HiPa: Hierarchical Partitioning for Fast PageRank on NUMA Multicore Systems

YuAng Chen
The Chinese University of Hong Kong, Shenzhen
yuangchen@link.cuhk.edu.cn

Yeh-Ching Chung
The Chinese University of Hong Kong, Shenzhen
ychung@cuhk.edu.cn

ABSTRACT

PageRank, weighing the importance of vertices in a graph, serves as an fundamental algorithm for graph-structured tasks in a variety of domains. However, the processing capacity of multicore systems is oftentimes poorly utilized for large-scale PageRank due to the irregular memory accesses and poor cache efficiency. In this paper, we present HiPa, a novel hierarchical partitioning methodology to accelerate PageRank by utilizing the memory-cache architecture of the multicore system. For the shared memory, HiPa subdivides the graph based on the NUMA characteristics to reduce remote memory access while ensuring workload balance. For the private cache, HiPa further splits the graph into cache-able partitions to promote in-core computing and cache locality. Based on the partitioning strategy, systematical optimizations are proposed, such as thread management and new data layout. These effectively alleviate thread migration and thread contention, thus enhancing the scalability of HiPa. The integration of NUMA- and cache-aware parallelism allows HiPa to harness the potential of multicore systems. The performance of HiPa is evaluated by comparing with the the start-of-the-art graph frameworks and hand-optimized implementations. Over the best among them, HiPa achieves accelerations from 1.11× to 1.45×, and reductions in remote memory accesses from 1.87× to 3.90×. Moreover, we investigate the behaviors of HiPa on different processor micro-architectures to push its performance closer to hardware limit.

CCS CONCEPTS

• **Computing methodologies** → **Shared memory algorithms**;
• **Computer systems organization** → **Multicore architectures**.

KEYWORDS

graph processing, memory hierarchy, multicore systems

ACM Reference Format:
YuAng Chen and Yeh-Ching Chung. 2021. HiPa: Hierarchical Partitioning for Fast PageRank on NUMA Multicore Systems. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3475737

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3475737

1 INTRODUCTION

Graph-structured data are commonly used to model the relation among a group of entities. It is widely adopted in real-world analytics for diverse purposes, such as social networks analysis [6], transportation planning[3], financial fraud detection [13]. Analyzing graphs in depth, PageRank is one of the most popular graph algorithms. It was initially proposed to measure the importance of a webpage for web search engines [29].

$$PR_{new}(v) = \frac{1-d}{|V|} + d \times \sum_{u \in E_{in}(v)} \frac{PR_{old}(u)}{|E_{out}(u)|} \quad (1)$$

The mathematical description of PageRank is presented in Eq. 1. d is a damping factor; $|V|$ is the total number of vertices in a graph; u are the in-neighbors of v; $|E_{out}(u)|$ is the number of outgoing edges of u. During the processing, the adjacency matrix of the graph is iteratively multiplied with an old rank vector to obtain a new rank vector. The mathematical representation of a skewed graph is a sparse adjacency matrix. Therefore, the computation of PageRank can be interpreted as iterative sparse matrix-vector multiplications (SpMV) [21]. Our discussions and optimizations proposed for PageRank can also be applied to SpMV as well as other graph algorithms.

In recent decades, information grows explosively in size and complexity. In order to handle large-scale graph data efficiently, significant research efforts are devoted to high-performance graph processing on various platforms, e.g., single-machine multicore systems [20, 32, 38] and multiple-machine distributed systems [11, 39, 43]. While the distributed graph processing is able to provide high scalability by utilizing multiple machines, it oftentimes delivers suboptimal cost efficiency and performance compared with its shared-memory counterpart running on multicores [43].

Equipped with sufficiently large DRAM, a modern server-class machine can process large-scale graph tasks within its main memory. Nevertheless, it still remains as a challenge to maximize the performance potential of the hardware resource. High memory latency, random cache accesses, thread divergence and etc., which are incurred by the irregularity of graph-structured data, restrains the effectiveness and efficiency of parallelism on multicore systems.

The intrinsic irregularity of graphs are commonly found in natural graph datasets that profile real-world phenomena, such as Internet topology [10]. A typical feature to exemplify such irregularity is the *skewed* power-law distribution of vertex degrees, where a tiny fraction (e.g., 10 percent) of vertices are responsible for a major fraction (e.g., 90 percent) of edges. For instance, a celebrity has massive social influence in a social network, and a search engine reaches out to billions of webpages in a web network.

To address the irregularity of graphs and accelerate graph processing, a variety of graph processing frameworks are developed

[20, 30, 32, 38, 40, 44]. In general, they can be categorized into three types: vertex-centric, edge-centric, and partition-centric. The vertex-centric paradigm is broadly implemented in contemporary frameworks [32, 38], where each thread processes the graph at the unit of vertex. To promote semi-sequential accesses and alleviate the burdens of synchronization primitives, edge-centric paradigm is thereby designed [30, 44]. Threads in this paradigm handle edges instead of vertices. To further enhance cache utility and eliminate synchronization overhead, partition-centric paradigm is proposed in recent works [20, 40]. There, threads operates on partitions, which are the vertex subsets split from the graph to fit into caches.

Our work in this paper lies in the area of the partition-centric paradigm with enormous inspirations from the vertex-centric one. The backbone of our approach is the *hierarchical partitioning* (HiPa) with the awareness of memory and cache architecture. Firstly, at the level of main memory, a graph is coarsely partitioned and placed within the local Non-Uniform Memory Access (NUMA) node. Then, at the level of cache, the graph data is further partitioned so that each subgraph is privately owned by a core. The joint exploitation of memory and cache allows HiPa to take the advantages of both vertex- and partition-centric paradigms. The benefits include minimizing remote memory accesses, improving cache locality and, consequently, significant speedup.

The contributions of this paper can be summarized as follows:

- We propose a hierarchical partitioning (HiPa) methodology to boost the performance of PageRank. Firstly, HiPa subdivides a graph onto the NUMA nodes to co-locate data and computation. Then, within each NUMA node, HiPa refines the subsets of data into disjoint cache-able partitions to promote in-core computing.
- Based on the partitioning strategy above, we further optimize HiPa by pinning data to threads and differentially placing data fields. The optimizations eliminate thread contention on hardware resources, reduce memory traffic and enhance scalability.
- Comprehensive experiments validate the effectiveness of HiPa in comparison with state-of-the-art frameworks and hand-optimized implementations. Outperforming the fastest of them, HiPa yields speedups by $1.11 - 1.45\times$ (with remote memory reduction by $2.87 - 4.99\times$). Also, in terms of remote memory accesses, HiPa achieves reductions by $1.87 - 3.90\times$ against the best alternative.
- We carefully investigate the difference between two generations of Intel CPUs. The upgrades in micro-architecture lead to non-trivial impact on the choice of partition size for partition-centric graph processing.

2 BACKGROUND

2.1 Basic Concepts

Graph Locality. The performance of PageRank on multicore systems is heavily affected by the data access pattern. We refer to such pattern as graph locality, which can be classified into two types: *temporal* and *spatial*. A group of vertices (or edges) show high temporal locality if frequently accessed; they exhibit high spatial locality if stored in nearby memory allowing for continuous reading and writing.

For the enhancement of graph locality, numerous optimization schemes are designed. For instance, via graph partitioning [38] or blocking [5], threads are limited to access a subset of graph vertices. The limitation eliminates the remote accesses to the long-distance vertices, thus enhancing the spatial locality. On the other hand, to facilitate temporal locality, hot vertices (e.g., whose degrees are higher than the average) are concentrated together by graph reordering [9] or semi-sorting [44].

2.2 NUMA Architecture

Modern server-class machines are typically featured with NUMA memory design. In these machines, a single motherboard utilizes more than one processors (i.e., sockets or CPUs). Each processor consists of multiple cores. The cores are connected with a local memory via a local bus, which together compose a NUMA node. The NUMA nodes are bound by a interconnect, finally building a multicore system [2].

Though the local memory of every NUMA node is globally shared with other nodes, there exists performance variability when a processor accesses different memory locations. The fastest speed is achieved if a processor interacts with its local memory within the same NUMA node. However, if a processor needs to reach out to a remote memory inside another NUMA node, the request must go through the remote memory controller of that node [12]. The indirect access to remote memory causes high memory latency, therefore leading to severe performance degradation. For instance, on a multicore systems with two Intel Xeon Silver 4210 CPUs (i.e., NUMA nodes), it costs a core 0.06 seconds to sequentially read 1GB data within its local memory, but 0.40 seconds for the remote one.

To exploit the benefits of NUMA architecture, a genre of NUMA-aware graph analytics frameworks and techniques are proposed [36, 38, 43]. In general, these approaches partition the graph into several subgraphs and allocate them to NUMA nodes. A balanced partitioning strategy (e.g., by edges or vertices) is required to promote workload balance among the NUMA nodes. Various optimization techniques are utilized, including NUMA-aware data layouts [38] and work-stealing scheduler [36]. The NUMA-awareness is typically implemented in vertex-centric graph systems. To the best of our knowledge, all existing partition-centric works are NUMA-oblivious [20, 40, 42].

2.3 Partition-centric Graph Processing

The essence of partition-centric paradigm lies in the *cache-able* disjoint partitions of a graph. All vertices of a graph are segmented into vertex subsets, the sizes of which equal the size of caches. When a graph is processed, the vertex subsets are taken as the input by threads for user-defined functions (e.g., PageRank).

The optimal partition size is empirically determined based on the characteristics of cache hierarchy. The cache hierarchy of a multicore system typically consists of three levels: level 1 (L1) cache, level 2 (L2) cache, and last level cache (LLC). Among them, L1 cache provides the fastest speed but the smallest storage, which is exactly opposite to LLC. As a compromise between speed and storage, L2 cache is widely selected as the partitioning basis for partition-centric graph frameworks [20, 21, 42].

By restricting each core to access a subset of graph vertices within its local L2 cache, the spatial locality is significantly improved as accesses to remote vertices are eliminated. Besides the vertex subset, each partition also includes a edge subset. If a edge

travels from a source vertex in one partition to a destination vertex in another partition, it is labeled as the inter-edge. Inter-edges incur core-to-core communication through the LLC or even the memory, which causes high latency. In comparison, when the source and destination vertices of a edge reside in the same core, this edge is labeled as the intra-edge. Intra-edges indict that the operations on vertices occur in local caches, which benefits the performance.

3 HIERARCHICAL PARTITIONING

Inspired by the NUMA characteristics of multicore systems and the partition-centric graph analytics, we proposed HiPa, a hierarchical partitioning methodology that leverages the memory-cache architecture. The abstraction of HiPa is illustrated in Fig. 1. It is designed with the goals of reducing remote memory accesses and improving cache efficiency. The details are described in following sections.

3.1 NUMA-aware Partitioning

The key to utilize a NUMA machine is co-locating the computation and data within the same NUMA node. Improper data allocation strategy might cause high overhead and workload imbalance for parallel graph processing, thus deteriorating the performance [38]. Therefore, it is a crucial task to partition the graph components, including vertices and edges, for the NUMA nodes.

When partitioning a graph, an intuitive idea is to evenly allocate vertices over the NUMA nodes. Define the graph as $G = (V, E)$, where V is the vertex set and E is the edge set. The disjoint vertex subsets V_i allocated to N NUMA nodes have the identical size of $|V|/N$, where $1 \leq i \leq N$. It should be emphasized that the vertex

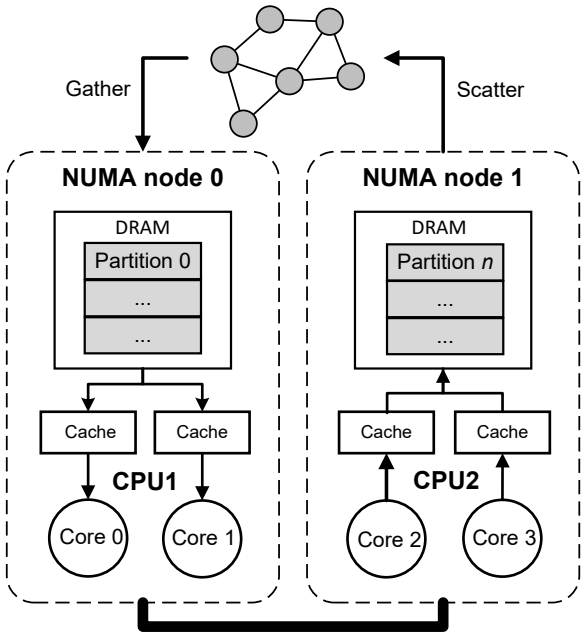

Figure 1: Conceptual Design of HiPa. HiPa applies hierarchical partitioning to a graph based on the characteristics of NUMA memory and private cache. Node 1 accumulates data in Gather phase for processing, and node 2 sends out updated data in Scatter phase.

subsets preserve the vertex order as in the original graph, and fulfill such conditions: $\cap_{i=1}^{N} V_i = \varnothing$ and $\cup_{i=1}^{N} V_i = V$. The same rules apply to edges too. Nevertheless, for graph algorithms where all graph edges are engaged in the processing, e.g., PageRank, the computation complexity mainly replies on the edges instead the vertices. For the skewed graphs, the even allocation of vertices leads to workload imbalance, thus slowing down the computation.

To cope with the above issue, previous works prefer to prioritizing edges for balanced partitioning [36, 38]. Specifically, each NUMA node is allocated with the same number of edges $|E|/N$. Thereby, the vertices are split into several subsets V_i with varied size $|V_i|$ holding these edges. In other words, the sum of vertex degrees $\sum_{v \in V_i} D(v)$ in every subset equals $|E|/N$, where v is a vertex and $D(v)$ is the degree of it. The allocations of edges and vertices on the i^{th} NUMA node can be mathematically described as follows:

$$\begin{aligned} |E_i| &= \frac{|E|}{N} \\ V_i &= \{v \in V | \textstyle\sum_{v \in V} D(v) = \frac{|E|}{N}\} \end{aligned} \tag{2}$$

Enlightened by the predecessors, HiPa also adopts the edge-oriented partitioning approach, but at a *coarser* granularity of cache-able vertex subsets, for instance, L2-cache-sized partitions (L2-partitions). The number of vertices allocated to a NUMA node must be a multiple of L2-partitions. The size of a L2-partition P is fixed to $|P|$= {L2 cache size} / {single vertex size}, and the resized vertex subset is denoted as $\tilde{V}_i$. Then, the number of vertices assigned to each NUMA node can be written as $|\tilde{V}_i| = n_i \cdot |P|$, where n_i is the least succeeding integer of $|V_i|/|P|$. Due to the resizing of vertex subsets, the corresponding edge subsets are resized too, which is denoted as $\tilde{E}_i$. Based on Eq. 2, the number of edges and vertices reallocated to the i^{th} NUMA node can be formulated as follows:

$$\begin{aligned} |\tilde{V}_i| &= ceil(\frac{|V_i|}{|P|}) \cdot |P| = (\frac{|V_i| - 1}{|P|} + 1) \cdot |P| = n_i \cdot |P| \\ |\tilde{E}_i| &= \textstyle\sum_{v \in \tilde{V}_i} D(v) \end{aligned} \tag{3}$$

In addition, the last NUMA node is an exception, which accommodates the leftover vertices and edges from other nodes. Also, the edges used for partitioning are either in-edges or out-edges. In this paper, the out-edges are selected for demonstration.

3.2 Cache-aware Partitioning

The NUMA-aware partitioning is aimed to minimize cross-node memory traffic and workload imbalance. Inside each NUMA node, the partitioning methodology is further refined.

In conventional partition-centric paradigm, threads handle the L2-partitions (the partitions with size equivalent to L2 cache) on a many-to-many basis. Any thread can arbitrarily access any partition. When a graph is being processed, the threads iterate over all graph partitions according to the first-come-first-serve policy. Threads have to contend for the ownership of a partition for exclusive processing within the private cache.

To avoid the potential contention of threads on the same partition, HiPa deploys a one-to-many model to the relationship between the threads and the partitions. Given Eq. 3, the i^{th} NUMA node contains n_i partitions. These partitions are distributed to local cores in groups. Each group G contains the same number of edges, which

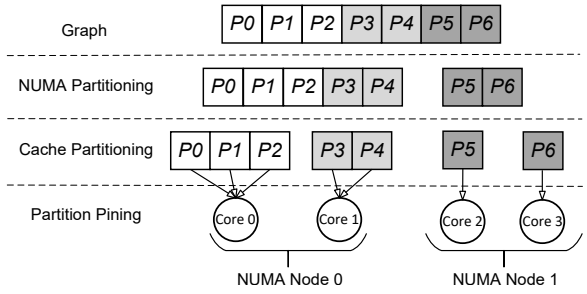

Figure 2: Partitioning result of HiPa. The boxes represent cache-able partitions of the graph data. Respectively, P0-2 hold 10 edges, P3-4 hold 15 edges, and P5-6 hold 30 edges. The final output of HiPa is that the cores are allocated with *unequal* numbers of partitions but equal number of edges.

applies edge-oriented partitioning again. When the graph is being processed, the access of a thread is confined to its belonging group of partitions. For example, assuming the i^{th} NUMA node consists of C cores, every thread (i.e., working core) in this node is assigned with a group of m_j partitions, where $1 \leq j \leq C$. Following conditions are satisfied:

$$\begin{aligned} n_i &= \textstyle\sum_{j=1}^{C} m_j \\ |G_j| &= m_j \cdot |P| \\ \frac{|\tilde{E}_i|}{C} &= \textstyle\sum_{v \in G_j} D(v) \end{aligned} \tag{4}$$

In brief, the partitioning methodology of HiPa imposes a two-level constraints on the working cores. Firstly, a graph is coarsely partitioned based on the NUMA design of memory, such that graph data can be processed by local processors. Secondly, within individual NUMA node, the accessibility of cores are further restricted to a group of cache-able partitions to eliminate thread contention.

Fig. 2 exemplifies the partitioning result of HiPa. Assume a graph can be subdivided into 7 cache-able partitions with equal number of vertices. In the first stage, the first five partitions and the last two are allocated to the NUMA node 1 and 2 respectively for workload balance (e.g., $n_1 = 5$ and $n_2 = 2$). Then, in the second stage, the first 3 sparse partitions are separated and fed into core 0. The results of cache-level partitioning are $m_1 = 3, m_2 = 2, m_3 = 1$, and $m_4 = 1$. Also, to deal with real-world graphs that cannot be precisely partitioned according to Eq. 4, we loosen the condition by allowing: $\sum_{v \in G_j} D(v) \geq |\tilde{E}_i|/C$.

3.3 Thread Management

3.3.1 Problem: Thread Contention. The Hyper-Threading is a popular technology offered by various modern CPUs [12]. It improves the parallelism of a multicore system by creating two logic cores from every physical core in the system. Each pair share the same hardware resources, including caches and bus, and incur light-weight overhead between context switches. Hence, the cooperation between the paired logic cores is more efficient than that of two physical cores. In addition, the number of logic cores stands for the maximum number of threads available for multithreading, which doubles the number of physical cores.

The Hyper-Threading is enabled by default in almost all graph processing frameworks to boost parallelization for speedup [20, 32, 38, 44]. Nevertheless, in the partition-centric paradigm, full utilization of logic cores often leads to a performance decline [20, 21, 40, 42]. In Section 3.3, experimental results show that existing partition-centric graph methodologies scales poorly once the number of working threads exceeds the number of physical cores. When all logic cores are utilized, the processing of graph is nearly slowed down by 2× (see Fig. 6).

The poor scalability results from the intensive contention among threads on hardware resources, including caches and memory bus [21]. In prior partition-centric works, each thread is allocated with L2-cache-sized data for computation, which requires full occupation of the private cache. This intensifies the thread contention, because the dual threads (i.e., logic cores) on the same physical core have to compete for the usage of L2 cache. Furthermore, the memory bandwidth is designed to scale with the physical core instead of logic cores [12]. The bandwidth is heavily congested when all logic cores are flushing the L2 caches [20].

To alleviate the thread contention, prior works simply use only half of available threads, which equates the number of physical cores [20, 21, 40, 42]. However, the problem of contention still exists due to the randomness of thread creation. When multithreading, the operating system (OS) arbitrarily generate threads from the pool of *logic* cores, regardless of the status of physical cores. Therefore, it might occur that two selected logic cores corresponds to the same physical core and thus contend for resources. Moreover, the threads schedule the processing of partitions on a first-come-first-serve policy. The competition amongst multiple threads for the ownership of a partition would further intensify the thread contention.

In order to alleviate the thread contention and enhance the scalability of partition-centric paradigm, we propose a thread-data pinning scheme for the coordination between data and threads on multicores. It consists of two steps: (1) bind the threads to NUMA nodes, and (2) pin the partitions to the bound threads. The details of this scheme are discussed in next section.

3.3.2 Solution: Thread-data Pinning. As outlined in Algorithm 1, the partition-centric processing of a graph can be generalized into a scatter-gather model that iteratively updates a graph [20]. The computation in every iteration envelops a scatter phase and a gather phase, where the updates of local vertices are propagated to or collected from neighbors. Within each phase, multiple threads are created to parallelize the task. In other words, one phase, either scatter or gather, includes the complete lifecycles of threads. A phase represents a parallel region. As a result, the graph computation is decomposed into numerous parallel regions

For instance, assuming a graph algorithm finishes in 10 iterations on a multicore system consisting of 2 NUMA node with 4 physical cores (i.e., 8 logic cores) per node. Within each iteration, the scatter and gather phases are respectively parallelized into discrete parallel regions. To avoid thread contention, only half of the logic cores are used. Therefore, inside each region, a thread pool that contains 8 threads are initialized to scatter/gather all graph partitions and then terminated after synchronization. Therefore, the overall computation, which consists of 10 iterations of scatter-gather phases, creates $10 \times 2 \times 8$ threads in an interleaved routine.

Algorithm 1: NUMA-oblivious scatter-gather model

```
   Input: partitions → set of partitions
   Output: Graph
1  for i ← 0 to iteration do
2  |  for p ∈ partitions do in parallel        ▷ parallel region
3  |  |  gather(p) ;
4  |  for p ∈ partitions do in parallel        ▷ parallel region
5  |  |  scatter(p);
6  Graph ← concatenate (partitions);
```

In the context of NUMA-aware processing, additional overheads are involved for thread manipulations, including *thread binding* and *thread migration*. To place the computation together with data on the same NUMA node, the computing threads are bound to the particular NUMA node where the data are stored. When a thread is initialized on a wrong NUMA node, thread binding would invoke thread migration, where a thread is migrated from current core to another core on the correct NUMA node. Thread migration incurs high latency as the thread context is transferred via memory. Such latency is increased in NUMA systems, because the transfer across NUMA nodes always proceeds through remote memory [1].

If we deploy NUMA-aware multithreading on the processing model of Algorithm 1, every threads created inside the parallel region have to be bound to a NUMA node. In the worst case, where all threads are initialized in the wrong NUMA nodes, the thread migration occurs up to 160 times, since there are 160 threads created during 10 iterations. The problem of thread migration is amplified by the massive production of threads.

In order to minimize the expensive thread migrations, a new processing model is proposed as in Algorithm 2. It reduces the frequency of thread creation by extending the functionality of threads. The lifetime of a thread covers the whole period of graph processing. Also, according to the one-to-many partitioning result obtained in Section 3.2, the cache-able partitions are pinned to threads (line 8). As a result, each thread is able to perform *complete* iterative scatter-gather computation over the *fixed* group of partitions.

Algorithm 2: Numa-aware scatter-gather model

```
   Input: numa_Partitions → numa-ly allocated partitions
   Input: numa_Threads → numa-ly bound threads
1  Function Th_Func(numa_part)
2  |  for i ← 0 to iter do
3  |  |  scatter(numa_part);
4  |  |  synchronize with other threads;
5  |  |  gather(numa_part);
6  for th ∈ numa_Threads do in parallel       ▷ parallel region
7  |  match th with p ∈ numa_Partitions;
8  |  th calls Th_Func(p);
9  Graph ← concatenate (numa_Partitions)
```

By pinning the partition to the thread on NUMA nodes, data traffic is mainly restricted within local memory, and thread contention are effectively eliminated. Hence, we can now take the advantage of Hyper-Threading and utilize all logic cores. In NUMA-aware model of Algorithm 2, the number of created threads during the overall graph processing equals to the number of available *logic* cores, which is 8×2. Hence, in the worst case of thread binding, the thread migration occur at most 16 times, which is significantly lower than the 160 times for Algorithm 1.

Finally, we summarize the advantages of the NUMA-aware thread-data pinning. (1) Remote memory accesses are reduced. (2) Thread migration is minimized. (3) Thread contention is alleviated. (4) Scalability is enhanced.

3.4 NUMA-aware Partition-based Data Layout

In order to facilitate the pinning between threads and data, an lookup table is constructed. It allows HiPa to identify the coverage of each thread on the graph data. As depicted in Fig. 3, it is a 2-layer tree that records the range of partitions permitted for every thread, and subsequently, the range of vertices covered by each partition. This table is globally accessible for all threads.

Furthermore, HiPa adopts the edge compressing technique proposed by the partition-centric work [21]. As previously discussed, the inter-edge, whose source and destination vertices reside in two different cores, induces expensive memory traffic. To reduce the cross-core communication, the inter-edges pointing to multiple vertices inside the same partition are *compressed* into single inter-edge. After the data is transferred via the compressed edge, it is then locally propagated to the destination vertices.

For the instance of Fig. 4, originally there are two inter-edges, $(v1, v6)$ and $(v1, v7)$, incurring two cross-core messages that carry the updated data from $v1$. Since the two inter-edges share the same source vertex, carry the same data and also reach to the same destination partition, it is redundant to keep both them. Hence, they are compressed into a single inter-edge, holding the data from one vertex to one *partition*, e.g, $(v0, p1)$. After received, this message is decoded and locally propagated to multiple vertices via the intra-edges $(p1, v6)$ and $(p1, v7)$ inside the destination partition [21].

However, unlike the predecessor, HiPa ensures the contiguous virtual address space for the graph data. Based on the partitioning result, graph vertices, edges and attributes (e.g., rank values and vertex degrees) are subdivided into discrete physical pages on different NUMA node. Since discrete address space (e.g., two arrays allocated on two NUMA nodes) requires additional overhead for indirect accesses, the attributes are mapped to contiguous virtual address to seamlessly access cross-node data. Also, within the same NUMA node, vertices as well as edges are placed in the contiguous physical (and virtual) address spaces, though they might belong to different partitions. This reduces the cost for a thread to iteratively

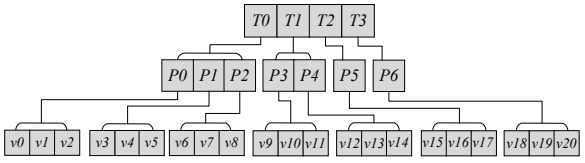

Figure 3: 2-level hierarchical lookup Table. The first level records the partition range for each thread, and the second level records the vertex range for each partition.

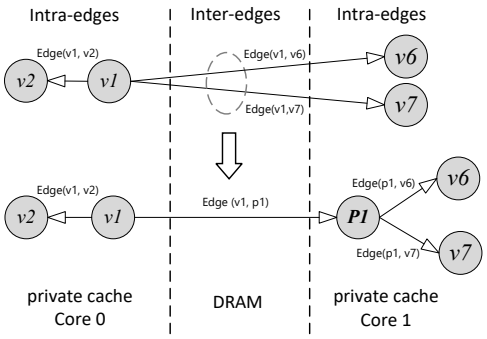

Figure 4: Scatter via inter-edges and intra-edges

process over multiple partitions, and avoid the risk of memory leaks due to multi-dimensional dynamic memory allocation[1].

4 EVALUATION

In this section, we evaluate the effectiveness of HiPa by comparing the performance of PageRank implemented under different methodologies/frameworks. Also, extensive experiments are conducted in order to study its impact on the multicore systems.

4.1 Experimental Setup

We perform experiments on a multicore machine with two Intel Xeon Silver 4210 processors (i.e., NUMA nodes). Each processor consists of 10 physical cores (contributing to 20 logic cores) and equipped with a 128GB local DRAM. The private L1 and L2 sizes per core are 64KB and 1MB respectively, and the shared L3 cache size is 13.75MB. The data types for vertices, edges and PageRank value are set to 4 bytes.

The code of HiPa is written in C++ and compiled using g++ 10.2.0 with optimization level O3. The partition size is empirically decided to 256KB (see Section 4.5) and 40 threads are fully utilized for multithreading. The results relating to execution time are reported using the average value over 5 executions, each of which runs for 20 iterations. To neutralize the effects of preprocessing on memory performance, such as loading graphs from disk, the experiments regarding memory and cache utilization are executed for 60 iterations.

Frameworks: We use the source codes of state-of-the-art frameworks, Polymer [38] and GPOP [20], for comparison. Polymer follows the vertex-centric paradigm, and is particularly designed for NUMA machines. GPOP follows the partition-centric paradigm. To our best knowledge, GPOP is the framework providing the fastest execution of PageRank [20]. Following authors' instruction, we set the partition size to be 1M and only use 20 threads for GPOP. However, many frameworks are designed with a trade-off between productivity (i.e., ease of use) and performance [31]. It is often possible for framework-based implementations to perform worse than well-tuned hand-optimized codes (see Section 4.2).

Hand-coded implementation: To provide a fair comparison, we use two hand-coded optimized variants of PageRank. One is a pull-based vertex-centric version (v-PR), where each vertex pulls the value from its in-neighbors for accumulation. This enables all columns of an adjacency matrix to be traversed asynchronously in parallel without storing the partial sum. The other variant is partition-centric PageRank (p-PR) with finely-tuned parameters, e.g., partition size 256KB and 20 threads. The sensitivity analysis regarding these parameters are provided in Section 4.4 and 4.5. We re-implement the source code offered by [21] with enhancement in memory safety. Additionally, Neither v-PR nor p-PR applies NUMA-awareness.

[1]we find such issue in [21] and report it to the authors.

Table 1: Graph Descriptions (K: Thousand, M: Million, B: Billion). Intra and inter stands for the intra-edges and the inter-edges per partition if size equals 1MB.

Graphs	Descriptions	#Vertices	#Edges	Intra	Inter
journal	Live Journal[22]	4.8M	68.5M	30.8K	7.9M
pld	Pay-Level-Domain[26]	42.9M	0.6B	7.2K	1.6M
wiki	Wiki Links[17]	18.3M	0.2B	74.9K	0.5M
kron	Synthetic Graph[4]	67M	2.1B	11.3K	2.8M
twitter	Twitter Follower[18]	41.7M	1.5B	10.5K	2.3M
mpi	Twitter Influence[7]	52.6M	2.0B	0.2M	1.6M

Six graph datasets with varied features are used for performance evaluation. *journal*, *twitter* and *mpi* are social networks illustrating the follower relationships among users. *pld* and *wiki* are hyperlink graphs extracted from the web pages by web crawlers. *kron* is generated by Graph500 Kronecker generator with scale of 23 [4]. Their statistic information is summarized in Table 1. In this work, we select the out-degree as the basis for graph partitioning.

4.2 Execution Time

Table 2 presents the execution time of PageRank following different implementing methodologies. We observe that HiPa consistently outperforms other methodologies on all graphs. It achieves the highest speedup against Polymer on graph *kron* by 10.65×. For p-PR, v-PR and GPOP, HiPa outperforms them by up to 1.45×, 4.56×, 3.62× respectively.

On the majority of graphs, the hand-coded implementations (e.g., HiPa, p-PR and v-PR) provide better performance than the framework-based ones (e.g, GPOP and Polymer) due to lightweight designs. For example, graph frameworks typically employ a $frontier$ for recording the active vertices in each iteration. It is used to avoid unnecessary accesses to the inactive vertices that do not participate in computation. However, the frontiers are completely redundant to PageRank since every vertex is processed in every iteration. In GPOP, this additional layer of can be disabled for PageRank. Hence, we only report the performance of simplified GPOP without frontier for evaluation in this paper, such as in Table 2.

Table 2: Execution time (in seconds) of PageRank with various implementations.

	HiPa	p-PR	v-PR	GPOP	Polymer
journal	**0.31**	0.41	0.54	1.14	1.72
pld	**2.43**	3.37	8.44	4.18	22.27
wiki	**1.74**	1.80	1.96	3.90	4.63
kron	**7.20**	10.06	32.82	11.29	76.62
twitter	**8.43**	9.83	12.09	14.91	41.06
mpi	**13.93**	17.54	24.41	33.90	64.00

Also, on the same design basis (e.g., hand-coded or framework-based), the partition-centric paradigm substantially outperforms its vertex-centric counterpart. For instance, HiPa and p-PR are faster than v-PR; GPOP is faster than Polymer. The acceleration results from the aggressive memory reduction promoted by the partition-centric paradigm, which is to be discussed in the next section.

Lastly, the overhead introduced by HiPa consists of graph partitioning and NUMA-aware data binding, but excludes the loading of graphs. The overheads for graphs *journal*, *pld*, *wiki*, *kron twitter*, *mpi* are 0.22s, 1.62s, 0.66s, 5.17s, 5.50s, 8.52s respectively. On average, they can be amortized by 12.7 iterations of PageRank in HiPa, while the normalized overheads for GPOP and p-PR are 9.61 and 12.44 iterations respectively. Plus the overheads, HiPa still delivers better end-to-end performance than other implementations on most of graphs, except v-PR on graphs *wiki* and *twitter*.

4.3 Memory Utility

With the optimizations of in-core computing and edge compressing, we expect HiPa to incur insignificant memory consumption compared with the vertex-centric methodologies. Moreover, in favor of the NUMA architecture, the remote memory traffic generated by HiPa is to be minimized as well.

Fig. 5 illustrates the memory accesses normalized by the number of edges in graphs. We refer to the memory accesses per edge as MApE. The percentages of remote MApE are significantly reduced in NUMA-aware designs, including HiPa and Polymer, which are 13.80% and 10.11% respectively. As for the NUMA-oblivious designs, such as p-PR, v-PR, and GPOP, the percentages of MApE are remarkably higher, which are 48.92%, 50.86%, and 53.00% respectively. Among all implementations, HiPa achieves the least number of *remote* memory accesses. The reduction in remote memory accesses allows HiPa to perform as the fastest methodology.

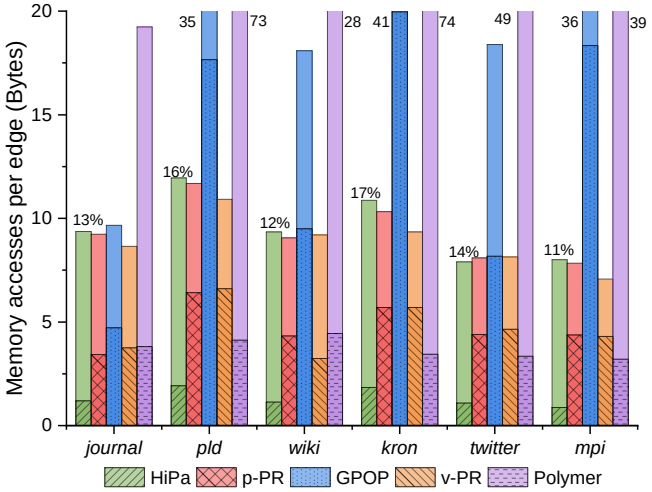

Figure 5: Memory accesses (normalized with graph edges) of PageRank in different implementing methodology. The total bar is the total accesses including remote and local memory accesses. The lower, shadowed bar segment is the remote memory accesses. The ratio of remote access for HiPa is denoted on top the total bar.

The partition-centric paradigm, deployed by HiPa, p-PR and GPOP, effectively reduces the memory traffic during graph processing. On average of all graphs, HiPa, p-PR and GPOP generate 9.57, 9.37 and 8.89 MApE respectively. The results are substantially lower than those of Polymer and v-PR, which are 26.66 and 47.31.

Among the partition-centric methodologies, GPOP produces the fewest total memory accesses, because of its largest partition size (i.e., 1MB) in comparison with HiPa and p-PR (i.e., both 256KB). The larger a partition, the better the compression (see Section 4.5). HiPa generates slightly higher memory accesses than p-PR, which are 9.57 vs. 9.37 MApE, due to the overhead of NUMA-aware optimizations. However, the remote memory accesses of p-PR account for 48.92%, which is notably higher than HiPa's 13.80%. As a result, HiPa delivers a considerably faster speed.

Polymer achieves the lowest *ratio* of remote memory accesses on the average of all graphs, which is 10.11% and even lower than HiPa's 14.17%. Nevertheless, the total memory accesses of Polymer is too high to justify the reduction in remote accesses. The inefficiency of Polymer (and also p-PR) results from the intrinsic disadvantages of vertex-centric paradigm. Suffering from atomic operations, low graph locality and irregular memory accesses [20], the performance of Polymer is severely limited.

4.4 Scalability

The scalability of HiPa as well as other methodologies is examined on graph *journal* by varying the number of threads for parallel processing. For each methodology, the results are normalized by its execution time using 40 threads to locate the peak-performance point (i.e., the lowest point). As depicted in Fig. 6, all plots finally converge to 1.

HiPa and other two vertex-centric implementations, Polymer and v-PR, exhibit high scalability. As the number of threads increase, the their performances show the tendency of improvement. When scaling from 2 to 20 threads, there is a sharp decrease in execution time. The scaling behavior become weak as more threads

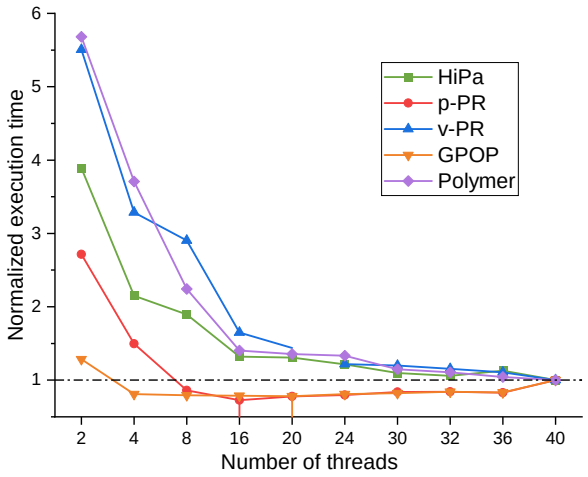

Figure 6: Execution time (normalized by the performance of 40 threads) with varied numbers of threads on *journal*.

are employed, e.g., from 20 to 40. The best performance is achieved when all 40 threads are utilized.

By contrast, the two partition-centric methodologies, p-PR and GPOP, demonstrate relatively poor scalability. Their best performance is obtained when using 16 threads and 20 threads respectively. When 40 threads are fully utilized, the execution time will be nearly doubled. This is because the conventional partition-centric methodologies cause high consumption of memory bandwidth when the cores load and offload partitions [21]. The bandwidth is saturated with approximately half of total threads [20]. Any further addition of threads would only aggregate the contention on bus and cache resources, and thus deteriorates the performance.

Though HiPa is built under the partition-centric paradigm, it still scales well on the multicore system because it forces two-level constrains for a thread to access a partition. Firstly, the NUMA-awareness of HiPa limits about 85% of memory traffic to occur within the local memories. It effectively avoids cores contending for interconnect bus between different NUMA nodes. Inside each NUMA node, the data of every partition are further pinned to a thread, so the conflicts of threads on the same partition are eliminated. As a result, the limitation of conventional partition-centric methodology is overcome, and hence the scalability of HiPa is considerably enhanced.

4.5 Sensitivity Analysis of Partition Size

The partition size plays an important role in the the effectiveness of partition-centric paradigm. Fig. 7 illustrates the impact of varied partition sizes on the execution time and LLC. In general, as the partition sizes become larger, the LLC hits and hit ratios continuously grow, which indicates the data accesses are increasingly concentrated in LLC. The best performance of HiPa is achieved when the partition size equals (or slightly less than) a quarter of L2 cache size, i.e., 256KB.

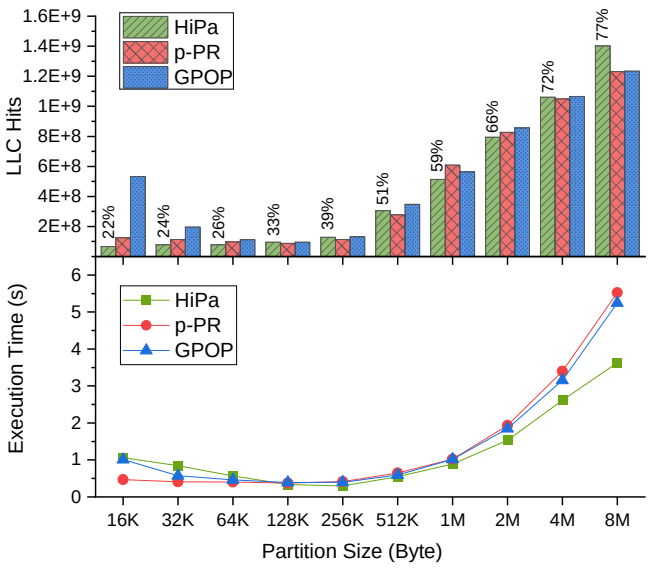

Figure 7: LLC cache hits (upper) and execution time (lower) with different partition sizes on graph *journal*. LLC hit ratios of HiPa are denoted.

When deciding the proper size for graph partitioning, there exists a tradeoff between cache locality and inter-edge compression. If the partition size is small (e.g., size $\leq$ 256KB), high spatial locality is ensured as random accesses are confined within the partitions in private caches. On the other hand, (too) small partitions, such as size=16KB, preserve a large number of inter-edges, which leads to high workloads and substantial memory traffic. Notice that GPOP incurs high LLC accesses, when the number of partitions is exceptional high with size = 16KB. This is because GPOP requires additional data fields for each partitions, such Flags, State [20].

As the partition size increases from 16KB to 256KB, inter-edges are continuously compressed, and workloads are accordingly reduced. Hence, the execution time keeps a steady decline. Nevertheless, once the size exceeds the threshold of 256KB, the intra-edge of a partition spill over from L2 cache to LLC. Cache spatial locality is not promised anymore. As Fig. 7 presents, there is a drastic surge in LLC hits (and hit ratios) for all partition-centric implementations when the partition size grows from 256KB to 8MB. Since LLC provides the highest latency in cache hierarchy, the execution is significantly decelerated.

Our experimental results show that the optimal partitioning size, or alternatively the threshold, is 256KB, which is a quarter of L2 size. This finding contrasts with the results in prior partition-centric works [20, 40]. In their settings, the partitions are usually set to fit into LLC or L2, which are 20MB or 1MB in our environment.

The conflict mainly originates from two factors. First of all, as mentioned beforehand, besides the vertex subset, a partition also contains an edge subset (including intra- and inter-edges). Their sizes are unequal for each partition. Both the edge subset and the buffer require additional memory and cache space for writing and reading. Hence, to allow a partition to be privately accommodated by a core, the size of a vertex subset is supposed to be smaller than the L2 cache size, so that the edge subset and buffer are co-located with the vertex subset within the same L2 cache.

Secondly, *the micro-architectures of processors are different.* our experiments in this paper are conducted on Intel 6th generation CPUs based on 14nm Skylake micro-architecture. All prior partition-centric methodologies were proposed according to the Intel 22nm Haswell micro-architecture, which is 2 generation older than Skylake. Architectural upgrades are deployed on Skylake, a major part of which relates to the cache hierarchy. For instance, in Haswell, the L2 cache is 256KB per core and the LLC is shared with 2.56MB per core. The LLC is inclusive, which guarantees every data exists in L2 cache must also exist in LLC. Skylake provides larger L2 cache of 1MB per core, and smaller shared LLC of 1.375MB per core. The LLC in Skylake is non-inclusive, which allows data in memory to be directly loaded to the L2 cache without through the LLC.

The new hardware characteristics lead to a variability in determining the size of cache-able partitions. To find the proper size on different micro-architectures, we extend the sensitivity analysis by examining the all partition-centric methodologies on Haswell and Skylake. All graphs except *kron* and *mpi* are tested, because these two exceed the memory capacity of our Haswell machine - Intel Xeon CPU E5-2667 with two NUMA nodes, 64GB main memory and 256KB L2 caches. To clearly present the trends of change around the threshold point, we normalize the execution time of each graph by the result of a particular size, for example, 128K on Haswell

Table 3: Normalized execution time using different partition sizes on Haswell and Skylake micro-architectures

Method.	Skylake				Haswell			
	64K	128K	**256K**	512K	64K	**128K**	256K	512K
HiPa	1.04	1.04	1.00	1.25	1.08	1.00	1.18	1.29
p-PR	0.85	0.89	1.00	1.30	1.00	1.00	1.08	1.25
GPOP	1.34	1.03	1.00	1.25	1.18	1.00	0.98	1.13
Average	1.08	0.99	1.00	1.27	1.09	1.00	1.08	1.22

and 256K on Skylake. The normalized time of all tested graphs are averaged and then listed in Table 3.

As Table 3 shows, the optimal partition size on Skylake strikes at 256KB for HiPa and GPOP, which enables the fastest execution. For p-PR, the best performance is at 128KB. The performance difference between 256KB and 128KB is minute, since the average of them are 1.00 and 0.99 respectively. As for the Haswell, the optimal sizes for three methodologies all hit at 128KB. When the size become larger than 256KB, both Skylake and Haswell are sharply decelerated due to the spillover of vertices into LLC.

In addition, the performance of HiPa deteriorates when deployed on single NUMA node with 20 threads, because all contentions are concentrated on one node. For instance, on graph *journal*, single-node HiPa costs 0.44s to finish 20 iterations of PageRank. With the same number of threads, 2-node HiPa, p-PR and GPOP cost 0.39s, 0.41s and 1.14s respectively to end the executions. Nevertheless, we expect the performance of HiPa to be further boosted in 4-node and 8-node machines.

5 RELATED WORK

Graph partitioning. The partitioning of a graph can be considered equivalent to the task allocation of a graph parallel computing system. There exist two essential types of lightweight partitioning schemes: vertex-based and edge-based. In the former case, vertices are assigned to disjoint segments. It is an intuitive method of partitioning and adopted in early graph frameworks such as GraphLab [23] and Pregel [24]. As for the latter, edges of a vertex are detached to different segments, and the vertex is accordingly replicated. This scheme gains its popularity in edge-centric frameworks, such as GridGraph [44] and X-stream [30]. For better performance, hybrid partitioning approaches are proposed to take advantages of the two, for example, PowerLyra [8] and CUBE [39]. The partitioning methodology proposed in our work falls into the hybrid category.

Moreover, numerous sophisticated partitioning algorithms are developed in order to obtain high-quality results, for example METIS [14, 15], KaHIP [27] and PULP [33]. Those algorithm extensively analyze the internal structure of the graphs. High-quality graph partitioning facilitates workload balance for parallel computation and reduces the communication volume between working units. As a result, it leads to a substantial performance boost for downstream applications [34].

Nevertheless, compared with the lightweight counterpart, sophisticated partitioning methods incurs high overhead and complicated implementation procedure. To this end, high-performance partitioning algorithms are designed, such as ParMETIS [16], ParHip [28] and XtraPULP [35]. In these partitioners, various optimization techniques are exploited, for instance, hierarchical partitioning [28] and distributed parallelism [35].

HiPa absorbs the nutrition from both lightweight and sophisticated partitioning strategies. It leverages the prior knowledge of vertex degrees for graph partitioning without investigating the structural property, so the analysis is simplified. For high quality and high performance, the hierarchical partitioning methodology is deployed in accordance with the memory hierarchy of multicore systems. As a result, HiPa achieves remarkable advantages in efficiency and effectiveness over contemporary works.

Graph analytics frameworks. HiPa is inspired by the graph frameworks designed for *shared-memory* multicore machines, such as Ligra [32], Ploymer [38] and GPOP [20]. Besides the shared-memory, there are a variety of hardware platforms targeted for the design of graph frameworks. GraphChi [19] and GridGraph [44] are proposed to handle large-scale graphs on a *disk-based* platform whose computing resource is rather limited, for instance, a laptop with 16 GB memory. Medusa [41] and Gunrock [37] exploits the massive parallelism from *GPU* to accelerate graph analytics.

When the graph data size is too large (e.g., terabytes) to fit into a single machine, the graph processing scales out to a cluster of *distributed* machines, for instance, Pregel [24] and GraphX [11]. The distributed platform imposes considerable overheads for inter-machine communication. Therefore, a major focus for distributed graph frameworks is to reduce network traffic, such as CUBE [39], Gemini [44]. However, the scalability of many distributed graph processing systems is established at the sacrifice of efficiency. Compared with their shared-memory counterparts, they oftentimes yield unsatisfactory cost efficiency as well as performance [43]. Furthermore, a study [25] showcases that, with optimizations, even single-threaded implementations are able to outperform many graph frameworks running on a cluster.

6 CONCLUSION AND FUTURE WORK

In this paper, we present HiPa, a hierarchical partitioning methodology that boosts the performance of PageRank on multicore systems. HiPa deploys two-level partitioning strategy based on the memory-cache architecture of the systems. Firstly, it subdivides the graph data into coarse-grained subsets according to the NUMA characteristics of the memory. Then, the graph subset in each NUMA node is refined into cache-able disjoint partitions to enable in-core computing. Furthermore, to co-locate computations and data, new data layout is designed and pinned to threads.

Extensive experiments are conducted to evaluate the performance of HiPa. In comparison with the state-of-the-art frameworks as well as hand-optimized implementations, it achieves the fast execution speed and the least remote memory accesses. Moreover, HiPa exhibits high scalability, because it effectively alleviates the thread contention on hardware resources, e.g., bus and caches. Also, we find that the optimal partitioning size for Skylake micro-architecture equals a quarter of L2 cache size, while for Haswell it is half of L2 cache size.

Albeit we use PageRank to demonstrate the effectiveness of HiPa, the methodology of HiPa can be deployed to more generic use scenarios. In the future work, we will explore the design space for

the extension of HiPa to diverse algorithms, e.g., SpMV, PageRank Delta and BFS.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China (No. 2018YFB1003505)

REFERENCES

[1] 2011. Intel® VTune™ Profiler Performance Analysis Cookbook . https://software.intel.com/content/www/us/en/develop/documentation/vtune-cookbook/top/tuning-recipes/os-thread-migration.html. [Online; accessed 22-April-2021].

[2] 2011. Optimizing Applications for NUMA. https://software.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html. [Online; accessed 21-April-2021].

[3] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav Marathe, and Dorothea Wagner. 2006. Implementations of routing algorithms for transportation networks. In *DIMACS Workshop on Shortest-Path Challenge*. Citeseer.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[5] Scott Beamer, Krste Asanović, and David Patterson. 2017. Reducing pagerank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 820–831.

[6] Peter J Carrington, John Scott, and Stanley Wasserman. 2005. *Models and methods in social network analysis*. Vol. 28. Cambridge university press.

[7] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P Gummadi. 2010. Measuring user influence in twitter: The million follower fallacy. In *fourth international AAAI conference on weblogs and social media*.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–15.

[9] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–13.

[10] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* 29, 4 (1999), 251–262.

[11] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[12] R Intel. 2014. Intel 64 and ia-32 architectures optimization reference manual. *Intel Corporation, Sept* (2014).

[13] N Kannan and S Vishveshwara. 1999. Identification of side-chain clusters in protein structures by a graph spectral method. *Journal of molecular biology* 292, 2 (1999), 441–464.

[14] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[15] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[16] George Karypis and Vipin Kumar. 1999. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review* 41, 2 (1999), 278–300.

[17] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. 1343–1350.

[18] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[19] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a {PC}. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 31–46.

[20] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2020. GPOP: A Scalable Cache-and Memory-efficient Framework for Graph Processing over Parts. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–24.

[21] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2018. Accelerating pagerank using partition-centric processing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 427–440.

[22] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.

[23] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012).

[24] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[25] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*.

[26] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science* 1 (2015).

[27] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2014. Partitioning complex networks via size-constrained clustering. In *International Symposium on Experimental Algorithms*. Springer, 351–363.

[28] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2625–2638.

[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[30] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[31] Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 979–990.

[32] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[33] George M Slota, Kamesh Madduri, and Sivasankaran Rajamanickam. 2014. PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, 481–490.

[34] George M Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2017. Partitioning trillion-edge graphs in minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 646–655.

[35] George M Slota, Cameron Root, Karen Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. 2020. Scalable, Multi-Constraint, Complex-Objective Graph Partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2789–2801.

[36] Jiawen Sun, Hans Vandierendonck, and Dimitrios S Nikolopoulos. 2017. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*. 1–10.

[37] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[38] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 183–193.

[39] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the hidden dimension in graph processing. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 285–300.

[40] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.

[41] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.

[42] Shijie Zhou, Kartik Lakhotia, Shreyas G Singapura, Hanqing Zeng, Rajgopal Kannan, Viktor K Prasanna, James Fox, Euna Kim, Oded Green, and David A Bader. 2017. Design and implementation of parallel pagerank on multicore platforms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–6.

[43] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 301–316.

[44] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*. 375–386.