Efficient and Simplified Parallel Graph Processing over CPU and MIC

Conference Paper · May 2015

DOI: 10.1109/IPDPS.2015.88

5 authors, including:

Linchuan Chen
The Ohio State University
9 PUBLICATIONS **54** CITATIONS
SEE PROFILE

Bin Ren
College of William and Mary
10 PUBLICATIONS **15** CITATIONS
SEE PROFILE

Gagan Agrawal
The Ohio State University
157 PUBLICATIONS **1,933** CITATIONS
SEE PROFILE

Efficient and Simplified Parallel Graph Processing over CPU and MIC

Linchuan Chen Xin Huo Bin Ren Surabhi Jain Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{chenlinc,huox,ren,jainsu,agrawal}@cse.ohio-state.edu

Abstract—**Intel Xeon Phi (MIC architecture) is a relatively new accelerator chip, which combines large-scale shared memory parallelism with wide SIMD lanes. Mapping applications on a node with such an architecture to achieve high parallel efficiency is a major challenge. In this paper, we focus on developing a system for heterogeneous graph processing, which is able to utilize both a many-core Xeon Phi and a multi-core CPU on one node. We propose a simple programming API with an intuitive interface for expressing SIMD parallelism. We develop efficient techniques for supporting our high-level API, focusing on exploiting wide SIMD lanes, massive number of cores, and partitioning of the work across CPU and accelerator, while handling the irregularity of graph applications. The components of our runtime system include a *condensed static memory buffer*, which supports efficient message insertion and SIMD message reduction while keeping memory requirements low, and specifically for MIC, a *pipelining scheme* for efficient message generation by avoiding frequent locking operations. Besides, a *hybrid* graph partitioning module is able to effectively partition the workload between the CPU and the MIC, ensuring balanced workload and low communication overhead. The main observations from our experimental evaluation using five popular applications are: for MIC executions, pipelining scheme is up to 3.36x faster than a naive approach using *locking based* message generation, and the speedup over OpenMP ranges from 1.17 to 4.15. Heterogeneous CPU-MIC execution achieves a speedup of up to 1.41 over the better of the CPU-only and MIC-only executions.**

I. Introduction

The motivation of our work arises from the emergence or greater applicability of a set of algorithms broadly referred to as *graph mining* [6], [31], [8], [37], [35], [14], [11]. Many challenging real-world scenarios can be modeled as graphs [4], [20], [30], [34] and graph analysis or mining algorithms can help solve key problems. However, the resulting graphs can often be very large, especially the ones derived from the World Wide Web or those representing *social networks*, such as the ones created by friendship links on Facebook. Because of the size of the graphs, it is natural to use parallel implementations to solve the problems. Recently, there has been much interest in the scalable graph mining. The work at CMU has developed a graph mining package based on the Hadoop implementation of MapReduce [15], whereas Google developed a more specialized API, called Pregel [26] for creating implementations of graph mining problems.

Over the last 6-7 years, high-end computing systems have changed significantly with respect to the *intra-node* architectures, with popularity of coprocessors. Over the last 3 years, as many as three of the five fastest supercomputers (at any time, based on the bi-annual top 500 list [1]) in the world involved coprocessors on each node, as they offer excellent performance-price and performance-power ratios. A recent development along the same lines has been the emergence of Xeon Phi chips, based on the Intel MIC architecture, which allows x86 compatible software to be used. More broadly, a node with a Xeon Phi represents many characteristics of a processing node that we can expect to see increasingly in the future – heterogeneity, large number of cores, combination of MIMD and SIMD parallelism, and only a small amount of memory per core. At the same time, our target class of applications, i.e., the graph processing algorithms, represent a class of irregular applications, for which extracting either large-scale shared memory parallelism or SIMD parallelism has been very hard.

Overall, effectively exploiting the power of a coprocessor like Xeon Phi requires that we exploit both MIMD and SIMD parallelism. While the former can be done through Pthreads or OpenMP, it is much harder to extract SIMD performance. This is because the restrictions on the model make hand-parallelization very hard. Particularly, in a Xeon Phi, the SIMD width has been extended to 512 bits (16 floats), potentially offering large benefits for applications. Even for MIMD parallelism, load balanced execution with a large number of cores and limited memory is challenging, especially for irregular applications. Yet another challenge is heterogeneity – a Xeon Phi coprocessor is connected to a multi-core CPU through PCIe bus. Compared with a Xeon Phi, the CPU has fewer cores, but better sequential performance on each core. Thus, it is important to utilize both of these devices for computation. This, however, leads to the challenge of partitioning the workload and minimizing inter-device sychronization and communication.

This paper describes a system for graph processing utilizing both the CPU and the MIC chip. We support a vertex-centric high-level API that can express a graph application easily. The system enables the users to write portable code that is simultaneously executed on both a Xeon Phi and a multi-core CPU. Overall, we address the following four challenges in our work: 1) Minimizing random memory accesses and exploiting the wide SIMD lanes: this is achieved through an innovative design of the *message buffer*, 2) Reducing contention overhead from concurrent vertex update: we create a pipelining implementation for the message generation step, which is suitable for operating with a large number of threads. 3) Load balancing among the large number of cores: we support a novel dynamic load balancing scheme for execution within a device. 4) Workload partitioning between CPU and MIC: we develop an effective *hybrid* graph partitioning scheme which achieves balanced workload and minimized communication. Even though the first three challenges arise from the MIC execution, and the optimizations are specifically designed considering its particular properties, the same code and optimizations are used for CPU execution.

We have evaluated our framework using five popular graph algorithms. For the MIC-only executions, the *pipelining scheme* we have introduced outperforms a naive *locking based* approach by 1.07x to 3.36x, and our approach for exploiting SIMD lanes delivers a speedup of between 5.16x and 7.85x for the message processing sub-step, and results in 1.18x to 1.23x performance improvement for the overall execution. Our system also outperforms OpenMP by up to 4.15x. Heterogeneous execution using our framework delivers a speedup of up to 1.41 over the better of the single device executions.

II. Background

A. Intel Xeon Phi Architecture

The x86-compatible Intel Xeon Phi coprocessor, which is a latest commercial release of the Intel Many Integrated Core (MIC) architecture, has already been incorporated in 9 of the top 100 supercomputers at the time of writing this paper [1]. MIC is designed to leverage existing x86 experience and benefit from traditional multi-core parallelization programming models, libraries, and tools.

In the available MIC systems, there are 60 or 61 x86 cores organized with shared memory. These cores are low frequency in-order ones, and each supports as many as 4 hardware threads. Additionally, there are 32 512-bit vector registers on each core for SIMD operations. The main memory sizes vary from 8 GB to 16 GB, and the memory is logically shared by all cores. Even though, the memory space is physically composed of separate GDDR5 memory channels, offering a very high parallel memory access bandwidth. The L1 cache is 32 KB, entirely local to each core, whereas each core has a coherent L2 cache, 512 KB, where cache for different cores are interconnected in a ring.

Our work focuses on three important features of Intel MIC architecture, which need to be exploited for obtaining high performance:

Wide SIMD Registers and Vector Processing Units (VPU): VPU has been treated as the most significant feature of Xeon Phi by many previous studies [24], [39], [29], [9]. The reason is that the Intel Xeon Phi coprocessor has doubled the SIMD lane width compared to Intel Xeon processor, i.e., 256-bit to 512-bit, which means that it is possible to process 16 (8) identical floating point (double precision) operations at the same time. In addition, we have a new 512-bit SIMD instruction set called Intel Initial Many Core Instructions (Intel IMCI), which has built-in *gather* and *scatter* operations that allow irregular memory accesses, a hardware supported *mask* data type, and *write-mask* operations that allow operating on some specific elements within the same SIMD register.

The SIMD instructions can be generated by the ICC compiler through the *auto-vectorization* option, or the programmers could use IMCI instruction set directly. The former needs low programming effort, though current compilation systems have several limitations and do not always obtain high performance. In comparison, the latter option can achieve the best performance, however, is tedious and error prone, and creates non-portable code.

Large Number of Concurrent Threads: Each Xeon Phi core allows up to 4 hyper-threads, in another word, we can have as many as 240/244 hardware threads sharing the same memory on Xeon Phi. This provides us with massive Multiple Instruction Multiple Data (MIMD) parallelism with shared memory, which has not been common in the past.

Source Code Portability with CPU: Similar to modern CPUs, a Xeon Phi coprocessor integrates x86 cores. This enables the same source code to be run on both a CPU and a Xeon Phi.

B. Graph Processing Algorithms

Though graph algorithms have been widely used, and for a long time, with rapid development of social network websites, graphs have become the most important data structure for representing relationships among people or other entities for social network websites such as Facebook, Twitter, Wikipedia, and others. A variety of graph mining algorithms have been proposed for discovering the relationships among people for social network graphs [4], [38], [21].

The most commonly used graph representations for storing a graph are the *adjacency matrices* or *adjacency lists*. *Compressed Sparse Row (CSR)* format is a well known format for efficient storage of sparse matrices, which is also widely used for storing sparse graphs. Figure 1 shows an example graph and its storage in CSR format (the associated vertex data or edge data are not shown).

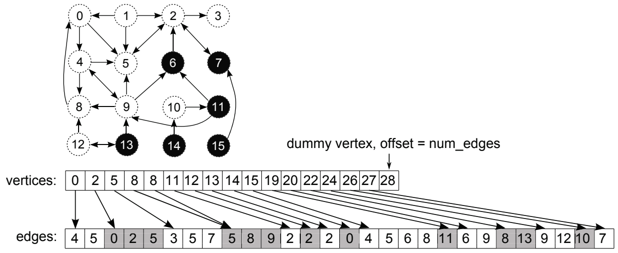


Fig. 1: An Example Graph and Its Representation in CSR Format

Despite a variation in the underlying logic across different graph algorithms, a significant fraction of graph algorithms share certain similarities. First, most graph algorithms are *vertex-centric*, i.e., the values and the processing are centered around vertices. These algorithms usually gradually update the values associated with vertices, using edges as the auxiliary information to perform the computations. The output of these algorithms is typically a set of vertex values. Second, edges are usually used for communication among vertices. These common characteristics have inspired the design of many graph processing programming models that have targeted different hardware platforms, including Pregel [26], GPS [32], Giraph [2], GraphLab [25] and Medusa [43], and also form the basis for the work presented in this paper.

III. Programming API

We base the programming API of our framework on the BSP model [36]. We treat a graph application as an iterative process, and each iteration is divided into three important steps: *message generation*, *message processing* and *vertex updating*, with synchronization between these steps. To use an example – *Single Source Shortest Paths Problem* (SSSP) aims to find a shortest path between a single source vertex and every other vertex in the graph. The algorithm we introduce here is applied to a positive weighted directed graph. The brief steps of the SSSP algorithm are: 1) *initialization*: for each vertex, an attribute *distance* representing the possible minimum distance from the source is maintained and initialized to a large constant, and the distance value for the source vertex is initialized to 0, 2) *relaxation*: for every vertex u, and for every incoming edge e ($e = (v, u)$), if $v.distance + e.weight < u.distance$, then update $u.distance$ to $v.distance + e.weight$, and 3) the above two steps are repeated until no more updates occur.

This algorithm can be easily expressed by a programming model with an API for message generation, message processing, and vertex updating. After the initialization step, the relaxation step could be expressed as a combination of these three steps. In each iteration, each vertex receives messages from its incoming neighbors, which are the updated minimum distances from the source. A minimum value of the values contained in these messages can then be calculated in (*message processing*). If this vertex has a new minimum potential distance value as a result of this calculation, it updates its distance with this new minimum value (*vertex updating*) and propagates this update through messages to its neighbors along the outgoing edges (*message generation*). The same procedure is repeated in the future iterations. Similar steps are used in Pregel [26], a distributed memory graph programming model, but there are also key differences considering the focus on shared memory and SIMD processing.

We explain our API using user-defined functions for SSSP. In addition to three main functions that are supported, an important characteristic of our API is simplified support for exploiting SIMD parallelism. This will be explained at the end of this section.

Listing 1: User-defined functions for SSSP

```
User-defined function for message generation:
template <class VertexValue,
          class EdgeValue,
          class MessageValue>
void generate_messages(size_t vertex_id,
        graph<VertexValue, EdgeValue> *g) {
    float my_dist = g->vertex_value[vertex_id];
    // Graph is in CSR format.
    for (size_t i = g->vertices[vertex_id];
        i < g->vertices[vertex_id + 1]; i++) {
        send_messages<MessageValue>(g->edges[i],
        my_dist + g->edge_value[i]);
    }
}
User-defined function for message processing:
template <class MessageValue>
void process_messages(vmsg_array<MessageValue> &vmsgs){
    // Reduce the vector messages to vmsgs[0].
    vfloat res = vmsgs[0];
    for (int i = 1; i < vmsgs.size(); ++i) {
        res = min(res, vmsgs[i]);
    }
    vmsgs[0] = res;
}
User-defined function for vertex updating:
template <class VertexValue,
          class EdgeValue,
          class MessageValue>
void update_vertex(MessageValue &msg,
    graph<VertexValue, EdgeValue> *g, size_t vertex_id){
    // Distance changed, will send msgs.
    if (msg < g->vertex_value[vertex_id]) {
        g->vertex_value[vertex_id] = msg;
        g->active[vertex_id] = 1;
    } else {
        // Distance not changed, no msgs will be sent.
        g->active[vertex_id] = 0;
    }
}
```

Basic API: The user-defined functions for SSSP is shown in Listing 1. The three key functions are as follows. *generate_messages()*, defines the way a certain vertex generates messages. The runtime system invokes this function for a vertex only when it is active (only the source vertex is active in the initial iteration). For an active vertex in this example, the function sends a message to every vertex it connects to, and specifically, it propagates its minimum distance from the source plus the weight of each edge.

process_messages() is invoked for a vertex (or simultaneously for multiple vertices) to process the received messages coming from other neighbors: in this particular example, it calculates the minimum value among all the messages that are received.

The last of the three functions, *update_vertex()* is invoked to update the value or the status of a particular vertex, typically as a result of processing of received message(s). Specifically, the parameter `msg` of function *update_vertex()* is the processing result from function *process_messages()* for the messages received by this vertex. In the example in Listing 1, the function compares the minimum distances received in the messages with its own distance from the source. If the received minimum distance is shorter, it updates its own value to the smaller value, and at the same time, it sets the status of vertex to *active*. Otherwise, this vertex will be *inactive*, since it is not updated in the current iteration. An inactive vertex may not participate in the message generation for next step.

Portable API for Exploiting SIMD Parallelism: Our framework provides a set of SSE primitives for automatic SIMD processing within each thread, which help to utilize the wide SIMD lanes. The core units of our SSE primitives are the set of *vector types (vtypes)*. The vtypes we currently support are `vint`, `vfloat` and `vdouble`. In contrast to scalar data types, each vtype contains a group of contiguous data elements. A set of common arithmetic/logical as well as assignment operations are *overloaded* to these vtypes, so that operations involving vector types, or vector type and scalar types could be easily supported without requiring users to perform complicated SSE intrinsics programming. These overloaded functions perform vectorization by invoking the built-in SIMD intrinsics. These functions are portable between MIC and CPU, that is, the same APIs are built on top of both KNC (for MIC), and SSE4.2 (for CPU), wrapping corresponding architecture-specific intrinsics. As we will explain later, our runtime system organizes received messages for each vertex in an aligned manner within a buffer, which makes SIMD processing of the messages possible.

Referring to the example in Listing 1, since the message type is `float`, the runtime passes the aligned messages as floating point vector arrays into the user-defined function. Users just need to use `vfloat` variables to process the messages in the SIMD fashion, in which case messages for up to 16 vertices (for MIC) are processed simultaneously. As we can see, the code is very similar to the serial code. It should be noted that SIMD processing of messages only applies to messages with basic data types that are supported by SSE, such as `int`, `float` and `double`, and are limited to associative and commutative reductions, such as sum, max, or min. However, such operations are very common in most graph applications. In the example in Listing 1, the user-defined function computes the minimum values among the received messages for up to 16 vertices (for MIC), and stores the results as the first element of the vector array `vmsgs`.

IV. Framework Implementation

This section describes the detailed design strategies, including data structures, algorithms, and optimizations that are applied.

The system has been designed to address the key features of the MIC architecture, as described in Section II and the nature of the graph applications. As the number of edges connecting a node can vary tremendously, we have difficulty in obtaining load balance and avoiding contention for memory accesses (both of these challenges accentuated due to the large number of cores sharing the same memory). Exploiting SIMD parallelism and dividing the work among the MIC and the CPU are other challenges.

A. System Components and Workflow

We first introduce the components and the workflow of our system, including showing how a graph application is constructed and processed using our framework.

The overall structure of an application built using our system is shown in Figure 2. Besides writing user-defined functions, which have been introduced in the previous section, users must write a driver code to read the input (with the help of distributed graph loading API), and to help drive the parameters such as the maximum number of iterations. The input to the system consists of two files: the graph file stored in an adjacency list format, and a graph partitioning file indicating

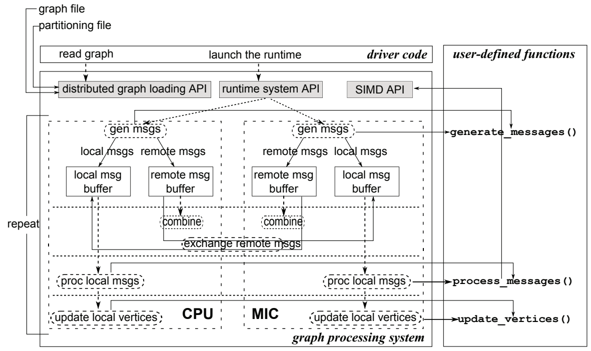


Fig. 2: System Components

which device each vertex belongs to. A separate module for generating the graph partitioning file will be described later in Section IV-E. The user-defined functions, in turn, are invoked by the runtime system. Symmetric runtime instances on the CPU and the Xeon Phi share the same source code and thus the same structure, though parameters such as numbers of threads running on each device are separately configured. The system is built using MPI symmetric computing, with CPU being Rank 0, and MIC being Rank 1. Multi-threading is used on each device.

On each device, a same set of steps are executed iteratively. In the first step, *message generation*, messages are generated into message buffers (either local message buffer or remote message buffer, depending on the message destination) through the system primitive `send_messages()`, called in the user function `generate_messages()`. A message is a data unit containing a value pair, in the form of $\langle$*dst_id, msg_value*$\rangle$.

Before the *message processing* step, an implicit *remote message exchange* step is performed between devices. To reduce the communication overhead, a *combination* is conducted to the remote message buffer. The combination result is sent to the other device as a single MPI message. Runtime system invokes the user-defined function `process_messages` for message combination. Received messages are inserted into local message buffer for further processing.

Local message buffer is then accessed by the *message processing* step to conduct either SIMD message reduction or scalar processing. It is also accessed to update the local vertex values in the *vertex updating* step. Thus, the message buffer is the core data structure in the entire system. The design of this data structure impacts the performance of all the major steps, and thus, the performance of the entire system.

B. Message Buffer Design

To utilize the SIMD lanes (especially the wide lanes on Xeon Phi) – more specifically – to support SIMD reduction of messages, we should organize the messages in a way that the memory space is aligned, and that messages for the same destination are loaded into the same lanes while being reduced. As a secondary consideration, our design must be cognizant of the memory limits, i.e., that the memory size on a MIC chip is only a few GBs. Suppose the width of SIMD lanes is w bytes, and the size of an element to be processed in the message is msg_size. To support SIMD processing, we should wrap w/msg_size messages together in a way that they are aligned with a multiple of w bytes. The runtime system stores the messages as *aligned vector types*, including `vint`, `vfloat`, and `vdouble`, where exactly w/msg_size scalar data elements are stored in an aligned fashion, same as the *vtypes* mentioned in Section III.

We pre-allocate the buffer in a *condensed static* fashion, before any iteration is executed. An example of this buffer is shown in Figure 3, where a buffer is created for the example graph in Figure 1. The *condensed static buffer* (CSB) is created for an input graph in the following steps: 1) Sort the vertices according to the in-degree in descending order. A *redirection map* is generated for redirecting messages from original *dst_ids* to the proper positions after sorted. The redirection map will be used in message insertion, to be stated later in this section. 2) Group the sorted vertices into *vertex groups*. Each vertex group contains $k \times w/msg\_size$ vertices, where k is a small constant and w is the SIMD lane width in bytes. In the example in Figure 3, k equals 2. 3) Obtain the maximum in-degree among the vertices in each vertex group (denoted as max_group_degree). Allocate k *aligned vector* arrays for each vertex group with an array size of max_group_degree.

Source	Messages (dst_id, value)
6	(2, value)
7	(2, value)
11	(6, value), (9, value)
13	(9, value), (12, value)
14	(10, value)
15	(7, value)

TABLE I: Messages Being Sent in the Example Graph

We use the directed graph shown in Figure 1 to show how the CSB shown in Figure 3 is constructed. The graph contains 16 vertices, with the maximum in-degree being 5 for Vertex 5, and minimum in-degree being 0 for Vertices 14 and 15. The vertices are sorted according to in-degrees in descending order, and the sorting result is shown in the table in figure 3. For simplicity, we assume the SIMD lane to be as wide as 4 messages ($w/msg\_size = 4$), and that we take the value of k as 2. Thus, we are combining 8 vertices into a same *vertex group*, resulting in two vertex groups in total. For the first vertex group, the maximum in-degree among all the vertices is 5, and for the second vertex group, the maximum in-degree is 1. For the first vertex group, 2 arrays of *aligned vector* type are created, with the length of each being 5. Similarly, for the second vertex group, 2 *aligned vector* arrays are created, with the length being 1.

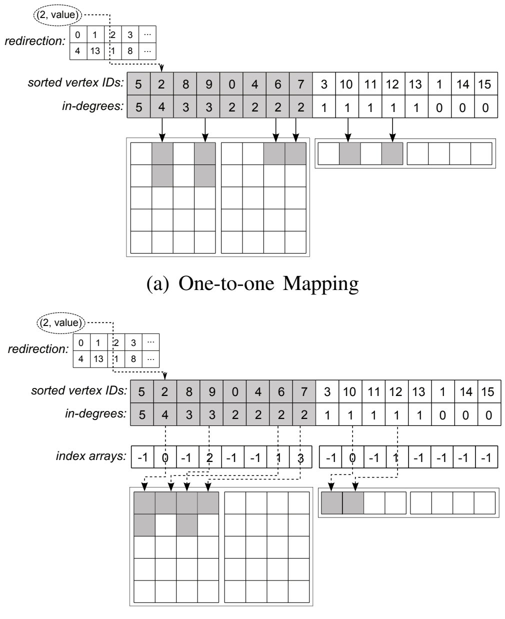
Such a buffer design significantly reduces the memory requirement because we group vertices with similar in-degrees together and avoid unnecessary memory consumption for small in-degree vertices. At the same time, this method allows likely use of SIMD lanes for processing one message each for a set of vertices (i.e., w/msg_size vertices).

C. Processing of Messages Using CSB

Despite the advantages in memory efficiency, there are certain challenges in processing messages using the CSB data structure we have introduced. Again, we use the example graph in Figure 1 and one iteration in the SSSP algorithm to show what challenges are faced, as well as how we address them.

Suppose at a certain iteration, vertices 6, 7, 11, 13, 14, 15 (shown in black) just updated their shortest distances from the source using the incoming messages from the previous iteration. Thus, these vertices are active in this iteration and are sending the updated values out to their neighbors in the current iteration. The messages that are being sent are shown in Table I. These messages are generated and sent in the *message generation step* of our system. Messages that are sent out from vertices are to be inserted into the message buffer.

Vertex-Column Mapping: The first problem is of mapping of vertices belonging to a vertex group to the columns of the vector arrays. The simplest method will be to use a predetermined mapping (e.g., direct one-to-one mapping) between the vertices and the columns in the vector arrays. In this example, among all the vertices, only Vertices 2, 6, 7, 9, 10, 12 have incoming messages. Whenever a message arrives, its destination vertex ID is translated to a position in the sorted table through the indirection map. The value of the message


(2, value)
redirection:
0 1 2 3 ···
4 13 1 8 ···
sorted vertex IDs: 5 2 8 9 0 4 6 7 3 10 11 12 13 1 14 15
in-degrees: 5 4 3 3 2 2 2 2 1 1 1 1 1 0 0 0
index arrays: -1 0 -1 2 -1 -1 1 3 -1 0 -1 1 -1 -1 -1 -1

(a) One-to-one Mapping

(b) Dynamic Column Allocation

Fig. 3: Condensed Static Buffer (CSB) and Message Insertion Methods

is inserted to the column in that position. After inserting the messages in Table I to the CSB, we get an insertion result that is shown in Figure 3(a): dark blocks in the buffer are the received messages, and others are empty. We can see that because a number of vertices in each vertex group do not receive any message, we are wasting SIMD lanes while conducting message reductions.

To solve this problem, we utilize a *dynamic column allocation*, which helps to condense the usage of columns. Facilitating such an implementation are an *index array*, and a *column offset*, which are maintained for every vertex group. Each element in the index array corresponds to a vertex and indicates the column index for the messages with this vertex as the destination. Before the message generation step in every iteration, all elements in an index array are initialized to -1, and the column offset is initialized to 0. Whenever a thread wants to insert a new message, according to the message's destination vertex ID, the thread checks the index array for that vertex. If the column index is not -1, the thread inserts the message to that column. Otherwise, it allocates the next available column from that vertex group, using locking in the process, by exclusively incrementing the column index and writing the allocated column index to the corresponding element in the index array. After all messages have been inserted within each vertex group, it is possible that only a subset of vector arrays in the front contain messages, while the remaining are completely empty. This allows for greater efficiency in using SIMD lanes. Recall that the width of a vertex group is k times the SIMD width, so we have the possibility that i ($i < k$) loop(s) of instructions may process all the vertices in the vertex-group. Figure 3(b) shows the result of message insertion using *dynamic column allocation* for the example.

SIMD Message Reduction: Now, let us see how messages from the CSB are reduced using SIMD parallelism. Iteratively, the runtime invokes the user-defined function `process_messages()` to process one *aligned vector* array from the CSB. Take the `process_messages()` function in Listing 1 as an example: users iterate on the `vfloat` type message array, and each time process a row of w/msg_size messages from the vector array. For MIC, because the SIMD lane width is 64 bytes and the messages being processed are of type `float`, simultaneously 16 messages participate in the overloaded `min()` function, which wraps the SSE intrinsic `_mm512_min_ps` for MIC. For CPU, 4 messages are processed simultaneously. Similarly, other arithmetic operations (e.g., $+$, $-$, $\times$, $\div$, etc.) also wrap the corresponding intrinsics to process w/msg_size messages in one invocation.

Message Insertion to Columns: Though the message buffer we designed makes it easy to exploit SIMD message processing and avoids excessive memory consumption, there are still other challenges to be addressed. An important question to be answered is how messages are to be inserted to the columns in the presence of parallelism. An intuitive approach is to use a *locking based* method – every thread inserts the messages it generates into the message buffer directly. However, because different vertices may send messages to same destinations (e.g. both Vertex 11 and Vertex 13 send messages to the Vertex 9 in Table I), and because different vertices are distributed to different threads, concurrent message insertions to the same buffer column is going to be common. To ensure correctness, locking operations must be used, i.e., whenever a message is being inserted to a message buffer column, the computing thread should lock the entire column. Such locking will be required frequently, and moreover, could lead to contention among the threads.

Based on this concern, we have implemented a *pipelining scheme* for message generation. In this scheme, we divide the computation threads into *worker threads* and *mover threads*. Worker threads are only responsible for computation and message generation. They do not insert messages into message buffers, instead, they temporarily store the messages in message queues, working sequentially. Mover threads are responsible for moving the messages from the message queues into appropriate locations in the message buffer.

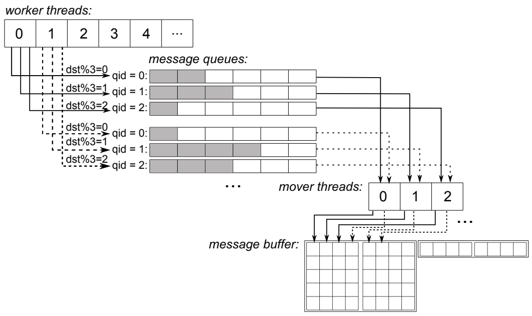


Fig. 4: Message Generation Pipelining Using Worker and Mover Threads

The example shown in Figure 4 uses 3 threads as mover threads. Each worker thread maintains private message queues, with the number of these queues being equal to the number of mover threads. After generating a message, the worker thread decides which message queue to insert it to, which is done based on the destination vertex ID using the expression: $queue\_id = dst\_id \bmod num\_mover\_threads$. A specific mover thread tid iterates over all worker threads' message queues whose $queue\_id$ equals tid, and moves the messages from these queues.

This strategy guarantees that each message queue is only written by only one thread, as well as read by only one thread. Similarly, a message buffer column can only be accessed by one mover thread. This is because messages are classified based on the modulo operation on the destination vertex IDs, and thus each message class (generated as a result of this classification) is handled by only one mover thread. Thus, a mover thread needs to use locking only at the time of buffer column allocation. Also, since the worker threads and mover threads work concurrently, the computation and memory accesses are overlapped effectively if the number of mover threads is chosen appropriately.

D. Intra-device Load Balancing

Because the numbers of incoming and outgoing edges to/from different vertices tend to vary, the amounts of processing associated with different vertices is different. Thus, if we simply evenly distribute the workload based on the number of vertices, load imbalance will likely occur.

We utilize a dynamic workload scheduling for all sub-steps during the execution. For the *message processing* step, we treat the set of all *vector arrays* from all *vertex groups* as *task units*. All threads dynamically retrieve these task units through a mutex-protected scheduling offset. To lower the task retrieving frequency and thus the scheduling overhead, a thread can obtain multiple tasks each time. Although the numbers of messages contained in different *vector arrays* can vary significantly, dynamic scheduling of these message processing tasks ensures that all threads are kept busy until no more tasks are available. Similarly, for the *vertex updating* step, vertices are also dynamically scheduled to all the threads in blocks. Finally, for the *message generation* step, vertices are dynamically scheduled to the *worker threads* for computation and message generation. For the mover threads, because we classify messages based on the modulo operation of the destination IDs, each message class tends to have a similar number of messages. Although each message class is statically mapped to one mover thread, workload distribution among mover threads is expected to be well balanced.

E. Graph Partitioning between CPU and MIC

Our framework is able to execute a graph application across CPU and MIC. An important issue in enabling this functionality is to partition the workload between CPU and MIC. Because a CPU-MIC node is on top of a distributed memory space, it is not feasible to utilize dynamic workload distribution, since it will lead to high data movement costs. Thus, the framework statically distributes vertices to devices, before an application is run, using a *partitioning ratio* (relative amounts of computation assigned to devices) specified by the users.

We now discuss how partitioning is performed by the system. The two desired properties from the partitioning method are: 1) load balance – suppose the expected workload ratio between the CPU and the MIC is $a : b$, $edges\_CPU : edges\_MIC$ should be close to $a : b$, where $edges\_CPU$ and $edges\_MIC$ are the number of edges processed by the CPU and MIC, respectively. 2) minimized communication volume: cross edges, i.e., the edges whose source and destination are on different devices should be as few as possible.

We now examine different partitioning methods and introduce our proposed *hybrid* partitioning method. An intuitive way of partitioning a graph is *continuous partitioning*: suppose the partitioning ratio indicated by the user is $a : b$, the first $\frac{a}{a+b} \times num\_vertices$ vertices are assigned to CPU, and the remaining vertices are assigned to MIC. The problem is that most graph datasets are power-law graphs, that is, the out-degrees of all vertices are not evenly distributed, and typically vertices with high out-degrees are together in a short range. If we simply partition graphs according to the number of vertices, the cumulative workload (cumulative out-degree) assigned to the devices are not proportional to the partitioning ratio provided by the users.

A candidate technique for solving the above problem is the round-robin assignment of vertices to devices: iterate over the vertices, for every $a+b$ vertices, the first a vertices are assigned to CPU, and the remaining b vertices are assigned to MIC. This technique ensures that vertices are assigned to devices in an interleaved way, and thus avoids the possibility that the clustered highly connected vertices are assigned to a single device. The problem is that this partitioning can lead to a very high volume of *cross edges* between two partitions, and thus high communication overhead between devices.

Based on the above observation, we derive a *hybrid partitioning* scheme. In this scheme, we first partition the vertices into small blocks, and then assign the blocks to the devices in a round-robin fashion. While partitioning the graph into small blocks, we use the *min-connectivity volume* partitioning scheme provided by the *Metis* software [16], so that the number of cross edges among the blocks is minimized. This partitioning not only maintains low communication overhead, but also keeps the computation ratio to be consistent with the expected partitioning ratio.

V. Experimental Results

In this section, we report results from a series of experiments evaluating our system from different perspectives – comparison between different execution schemes, i.e., *locking based execution* and *pipelining execution*, comparison with OpenMP code, benefit of using both CPU and MIC compared with using a single device, as well as the benefits of using SIMD parallelism. We also evaluate the effectiveness of graph partitioning module, and the overall performance by showing speedups over sequential executions.

A. Evaluation Environment

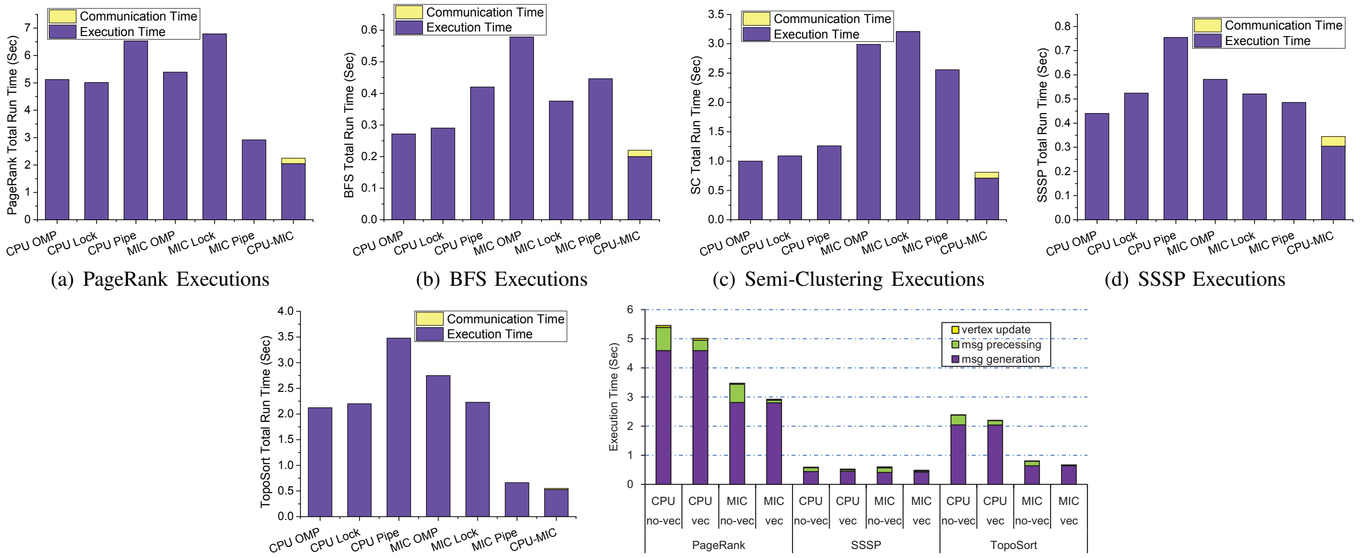
Our experiments were conducted on a node with an Intel Xeon E5-2680 CPU and an Intel Xeon Phi SE10P coprocessor. The CPU has 16 cores, each running at 2.70 GHz. The size of the main memory is 63 GB. The Xeon Phi has 61 cores each running at 1.1 GHz, with four hyperthreads per core. The total size of the memory on this card is 8 GB. We used mpic++ 4.1, from Intel IMPI library, to compile all the codes, with *-O3* optimization enabled. The mpic++ was built on top of icpc version 13.1.0. For the CPU-MIC symmetric computing, we compile separate binaries for MIC and CPU from the same source code, with *-mmic* flag for MIC, and *-xhost* for CPU.

B. Applications Used

Five commonly used graph applications are used to evaluate our system. Among them, **PageRank** is a graph algorithm [4] used to rank the importance of websites based on the number and quality of incoming links. Implementing a PageRank algorithm using our programming model involves treating each website as a vertex, and the value associated with each vertex (PageRank value) is initialized to 1. In each iteration, the message generation sub-step propagates the PageRank value of each vertex to its neighbors, by dividing the value by the number of outbound edges. The message reduction sub-step sums up the received PageRank values from the neighbors, utilizing SIMD processing. The vertex update sub-step updates each vertex's PageRank value using the sum. A directed graph *Pokec* [23] containing 1.6 million vertices and 31 million edges is used as the input.

BFS, Breadth First Search, is a very popular graph traversal algorithm. While implemented using our framework, initially, the source vertex is set as active, and its vertex value, level, is 0, while other vertices are inactive. In each iteration, active vertices send their level value plus 1 as messages to neighbors. Unvisited vertices which receive messages set their level, using any message that is received, and set themselves as active, and thus, message reduction is not needed. The execution ends when no active vertex exists. The same dataset with PageRank is used for this application.

SC or Semi-Clustering – is a graph based clustering algorithm, typically used for social network graphs. The algorithm



(a) PageRank Executions

(b) BFS Executions

(c) Semi-Clustering Executions

(d) SSSP Executions

(e) Topological Sorting Executions

(f) Effect of SIMD Processing (Vectorization) on Execution Times

Fig. 5: (a) - (e): Comparison of Different Versions of Five Applications. (f): Impact of SIMD Processing.

is applied to an undirected graph, with each vertex representing a person, and an edge representing a connection between two individuals. Semi-clusters are groups of people such that people within a semi-cluster interact frequently with each other. Each vertex may belong to more than one semi-cluster. A semi-cluster is represented as an array of vertex IDs. Each vertex has an associated value, which is a vector containing at most a number (a pre-defined maximum value) of semi-clusters. Each edge has a weight representing the interaction frequency. A score for each cluster is computed and the clusters associated with every vertex are sorted according to the score in descending order. In the message generation sub-step, each vertex sends the top-score clusters to all of its neighbors. In the message processing sub-step, each vertex combines the received clusters with the clusters from its own vertex value, and sorts them according to the score. Clusters with highest scores are used to update the vertex value. Because the message processing step is not associative and commutative, and the message type is not basic data type, SIMD reduction is not utilized. The dataset we used is a smaller undirected graph *DBLP* [41] containing 436 K vertices and 1.1 M edges. The reason why we use a smaller dataset is that the message size and vertex value size are much larger than the other applications and the dataset size is limited by the volume of the main memory on MIC. We converted the undirected graph to a directed graph by duplicating each edge, in order to fit the input into our system.

SSSP or Single Source Shortest Paths, was used as the running example throughout the paper. Same as PageRank, the message reduction sub-step in each iteration utilizes SIMD processing. The same dataset with PageRank is used in this application. In addition, we randomly generated weight value for each edge.

TopoSort, Topological Sorting, outputs a linear ordering for the vertices in a DAG (Directed Acyclic Graph), such that if there is an edge pointing from u to v, then u will appear before v in the ordering. Programming this algorithm on our framework involves the following steps: initially, vertices with zero in-degree are set as active, and other vertices are inactive. In each iteration, active vertices send messages containing value 1 to their neighbors, and set themselves as inactive. Vertices receiving messages sum up the messages, and decrease their in-degree value using the sum. If a vertex's in-degree becomes 0 after the substraction, it sets itself as active. The algorithm ends when no vertices are active. We use a randomly generated DAG containing 40K vertices and 200M edges as the input.

C. Overall Performance

In this section, we evaluate the performance of each application, and compare the different versions. Specifically, we evaluate both single-device executions and CPU-MIC executions. For single-device executions, besides the versions written using our framework, we also examine how our framework compares with OpenMP. Both OpenMP and our system provide a high level programming API, and both support SIMD parallelism either provided by the ICPC compiler with OpenMP directives or by the vector API of our programming model. The OpenMP version codes were compiled with ICPC at *-O3* with flags *-openmp -parallel*. Although the Intel compiler supports *offload* pragma, so that a program could be executed on both CPU and MIC, it is not easy to write an offloaded graph algorithm using it, since the communication handling is non-trivial and hard to achieve efficiency. Thus, while comparing with OpenMP, we only investigate single-device executions.

The results are shown in Figure 5(a) to Figure 5(e). The *CPU OMP* and *MIC OMP* versions are written with OpenMP directives on sequential code, with proper use of synchronization (OpenMP locks). *CPU Lock* and *MIC Lock* are single-device executions written with our framework, using locking-based message generation. Similarly, *CPU Pipe* and *MIC Pipe* are single-device versions on our framework using pipelining message generation. *CPU-MIC* version is the heterogeneous version written with our framework.

We investigated the compiler vectorization report for OpenMP codes (both on CPU and MIC). It turns out that the major loops of the applications written in OpenMP are not vectorized, and thus OpenMP code could not benefit from SIMD parallelism of the MIC architecture. This is because of the random memory access pattern of graph applications. The message organization and SIMD APIs in our system enables the efficient utilization of SIMD, the details of which will be shown later in this section.

CPU-only Executions: The framework execution strategies we designed are more suited for the MIC architecture, which has a larger thread number and high parallel memory bandwidth. In contrast, CPU has a much smaller number of threads and thus contention overhead is not severe. Also, CPU has a much smaller memory bandwidth so that the overhead of messages storage in our system offsets the benefits from reduced contention and SIMD message reduction. On average, for CPU-only executions, OpenMP outperforms our framework by 2.5%. Also, locking-based executions outperform the pipelining executions, due to that the locking-based method does not need to store messages in message queue before they are inserted to message buffer, and that no threads need to solely work on message movement. For all the applications, best performance was delivered with a total of 16 threads, i.e, 1 thread per core.

MIC-only executions: With the exception of BFS, pipelining executions (*MIC Pipe*) outperform locking-based (*MIC Lock*) executions, as well as OpenMP versions (*MIC OMP*), though the relative performance varies depending on the property of applications. For `PageRank`, all vertices generate messages along all edges every iteration, and thus the number of messages generated is large. As a result, contention is severe while locking is used. The pipelining version is 2.33x faster than the locking based method, and 1.85x faster than the OpenMP version. For both locking-based framework execution and OpenMP version, 240 threads was used to achieve the best performance, while 180 worker threads + 660 mover threads achieve the best performance for pipelining framework execution (same thread configurations are used for the remaining applications as well). For `BFS`, in each iteration, only a subset of vertices are active and the amount of messages generated is small, and thus the contention overhead is less severe. The locking-based framework execution is 1.19x faster than pipelining execution. *MIC lock* and *MIC pipe* are 1.54x and 1.30x faster than *MIC OMP*, respectively. Even though neither OpenMP or framework use SIMD for message processing, OpenMP is slower, likely due to the more expensive locking operations. For `SC` (Semi-Clustering), pipelining version performs better than locking version (1.25x faster), as well as OpenMP (1.17x faster). The speedup achieved from pipelining execution solely comes from reduced contention overhead, as SIMD reduction is not applied in this application. For `SSSP`, both *MIC Lock* and *MIC Pipe* run slightly faster than OpenMP (1.11x, and 1.20x speedup, respectively). The speedup of *MIC Lock* over OpenMP mainly comes from SIMD message reduction, while *MIC Pipe* also benefits from the reduced contention. The last application, `TopoSort`, uses a highly connected graph, which implies that in each iteration, a large number of messages are sent to a single vertex. The expensive locking used in OpenMP adversely impacts the performance. Pipelining execution is 4.15x and 3.36x faster than OpenMP version and locking-based framework execution, respectively.

CPU-MIC executions: For each graph dataset, a min-connectivity blocked partitioning (256 partitions) result is generated using *Metis*. The blocked partitioning result is reused for generating hybrid partitioning results for different ratios. Although the blocked partitioning takes time, it is only applied to each dataset once, and thus we did not include the partitioning time. We tried different partitioning ratios between CPU and MIC, and the results reported here are from the ratios that gave the best load balance. Locking-based execution was used for CPU, as it is faster than pipelining execution, while for MIC, pipelining execution was used except for BFS. Both the execution time and the communication time are separately shown in result figures.

For `PageRank`, *MIC Pipe* is 1.72x faster than *CPU Lock*, and CPU-MIC achieves a 1.30x speedup over the faster of single-device executions, i.e., *MIC Pipe*, at a graph partitioning ratio of 3:5 (CPU 3, MIC 5). For `BFS`, *CPU Lock* is 1.30x faster than *MIC Pipe*. Using CPU and MIC together results in a 1.32x speedup over *CPU Lock* version, at a graph partitioning ratio of 4:3. CPU performs much faster than MIC for `SC`, due to the more complex conditional instructions involved, which CPU is better at. CPU-MIC version is 1.29x faster than *CPU Lock* version, using a partitioning ratio of 2:1. For `SSSP`, CPU and MIC have a very similar performance, and using both achieves a speedup of 1.41 (using a equal graph partitioning). The last application, `TopoSort`, MIC is 3.32 times faster than CPU. Using CPU and MIC together has a smaller speedup of 1.20x over the MIC only execution (using 1:4 graph partitioning). Also note that the input for TopoSort is a highly connected graph, and the number of vertices is much smaller compared with the large number of edges, so that the communication overhead is negligible.

D. Benefits of SIMD Execution for Message Processing

Among the five applications, two applications involve no SIMD message reduction – specifically, SC uses sorting in message processing step, and BFS does not have message reduction sub-step. The other three applications perform message reduction through nuanced runtime SIMD parallelization supported in our framework. We now quantify the benefits obtained from runtime SIMD parallelization. For this purpose, we re-wrote the message processing sub-step for these three applications in a scalar way, for both CPU-only and MIC-only executions.

Figure 5(f) shows the execution times with and without vectorization. All reported data is from execution strategies and thread configurations that deliver the best results. For CPU executions, the vectorized versions achieve a speedup of 2.24, 2.35, and 2.22 over non-vectorized versions for the message processing sub-step for PageRank, SSSP, and TopoSort, respectively. Similarly, for MIC executions, 6.98x, 5.16x and 7.85x speedups are achieved for message processing steps. Although up to 4(CPU)/16(MIC) floating point messages are processed simultaneously by the SIMD lanes on each core, the achieved speedup is limited due to several factors: bubbles in the lanes due to the difference in the number of received messages for each vertex, and processing can become memory bound after a certain point. The benefits of SIMD processing on the overall performance depends upon the relative amount of time spent on the message processing phase. For CPU executions, these speedups are: 9%, 13% and 8% for PageRank, SSSP and TopoSort, respectively. For MIC executions, the speedups are: 18%, 23%, and 21%, respectively, for these applications.

E. Effect of Hybrid Graph Partitioning for CPU-MIC Execution

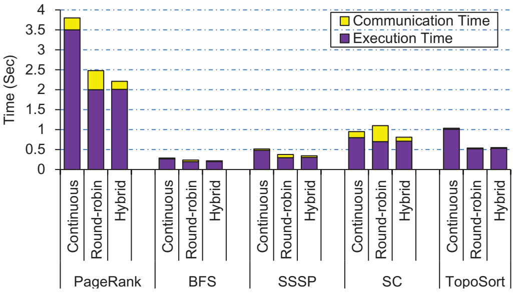


Fig. 6: Impact of Different Graph Partitioning Methods

To illustrate the effectiveness of the hybrid graph partitioning, we executed the CPU-MIC versions with *continuous partitioning*, *round-robin partitioning*, and *hybrid partitioning*. The partitioning ratio used for each application is the same as that is used for achieving the best CPU-MIC execution in Section V-C. Figure 6 shows the time distribution of each application using the different partitioning schemes. The execution time is determined by the slower device, and the communication time is the time spent on exchanging messages between devices. For PageRank, a 3:5 partitioning ratio is used as the input to the partitioning modules. Both round-robin and hybrid partitioning methods achieve balanced partitioning results: the ratio of cumulative out-degrees on CPU and MIC is close to the expected workload ratio (3:5). Because vertices with higher out-degrees are concentrated at the front of the graph *Pokec*, continuous partitioning leads to a larger than expected number of edges on CPU, but less than expected number of edges on the MIC. This leads to a much longer overall running time due to significant load imbalance. Round-robin partitioning results with 2.27x more cross edges than hybrid partitioning, and thus much longer communication time. Execution using hybrid partitioning is 1.72x and 1.13x faster than continuous and round-robin partitionings, respectively. Similar trends can be seen from the other applications: for BFS, using hybrid partitioning is 1.31x and 1.09x faster than the other two partitioning methods; for SSSP, the speedups are 1.50 and 1.10; and for SC, 1.17 and 1.36. The graph *DAG* used in TopoSort has almost equal number of cross edges using round-robin and hybrid partitionings, and thus similar execution times using either partitioning, though running with continuous partitioning result is much slower than the other two due to the imbalanced partitioning result.

F. Overall Performance Gained from Our Framework

To see the efficiency and scalability of our framework, we compare different versions using our framework with *sequential* versions (written in C/C++ and executed by one core). We run sequential versions on both CPU and MIC.

	Execution times (Sec)				
	PageRank	BFS	SC	SSSP	TopoSort
CPU Seq	18.01	1.46	8.29	2.62	8.42
MIC Seq	181	12.19	134.06	24.07	85.17
CPU Multi-core Speedup over CPU Seq	5.01 (3.6x)	0.29 (5x)	1.09 (7.6x)	0.52 (5x)	2.20 (3.8x)
MIC Many-core Speedup over MIC Seq	2.92 (62x)	0.38 (32x)	2.56 (52x)	0.49 (49x)	0.66 (129x)
CPU-MIC Best Speedup over CPU Seq	2.25 (8x)	0.22 (6.7x)	0.81 (10.2x)	0.34 (7.7x)	0.55 (15.3x)

TABLE II: Parallel Efficiency Obtained from Our Framework

The results are shown in Table II. Note that even though the clock frequency of a CPU core is only 2.4 times faster than a core on MIC, a CPU core runs the same sequential code around 11x faster, on the average, because of out-of-order execution and other enhancements. The speedup of CPU multi-core execution on our framework ranges from 3.6x to 7.6x. Similarly, MIC many-core execution has a speedup of 32x to 129x, compared with MIC sequential execution. The last application, Topological sorting, achieves the highest speedup on MIC, due to the high connectivity of the DAG, leading to a high computation density. The combined execution of CPU and MIC, while compared with CPU sequential version, achieves a speedup of between 6.7x and 15.3x.

VI. Related Work

There have been many efforts on parallelizing various graph applications on parallel systems. Pregel [26], a programming model proposed by Malewicz *et al.*, provides programmers a general API, and was built for distributed graph processing. The default graph partitioning used in Pregel and some of the open source implementations [2] is hash based and thus does not consider the amount of communication volume. GPS [32], another open source implementation of Pregel, also utilizes Metis to balance the edges in partitions and minimize cross edges, but in a more straightforward way. Our method is able to reuse the blocked partitioning results of Metis for different partitioning ratios. Mizan [17] implements Pregel system using vertex migration. Compared with Mizan, our approach is more lightweight, and more suitable for the case of using a limited number of devices. Other parallel graph libraries such as the Parallel Boost Graph Library [10], and CGMgraph [5] have been on distributed memory systems. They provide implementations for particular graph algorithms, instead of general-purpose frameworks. Sedge [42] proposes a partitioning method for local graph queries on a concentrated subset of vertices across clusters. Cyclops [7] is a distributed graph programming framework executing in a different way from BSP models. Seraph [40] is a distributed graph processing system aiming to minimize the memory consumption and fault tolerance costs on clusters.

Galois project [27], [22] extensively studies and automatically explores *amorphous* data-parallelism present in irregular algorithms including graphs, and it can execute *Galoized* serial code in parallel on shared-memory machines. Our work cannot handle data-driven or speculative parallelism, but focuses on issues associated with large-scale shared memory and SIMD parallelization for data-parallel graph applications. MTGL [3] is another parallel graph library, which is on shared memory system, however, it focuses on graph query algorithms.

Recently, there have also been many efforts focusing on parallelizing graphs and trees applications on SIMD accelerators and GPUs. Merrill *et al.* [28] parallelize Breadth-first Search, an important graph traversal algorithm, on the GPUs architecture by focusing on fine-grained task management. This work shares some similarities to our work, however, is not aimed towards providing a high-level API for graph applications. Hong *et al.* [12] developed a novel virtual warp-centric programming method to address the work imbalance problem in graph algorithms, but again, is not providing a high-level API. Jo *et al.* [13] designed a novel scheduling mechanism for efficient SIMD execution of tree traversal applications, and Kim *et al.* [19] designed *FAST*, an architecture-sensitive layout of the index tree on both CPU with SIMD and GPU architectures. Both of these efforts focus on tree structures, which involves different characteristics (especially from the layout reorganization consideration) than our target class. Most closely related to our work, Zhong and He [43] proposed a programming framework, *Medusa*, to parallelize graph processing applications on GPUs by adapting Pregel's design. They focus on issues with SIMT architecture, whereas our design is for exploiting the more rigid SSE-like parallelism as well as larger-scale MIMD (shared memory) parallelism. CuSha [18] optimizes graph processing on GPUs with intensive usage of shared memory, by re-organizing the graph data in *shards*. Again, the technique is specifically for GPU architecture.

Recently, several efforts have also parallelized irregular applications on the Xeon Phi architecture, and thus share some similarities with our work. Saule and Catalyurek [33] provided a preliminary evaluations on graph applications on the Xeon Phi architecture, but did not report a system design. Liu *et al.* [24] parallelized Sparse Matrix-Vector Multiplication, and Pennycook *et al.* [29] parallelized *Molecular Dynamics* on Xeon Phi, with emphasis on long vector SIMD parallelization. Our work has focused on supporting a system with a general and high-level API, for a set of graph processing applications.

VII. Conclusions and Future Work

With growing density of transistors and increasing focus on energy efficiency, future processors will have increasing number of cores, parallelism at multiple levels, and relative small amount of memory per core. Such a trend is reflected through a new coprocessor, Intel Xeon Phi. More specifically, Xeon Phi is an example of a system where large-scale shared memory (MIMD) as well as SIMD parallelism must be exploited to achieve high efficiency.

This paper has focused on the challenge of exploiting these two types of parallelism simultaneously for graph processing applications. We have supported a simple graph programming API. The innovative components of our runtime system include a condensed static memory buffer, which supports efficient message insertion and SIMD message reduction while keeping the memory requirements low, and a pipelining scheme for efficient message generation by avoiding frequent locking operations. Although these techniques are specifically designed for the MIC architecture, they are able to run on a multi-core CPU, with a reasonable performance, compared with OpenMP. Our framework is able to execute a graph application across the CPU and MIC, with the assistance of our proposed *hybrid* graph partitioning. We believe that the underlying ideas of our approach can be used for other classes of applications, especially irregular reductions and N-body problems, which share some similarities with graph algorithms. Our future work includes applying and/or extending the current system for other classes of applications, as well as providing additional functionality for graph applications, such as auto-tuning for deciding the optimal number of worker/-mover threads, as well as the partitioning ratio between CPU and MIC.

References

[1] http://www.top500.org/lists/2013/11/.

[2] Giraph. http://giraph.apache.org/.

[3] Jonathan W Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. IPDPS '07, pages 1–14. IEEE.

[4] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[5] Albert Chan, Frank Dehne, and Ryan Taylor. Cgmgraph/cgmlib: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines. *International Journal of High Performance Computing Applications*, 19(1):81–97, 2005.

[6] Jiyang Chen, Osmar R. Zaïane, and Randy Goebel. Detecting Communities in Social Networks Using Max-Min Modularity. In *SDM*, pages 978–989, 2009.

[7] Rong Chen, Xin Ding, Peng Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Computation and Communication Efficient Graph Processing with Distributed Immutable View. HPDC '14.

[8] Jiefeng Cheng, Jeffrey Xu Yu, Bolin Ding, Philip S. Yu, and Haixun Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[9] Jiri Dokulil, Enes Bajrovic, Siegfried Benkner, Sabri Pllana, Martin Sandrieser, and Beverly Bachmayer. Efficient Hybrid Execution of C++ Applications using Intel Xeon Phi Coprocessor. *CoRR*, abs/1211.5530, 2012.

[10] Douglas Gregor and Andrew Lumsdaine. Lifting Sequential Graph Algorithms for Distributed-memory Parallel Computation. In *OOPSLA*, pages 423–437. ACM, 2005.

[11] Daniel S. Hirschberg, Ashok K. Chandra, and Dilip V. Sarwate. Computing Connected Components on Parallel Computers. *Commun. ACM*, 22(8):461–464, 1979.

[12] Sungpack Hong, Sang Kyun, Kim Tayo, and Oguntebi Kunle Olukotun. Accelerating CUDA Graph Algorithms at Maximum Warp. In *In PPoPP*. ACM, 2011.

[13] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. Automatic Vectorization of Tree Traversals. PACT '13. IEEE Press.

[14] U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop. Technical Report CMU-ML-08-117, School of Computer Science, Carnegie Mellon University, 2008.

[15] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System. In *ICDM*, pages 229–238, 2009.

[16] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[17] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. EuroSys '13, pages 169–182. ACM, 2013.

[18] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric Graph Processing on GPUs. HPDC '14.

[19] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. SIGMOD '10.

[20] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. In *SODA*, pages 668–677, 1998.

[21] Brian Kulis, Sugato Basu, Inderjit Dhillon, and Raymond Mooney. Semi-supervised Graph Clustering: A Kernel Approach. ICML '05.

[22] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. How Much Parallelism is there in Irregular Applications? In *PPOPP*, pages 3–14. ACM, 2009.

[23] M. Zabovsky L. Takac. Data Analysis in Public Social Networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations*, 2012.

[24] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. ICS '13. ACM.

[25] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A New Framework for Parallel Machine Learning. *Conference on Uncertainty in Artificial Intelligence*, 2010.

[26] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[27] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, Muhammad Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven Optimizations for Amorphous Data-parallel Programs. In *PPOPP*, pages 3–14. ACM, 2010.

[28] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. PPoPP '12. ACM.

[29] Simon J Pennycook, Chris J Hughes, M Smelyanskiy, and SA Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors. IPDPS '13. IEEE.

[30] Tieyun Qian, Jaideep Srivastava, Zhiyong Peng, and Phillip C.-Y. Sheu. Simultaneously Finding Fundamental Articles and New Topics Using a Community Tracking Method. In *PAKDD*, pages 796–803, 2009.

[31] Sayan Ranu and Ambuj K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *ICDE*, pages 844–855, 2009.

[32] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. SSDBM, 2013.

[33] Erik Saule and Umit V Catalyurek. An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture. IPDPSW '12. IEEE.

[34] Nisheeth Shrivastava, Anirban Majumder, and Rajeev Rastogi. Mining (Social) Network Graphs to Detect Random Link Attacks. In *ICDE*, pages 486–495, 2008.

[35] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. DOULION: Counting Triangles in Massive Graphs with a Coin. In *KDD*, pages 837–846, 2009.

[36] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, August 1990.

[37] Nan Wang, Srinivasan Parthasarathy, Kian-Lee Tan, and Anthony K. H. Tung. Csv: visualizing and mining cohesive subgraphs. In *SIGMOD*, pages 445–458, 2008.

[38] Takashi Washio and Hiroshi Motoda. State of the Art of Graph-based Data Mining. *SIGKDD Explor. Newsl.*, 5(1), July 2003.

[39] Samuel Williams, Dhiraj D. Kalamkar, Amik Singh, Anand M. Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid Oliker. Optimization of Geometric Multigrid for Emerging Multi- and Manycore Processors. SC '12, 2012.

[40] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: An Efficient, Low-cost System for Concurrent Graph Processing. HPDC '14.

[41] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-truth. MDS '12. ACM.

[42] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards Effective Partition Management for Large Graphs. SIGMOD '12. ACM.

[43] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *TPDS*, 99(1):1, 2013.