Optimizing CPU Cache Performance for Pregel-Like Graph Computation

Songjie Niu, Shimin Chen*
State Key Laboratory of Computer Architecture
Institute of Computing Technology
Chinese Academy of Sciences
{niusongjie,chensm}@ict.ac.cn

***Abstract*—In-memory graph computation systems have been used to support many important applications, such as PageRank on the web graph and social network analysis. In this paper, we study the CPU cache performance of graph computation. We have implemented a graph computation system, called *GraphLite*, in C/C++ based on the description of Pregel. We analyze the CPU cache behavior of the internal data structures and operations of graph computation. Then we exploit CPU cache prefetching techniques to improve the cache performance. Real machine experimental results show that our solution achieves 1.9–2.2x speedups compared to the baseline implementation.**

I. Introduction

Many real-world data sets with complex relationships can be modeled as graphs $G = (V, E)$. For example, in a web graph, each vertex $v \in V$ represents a web page and each directed edge $e_{vu} \in E$ represents a hyperlink in web page v that points to web page u. In a social network graph, a vertex v represents a user in a social network, and an edge e_{vu} can represent the follow relationship from user v to user u. Other examples of real-world graphs include road networks, publication citation relationships, and semantic networks.

A large number of algorithms have been developed on top of the graph data model, including PageRank [3], shortest path [6], betweenness centrality [7], community structure discovery [9]. For example, PageRank is a representative graph-based algorithm. It computes a ranking score for each vertex in a web graph, which is used to rank the web search results in search engines. The PageRank algorithm assigns an initial value to all vertices. Then it exploits the graph structure to repeatedly update the PageRank values of vertices until the values converge or other predefined criteria are met (e.g., after a pre-defined number of iterations). Such iterative computation is typical for most graph algorithms.

A number of graph processing systems have emerged to support graph analysis. Pregel [16] is Google's proprietary system based on the principles of Bulk Synchronous Parallel (BSP) model and vertex-centric computation. Open source implementations of Pregel are mainly Java based, including Giraph [8], Hama [11], GPS [19]. GraphLab [15] is another popular graph computation system. It allows asynchronous updates on vertex and edge values, which can speed up many machine learning algorithms. Recent studies have investigated various aspects of graph computation, including distributed asynchronous graph computation [14], [10], disk I/O optimizations [13], coarser grain computation units [20], [21], and algorithm-specific optimizations [18].

In this paper, we are interested in the CPU cache performance aspect of graph computation because most graph processing systems are main memory based. CPU cache performance has been extensively studied for relational database systems [2], [17], [5], [1], [12]. However, it has not been studied in depth for graph computation before.

We have implemented a Pregel-like graph computation system in C/C++[1], called GraphLite. We analyze the internal data structures and operations of GraphLite, and optimize their CPU cache performance. We find that accessing vertex-to-vertex messages often incurs poor CPU cache behaviors. The approach to storing all messages in contiguous arrays requires sorting the message array in each iteration. This incurs significant cost because message sizes are comparable to the cache line size (e.g., 0.5 cache line large in PageRank). Instead, we propose to exploit CPU cache prefetching to reduce the impact of cache misses. Real machine experiments show that our approach can significantly improve the performance of graph computation on a single machine node.

The contributions of this paper are three fold. First, to our knowledge, this is the first research study that focuses on the CPU cache performance of graph computation. Second, we analyze the CPU cache behaviors of graph computation in depth and exploit CPU cache prefetching to improve the cache performance. Finally, preliminary experimental results show the significance of our proposed solution.

The rest of the paper is organized as follows. Section II describes the programming interface and the baseline implementation of GraphLite. Section III analyzes the CPU cache behaviors of GraphLite, and proposes prefetching schemes to improve performance. Section IV presents preliminary experimental results. Finally, Section V concludes the paper.

II. Pregel-Like Graph Computation System

We have implemented a graph computation system, called *GraphLite*, in C/C++ based on the description of Pregel [16]. In this section, we first describe the vertex centric graph

*Corresponding author

[1]Open source implementation of Pregel is mostly based on Java. Because of JVM, Java programs often have limited control of the location and the memory layout of their allocated objects. Therefore, we find it easier to analyze and improve CPU cache performance in C/C++ programs.

programming model, then illustrate the internal data structures and operations in GraphLite.

A. Graph Programming Model

Pregel performs synchronous, vertex-centric graph computation. It follows the Bulk Synchronous Parallel (BSP) model. Under this model, the entire computation is composed of a series of supersteps. In each superstep, the system runs a user-defined `compute` function on each vertex in the graph (conceptually) in parallel without any synchronization. Synchronization occurs at the start of each superstep. The system waits for all the operations in the previous superstep to complete before starting the next superstep.

Figure 1 shows the vertex-centric programming interface supported by GraphLite. The interface defines a template class `Vertex`. An application program will implement a subclass of `Vertex` by overloading the `compute` method in the class. Typically, `compute` performs local computation to update the value of the current vertex, and communicates with neighbor vertices through messages. The other methods in the class are provided by the system and can be used in the implementation of the user defined `compute` method. We describe the semantics of the important methods and concepts as follows:

- **Vertex value:** An application program can add a fixed sized value of arbitrary type to each vertex. It can be accessed with `getValue` and `mutableValue`. `getVSize` returns the overall size of a user-defined `Vertex`, which is used to allocate space for vertices in GraphLite.
- **Outgoing messages:** The two send message methods are used to send outgoing messages from `compute`. Typically, a graph algorithm will send messages along all out-edges from a vertex. If this is the case and all the messages are the same, then `sendMessageToAllNeighbors` simplifies the algorithm implementation. Outgoing messages will be delivered in the next superstep.
- **Incoming messages:** Incoming messages that are sent in the previous superstep can be accessed by using the `MsgIterator` parameter of `compute`.
- **Global aggregates:** Aggregates are useful for computing global values such as total error and statistics. GraphLite supports sum, avg, min, and max by default, and provides an interface to implement customized aggregates. The `accumulate` method accumulates a new value to (the local copy of) the aggregate specified by `agg_id`. The global value of the aggregate is updated at the start of each superstep at the synchronization time. This global value can be obtained by `getAggregate`.
- **Computation completion criteria:** A vertex is in either the active or the inactive state. A vertex is active in superstep 0. `compute` transfers a vertex into the inactive state by calling `voteToHalt`. A vertex becomes active again if it receives an incoming message. The graph computation completes if all vertices are inactive and there are no messages at the start of a superstep.

An implementation of the PageRank [3] algorithm on GraphLite is shown in Figure 2. PageRank models a user's web browsing behavior as a random walk on the web graph.

```
1  template<typename V, typename E, typename M>
2  class Vertex : public VertexBase {
3   public: // user must implement compute()
4    virtual void compute(MsgIterator * msgs)=0;
5
6   public: // methods provided by the system
7    const V & getValue();
8    V * mutableValue();
9
10   OutEdgeIterator getOutEdgeIterator();
11   void sendMessageTo(const int64_t& dest_vertex,
12                      const M & msg);
13   void sendMessageToAllNeighbors(const M & msg);
14
15   void accumulate(const void * p, int agg_id);
16   const void* getAggregate(int agg_id);
17
18   void voteToHalt();
19
20   const int64_t & vertexID() const;
21   int superstep() const;
22   int getVSize() { return sizeof(V); }
23   int getESize() { return sizeof(E); }
24 };
```

Fig. 1. Vertex programming interface.

```
1  #define AGGERR 1
2  void compute(MsgIterator * msgs)
3  {
4   double val;
5   if (superstep() == 0) {
6     val= 1.0;  // initial value
7   }
8   else {
9     // check if converged
10    if (superstep() >= 2 &&
11        *(double *)getAggregate(AGGERR) < TH) {
12      voteToHalt(); return;
13    }
14    // compute pagerank
15    double sum= 0.0;
16    for (; !msgs->done(); msgs->next()) {
17       sum += msgs->getValue();
18    }
19    val = 0.15 + 0.85 * sum;
20    // accumulate delta pageranks
21    double acc = fabs(getValue() - val);
22    accumulate(&acc, AGGERR);
23   }
24   // set new pagerank value and propagate
25   *mutableValue() = val;
26   int64_t n = getOutEdgeIterator().size();
27   sendMessageToAllNeighbors(val / n);
28 }
```

Fig. 2. PageRank program using vertex programming interface.

The user clicks a random hyperlink in the current web page with a probability of 0.85 or goes to a random web page (by entering a URL) with a probability of 0.15. As shown in Figure 2, a vertex v in the graph represents a web page, and an out-edge e_{vu} represents a hyperlink from v to u. The implementation uses the vertex value to store the PageRank score. It models hyperlink clicks by sending messages along out-edges that carry partial PageRank scores. `compute` initializes all PageRank scores to 1 in superstep 0 (Line 5–7). In each subsequent superstep, it combines the incoming messages to compute the updated PageRank score for the current vertex (Line 14–19). At the end of a superstep, `compute` divides the score of the current vertex by the number of out-edges and sends the quotient as a message along each out-edge (Line 26–27). The computation continues iteratively until convergence.

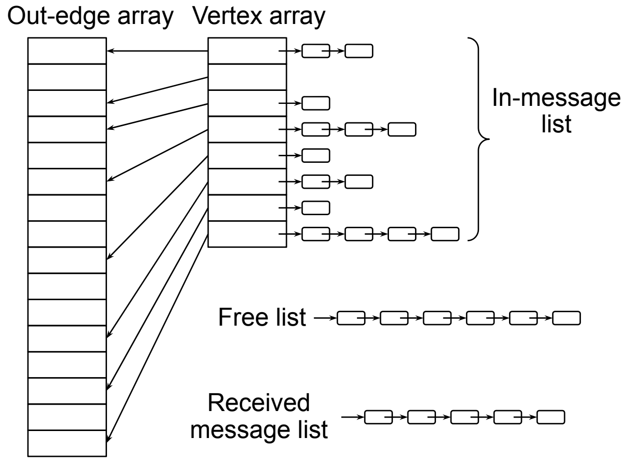

Fig. 3. Data structures in the baseline implementation.

The implementation computes the total absolute difference of PageRank scores as an aggregate (Line 21–22), and compares it against a predefined threshold `TH` (Line 9–13) to determine if the computation should complete.

B. GraphLite Baseline Implementation

GraphLite performs distributed computation by running a master process and a set of worker processes on different machines and/or different CPU cores. The master coordinates the global synchronization of each superstep and manages the global aggregates, while each worker performs in-memory computation on a partition of the graph.

In the following, we focus on the internal data structures and operations of a worker, which provide the basis for understanding and optimizing the CPU cache performance of graph computation in this paper. (Please note that the Pregel paper [16] mainly focuses on the distributed computation aspect of the system, and does not provide a detailed description of the internals of a worker.)

Figure 3 depicts the internal data structures to support graph computation in a worker. GraphLite allocates an array of vertices and an array of out-edges at the initialization time. While application programs can use arbitrary types for vertex value `V` and edge value `E`, GraphLite utilizes the `getVSize` and `getESize` methods to obtain their sizes[2]. The out-edges are sorted according to the source vertices. Then each vertex points to a contiguous number of edges in the out-edge array.

GraphLite maintains three data structures for messages as shown in Figure 3. First, each vertex points to an in-message list that contains the messages targeting this vertex received in the last superstep. The `MsgIterator` for `compute` is built on top of this in-message list. Second, there is a global list for messages received in the current superstep. The destination vertex of a message sent by `compute` can be on the same worker or on a different worker. For the former case, GraphLite puts the message directly into the global received message list. For the latter case, the worker puts the message into an outgoing buffer for the destination worker node. When the buffer is full, or when the current superstep completes, the worker performs the actual communication to send the messages in the outgoing buffer to the destination worker. Messages received from other workers are also put into the the global received message list. Finally, there is a global free list to facilitate the memory management for messages.

At the beginning of a superstep, a worker delivers every message in the received message list to the in-message lists of the associated destination vertices. It then calls `compute` on each vertex. A `compute` on vertex v will visit v's in-message list to retrieve all messages received in the last superstep, and send new messages. After `compute` on v returns, the worker will free all the messages in v's in-message list. The allocation of new messages is handled by the send methods. The new messages will be put into the received message lists of the destination workers before the start of the next superstep.

III. Analyzing and Optimizing CPU Cache Performance for Graph Computation

In this section, we first analyze the CPU cache behavior of the baseline implementation and then discuss how to optimize the CPU cache performance for GraphLite.

A. Analyzing CPU Cache Behavior

Let us consider the CPU cache behavior for accessing the internal data structures in the baseline implementation as shown in Figure 3. We assume that the size of the vertex array, the size of the edge array, and the size of all messages are all substantially larger than the size of the CPU cache.

- **Vertex and Edge Data:** In every superstep, a worker visits every vertex and calls `compute` in a loop. This is sequential memory access to the vertex array. `compute` often iterates through the out-edges of the current vertex to send messages. Since the out-edges are sorted in the order of the source vertex, the access to the out-edge data is also sequential. Sequential access is supported well by CPU caches. Therefore, both vertex and edge data see very good CPU cache behavior.
- **In-Message List:** In-message lists are populated at the beginning of a superstep when the worker distributes messages from the received message list. Since the messages are not in the order of the destination vertices, it is likely that the delivery of each message incurs a cache miss to access the in-message list head in the destination vertex. Since this distribution operation shuffles the messages across the vertices, the messages in an in-message list of a vertex are likely to be scattered across the memory. As a result, when reading the in-message list, `compute` is likely to incur a CPU cache miss for every message in the list. Moreover, because of the linked list organization, the memory address of message i in the list is available only after message $i-1$ is loaded into the cache. Consequently, the cache miss to load message i will be fully exposed. It will not be overlapped with the cache miss that loads message $i-1$.
- **Received Message List:** The above message distribution operation reads the messages in the received message list. Like reading in-message lists, the read operation performs a linked list traversal, and is likely to incur a fully exposed cache miss for every message in the list. On the other

[2]GraphLite currently supports only fixed sized values and edges, which can support many graph algorithms, including PageRank, shorted path, centrality computation. We plan to investigate the support for variable sized values and edges, whose sizes may change during computation.

hand, unlike in-message lists, the operation to insert a new message into the list has good cache performance because the global head of the received message list is frequently visited and usually stays in the cache.

- **Free List:** New messages are allocated in the send methods that are called in `compute`. Messages in the in-message list are freed after `compute`. Therefore, the free list sees frequent allocation and free operations. Freed memory is likely to be reused very soon. In fact, in a graph computation (e.g., PageRank), the number of in-messages (out-messages) for a vertex v often equals to the in-degree (out-degree) of v. Therefore, the length of the free list is equal to the difference between the total in-degrees and the total out-degrees seen so far. If the difference is not very large, the free list is likely to stay in the CPU cache. Therefore, the free list often sees good CPU cache performance.

We analyze the CPU cache behavior from the perspective of a message m. m is generated by the send method called by `compute`. The message allocation and the insertion of m into the received message list are cache friendly. However, the distribution of m to m's destination vertex v incurs two cache misses, one for m and one for v. Visiting m in the in-memory list incurs another cache miss. Therefore, each message is likely to incur three cache misses.

B. Optimizing CPU Cache Performance

We have chosen the linked list structure in our baseline design because it is simple to implement and it is effective to support memory reuse so as to fully utilize memory for graph computation. However, the above analysis shows that the linked list structure incurs poor CPU cache behavior for in-message lists and the received message list. We would like to re-design the data structure to improve CPU cache performance while maintaining good space efficiency.

One approach is to discard the linked list structure altogether. Instead we store the messages in arrays. All the in-message lists are stored in an array. This in-message array is sorted in the order of the destination vertices. In this way, reading the in-messages in `compute` performs cache-friendly sequential memory access. Similarly, the received message list is implemented as an array. Then the distribution process performs a sorting operation on the received message array to sort the messages in the order of the destination vertices.

Suppose the size of a message is S, the total number of messages is N, the CPU cache size is C, and the cache line size is L. This approach will write NS bytes to the received message array and read NS bytes from the in-message array. The sorting will read and write the entire array $\lceil log_b \frac{NS}{C} \rceil$ times until the sub arrays fit into the CPU cache. For quick sort, $b = 2$. One can also design a distribution based sorting algorithm to achieve a larger b. The idea is to divide an array into $b > 2$ buckets in each pass. Note that b is limited by the number (e.g., 32) of DTLB entries so that the distribution operation will not incur TLB misses. Every pass of the distribution sort will need to read the array one more time to count the number of entries in each bucket. Overall, this approach incurs $n = \frac{S}{L}(2 + 2\lceil log_b \frac{NS}{C} \rceil)$ cache misses per message. When the graph is large, $\lceil log_b \frac{NS}{C} \rceil \geq 2$. Since a message contains the source vertex ID, the destination

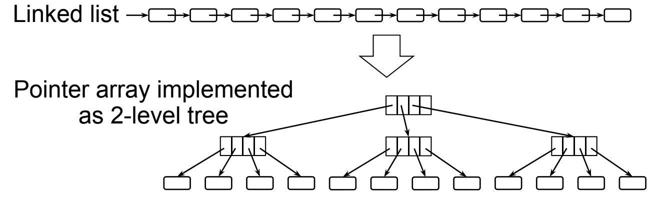

Fig. 4. Replacing linked lists of messages by pointer arrays in order to remove data dependence and employ CPU cache prefetching. For very long lists, pointer arrays are implemented as 2-level trees.

vertex ID, and the message content, $\frac{S}{L}$ is often non-trivial. For example, in PageRank, $\frac{S}{L} = 0.5$ and thus $n \geq 3$. Therefore, the number of cache misses per message is comparable or worse than that of the baseline implementation. The benefit of the array based solution will quickly diminish as the size of the message increases.

We propose to employ CPU cache prefetching to reduce the impact of cache misses. First, we replace all the linked list structures with pointer arrays. Note that this does not require additional space since we essentially move the pointer space from every message to the pointer arrays. For in-message lists, we use `std::vector`, which is a dynamic array. However, for the global received message list, the pointer array would be very large, and dynamic allocation and resizing would be costly. Therefore, we design a 2-level tree structure as shown in Figure 4. The tree node of the lower level is each 1MB large, which can hold up to 128K 8-byte pointers.

Second, we employ simple prefetching for reading the in-message list. Let d be the prefetching distance, which is a tunable parameter. In the constructor of the `MsgIterator` of `compute`, we issue prefetch instructions for the first d messages in the list. Then when message i is accessed, we issue a prefetch instruction for message $i + d$. We make sure that prefetching does not go beyond the end of the list.

Finally, we employ prefetching for the message distribution process. The distribution of a message incurs two dependent cache misses, one for the message and one for the destination vertex. The above simple prefetching solution cannot be used to prefetch the latter since the destination vertex is computed from the message content. A prefetch would be issued too late for the destination vertex. Instead, we exploit more sophisticated prefetching techniques, namely group prefetching and software pipelined prefetching, as proposed previously for cache efficient hash joins [4]. Figure 5 shows the group prefetching implementation for the distribution process. The algorithm delivers a group of messages at a time. For each group, step (1) prefetches all the messages. In this way, the cache misses to load the messages will be serviced in parallel. Next, step (2) computes and prefetches all the destination vertices (a.k.a. nodes). The computation assumes hash based graph partitioning. Similarly, the cache misses to load the nodes will be serviced in parallel. Step (3) delivers the messages. Here, $pref_group_size$ is a tunable parameter. Like group prefetching, software pipelined prefetching also takes advantage of the fact that each message is independent of the other messages. It builds a three stage software pipeline (corresponding to the three steps in group prefetching), and processes a different message at each stage in every iteration.

```
Msg  *msg[pref_group_size], *mp;
Node *nd[pref_group_size], *np;
int64t i, j, index;

PointerArray::Iterator* it =
                received_msg->getIterator();
int64_t total_num= received_msg->total();

// deliver a group of messages per iteration
i= pref_group_size;
for (; i < total_num; i+= pref_group_size) {
  // (1) prefetch the group of messages
  for (j= 0; j < pref_group_size; j++) {
    msg[j]= (Msg *)it->next();
    prefetch(msg[j]);
  }
  // (2) prefetch the group of nodes
  for (j = 0; j < pref_group_size; j++) {
    index= msg[j]->dest_id / worker_cnt;
    nd[j]= getNode(index);
    prefetch(nd[j]);
  }
  // (3) deliver the group of messages
  for (j= 0; j < pref_group_size; j++) {
    nd[j]->recvNewMsg(msg[j]);
  }
}

// deliver the rest of the messages
mp= (Msg *)it->next();
for (; mp; mp= (Msg *)it->next()) {
  index= mp->dest_id / worker_cnt;
  np= getNode(index);
  np->recvNewMsg(mp);
}
```

Fig. 5. Group prefetching for message delivery.

IV. Preliminary Evaluation

In this section, we perform real-machine experiments to understand the impact of our proposed optimizations on graph computation. We first describe the setup for running the experiments and then discuss the experimental results.

A. Experimental Setup

Machine configuration. In our preliminary evaluation, we focus on the CPU cache performance of a single worker. We perform all the experiments on an Intel x86-64 machine equipped with a 3.40GHz Intel i7-4770 CPU, 8 MB L3 cache, and 16 GB main memory. The machine runs Ubuntu 13.10 with Linux 3.11.0-12-generic kernel. All the code is compiled with g++ 4.8.1 with the optimization level -O3. Before running the experiments, we disabled dynamic power management of the machine and set the CPU frequency to the maximum. For each experiment, we perform 10 runs and report the average result across the 10 runs.

GraphLite implementations. We report the results for six implementations. Their labels and meanings are as follows. (1) *base*: the baseline implementation as described in Section II-B. (2) *imm*: this is a variant of the baseline. We do not maintain the global received message list. Instead, we add a local received message list in every vertex. Once a message arrives, we immediately insert it into the received message list in the destination vertex. (3) *sort*: we store messages in arrays and apply quick sort as described in Section III-B. (4) *opt1*: we perform group prefetching for message distribution. The results of software pipelined prefetching are similar to those of group prefetching. Hence, we omit results for software pipelined prefetching. (5) *opt12*: besides opt1, we employ simple prefetching for in-message lists. (6) *opt123*: besides opt12, we further employ prefetching for the free list. This is not described in Section III-B because our analysis in Section III-A finds that the access to the free list often has good cache performance. We would like to see if the experimental results actually support our analysis.

TABLE I. Data set description.

Name	Type	Vertices	Edges	Description
email-Enron	Directed	36,692	367,662	Email communication network from Enron
amazon0505	Directed	410,236	3,356,824	Amazon product co-purchasing network from May 05 2003
soc-Pokec-rel	Directed	1,632,803	30,622,564	Pokec social network user relationship data
soc-LiveJournal1	Directed	4,847,571	68,993,773	LiveJournal online friendship social network

Note: Data sets are downloaded from http://snap.stanford.edu/data/.

Workload. We run the PageRank algorithm on four real-world graphs with increasing numbers of vertices and edges, as listed in Table I. The largest graph, soc-LiveJournal1, contains about 69 million edges. Since the PageRank algorithm sends a message along every edge in each iteration and the message size is 32 byte in our current implementation, the total data size for incoming messages in an iteration is 2.2GB, which is much larger than the CPU L3 cache size. We modify the PageRank implementation in Figure 2 so that the computation completes after 10 supersteps.

B. Experimental Results

Figure 6 shows the elapsed times of the PageRank computation on six variants of GraphLite implementations. From left to right, the four figures correspond to the four graph data sets in Table I. The number of vertices in the graphs increases from 36.7 thousand to 4.8 million, and the number of edges increases from 368 thousand to 69 million. As expected, the larger the graph, the longer the execution time of PageRank.

Comparing *base* and *imm*, we see that *imm* improves *base* by a factor of 1.2–1.4x. This shows that it is a good idea to deliver the messages immediately when the messages are received. Compared to the global message distribution process in *base*, *imm* saves the cache misses for traversing he global received message list to retrieve the messages.

Comparing *sort* and the two baselines *base* and *imm*, we see that *sort* only slightly improves *base* but it is significantly worse than *imm*. Note that our current implementation uses the quick sort algorithm. A more efficient sorting algorithm may achieve better performance. Nevertheless, the results confirm our observation in Section III-B. Because the message size in graph computation is comparable to the cache line size, it is less beneficial to employ the *sort* approach.

The three optimization schemes employ CPU cache prefetching to improve the CPU cache performance of graph computation. We see that group prefetching for the message distribution process (*opt1*) achieves 1.2–1.6x improvements over *base*. Applying prefetching for the in-message lists (*opt12*) achieves an additional 1.3–1.6x improvements over

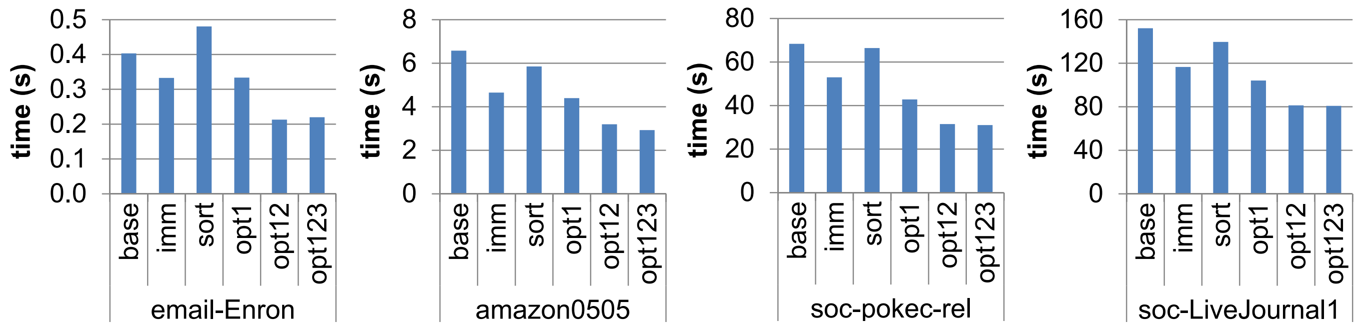

Fig. 6. Elapsed time for PageRank. (*base*: baseline implementation; *imm*: immediately deliver messages in send methods; *sort*: store messages in arrays and do sorting; *opt1*: group prefetching for the distribution process; *opt12*: opt1 + prefetching for in-message lists; *opt123*: opt12 + prefetching for free list.)

opt1. However, prefetching for the free list (*opt123*) sees little benefit over *opt12*. The experimental results confirm our analysis in SectionIII. Accessing the in-memory lists and the global received message list incur fully exposed cache misses for the messages and/or the vertex data. CPU cache prefetching can effectively overlap multiple memory loads that serve the cache misses, thereby reducing the impact of cache misses on the elapsed time. On the other hand, the free list often stays in the cache because of the frequent allocation and free operations. Therefore, *opt123* is almost the same as *opt12*.

Overall, our optimizations that employ CPU cache prefetching to reduce the impact of cache misses are very effective. *opt12* achieves 1.9–2.2x speedups over *base*, and 1.4–1.7x speedups over *imm*.

V. Conclusion and Future Work

Graph is a popular data model for big data analysis. As graph systems typically hold and process graph data in main memory, their CPU cache behavior plays an important role in the efficiency of the computation.

In this paper, we analyze the internal data structures and operations of a Pregel-like graph computation system. We find that the system often incurs expensive fully exposed cache misses for processing inter-vertex messages. We propose to exploit CPU cache prefetching to reduce the impact of cache misses, while maintaining good memory space utilization of the baseline implementation. Preliminary experimental results show that CPU cache prefetching can significantly improve graph computation, achieving 1.9–2.2x speedups compared to the baseline implementation.

This paper reports our on-going effort to optimize the CPU cache performance of Pregel-Like graph computation system. We plan to extend this study in the following three dimensions: (1) studying more graph algorithms that have different computation and communication characteristics; (2) performing experiments in a distributed environment for larger graphs; and (3) performing in-depth quantitative analysis of the CPU cache behaviors for graph computation.

Acknowledgment

The second author is partially supported by the CAS Hundred Talents program and by NSFC Innovation Research Group No. 61221062.

References

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB*, 2001, pp. 169–180.

[2] P. A. Boncz, S. Manegold, and M. L. Kersten, "Database architecture optimized for the new bottleneck: Memory access," in *VLDB*, 1999.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry, "Improving hash join performance through prefetching," in *ICDE*, 2004.

[5] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," in *SIGMOD*, 2001.

[6] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Math. Program.*, vol. 73, pp. 129–174, 1996.

[7] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, no. 1, pp. pp. 35–41, 1977.

[8] A. Giraph, http://giraph.apache.org.

[9] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proc Natl Acad Sci U S A*, vol. 99, no. 12, pp. 7821–7826, June 2002.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 17–30.

[11] A. Hama, http://hama.apache.org.

[12] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious B^+-trees," in *SIGMETRICS*, 2003.

[13] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, 2012, pp. 31–46.

[14] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *UAI*, 2010, pp. 340–349.

[16] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD Conference*, 2010, pp. 135–146.

[17] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *VLDB*, 1999, pp. 78–89.

[18] Salihoglu and J. Widom, "Optimizing graph algorithms on pregel-like systems," *PVLDB*, vol. 7, no. 7, pp. 577–588, 2014.

[19] S. Salihoglu and J. Widom, "GPS: a graph processing system," in *SSDBM*, 2013, p. 22.

[20] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[21] W. Xie, G. Wang, D. Bindel, A. J. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *PVLDB*, vol. 6, no. 14, pp. 2014–2025, 2013.