Cache-Conscious Graph Collaborative Filtering on Multi-socket Multicore Systems

Lifeng Nai[1], Yinglong Xia[2], Ching-Yung Lin[2], Bo Hong[1], and Hsien-Hsin S. Lee[1]
[1]Georgia Institute of Technology, Atlanta, GA 30332, USA
[2]IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA
{lnai3,bohong,leehs}@gatech.edu[1], {yxia,chingyung}@us.ibm.com[2]

ABSTRACT

Recommendation systems using graph collaborative filtering often require responses in real time and high throughput. Therefore, besides recommendation accuracy, it is critical to study high performance concurrent collaborative filtering on modern platforms. To achieve high performance, we study the graph data locality characteristics of collaborative filtering. Our experiments demonstrate that although an individual graph traversal exhibits poor data locality, multiple queries have a tendency of sharing their data footprints, especially in the case of queries with neighboring root vertices. Such characteristics lead to both inter- and intra-thread data locality, which can be utilized to significantly improve collaborative filtering performance.

Based on these observations, we present a cache-conscious system for collaborative filtering on modern multi-socket multicore platforms. In this system, we propose a cache-conscious query scheduling technique and an in-memory graph representation, and to maximize cache performance and minimize cross-core/socket communication overhead, we address both inter- and intra-thread data locality. To address the workload balancing issue, this study introduces a dynamic work-stealing mechanism to explore the trade-off between workload balancing and cache-consciousness.

The proposed system was evaluated on a Power7+ system against the IBM Knowledge Repository graph dataset. The results demonstrated both good scalability and throughput. Compared with the basic system that does not perform cache-conscious scheduling, inter-thread scheduling improves throughput by up to 18%. Intra-thread scheduling can further improve throughput by as much as 22%. By enabling dynamic work-stealing, the proposed technique balances workloads across all threads with a low standard deviation of the per-thread processing time.

Keywords

collaborative filtering; cache-conscious; parallel computing;

1. INTRODUCTION

Numerous e-commerce websites rely on recommendation systems to help online customers avoid information overload by making suggestions regarding which information is the most relevant to them [1]. To achieve better recommendation results, researchers have proposed several algorithms. One of the most popular algorithms is collaborative filtering, which they have studied, enhanced, and implemented in prior proposals, such as Tapestry [2], Ringo [3], and Video Recommender [4], using various approaches.

In this paper, instead of focusing on the accuracy of collaborative filtering results, we concentrate on system performance. We study an item-based collaborative filtering system in which recommendation results are achieved by graph breadth-first-search and similarity sorting (algorithm details can be found in Section 6.1). In this system, which requires that a large number of concurrent queries be processed in real time, throughput is the major consideration. Therefore, data locality and its impact on cache performance become key optimization goals [5][6][7].

Graph algorithms have poor data locality because of their irregular access behaviors, which follow a pointer-chasing pattern. Optimizing an individual instance of a graph algorithm is known to be difficult [8][9]. However, for our target collaborative filtering problem, we observed that although a single query typically shows poor data locality, multiple queries have a tendency of sharing their data, which can lead to data locality between inter- and intra-thread queries. Such locality significantly impacts cache performance and communication overhead. The impact is both modeled and experimentally measured. We observed significant throughput, which was the motivation for developing the proposed technique. Meanwhile, we notice that brute-force data locality aware methods tend to lead to unbalanced workload scheduling. To handle such issues, we also accounted for the balance between workload balancing and data locality in the proposed technique.

Based on the above observations, we propose a cache-conscious implementation of high throughput collaborative filtering. In the proposed technique, we schedule queries in accordance with their data locality behaviors to maximize cache performance and minimize communication overhead and use a dynamic work-stealing method to ensure proper balance between cache-consciousness and workload balancing. Our observations and techniques are not limited by specific algorithms. Many graph analytic queries tend to traverse only a subset of the whole graph. In these cases, data localities between different queries also exist. Therefore, our techniques can easily be generalized to other throughput-oriented graph algorithms and systems.

Finally, to evaluate the proposed methods, we conduct extensive experiments. Compared to the basic system with simple round-robin scheduling, our graph representation and workload partitioning method achieves good scalability and throughput. With inter-thread scheduling, the proposed technique can improve throughput by up to 18% over the basic system. When both inter- and

CF'14 May 20 - 22 2014, Cagliari, Italy

http://dx.doi.org/10.1145/2597917.2597935.

intra-thread scheduling are applied, the technique achieves 22% improvement. By enabling dynamic work-stealing, the technique balances workloads with limited variance in per-thread processing time. In general, it achieves cache-conscious high-throughput implementation of collaborative filtering.

The main contributions of this paper are as follows:

- To the best of our knowledge, this is the first study of data locality of a collaborative filtering system on modern multi-socket multicore platforms. We analyze data locality characteristics between collaborative filtering queries and observe and demonstrate both inter- and intra-thread data locality.
- We propose a complete cache-conscious implementation of collaborative filtering system, which includes an in-memory graph representation, a lock-free vertex data structure, a cache-conscious inter- and intra-thread scheduling technique, and a dynamic work-stealing method.

The rest of the paper is organized as follows. Section 2 analyzes data locality and workload balancing issues that motivate our work, and Section 3 provides the details of the implementation. Section 4 examines our technique for both scalability and throughput. Section 5 summarizes our work. The last section, Appendix 6, introduces the baseline collaborative filtering algorithm used in our work and discusses the background and previous proposals of collaborative filtering.

2. MOTIVATION

As a method of predicting user interest, collaborative filtering has been employed in various previously proposed approaches [2] [3] [4]. Researchers have typically focused more on the accuracy and scalability of such algorithms, overlooking the impact of the cache hierarchy on performance. Because of their random access patterns, graph collaborative filtering applications typically have poor data locality. However, in certain situations, data locality may still exist and enhance system performance. This work uses an item-based method as the target collaborative filtering system. Each query performs an item-to-item breadth-first-search to find the most relevant items (refer to Section 6.1 for more details). Although data locality can rarely be found within queries, multiple queries may still share a significant amount of data. Such data locality, which can be represented as both inter- and intra-thread locality, improves performance. In our target collaborative filtering system, overall throughput is the major concern. Thus, we should carefully analyze data locality between queries and its impact on cache performance.

2.1 Data locality

As the major component of collaborative filtering, breadth-first-search (BFS) is usually considered as a poor data locality operation [10]. Data access pattern of BFS largely relies on graph dataset itself. The irregularity in graph edges makes it difficult to exploit its data locality, leading to poor cache performance. However, different runs of BFS traversal may still share a significant portion of their accessed data.

To approximately estimate the access behavior of BFS traversal, the model below is used.

Notations:
T_{vertex}: total number of vertices accessed in N level BFS traversals
O_d: overlapped vertices between two BFS traversals with distance d
V_k: number of vertices accessed in level k
E: average number of edges per vertex

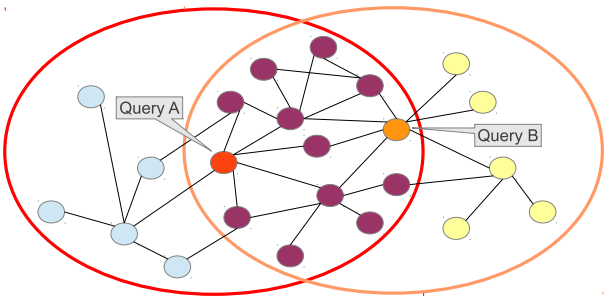

Figure 1: Example of neighbor queries

N: number of BFS traversal levels
p_i: probability of accessing a new vertex in level i
α_k: probability of a vertex sharing by two BFS traversals

The number of new vertices accessed in level k will be:

$$V_k = V_{k-1} \times E \times p_k \tag{1}$$

With $V_1 = E$ and $p_i = 1$, we can have

$$V_k = E^k \prod_{i=1}^{k} p_i \tag{2}$$

Therefore, the total number of accessed vertices is:

$$T_{vertex} = \sum_{k=1}^{N} V_k = \sum_{k=1}^{N} E^k \prod_{i=1}^{k} p_i \tag{3}$$

Meanwhile, for two BFS traversals with distance d, the vertices accessed in level below N-d will be overlapped. Besides them, other vertices are also possible to be overlapped depending on graph structures. Thus, we can have:

$$O_d = \sum_{k=1}^{N-d} V_k + \sum_{k=d}^{N} \alpha_k V_k \tag{4}$$

As shown in Equation 4, the number of overlapped vertices of two BFS traversals are decided by two factors, the distance d and the probability of vertices sharing by two BFS traversals α_k. Although the probability α_k is a parameter determined by graph dataset, Equation 4 clearly shows that two BFS traversals can share significant amount of vertices if distance is short.

An illustration example is also shown in Figure 1. In this example, collaborative filtering query A and B are performing BFS traversal starting from two different vertices with short distance. They are considered as **neighbor queries**. As shown in Figure 1, these two neighbor queries share significant amount of vertices in their traversal. Apparently, the overlapped vertices between them can result in data locality and help cache performance. Meanwhile, such data locality exists between both intra- and inter-thread queries. The details will be explained in sections below.

2.2 Intra-thread locality

For queries within the same thread, neighbor queries have the potential of sharing their accessed vertices and result in intra-thread locality. In order to achieve performance benefits from the locality, the scheduling of intra-thread queries plays a crucial role. Intuitively, the neighbor queries should be scheduled together to maximize benefits of data locality.

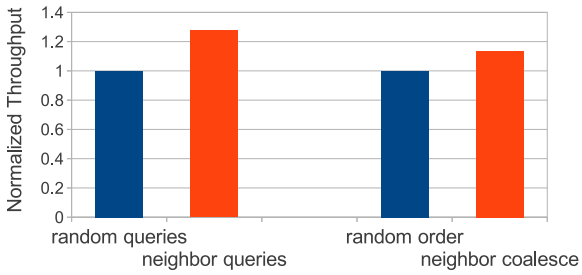

Figure 2: Motivation experiments for intra-thread locality (see text below for explanation of the terms)

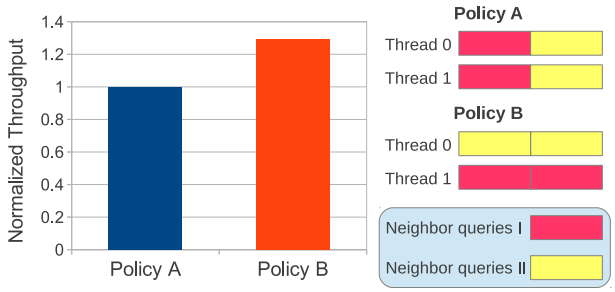

Figure 3: Motivation experiment of different inter-thread scheduling policies (see text below for details of policies A and B)

In graph traversals, when visiting each vertex, both outgoing edges and vertex property will be accessed. The total volume of data accessed when visiting each vertex can be represented as Equation 5

Notations:
Φ*: total accessed data size when visiting a vertex*
S_e*: edge data size in bytes*
S_v*: vertex property data size in bytes*

The total accessed data size will be:

$$\Phi = S_e \times E + S_v \tag{5}$$

The dataset used in our collaborative filtering system is the IBM Knowledge Repository graph dataset (refer to Section 4.1 for more details). In this dataset, E is around 10, while S_e is 24 bytes and S_v is 48 bytes. Thus, the data size Φ of one vertex is close to 300 bytes. According to Equation 3, the total accessed data size is around 450 KB for a 4 level BFS traversal, which is almost twice of L2 cache size and 1/8 of L3 cache size. Therefore, even if two queries have locality, few other unrelated queries can easily evict the useful data out of cache and reduce the impact of data locality. To further explain this observation, two motivation experiments are performed.

In the first experiment, two different groups of queries are processed independently. In both groups, we choose queries with similar complexity but different distances, **neighbor queries** in the first group and **random queries** in the second one. From the result shown in Figure 2, we can clearly see that neighbor queries show a 27.4% improvement of throughput than random queries. It well supports our intuition that the vertices accessed by neighbor queries may be significantly overlapped. Such a pattern will result in data locality and benefit cache performance.

While in the second experiment, these two groups of queries are processed together in the same thread with different interleaving patterns. Neighbor queries are coalesced together in the second test (**neighbor coalesce**) while queries in the first test are interleaved randomly (**random order**). The performance results are shown in Figure 2. Comparing with random order, simply scheduling neighbor queries together (neighbor coalesce) improves throughput by 13.5%. Although data locality exists between neighbor queries, processing order of queries still matters. Therefore, in order to maintain the benefit of data locality, query scheduling must be aware of intra-thread locality between queries.

2.3 Inter-thread locality

In addition to the intra-thread data locality, neighbor queries of different threads may also contain data locality, which leads to inter-thread data locality and brings impact on cross-core/socket communication overhead.

In multi-socket multicore platforms, besides the overhead due to cache misses, intensive inter-cache communication can also lead to significant performance degradation. Memory accesses that hit remote caches will trigger data transfer across sockets or cores via cache coherence messages. For data intensive applications, if different threads are sharing significant amount of data, cache coherence messages can introduce extra communication overhead. Worse yet, data ping-pong effect can take place when write operations are involved. The communication overhead depends on the underlying cache hierarchy and interconnection structure. For private caches within the same chip, the overhead comes from on-chip interconnection, while for cores of different sockets, it is caused by inter-socket communication, which is even more severe due to the limited cross-socket bandwidth. Therefore, inter-thread data locality and its impact on communication traffic must be taken into account.

Consider the experiment presented in Figure 3. 32 queries from two neighborhoods are partitioned into two separate threads running on cores from different sockets. Two types of scheduling policies are performed. Neighbor queries (refer to Section B for definition) are processed by the same thread in **policy B**, while in **policy A**, neighbor queries are divided evenly into two threads with the same intra-thread order. As shown in Figure 3, by grouping queries with shared data in the same thread, policy B improves the throughput by 28.1%, which is in accordance with our previous explanation. In policy B, most of data accesses do not hit remote caches. High communication traffic and ping-pong effect are both avoided. Accordingly, in our proposed collaborative filtering implementation, besides intra-thread data locality, data locality between threads also needs to be considered to minimize the inter-cache communication overhead.

2.4 Workload balancing

To fully utilize multi-core architectures, scheduling queries in a balanced way is crucial [11]. However, the requirement of workload balancing is contradicting with data locality considerations, which presents challenges for workload balancing mechanism.

To explore intra- and inter-thread locality, it is desirable to merge queries into a small number of threads to improve cache performance. However, this would also lead to unbalanced workloads in these threads. As shown in Figure 4's example, 10 queries are requested. Seven of them are from one neighborhood, while the

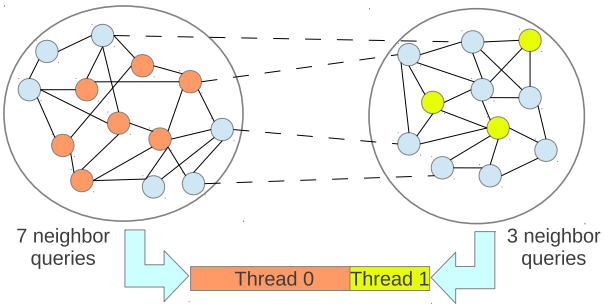

Figure 4: Example of workload balancing issue

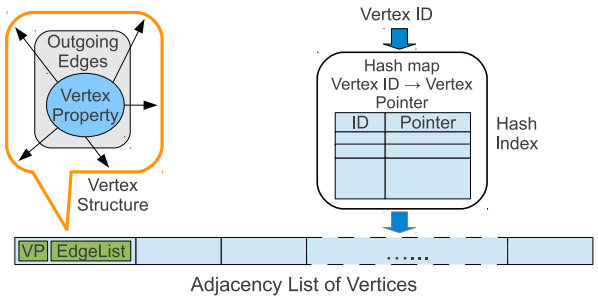

Figure 5: In-memory graph representation

other three are from another remote neighborhood. Considering inter- and intra-data locality, neighbor queries should be scheduled back to back together in the same thread. As such, the seven queries should be processed in thread 0, while the other 3 queries should be processed in thread 1. In this case, the workload of each thread is highly unbalanced and heavily loaded thread 0 will take longer processing time while thread 1 is idle for most of the time.

To address this challenge, we need to dynamically balance workload among cores and be aware of both intra- and inter-thread data locality at the same time. All these tradeoffs must be addressed in a complete framework to strike an optimal balance.

3. CACHE-CONSCIOUS COLLABORATIVE FILTERING

Modern computer architectures are becoming more and more complicated. They usually consist of multi-sockets of multi-core chips, sophisticated memory hierarchy, and cross-socket/core communication mechanism. To achieve better performance of collaborative filtering, we should try to avoid performance bottlenecks. Applications should aim to maximize the cache hit rate, minimize the cross-core/cross-socket communication overhead, and balance the CPU utilization. To achieve these targets, we implement our collaborative filtering system based on the following components: (1) a workload partitioning method, (2) an in-memory graph representation, (3) a lock-free data structure for concurrent queries, (4) a cache-conscious query scheduling technique, and (5) a dynamic work balancing method. The details are explained below.

3.1 Workload partitioning

As mentioned in Section 6.1, the collaborative filtering workload contains multiple incoming queries that need to be processed in real time. In order to achieve high throughput and full utilization of underlying computation resources, the multiple queries need to be properly partitioned into different threads.

As introduced in Section 6.1, the key operation of collaborative filtering is BFS graph traversal. An intuitive way of workload partitioning is to partition graph traversal operations. Several parallel implementations of the graph traversal algorithm have been recently proposed, which can traverse graph data on a distributed system with multiple nodes, or on a single-node multi-core/socket system [12][13][14][15]. Although all graph traversal algorithms share significant similarity fundamentally, the traversal operations in collaborative filtering is accessing only a small subset of the whole graph with a limited number of hops. It will lead to fine-grained workload partitioning if we perform partitioning within graph traversal operations. Consequently, each workload chunk will be too small to hide the communication and synchronization overhead between parallel threads. Previously proposed parallel algorithms become infeasible in this case.

Thus, a coarser grained partitioning method should be used. Considering the number of incoming queries, partitioning workload at the granularity of queries becomes another choice. In this way, sophisticated thread level synchronization and scheduling can be avoided. The partitioned unit is naturally one query, which is easier for both scheduling and partitioning. Considering these advantages, query granularity partitioning is used in our proposed collaborative filtering implementation.

3.2 In-memory graph representation

Graph operations usually suffer from poor cache performance due to the dynamic nature of graph datasets. Despite of that, a proper in-memory representation of graph can still significantly affect the cache performance of graph operations. In our collaborative filtering implementation, data structure should be graph traversal friendly and flexible enough to support timely graph update. Therefore, we propose a vertex-centric data structure to represent a graph in memory.

As shown in Figure 5, a vertex is the basic unit of a graph. The vertex property and the outgoing edges stay within the same vertex structure. Such a design is in accordance with our graph traversal pattern. When traversing a graph via BFS, the vertex property and edges will be accessed together. By organizing them in contiguous memory blocks, cache residents have larger chance of reuse. Vertices are organized in an adjacency list with hash indices. Each vertex is indexed by a unique integer vertex id. Compared with a vector structure, a list does not compromise the cache performance because of the dynamic vertex access pattern of BFS. Moreover, unlike vector structure, allocating new vertices only introduce limited overhead in a list. No extra memory allocation and copy operations are required. Thus, in our implementation, graph is represented in a two-level structure. In the first level, vertices stay in a list with hash index, which can benefit graph update. While in the second level, the vertex property and edges are arranged in a sequential manner to increase the cache data reuse rate.

3.3 Lock-free vertex data structure for concurrent BFS

BFS in our collaborative filtering system performs graph read operations most of time. However, several data elements in the vertex property are still volatile and will be updated during graph traversal, such as color, traversal depth and path number. Since graph data are shared by all threads, concurrent queries of different threads have data races over these volatile data. To solve this issue, instead of using locks of different granularities, we maintain an array of volatile data in vertex properties. The array size is the same

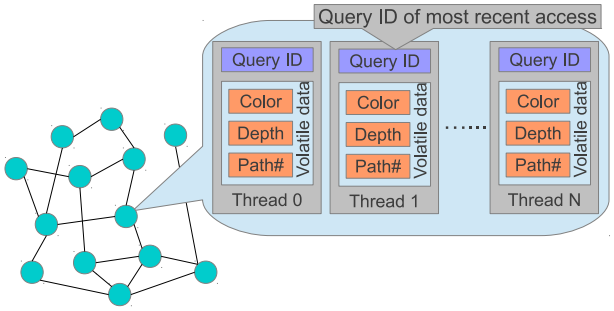

Figure 6: Lock-free data structure in each vertex

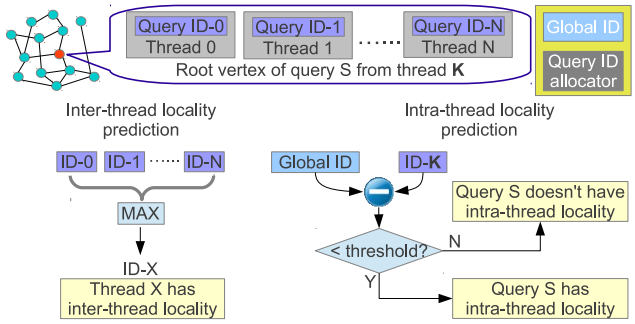

Figure 7: Illustration of data locality prediction

as the total number of threads. As shown in Figure 6, each thread will operate only on their own data segment in vertex properties. Data races are completely avoided without any lock.

Besides, queries in the same thread also have conflicts on volatile data. The color and depth value updated by the previous query need to be initialized again before the next query. However, this initialization operation is non-trivial and consumes significant amount of time. To minimize the overhead from intra-thread data races, we allocate a unique query ID to each query. An array of query ID records will be stored in the vertex property together with other volatile data. It is used to indicate the query ID of last access from current thread. The query ID allocation is achieved by performing a simple atomic fetch-and-add operation on a global ID variable. Hence, the query ID is globally unique and monotonic increased by time and the global ID variable indicates the number of processed queries.

3.4 Cache-conscious query scheduling

In order to maximize the benefits of intra- and inter-thread data locality and workload balancing, we propose a cache-conscious scheduling technique, in which queries are scheduled according to the data locality prediction results and balanced by the work-stealing mechanism. Details of our proposed technique are explained as following.

3.4.1 Data locality prediction

As a first step of our proposed scheduling policy, a low complexity locality prediction method is used. The method is based on a simple observation explained in Section 2 that neighbor queries have higher possibility of sharing data. Considering the fact that collaborative filtering performs only low-hop BFS, a new query A is the previous query B's neighbor if its root vertex was last accessed by query B. Therefore, queries' data locality can be reflected

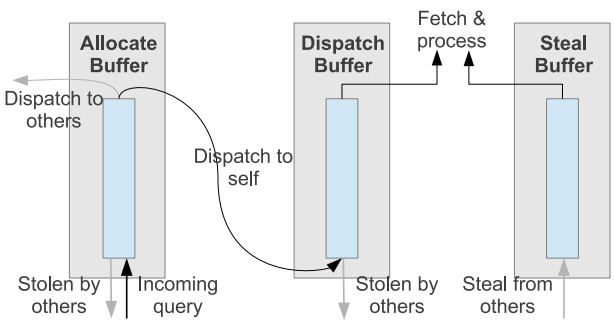

Figure 8: Structure of task buffers in each thread

by root vertices' access history. Such history record can be obtained from the query ID record in the vertex property.

As mentioned in the previous section, a query ID is allocated by the global ID variable and is increased over time. Each vertex has an array of query ID records corresponding to each thread. For a new query, if the query ID record of its root vertex shows that thread X has the largest query ID, it can be inferred that thread X recently just processed a query which is the neighbor of the current query. In this way, we predict data locality according to the root vertices' query ID record. The thread with the largest query ID is predicted to have inter-thread data locality with the current query. Meanwhile, by comparing global ID variable and query ID record of a certain query, we can get the information that how many queries have been processed after a neighbor query. If the difference is below the threshold G, we can predict that a neighbor was just processed and intra thread data locality exists.

An illustration example is shown in Figure 7. In this example, both inter- and intra-thread locality are predicted for a new query from thread K. The prediction is achieved by processing the query history information of its root vertex. In the inter-thread locality prediction, thread X (ID-X) has the largest ID among all query ID records. Thus, thread X is predicted to have inter-thread locality with current query. Based on our previous discussion in Section 2, current query should be processed by thread X. Meanwhile, in the intra-thread locality prediction in thread K, global ID variable is compared against ID-K. If the difference is below threshold, current query is predicted to have intra-thread locality in thread K and therefore should have higher priority to be processed.

3.4.2 Cache-conscious scheduling

As discussed in Section 2, both intra- and inter-thread locality should be considered in our cache-conscious scheduling technique. To achieve this target, we preform intra- and inter-thread scheduling based on the locality prediction results. As shown in Figure 8, three task buffers are established in each thread, allocate buffer, dispatch buffer, and steal buffer. They're all organized by insertion order in list data structures. The **steal buffer** is for the purpose of workload balancing and will be further explained in the next section. The previous two task buffers will be discussed in this section.

Whenever a new query is allocated for the current thread, the query is appended at the end of its **allocate buffer** for later scheduling. Meanwhile, a local thread always checks the **dispatch buffer** for queries to be processed. If the dispatch buffer is empty, D queries will be fetched from the allocate buffer for dispatch. The query dispatch procedure is performed according to the locality prediction outcome. As explained in previously, the thread with the largest query ID is predicted to have potential inter-thread locality with the current query. Therefore, it is inserted into the dispatch

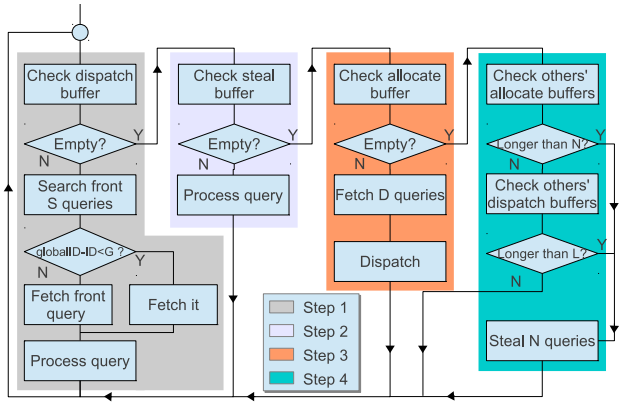

Figure 9: Complete algorithm in detail steps

buffer of the thread with a largest query ID. In this way, queries will always be dispatched into threads with the highest possibility of inter-thread data locality. Inter-thread data sharing as well as the communication overhead are avoided.

For queries inside a local thread's dispatch buffer, processing by insertion order obviously is not sufficient. Their processing order should be decided by considerations of intra-thread data locality. If a neighbor query is processed recently by the current thread, it should have higher priority to be processed earlier. Thus, when processing queries in the dispatch buffer, a local thread will check the oldest S queries for their query ID records. If the gap between their query ID records and global ID variable is within threshold G, the current query is predicted to have intra-thread data locality with recently processed query. Thus, this query is processed first. If no qualified queries is found, the original insertion order is used by fetching queries from the head of the buffer.

3.4.3 Workload Balancing

As discussed in Section 2, data locality intends to coalesce all queries into fewer threads. To fully utilize hardware resources, a workload balancing method is necessary. In our proposed implementation, a dynamic work-stealing technique is used.

A stealing operation is triggered when both the dispatch buffer and allocate buffer are empty. In that case, a local thread does not have any workload, neither queries to process or queries to dispatch. Instead of waiting for future incoming queries, the local thread first tries to steal N workloads from others' allocate buffer and appended them to its own local **steal buffer**. As queries pending for dispatch, stealing them brings only limited negative impact on data locality. These queries will later be processed regardless of their data locality. If no threads' allocate buffer has more than N queries, the dispatch buffers having greater than L elements are considered. Those threads usually have accessed more vertices and hence are getting even higher possibility to get new dispatched queries. Stealing queries from them can better balance each thread's size of accessed data. It will further result in balancing each thread's priority of inter-thread data locality. Meanwhile, stealing granularity N and threshold L can significantly affect stealing aggressiveness. These parameters need to chosen carefully for tradeoffs between cache-consciousness and workload balancing.

3.4.4 Full Algorithm/Implementation

As explained in previous sections, the complete collaborative filtering implementation is delineated in Figure 9. We now describe our mechanism in detail steps.

Table 1: Evaluation dataset

Feature	Description
Dataset	IBM Knowledge Repository graph dataset
Graph type	Bipartite graph: users and documents
Vertex	Users: 72.3K, Documents: 82.1K
Edge	User retrieves document: 1.74M edges
Query	Recommend N most relevant documents

1. *Intra-thread scheduling*: Check dispatch buffer. If it's empty, continue to step 2. Otherwise, search front S queries and process the query whose last processed query ID is within threshold G of current global ID. If such query cannot be found, query at the head of the buffer is fetched and processed. After processed this query, jump back to the starting point.

2. *Workload balancing*: Check steal buffer. If it's not empty, fetch head query and process it. Otherwise, continue to step 3.

3. *Inter-thread scheduling*: If allocate buffer is empty, continue to step 4. If not, fetch D queries and dispatch them according to locality prediction outcome. Locality prediction is achieved by checking root vertices' query ID array. The query will be dispatched to the thread with largest query ID. Then jump to step 1.

4. *Dynamic work-stealing*: Randomly check others' allocate buffers. If buffer size is longer than N, steal N queries from the allocate buffer and insert them into local steal buffer. If fail to find such buffer, check others' dispatch buffers instead. If success, steal N queries from the dispatch buffer. Then jump to step 1 again.

4. EVALUATION

4.1 Evaluation methodology

In our evaluation, experiments were performed on a Power7+ platform with four sockets. Each socket contains eight cores and four Simultaneous Multi-Threading (SMT) threads per core. In total, the platform has 32 cores and 128 hardware threads. The Power7+ processor executes instructions out-of-order and maintains 12 execution units shared by four SMT threads. Each core has 32KB L1, 256KB L2, and 4MB L3 private caches. To avoid the impact of OS thread scheduling, we bind threads with specific cores. To maximize utilization of hardware resources, threads are bound to different sockets first, then different cores. If more than 32 threads are required, SMT threads are utilized.

We ran our experiments on the IBM Knowledge Repository graph dataset from a document recommendation system used by IBM internally. In this dataset, two types of vertices, users and documents, form up a bipartite graph. As shown in Table 1, the graph contains 72.3K users, 82.1K documents, and 1.74M edges. The collaborative filtering query starts from a document and recommend N most relevant documents.

In the following sections, we evaluate our proposed technique in three types of experiments. First, a simple multi-threaded collaborative filtering is performed as a basic system. In the basic system, graph is represented as we proposed in Section 3 and workload is partitioned at the query granularity. However, queries are allocated randomly with only a simple round-robin scheduling. Then, we apply our inter-thread scheduling policy on it with work-stealing

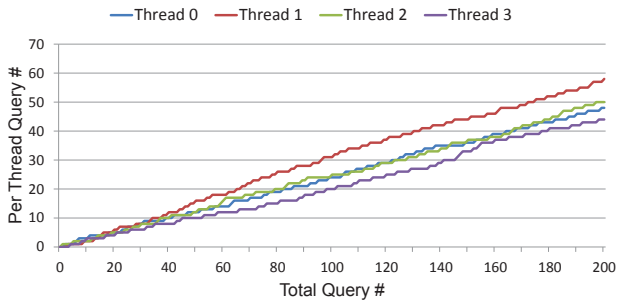

Figure 10: Query distribution with work-stealing

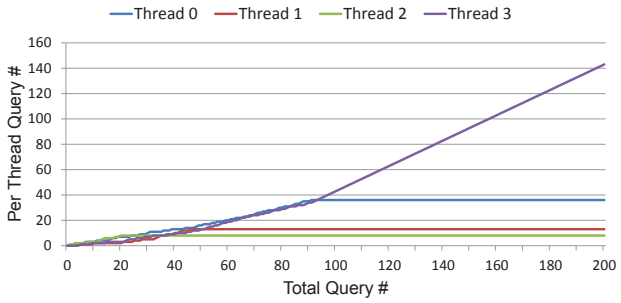

Figure 11: Query distribution without work-stealing

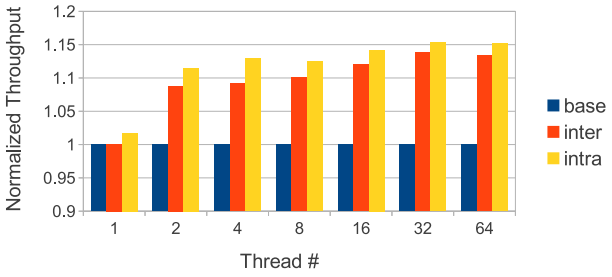

Figure 12: Normalized throughput

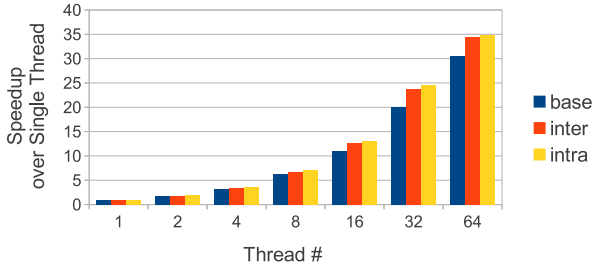

Figure 13: Speedup over single thread

enabled. After that, intra-thread scheduling is also introduced to work together with inter-thread scheduling. In each experiment, throughput, speedup, and scalability are analyzed in details.

4.2 Experimental Results

4.2.1 Necessity of work-stealing

In our proposed technique, inter-thread data locality is the major consideration when scheduling queries across threads. However, considering only inter-thread locality, it intends to combine queries into few threads and results in unbalanced workloads. To address this issue, a dynamical work-stealing method is used to ensure workload balancing among threads. In order to show the impact of our dynamic work-stealing mechanism, an experiment with four threads are performed. In the experiment, 200 queries are performed with our inter-thread scheduling technique and the number of processed queries over time for each thread is collected. As shown in Figure 11, without dynamic work-stealing, queries are well distributed over all threads at the early stage. However, when time goes by, it shows extremely unbalanced query distribution. Conversely, in Figure 10, queries are well balanced across all threads when dynamic work-stealing is enabled. Therefore, our dynamic work-stealing mechanism is enabled in all following experiments to avoid impact of unbalanced workloads.

4.2.2 Cache-conscious scheduling

In the experiments of this section, we randomly generate 32,768 queries. To better estimate system throughput, all queries are streamed into our collaborative filtering system in a burst way for the stress testing purpose. For the scalability analysis purpose, different numbers of threads were evaluated, starting from one to 64 threads. Since there are only 32 cores in our evaluation platform, we do not further perform experiments for 128 threads. Because the SMT threads share the same core's cache resources and therefore cannot well demonstrate the impact of cache related techniques. To simplify the notation, we denote the basic system, inter-thread scheduling, and intra-thread scheduling (with inter-thread scheduling) as base, inter, and intra.

The throughput results of three types of experiments are shown in Figure 12. The throughput number is calculated based on the average time per query. To better scale the results, they are all normalized to the basic system's throughput. From the results, we can see that the inter-thread scheduling alone can achieve up to 14% throughput improvement with 32 threads, while the intra-thread scheduling can further improvement that number to 16%. For the single thread, the inter-thread cannot bring any benefit and intra-thread shows limited benefit due to the constraint of the intra-thread search distance and a huge number of incoming queries. When the thread number is increased from two to 32, the throughput benefits of both inter- and intra-thread scheduling are increasing monotonically. That is because cross-core/socket communication overheads become more severe with more cores involved. However, for 64 threads, the improvement decreases slightly. It is in accordance with underlying hardware architecture. When more than one SMT thread are utilized in each core, the communication overhead still remains unchanged while the scheduling overhead increases with the number of threads.

The speedup results are shown in Figure 13. The overall processing time of all queries are measured and calculated as the speedup over a single-thread basic system. As shown, the speedup increases with the number of threads in all three types of experiments. Inter-thread scheduling achieves better performance than the basic system while an even better speedup is achieved with intra-thread scheduling. The detailed improvement of both inter- and intra-scheduling is illustrated in Figure 14, in which the speedup over the basic system is shown. As shown in Figure 14, the improvement of the intra-

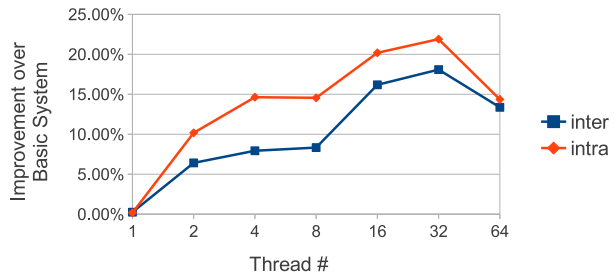

Figure 14: Improvement of inter- and intra-thread scheduling

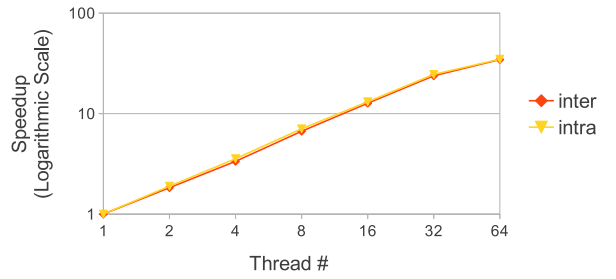

Figure 15: Scalability result

thread scheduling reaches as high as 22% for 32 threads. With 32 threads, the inter-thread scheduling also achieves the largest improvement of 18%. Besides, the improvement is increasing monotonically from 2 threads to 32 threads with a jump at 16 threads. This is because the improvement of the inter-thread scheduling is achieved by reducing communication traffic across cores. With more cores, the cross-core traffic overhead is more severe. Moreover, when core number is larger than 8, the cross-socket traffic is involved and brings much larger overhead. Nevertheless, when the number of threads is increased to 64, the improvement drops back to 14.3%. This is caused by the SMT threads in the same core. They share caches and interfere against each other. In this case, both our inter- and intra-thread scheduling becomes infeasible within the same core.

Figure 15 plots at a log scale to show the scalability of our proposed technique. As shown in the results, our methods are showing close to perfect scalability except for the case of 64 threads, in which the SMT threads over utilize the hardware resources and bring negative impact on the scalability. Besides, intra-thread scheduling further improves the scalability slightly.

4.2.3 Workload balancing

To further prove the effectiveness of our work-stealing method, we use the standard deviation and relative standard deviation of per thread processing time as the metric to measure workload balancing. We calculated the standard deviation from the processing time results of each thread in previous scalability experiments. The standard deviation results range from 1.6 to 3.1. Compared with the long processing time of each thread, it shows relatively small variance. Meanwhile, the relative standard deviation results are all below 10%. Such result indicates that our proposed mechanism reaches good workload balancing.

5. CONCLUSION AND DISCUSSION

In this paper, we studied high-throughput collaborative filtering on modern multi-socket multicore platforms. Our analysis and experiments showed that an individual query of collaborative filtering exhibits suboptimal data locality, but when multiple queries were processed, data locality was exhibited at both inter- and intra-thread levels. We also observed that the locality consideration usually contradicted the workload balancing requirements, which revealed a tradeoff between these two factors.

Based on the above observations, we presented a cache-conscious implementation for a collaborative filtering system. In this system, we first proposed an in-memory graph representation that achieved a balance between locality and flexibility and then presented a data locality prediction method based on the query ID record. With the data locality prediction results, we presented a cache-conscious scheduling technique that scheduled queries within and between threads in accordance with the intra- and inter-thread data locality, respectively. To balance the workload among the threads, we also introduced a dynamic work-stealing mechanism. By stealing dynamically, the mechanism rescheduled queries across threads, improving the overall workload balance.

Our study showed useful observations for the data locality behavior of graph algorithms. To improve throughput and scalability, it exploited the locality between queries. In the experiments on the Power7+ platform, our proposed technique demonstrated high throughput and good scalability and improved performance by as much as 22% over the basic system.

In the evaluation experiments, we used a Power7+ platform, which features private L3 caches. However, several other platforms such as Intel Xeon processors feature shared L3 caches. Compared to private caches, shared L3 caches lead to smaller communication overhead within the same chip. In this case, the impact of our proposed inter-thread scheduling technique is less significant when the number of threads is small. Nevertheless, because of the large overhead of cross-socket communication, the inter-thread scheduling technique still has significant impact as the number of threads increases.

Although our work focuses on system throughput, our solution can also be easily extended to address other scheduling schemes with minor changes. For example, to better support fairness requirements, we can add a simple timer mechanism that ensures maximum waiting time of each query or an epoch refresher that refreshes the system status before each epoch. Depending on specific datasets and query types, fairness requirements sometimes negatively impact system throughput. Throughput and fairness requirements pose an interesting tradeoff, which will be further analyzed in our future work. Moreover, in large-scale distributed systems, similar behaviors of data locality may exist at different scales. In our future work, we plan to also analyze inter-query locality behaviors in large-scale systems.

6. APPENDIX

6.1 Baseline Collaborative Filtering System

We use the following notations to briefly describe our document recommendation system which is essentially a collaborative filtering. Given a set of documents D and a set of users U, we build an edge between $d \in D$ and $u \in U$ as long as the document d is retrieved by user u. By normalizing the total number of retrievals of d with respect to u, we have an edge weight $w_{d,u} \in W$. Therefore, we have a weighted bipartite graph $G(D, U, E, W)$, where $E = \{(d, u) | d \in D, u \in U, w_{d,u} \neq 0\}$. Given the weighted

bipartite graph G and a root vertex $d_r \in D$, the collaborative filter returns a list of relevant documents $R \subset D$. Simply speaking, the relevance between document d_r and d' is measured by the total weight of edges in the paths from d_r to d', that is,

$$s_{d'} = \sum_{(v_1,v_2)\in \cup l_{d_r \to d'}} w_{v_1,v_2}, \quad \forall l_{d_r \to d'} \tag{6}$$

where $l_{r\to d'}$ represents a path from d_r to d'. Note that when the path length is 2 and the edges have equal weight, the relevance is equivalent to the number of users reading both documents. In practice, a collaborative filtering system must handle multiple queries simultaneously. Thus, there is a list of roots to process. The collaborative filter that schedules those queries out of order for improving the overall throughput is called *collective collaborative filter*. Note that the algorithm works on arbitrary graphs, although the application scenario requires them to be bipartite. Such algorithm is also internally used in IBM Knowledge Repository, a large scale enterprise-wide document database.

Algorithm 1 Collective graph collaborative filtering

Input: weighted bipartite graph $G(D, U, E, W)$, query root set Q, request capacity N
Output: result set $\mathcal{R} = \{R_r\}$

```
1: for all r ∈ Q do
2:    L_r = {r}, s_v^r = 0, n_r = 0, c_v^r = rand()
      {BFS-like weight propagation}
3:    while L_r ≠ ∅ and n < N do
4:       L'_r = ∅
5:       for v ∈ L_r do
6:          for v' ∈ Γ_v do
7:             if c_v'^r ≠ c_v^r then
8:                c_v'^r = c_v^r, L'_r ← v'
9:                s_v'^r = s_v^t + w_{v,v'}
10:            end if
11:         end for
12:      end for
13:      L_r = L'_r, R_r = R_r ∪ L'_r, n_r = n_r + |L'_r|
14:   end while
15: end for
```

Algorithm 1 is an approximate implementation of the above collaborative filter, which forms the foundation of our work in this paper. It works in batch mode by processing a set of queries, each starting from a document in set Q. For each query, the algorithm returns no less than N relevant documents, if they can be found. In a query rooted at r, we let s_v^r denote the relevance score which is initialized to 0. n_r counts the number relevant documents to return. c_v^r receives a random flag associated with a query from the random flag generator `rand()`, which indicates if a vertex has been visited in the query with the same flag. The use of a random flag is to spare flag cleaning for future traversal. Lines 3-14 are essentially a BFS-like traversal for propagating the score s_v^r to each vertex. In Line 8, we propagate the random flag and put the newly visited vertex into L'_r for the next iteration. Line 9 updates the relevance score according to Eq. 6. In Line 13, the parameters are updated for checking the termination status. Note that the scored vertices are ranked in some appliations, where the vertices closer to the root is ranked at a higher position; while the vertices at the same BFS level are ranked (sorted) according to their scores. We omit the ranking process in the algorithm, as it is quite trivial.

6.2 Background and related work

Collaborative filtering (CF) has been studied in a number of approaches and implemented with various focuses. For example, the item based collaborative filtering systems identify users with similar interest spaces or objects (say, documents) with similar characteristics. It assumes that the items favored by users with similar interest space is potentially recommendable to the new users with similar interest space. Such approach motives us to model users' interest spaces or characteristic clusters of objects.

An earlier implementation of collaborative filtering is Tapestry [2], which relies on the explicit opinions of people from a close-knit community. However, it is not practical in larger communities to assume each person knowing the others. A pseudonymous collaborative filtering solution for Usenet news and movies was proposed by GroupLens research system [16][17]. Ringo [3] and Video Recommender [4] generate recommendations on music and movies, respectively. More different collaborative filtering systems can be found in a special issue of the Communications of the ACM [18].

In addition to the item based approaches, Bayesian networks, clustering, and Horting are also used in recommendation systems. Bayesian networks create a model based on a training data set which can be built offline for hours or even days. The resulting model is small, fast, and essentially as accurate as the nearest neighbor methods [19]. Horting is based on graphs, where the nodes represent users and edges indicate similarity between two users [20]. Predictions are produced by walking the graph to nearby nodes and combining the opinions of the nearby users. In 2011 ACM KDD CUP contest, there is a collaborative filtering solution that addresses the unique item taxonomy characteristics and dataset volume. This proposed technique is implemented as part of GraphLab's collaborative filtering library [21]. All the above collaborative filtering implementations focus on providing rich functionality to data analytics, where the map of the algorithm onto particular processor/system architectures are not addressed properly.

In contrast to the existing work on collaborative filtering, we focus on the cache-consciousness issue of a straightforward item-based approach. It is performed towards a document recommendation system in the IBM Knowledge Repository. In this application, the system must concurrently handle a large number of queries in real time. To achieve the requirements of target system, we propose a cache-conscious graph collaborative filtering technique.

Data sharing in general applications is a common topic in various prior works. However, the special data representation and access pattern of graph operations contain unique features. Therefore, directly applying the solutions of general applications is typically infeasible in graph systems. In our work, we performed the first study of data locality behaviors in a collaborative filtering system. By making use of the data locality between inter- and intra-thread graph queries, our proposed technique achieves high throughput and good scalability.

7. REFERENCES

[1] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: item-to-item collaborative filtering," *Internet Computing, IEEE*, vol. 7, no. 1, pp. 76–80, 2003.

[2] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, "Using collaborative filtering to weave an information tapestry," *Commun. ACM*, vol. 35, no. 12, pp. 61–70, Dec. 1992.

[3] U. Shardanand and P. Maes, "Social information filtering: algorithms for automating word of mouth," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '95. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 210–217.

[4] W. Hill, L. Stead, M. Rosenstein, and G. Furnas, "Recommending and evaluating choices in a virtual community of use," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '95. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 194–201.

[5] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro, "Cache-oblivious priority queue and graph algorithm applications," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, ser. STOC '02. New York, NY, USA: ACM, 2002, pp. 268–276.

[6] R. E. Ladner, J. D. Fix, and A. LaMarca, "Cache performance analysis of traversals and random accesses," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '99. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999, pp. 613–622.

[7] V. K. Pingali, S. A. McKee, W. C. Hseih, and J. B. Carter, "Computation regrouping: Restructuring programs for temporal data cache locality," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002, pp. 252–261.

[8] G. Cong and S. Sbaraglia, "A study on the locality behavior of minimum spanning tree algorithms," in *Proceedings of the 13th International Conference on High Performance Computing*, ser. HiPC'06. Berlin, Heidelberg: Springer-Verlag, 2006.

[9] D. A. Bader, G. Cong, and J. Feo, "On the architectural requirements for efficient execution of graph algorithms," in *Proceedings of the 2005 International Conference on Parallel Processing*, ser. ICPP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 547–556.

[10] L. Yuan, C. Ding, D. Tefankovic, and Y. Zhang, "Modeling the locality in graph traversals," in *Parallel Processing (ICPP), 2012 41st International Conference on*, 2012, pp. 138–147.

[11] S. Krishnamoorthy, U. Catalyurek, J. Nieplocha, and P. Sadayappan, "An approach to locality-conscious load balancing and transparent memory hierarchy management with a global-address-space parallel programming model," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, pp. 8 pp.–.

[12] D. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *Parallel Processing, 2006. ICPP 2006. International Conference on*, 2006, pp. 523–530.

[13] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 2005, pp. 25–25.

[14] B. Derbel and M. Mosbah, "Distributed graph traversals by relabelling systems with applications."

[15] B.-Y. Su, T. Brutch, and K. Keutzer, "Parallel bfs graph traversal on images using structured grid," in *Image Processing (ICIP), 2010 17th IEEE International Conference on*, 2010.

[16] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl, "Grouplens: applying collaborative filtering to usenet news," *Commun. ACM*, vol. 40, no. 3, pp. 77–87, Mar. 1997.

[17] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "Grouplens: an open architecture for collaborative filtering of netnews," in *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, ser. CSCW '94. New York, NY, USA: ACM, 1994, pp. 175–186.

[18] P. Resnick and H. R. Varian, "Recommender systems," *Commun. ACM*, vol. 40, no. 3, pp. 56–58, Mar. 1997.

[19] J. S. Breese, D. Heckerman, and C. Kadie, "Empirical analysis of predictive algorithms for collaborative filtering," in *Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence*, ser. UAI'98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 43–52.

[20] C. C. Aggarwal, J. L. Wolf, K.-L. Wu, and P. S. Yu, "Horting hatches an egg: a new graph-theoretic approach to collaborative filtering," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '99. New York, NY, USA: ACM, 1999, pp. 201–212.

[21] Y. Wu, Q. Yan, D. Bickson, Y. Low, and Q. Yang, "Efficient multicore collaborative filtering," in *ACM KDD CUP workshop*, 2011.