g. However, at the 2th SuperStep, $\delta(e)$ is updated to 2 again (instead of 3). Then, the previous messages are invalid. Consequently, f, g, h, i are processed twice.

5.2 Across-Step Message Pruning

For BSP, δ is updated only by depending on messages from the last SuperStep (in Section 3.1). If we can cumulate more messages before updating δ, then the

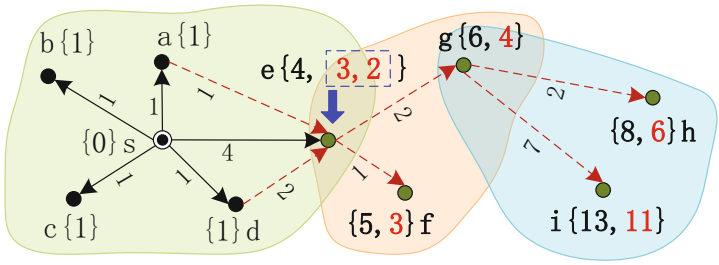


Fig. 3. The analysis of P-SSSP

impact of Multipath-Vertices will be relieved greatly. Consequently, we propose the EBSP model by extending BSP.

Definition 2. *EBSP*

Let M_{i+1} be the set of messages for the SuperStep t_{i+1}. At t_i, it is possible that $M^s_{i+1} \neq \phi$, $M^s_{i+1} \subseteq M_{i+1}$, if messages are sent asynchronously. When processing vertices at t_i, the domain of referenced messages is $M_i \cup M^s_{i+1}$.

EBSP will not affect the correctness of P-SSSP. Based on EBSP, we propose a novel technique called Across-step Message Pruning (ASMP) to relieve the phenomena of disseminating redundant messages. Algorithm 2 introduces the processing of one Message Node H^k. First, we load all received messages from t-1 into memory and put them into M^k_t after combining (in Section 4.1). Then the memory-resident received messages (without combination) of t+1 will be used to prune messages in M^k_t (Steps 3-8). A leaf node will be skipped if all of its messages in M^k_t are pruned. By this policy, new messages of t+1 will be obtained to optimize the synchronous update mechanism. Instead of combining existing messages, our policy is denoted to avoiding the generation of redundant messages, which is more effective. It can improve the performance of communication and disk-accesses. The scale of redundant messages which are eliminated by ASMP can be estimated by Theorem 2.

Theorem 2. *In Algorithm 2, for one vertex v_r, if $\delta(v_r) > msg^k_t(v_r) > msg^{k_s}_{t+1}(v_r)$, then the maximal number of pruned messages is $\Gamma(v_r)$:*

$$\Gamma(v_r) = \begin{cases} |adj(v_r)|, v_r = v_{maxHOP} \\ |adj(v_r)| + \sum\limits_{\forall v_m \in adj(v_r)} \Gamma(v_m), v_r \neq v_{maxHOP} \end{cases} \quad (4)$$

where v_{maxHOP} is the farthest one among reachable vertices of v_r.

Proof. Normally, if $\delta(v_r) > msg^k_t(v_r)$, $\delta(v_r)$ will be updated and then messages will be sent to $adj(v_r)$. However, in Algorithm 2, $msg^k_t(v_r)$ will be pruned if $msg^k_t(v_r) > msg^{k_s}_{t+1}(v_r)$. Recursively, at the SuperStep $t+1$, $\forall v_m \in adj(v_r)$, $\delta(v_m)$ will not be updated if $m^k_{t+1}(v_m) > m^{k_s}_{t+2}(v_m)$ or $m^k_{t+1}(v_m) \geq \delta(v_m)$. The pruning effect will not stop until v_{maxHOP} is processed.

Algorithm 2. Across-step Message Pruning

Input : message set for H^k at the SuperStep t and $t+1$: M_t^k, $M_{t+1}^{k_s}$
Output: message set after pruning: $\mathbb{M}_t^k$

```
1 V_t^k ← extract vertex IDs from M_t^k
2 V_{t+1}^{k_s} ← extract vertex IDs from M_{t+1}^{k_s}
3 foreach u ∈ V_t^k ∩ V_{t+1}^{k_s} do
4     msg_t^k(u) ← getMsg(M_t^k, u)
5     msg_{t+1}^{k_s}(u) ← min{getMsg(M_{t+1}^{k_s}, u)}
6     if msg_t^k(u) > msg_{t+1}^{k_s}(u) then
7         put msg_t^k(u) into the Pruning Set M_p
8 M_t^k = M_t^k − M_p
9 return M_t^k
```

We notice that if $\mathbb{M}_t^k = M_t^k \bigcup M_{t+1}^{k_s}$, δ will also be updated across-step, which is called an Across-step Vertex Updating (ASVU) policy. ASVU can accelerate the spread of messages. Therefore, the iteration will converge in advance compared with ASMP. However, $M_{t+1}^{k_s}$ is only a subset of M_{t+1}^k, so its elements may not be the minimal message value of t+1. For example, if $\delta(u) > msg_t^k(u) > msg_{t+1}^{k_s}(u)$, then δ will be updated at t. However, if $msg_{t+1}^{k_s}(u) > msg_{t+1}^k(u)$, $msg_{t+1}^k(u) \in M_{t+1}^k$, messages generated at t are also redundant, which offsets the pruning gains. Specially, compared with ASMP, if $u \in V_{t+1}^{k_s} \wedge u \notin V_t^k$ and $\delta(u) > msg_{t+1}^{k_s}(u) > msg_{t+1}^k(u)$, extra redundant messages will be generated.

6 Experimental Evaluation

To evaluate our patterns, we have implemented a disk-resident prototype system based on EBSP, namely DiterGraph. Data sets are listed in Table 1. The weight of unweighted graphs is a random positive integer. All optimization policies are evaluated over real graphs [19,20,21]. Then we validate the data processing capacity of DiterGraph over synthetic data sets and compare it with Giraph-0.1, Hama-0.5 and Hadoop-1.1.0. Our cluster is composed of 41 nodes which are connected by gigabit Ethernet to a switch. Every node is equipped with 2 Intel Core i3-2100 CPUs, 2GB available RAM and a Hitachi disk (500GB and 7,200 RPM).

6.1 Evaluation of Tunable Hash Index and Static Hash Index

Fig 4 illustrates the effect of our tunable hash index by comparing it to a static hash index. Their initial bucket number computed based on B_s is equivalent. However, the bucketing granularity of the static hash index will not be adjusted dynamically. In our experiments, we set B_s as 4000 according to the speed of communication and disk-accesses (described in Section 4.1). For USA-RN, we

Table 1. Characteristics of data sets

Data Set	ABBR.	Vertices	Edges	Avg. Degree	Disk Size
Social-LiveJournal1	S-LJ	4,847,571	68,993,773	14.23	0.9GB
Full USA Road Network	USA-RN	23,947,347	5,833,333	0.244	1.2GB
Wikipedia page-to-page	Wiki-PP	5,716,808	130,160,392	22.76	1.5GB
Synthetic Data Sets	Syn-D_x	1-600M	13-8100M	13.5	0.2-114GB

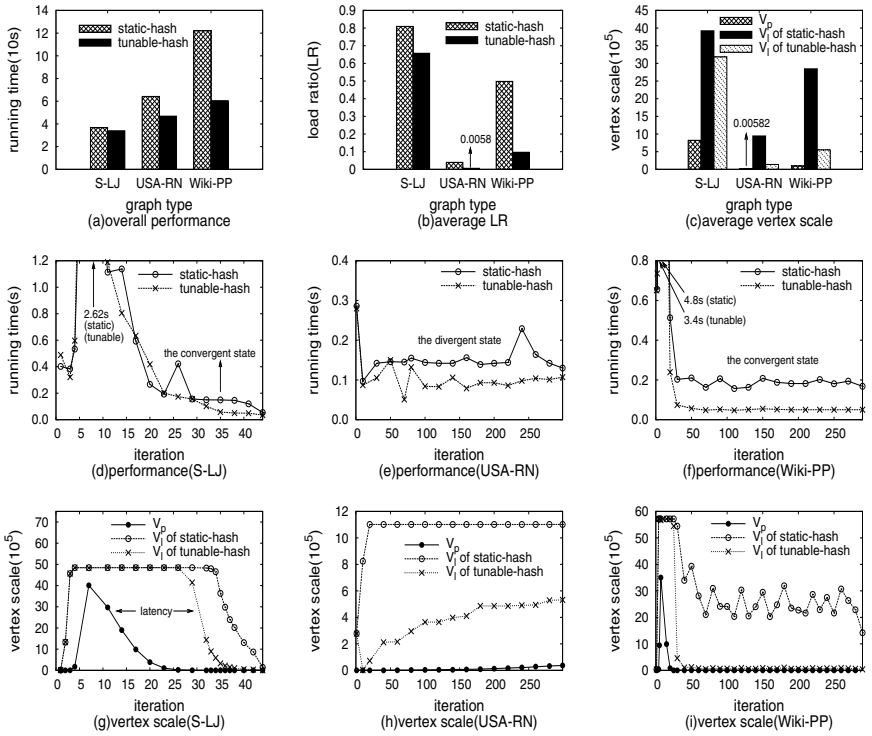


Fig. 4. Tunable hash index vs. static hash index (real data sets, 20 nodes)

only show the statistics of the first 300 iterations. In fact, it requires hundreds of iterations to fully converge because the graph has a huge diameter.

As shown in Fig 4(a), for Wiki-PP, the speedup of our tunable index compared to the static index is roughly a factor of 2. The tunable hash index has reduced its average LR of one iteration by 80.6% (Fig 4(b)), which means a large portion of *wasteful data* have been skipped. Therefore, its average LE ($|V_p|/|V_l|$) is improved by roughly 5 times (Fig 4(c)). The average LR of USA-RN is also reduced by up to 86%, but the overall performance is only improved by 28%, which is less than Wiki-PP. The reason is that USA-RN is a sparse graph, then the essential cost of warm-up (e.g., the initialization overhead of disk operations

and communication) occupies a considerable portion of the running time, which affects the overall effect of our index.

For S-LJ, the gain is not as obvious as that of USA-RN and Wiki-PP. By analyzing the performance of every iteration (Fig 4(d)-(f)), we notice that, for USA-RN and Wiki-PP, their resident time of the divergent or convergent state is much longer than that of S-LJ. During these two states, just as illustrated in Fig 4(g)-(i), the scale of *wasteful data* is reduced efficiently by the tunable hash index. For example, P-SSSP over Wiki-PP took 290 iterations to converge. Fig 4(i) shows a large subset of vertices have found the shortest distance within the first 40 iterations. The remaining 250 iterations update less than 3% of δ. Therefore, the cumulate effect of adjustments is tremendous. However, for S-LJ, the number of iterations is only 44. Considering the latency of adjustments (Fig 4(g)), the overall gain is not remarkable.

6.2 Evaluation of ASMP and ASVU

This suit of experiments is used to examine the effect of ASMP and ASVU (described in Section 5.2). They are implemented based on the tunable hash index. As a comparative standard, the baseline method in experiments does not adopt any policies (both ASMP and ASVU) to optimize the message-processing.

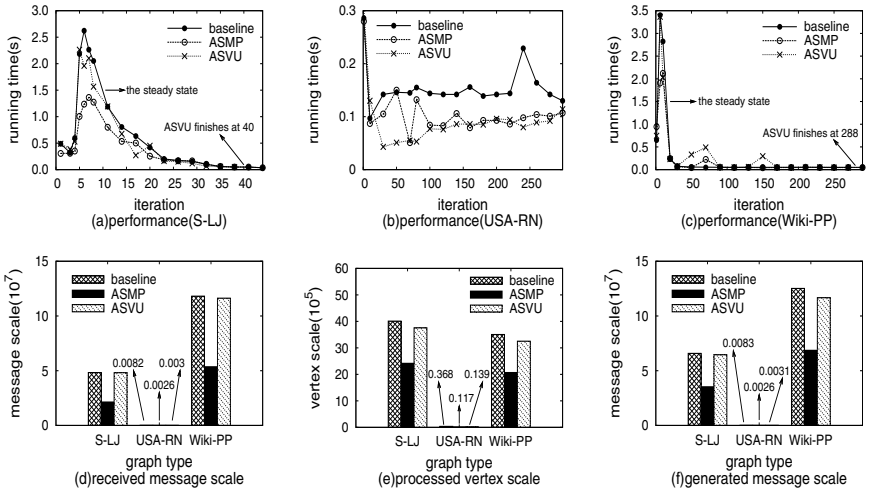


Fig. 5. Analysis on ASMP and ASVU (real data sets, 20 nodes)

As shown in Fig 5(a)-(c), the ASMP policy can optimize the performance of the steady state obviously. Especially for S-LJ, the overall performance can be improved by up to 23% because its resident time of the steady state is relatively longer than that of USA-RN and Wiki-PP. As illustrated in Fig 5(d)-(f), the effect of ASMP is tremendous at the iteration where the received message scale

has reached the peak. Exemplified by S-LJ, the number of received messages (M_t) can be reduced by 56%. Then, compared with the baseline method, 45% vertices will be skipped at this iteration, which reduces the cost of disk-accesses (Fig 5(e)). Finally, the scale of new messages also decreases by 46% (Fig 5(f)), which reduces the communication cost. We notice that the iterations of S-LJ and Wiki-PP with ASVU are both completed in advance (Fig 5(a) and (c)) because ASVU can accelerate the spread of messages. However, considering the impact of redundant messages (Fig 5(d)-(f)), the contributions to overall performance of ASVU is not as obvious as that of ASMP. Especially, for S-LJ, the performance of ASMP is 16% faster than that of ASVU.

6.3 Evaluation of Data Processing Capacity and Overall Efficiency

Compared to Giraph, Hama and Hadoop, the P-SSSP implementation on DiterGraph can be executed over large graphs efficiently with limited resources. First, we set the number of nodes as 10. As shown in Figure 6(a), benefitted from our tunable hash index and ASMP, the running time of DiterGraph is two times faster than that of Giraph. Compared with Hadoop and Hama, the speedup is even 21 to 43. We are unable to run P-SSSP on Giraph when the vertex scale is more than 4 million, as the system runs out of memory. Second, we evaluate the scalability of DiterGraph by varying graph sizes and node numbers (Figure 6(b)). Given 40 nodes, when the number of vertices varies from 100 million to 600 million, the increase from 415 seconds to 3262 seconds demonstrates that the running time increases linearly with the graph size. Given the graph size, such as 600 million, the running time decreases from 9998 seconds to 3262 seconds when the number of nodes increases from 10 to 40.

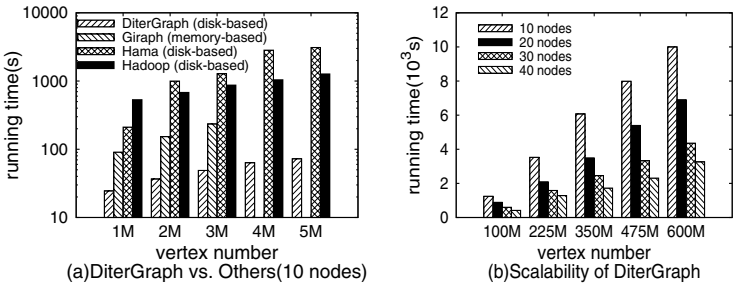


Fig. 6. Data processing capacity and overall efficiency (synthetic data sets)

7 Conclusion and Future Work

In this paper, we propose a novel state-transition model for P-SSSP. Then a tunable hash index is designed to optimize the cost of disk-accesses. By extending BSP, we propose the ASMP policy to reduce the message scale. The

extensive experimental studies illustrate that the first policy can optimize the performance during the divergent and convergent states. And the second policy is effective for the steady state. In future work, we will extend our methods for incremental-iterative algorithms, such as the connected components computation, belief propagation and the incremental PageRank computation.

Acknowledgments. This research is supported by the National Natural Science Foundation of China (61272179, 61033007, 61003058), the National Basic Research Program of China (973 Program) under Grant No.2012CB316201, and the Fundamental Research Funds for the Central Universities (N110404006, N100704001).

References

1. Gao, J., Jin, R.M., Zhou, J.S., et al.: Relational Approach for Shortest Path Discovery over Large Graphs. PVLDB 5(4), 358–369 (2012)
2. Malewicz, G., Austern, M.H., Bik, A.J.C., et al.: Pregel: A System for Large-Scale Graph Processing. In: Proc. of SIGMOD, pp. 135–146 (2010)
3. Apache Incubator Giraph, `http://incubator.apache.org/giraph/`
4. Ewen, S., Tzoumas, K., Kaufmann, M., et al.: Spinning Fast Iterative Data Flows. PVLDB 5(11), 1268–1279 (2012)
5. Apache Hama, `http://hama.apache.org/`
6. Meyer, U., Sanders, P.: Δ-Stepping: A Parallel Single Source Shortest Path Algorithm. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 393–404. Springer, Heidelberg (1998)
7. Thorup, M.: Undirected Single-Source Shortest Paths with Positive Integer Wights in Linear Time. JACM 46(3), 362–394 (1999)
8. Meyer, U., Osipov, V.: Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm. In: Proc. of ALENEX, pp. 85–96 (2009)
9. Cheng, J., Ke, Y., Chu, S., et al.: Efficient Processing of Distance Queries in Large Graphs: A Vertex Cover Approach. In: Proc. of SIGMOD, pp. 457–468 (2012)
10. Valiant, L.G.: A Bridging Model for Parallel Computation. Communications of the ACM 33(8), 103–111 (1990)
11. Apache Hadoop, `http://hadoop.apache.org/`
12. Fagin, R., Nievergelt, J., Pippenger, N.: Extendible Hashing - A Fast Access Method for Dynamic Files. TODS 4(3), 315–344 (1979)
13. Litwin, W.: Linear Hashing: A New Tool for File and Table Addressing. In: Proc. of VLDB, pp. 212–223 (1980)
14. Xiao, Y.H., Wu, W.T., Pei, J.: Efficiently Indexing Shortest Paths by Exploiting Symmetry in Graphs. In: Proc. of EDBT, pp. 493–504 (2009)
15. Wei, F.: TEDI: Efficient Shortest Path Query Answering on Graphs. In: Proc. of SIGMOD, pp. 99–110 (2010)
16. Trinity, `http://research.microsoft.com/en-us/projects/trinity/`
17. Bu, Y., Howe, B., Balazinska, M., et al.: HaLoop: Efficient Iterative Data Processing on Large Clusters. PVLDB 3(1-2), 285–296 (2010)
18. Twister: Iterative MapReduce, `http://www.iterativemapreduce.org/`
19. SNAP: Network dataset, `http://snap.stanford.edu/data/soc-LiveJournal1.html`
20. 9th DIMACS, `http://www.dis.uniroma1.it/challenge9/download.shtml`
21. Using the Wikipedia link dataset, `http://haselgrove.id.au/wikipedia.htm`