NUMA-Aware Graph-Structured Analytics

Kaiyuan Zhang Rong Chen Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, China

Abstract

Graph-structured analytics has been widely adopted in a number of big data applications such as social computation, web-search and recommendation systems. Though much prior research focuses on scaling graph-analytics on distributed environments, the strong desire on performance per core, dollar and joule has generated considerable interests of processing large-scale graphs on a single server-class machine, which may have several terabytes of RAM and 80 or more cores. However, prior graph-analytics systems are largely neutral to NUMA characteristics and thus have suboptimal performance.

This paper presents a detailed study of NUMA characteristics and their impact on the efficiency of graph-analytics. Our study uncovers two insights: 1) either random or interleaved allocation of graph data will significantly hamper data locality and parallelism; 2) sequential inter-node (i.e., remote) memory accesses have much higher bandwidth than both intra- and inter-node random ones. Based on them, this paper describes Polymer, a NUMA-aware graph-analytics system on multicore with two key design decisions. First, Polymer differentially allocates and places topology data, application-defined data and mutable runtime states of a graph system according to their access patterns to minimize remote accesses. Second, for some remaining random accesses, Polymer carefully converts random remote accesses into sequential remote accesses, by using lightweight replication of vertices across NUMA nodes. To improve load balance and vertex convergence, Polymer is further built with a hierarchical barrier to boost parallelism and locality, an edge-oriented balanced partitioning for skewed graphs, and adaptive data structures according to the proportion of active vertices. A detailed evaluation on an 80-core machine shows that Polymer often outperforms the state-of-the-art single-machine graph-analytics systems, including Ligra, X-Stream and Galois, for a set of popular real-world and synthetic graphs.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming

Keywords Graph-structured Analytics; Non-uniform Memory Access (NUMA)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3205-7/15/02...$15.00
http://dx.doi.org/10.1145/2688500.2688507

1. Introduction

Many machine learning, data mining and scientific computation can be modeled as graph-structured computation, resulting in a new application domain called graph analytics. Nowadays, it has been widely adopted in areas including social computation, web search, natural language processing, and recommendation systems [7, 31, 40, 44, 47]. The strong desire for efficiency has driven the design of a number of distributed graph analytics frameworks such as Pregel [35], GraphLab [22, 32] and Cyclops [13].

The multicore evolution has led to drastic increases in CPU core counts and memory sizes. Actually it is now not uncommon to see server-class machines with 80 or more cores and several terabyte of memory. This has led to the design and implementation of several recent single-machine graph-analytics systems to achieve more performance per core, dollar and joule, due to the significantly cheaper cost and lower latency in communication. Examples include GraphChi [28], Ligra [43], X-Stream [42], and Galois [39], which can usually process graphs on a single machine with hundreds of billions of edges.

On the other hand, commodity multicore machines have shifted into cache-coherent NUMA (cc-NUMA) architectures, where the latency of memory accesses on remote chips is much higher than that on local ones [18]. Though the NUMA effects may have significant impact on the efficiency of graph analytics, existing systems are largely NUMA-oblivious but focus on other aspects such as improving out-of-core accesses [28], selecting appropriate execution modes [43], supporting sophisticated task scheduler [39], and reducing random operations on edges [42].

In this paper, we make a comprehensive study on the characteristics of commodity NUMA machines and how they affect the efficiency of existing single-machine graph analytics systems. Though conventional wisdom is that remote memory accesses have higher latency and lower throughput than local ones, we quantitatively show that sequential remote accesses have much higher bandwidth than both random *local* and random remote ones (2.92X and 6.85X on our tested machines). Further, either interleaved or centralized allocation of graph data on existing graph analytics systems causes poor data locality and limited parallelism.

Based on the above observations, this paper describes Polymer[1], a NUMA-aware graph-analytics system that inherits the scatter-gather programming interface from Ligra [43] but is built with several NUMA-aware designs. The key of Polymer is to minimize both random and remote memory accesses by optimizing graph data layout and access strategies.

Polymer adopts a general design principle for NUMA machines by co-locating graph data and computation within NUMA-nodes as much as possible, with the goal of reducing remote memory accesses and balancing cross-node interconnect bandwidth. Specifi-

[1] The source code and a brief instruction of how to use Polymer are at `http://ipads.se.sjtu.edu.cn/projects/polymer.html`

cally, Polymer differentially allocates and places graph data according to their access patterns. For *graph topology* data such as vertices and edges that are always accessed only their own threads, Polymer uses corporative allocation by letting an accessing thread to allocate the memory in its local memory node, and thus eliminating remote accesses. Second, for *application-defined data* (like the *ranks* of a web page in PageRank) whose memory locations are static but data will be updated dynamically, though corporative allocation may eliminate a lot of remote accesses, there are still inevitable remote accesses due to frequent exchanges of application-defined data during computation. Hence, Polymer allocates such data with contiguous virtual addresses, but distributes actual physical memory frames to the NUMA-node of the owning thread. This makes it seamless to access cross-node data. For mutable graph *runtime states* such as the current active vertices, as they are dynamically allocated in each iteration, Polymer allocates and updates such data in a distributed way but accesses it through a global lookup table to avoid contention.

Graph analytics has been long recognized to have many random access and poor data locality [33]. Based on the observation that *sequential inter-node (i.e., remote) memory accesses have much higher bandwidth than both intra- and inter-node random ones*, Polymer borrows the idea from distributed graph systems [22] by replicating vertex data across NUMA-nodes in a lightweight way. (This essentially follows the philosophy of current multicore OS designs by treating a large-scale machine as a distributed system [5]). Specifically, a vertex only conducts computation on edges within the local NUMA-node and uses its replicas in other NUMA-node to initiate the computation on other edges. Unlike distributed graph systems, Polymer does not distribute application-defined data but still applies all updates to a vertex within a single copy of application-defined data.

Polymer has three optimizations to improve scheduling on NUMA machines and handle different properties of graphs. First, being aware of the hierarchical parallelism and locality in NUMA machines, Polymer is extended with a hierarchical scheduler to reduce the cost for synchronizing threads on multiple NUMA-nodes. Second, inspired by vertex-cuts [22] from distributed graph systems, Polymer improves the balance of workload among NUMA-nodes by evenly partitioning edges rather than vertices for skewed graphs [20]. Finally, as most graph algorithms converge asymmetrically, using a single data structure for runtime states may drastically degrade the performance, especially for traversal algorithms on high-diameter graphs, Polymer adopts adaptive data structures boost performance.

We have implemented Polymer and several typical graph applications, which comprise about 5,300 lines of C++ code. Our evaluation on both an 80-core Intel machine and a 64-core AMD machine with a number of graph algorithms using both real-world and synthetic graph datasets shows that Polymer often outperforms existing state-of-the-art graph-parallel systems and scales well in terms of the number of sockets, due to its graph-aware data layout, NUMA-aware computation, reduced synchronization and traversal cost, and improved load balance.

This paper makes the following contributions:

- A comprehensive analysis that uncovers several NUMA characteristics and issues with existing NUMA-oblivious graph analytics systems (Section 3).
- The Polymer system that exploits both NUMA- and graph-aware data layout and memory access strategies (Section 4)
- Three optimizations that improve global synchronization efficiency, load balance and data structure flexibility (Section 5).
- A detailed evaluation that demonstrates the performance and scalability benefit of Polymer (Section 6).

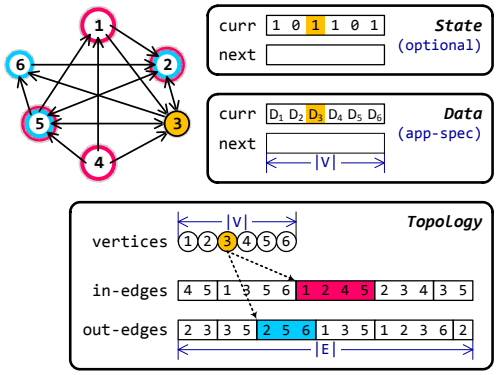


Figure 1. The in-memory data structure of scatter-gather model.

2. Background

2.1 Graph Analytics

In-memory data structure: It mainly consists of three parts: graph topology data, application-defined data and graph runtime states. As shown in Figure 1, existing systems commonly split graph topology into separately ordered arrays for vertices and edges. The array for `in-edges` of all vertices are partitioned by their target vertex and storing the source vertices. Similarly, the array for `out-edges` of all vertices are partitioned by their target vertex and storing the source vertices. Both of them are optional. The metadata of all `vertices` are kept in one array, in which each one stores the start of its in-edge and out-edge partitions, and also maintains the in-degree and out-degree. Two arrays store two versions of the application-defined vertex data separately. One (i.e., `current`) maintains the value computed in the previous iteration, and the other (i.e., `next`) keeps the update value computed in current execution. The application-defined edge data can be handled similarly if necessary. The optional runtime states, i.e., active or not, of vertex for `current` and `next` execution are also stored in two arrays respectively to support dynamic computation [32].

Graph computation: Existing graph analytics systems follow a scatter-gather iterative computation model [22], which abstracts the computation in each iteration as consecutive *scatter* and *gather* phases. The scatter phase propagates the current value of a vertex to its neighbors along edges, while the gather phase accumulates values from neighbors to compute the next value of a vertex. There are two main approaches to implementing the scatter-gather model, namely *vertex-centric* [35] and *edge-centric* [42].

The **vertex-centric** system such as Ligra [43] iterates over all active vertices in both scatter and gather phases. The propagation of vertex data in scatter phase can be implemented in either push or pull mode. The left part of Ligra system in Figure 2 illustrates the execution flow and in-memory data access patterns under the two modes for vertex 3.

In the *push* mode, the worker thread first scans the `current` state array (`SEQ|R`) to identify an active vertex in `vertices` array (`SEQ|R`), and then obtains its neighbors through the `out-edges` array (`SEQ|R`). Further, the worker thread pushes the value of the active vertex in the `current` data array (`SEQ|R`) to its neighbors in the `next` data array (`RAND|W`), and sets the `next` state array (`RAND|W`). For example, the value of vertex 3 will be pushed to its neighboring vertex 2, 5 and 6 along out-edges, which are labeled bold arrow on the sample graph.

In the *pull* mode, for each vertex in the `vertices` array (`SEQ|R`), the worker thread first obtains its active neighbors through the `in-edges` (`SEQ|R`) and `current` state arrays (`RAND|R`). Further, the worker thread pulls the value from active neighbors in the

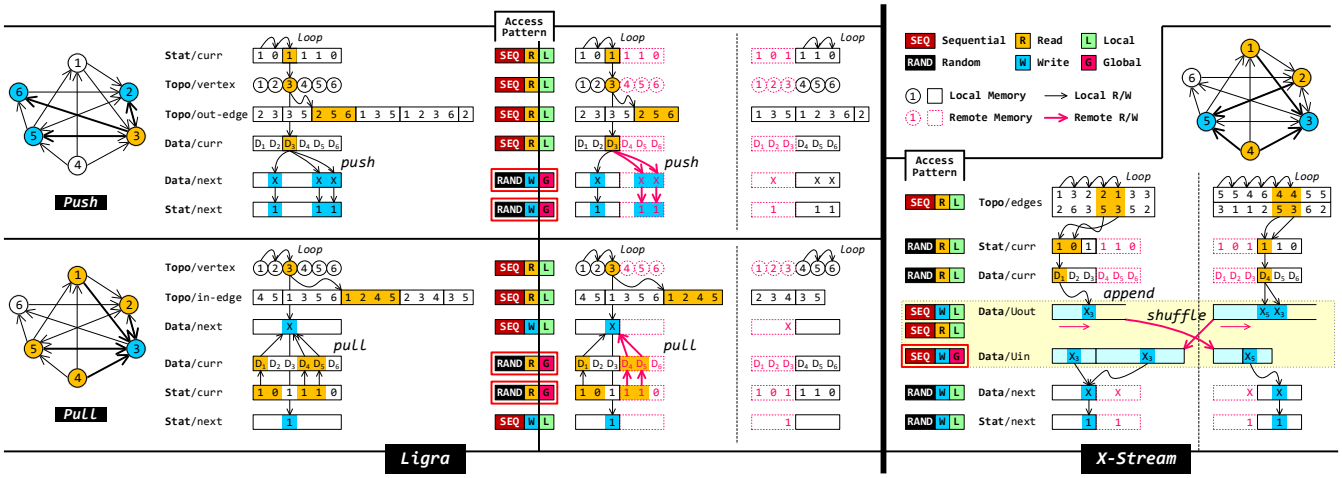
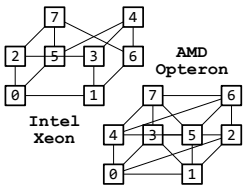


Figure 2. The in-memory data layout and execution flow of existing graph-analytics systems for a sample graph. The pattern to access in-memory data is labeled by SEQ/RAND, R/W and L/G. Yellow indicates source vertex, and blue indicates target vertex.



(a) System topology

Inst.	0-hop	1-hop	2-hop
80-core Intel Xeon machine			
Load	117	271	372
Store	108	304	409
64-core AMD Opteron machine			
Load	228	419	498
Store	256	463	544

(b) Latencies (cycles) on the distance

Figure 3. The characteristics of NUMA machines for experiments.

Access	0-hop	1-hop	2-hop	Interleaved
80-core Intel Xeon machine				
Sequential	3207	2455	2101	2333
Random	720	348	307	344
64-core AMD Opteron machine				
Sequential	3241	2806/2406	1997	2509
Random	533	509/487	415	466

Figure 4. The bandwidth (MB/s) of memory access on the distance.

`current` data array (`RAND|R`) to update the vertex in the `next` data array (`SEQ|W`) and set the `next` state array (`SEQ|W`). For example, the vertex 3 will pull the value of its neighboring vertex 1, 2, 4 and 5 along in-edges to update its value. The inactive vertex 2 will be skipped.

The **edge-centric** system such X-Stream [42] iterates over all edges rather than vertices to avoid random accesses to edges, instead of sequentially accessing them. Further, it introduces an additional *shuffle* phase and "tiling strategy" to improve random access to vertices. As shown in Figure 2, all edges are split by their source vertex into two partitions and the engine processes one partition associated with their source vertices at a time. In the scatter phase, the worker thread iterates over `edges` array (`SEQ|R`) and appends the update from the `current` data array (`RAND|R`) to a list `Uout` (`SEQ|W`) for active edges, which are identified by the state of its source vertex from the `current` state array (`RAND|R`). In the shuffle phase, all updates in the list `Uout` (`SEQ|R`) are re-arranged to the update list `Uin` (`SEQ|W`) by their target vertex. Finally, all updates will be applied to the `next` data array (`RAND|W`) in the gather phase, and the `next` state array (`RAND|W`) will be set simultaneously. For example, in the second partition, the update along the edge from vertex 4 to vertex 3 will be appended to local `Uout` first and then shuffled to the `Uin` of partition 1, and finally written to vertex 3.

2.2 NUMA Characteristics

A commodity NUMA machine consists of several processor nodes (i.e., socket), each of which contains multiple cores and a local DRAM. The nodes are connected by the high-speed interconnect into a cache-coherent system, forming a globally shared memory abstraction to applications.

The distribution of memory to processor nodes leads to non-uniform memory access: the latency of accessing locally-attached cache and memory is significantly lower than those attached to other processor. Further, the latency highly depends on the distance (i.e., hops) between nodes.

To make a quantitative study on the non-uniform feature of commodity NUMA machines, we measure the latency of memory accesses along with distance on both an 80-core (8 sockets X 10 cores) Intel machine and a 64-core (4 sockets X 2 dies X 8 cores) AMD machine. The dies within a socket have a 1-hop distance. The topologies of the two machines are depicted in Figure 3(a)[2].

As shown in Figure 3(b), for the 80-core machine, the latency of load and store on the same node through memory are 117 and 108 cycles respectively, whereas an access over one or two hop(s) is approximately 2 or 3 times more expensive than an access on the same node accordingly[3]. The 64-core machine has a close trend.

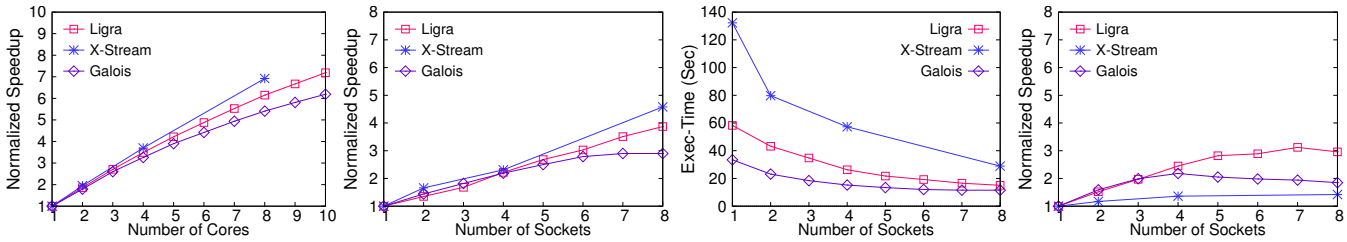
Figure 4 shows the bandwidth of memory access along with distances on the 80-core machine[4]. The bandwidth of remote accesses is unsurprisingly quite lower than that of local access, up to 34% and 57% performance degradation for sequential and random access over two hops respectively. Further, the bandwidth of interleaved access, where the maximum distance between the two cores is 2 hops, is even lower than that of remote access with 1-hop distance. The results on 64-core AMD machine are similar, and the two values for 1-hop distance indicate within a socket or not.

Further, it is a little bit surprising that *sequential remote accesses, have much higher throughput than random ones (both local and remote)*, by factors from 2.92X to 6.85X with increasing distance on 80-core Intel machine (see Figure 4). For example, random local accesses only have 720 MB/s throughput, which is far smaller than sequential accesses with 2-hop distance (2101 MB/s).

[2] Detailed machine configurations can be found in Section 6

[3] The latency is evaluated by `ccbench` [18].

[4] The bandwidth is evaluated by `numademo` [3] with some extensions.



(a) Cores (b) Sockets (c) Runtime (d) Sockets(AMD)

Figure 5. (a) The speedup with the increasing number of cores within one socket. (b) and (c) The speedup and execution time with the increasing number of sockets with fixed 10 cores per socket. (d) The speedup with the increasing number of sockets with fixed 8 cores per socket on the 64-core AMD machine.

3. Challenges and Issues

This section discusses why prior single-machine graph-analytics systems fall shorts on NUMA platforms, by attributing them to two issues: data layout and access strategy.

3.1 Issue 1: Data Layout

By default, Linux employs the "first-touch" policy to bind virtual pages to physical frames locating on a memory node where a thread *first* touches the pages.

Graph analytics usually consists of two stages: graph construction and computation. In existing frameworks, the long-term in-memory data structures, including both graph topology and application-defined data, are allocated and initialized by multiple constructing threads on different NUMA-nodes during the construction stage. Such data is then accessed by graph processing threads bound to such NUMA-nodes at the computation stage. Under the default allocation policy, namely "first-touch", there is mismatch between allocation threads and processing threads, which forms *interleaved* page allocation. This notably hurts locality and degrade performance due to a large number of remote and random memory accesses. As shown in Section 2.2, both remote random memory accesses have much higher latency and throughput than sequential and random ones.

Further, the short-term in-memory data structures, such as runtime states, are allocated and initiated by the main thread at the beginning of each iteration, and accessed by all processing threads from NUMA-nodes in each iteration. Such *centralized* allocation will result in not only excessive remote memory accesses, but also congestion on interconnects and memory controllers [17].

3.2 Issue 2: Access Strategy

For graph analytics, it is inevitable to access remote memory even under ideal data layout, due to the lack of access locality when traversing edges. However, as we observed in Section 2.2, sequential remote accesses have much higher bandwidth than both random local and random remote accesses. Yet, the access pattern (i.e., sequential or random) for the remote memory access is either completely overlooked or restricted in an inefficient way, leading to sub-optimal performance and scalability by prior work.

In **vertex-centric** system such as Ligra [43], even if the local accesses to in-memory data structures have been carefully arranged in a sequential order, the remote accesses to `next` or `curr` data and state arrays for *push* or *pull* mode are coupled with random orders accordingly. Worse even, concurrent remote accesses by threads bound on different processors will further cause congestion on interconnect.

The right part of Ligra system in Figure 2 shows the execution on a 2-node NUMA machine using push and pull modes. Note that each operation is labeled with access pattern, and *local*(`L`) or *global*(`G`) symbol associated for the NUMA case. In the *push* mode, the worker thread randomly pushes the value of the vertex 3 to its neighbors allocated on remote memory nodes (e.g., vertex 5 and 6), and also randomly writes the state array for these neighbors. They are labeled as random, write and global (i.e., `RAND|W|G`). In the *pull* mode, the worker thread randomly pulls the value from active neighbors allocated on remote memory nodes (e.g. vertex 4 and 5) to update vertex 3. They are labeled as random, read and global (i.e., `RAND|R|G`).

In **edge-centric** system such as X-Stream [42], based on "tiling strategy" [10], each core can independently process one partition in both the gather and the scatter phases, and only exchanges data among all of memory nodes in the shuffle phase. Even if the remote access is sequential, the shuffle phase causes additional memory allocation and remote copy overhead. Worse even, it is quite costly to identify the state of edges instead of vertices, since the number of edges is commonly a few tens or hundreds of times than the number of vertices. It will significantly degrade performance and scalability for traversal algorithms (e.g., SSSP), especially for high-diameter graphs like road networks (see Table 3).

The X-Stream system in Figure 2 also illustrates the execution on a 2-node NUMA machine. The updates to target vertices in a set of `Uouts` allocated on different memory node are shuffled to different `Uins` by their target vertices. It is labeled as sequential, write and global (i.e., `SEQ|W|G`). For example, all updates on vertex 3 will be grouped and eventually updated to the `next` data array on the same node.

3.3 Quantitive Performance and Scalability on NUMA

We study the scalability of Ligra, Galois and X-Stream using PageRank algorithm for Twitter follower graph from two dimensions. We increase the number of cores within one socket first and then increase the number of sockets with fixed 10 cores per socket on our 80-core machine[5].

Figure 5(a) shows that existing systems can scale well in terms of number of cores, up to 6.92X using 8 cores on X-Stream. However, none of them has a very good scalability in terms of number of sockets, because of inefficient data layout and access strategy on NUMA machines. As shown in Figure 5(b) and (c), X-Stream has a slightly better scalability (4.58X) but the worst performance due to additional time-consuming shuffling. On the contrary, Galois has currently best performance due to fully optimized infrastructure, while its scalability is very poor (2.90X on 8 sockets). We also reran the experiment on our 64-core AMD machine, and gained a worse scalability (see Figure 5(d)). The performance of X-Stream and Galois even degrades when the number of sockets exceeds 4, since the HyperTransport interconnect in AMD systems can only ensure the distance between two nodes to one hop for at most 4 sockets. In short, existing systems are still largely NUMA-oblivious, resulting in less optimal performance and scalability.

[5] We select sockets with minimized total distances. Since the number of threads in X-Stream must be a power of 2, we evaluate it using up to 8 cores in each socket.

4. NUMA-Aware Graph-Analytics

This section describes Polymer, a NUMA-aware graph analytics system, which provides a vertex-centric programming interface. The key to Polymer is aligning graph-specific data structure and computation with the NUMA characteristics to reduce remote and random memory accesses as much as possible.

4.1 Computation Model and Interface

Polymer follows the typical scatter-gather model by providing two interfaces inherited from Ligra [43]: `EdgeMap` and `VertexMap`. A graph program P in Polymer *synchronously* runs on a directed graph $G = (V, E)$, where V is the vertex set and E is the edge set. For undirected graphs, each of their edges is presented by a pair of directed edges, one per direction. The topology data of vertex v includes the set of in- or out-neighbors ($N_{in}(v)$ or $N_{out}(v)$) and its in- or out-degree ($|N_{in}(v)|$ or $|N_{out}(v)|$). A `SubVertex` type is used to define a subset of vertices $U \subseteq V$. Like most prior systems, Polymer assumes that the graph topology is immutable during graph computation; how to extend Polymer to support mutable topology is our future work.

The following describes the main interface of Polymer.

1. `EdgeMap`(G, A, F) : *SubVertex*
   For the input directed graph $G = (V, E)$, `EdgeMap` applies the application-define function F to all edges whose source vertices belong to an active vertex set A. More precisely, for an active edge set $E_{active} = \{(v,u) \in E \mid v \in A\}$, the function F is applied to each edge in E_{active}, and returns an updated vertex set `SubVertex`: $R = \{u \mid (v,u) \in E_{active} \wedge F(v,u) = true\}$.
2. `VertexMap`(G, A, F) : *SubVertex*
   `VertexMap` applies the application-define function F to all vertices in an active set A, and also returns an updated vertex set `SubVertex`: $R = \{v \in A \mid F(v) = true\}$.

Algorithm 4.1 illustrates the pseudocode of PageRank [7] in Polymer. It assumes that the graph topology data is stored in G. D_{curr} is used to store the current rank of vertices and initiated to $1/|V|$ (line 1), while the new rank of vertices will be stored to D_{next}, which is initiated to 0 (line 2). The `PREdgeF` function (line 3-6) passed to `EdgeMap` atomically pushes the current rank of a source vertex v to the target vertex t by `AtomicAdd` (line 4). The `PRVertF` function (line 7-12) passed to `VertexMap` normalizes the sum of updates from in-neighbors by multiplying a factor of 0.85, and adds a rank of "random surfer" (line 8). If the absolute difference of the new rank is larger than a user-defined threshold ε, the vertex will be alive in next iteration (line 9). The main function, namely `PageRank`, iteratively calls `EdgeMap` and `VertexMap` until the iteration number exceeds the threshold or all vertices have converged (line 17-23). The runtime states A maintains the active vertices.

To enjoy the benefit of appropriate execution modes, like Ligra [43], Polymer also provides two execution modes (i.e., push and pull). Figure 6 demonstrates the execution flow and in-memory data access pattern under two modes on a 2-node NUMA machine. In the following sections, we will introduce our design on data layout and graph computation in detail based on this figure.

4.2 Graph-Aware Data Structure and Layout

The in-memory data structure of Polymer is similar with other vertex-centric systems, which consist of graph topology, application-defined data and runtime states. However, Polymer is designed specifically for NUMA by co-locating graph data and computation within the same NUMA node as much as possible, with the goal of reducing remote memory accesses and balancing cross-node interconnect bandwidth. The key is leveraging the unique access patterns of graph computation to place the in-memory graph data.

Algorithm 4.1: PageRank

```
D_curr ← {1/|V|, 1/|V|, ... 1/|V|}
D_next ← {0.0, 0.0, ... 0.0}

PREdgeF(s,t) begin
    AtomicAdd (&D_next[t], D_curr[s]/|N_out(s)|)
    return true
end

PRVertF(v) begin
    D_next[v] ← 0.15/|V| + (0.85 × D_next[v])
    alive ← |D_next[v] − D_curr[v]| > ε
    D_curr[v] ← 0.0
    return alive
end

PageRank(G, ε, max_iter) begin
    iter ← 0
    A ← V
    while iter ≤ max_iter ∧ A ≠ ∅ do
        A ← EdgeMap (G, A, PREdgeF)
        A ← VertexMap(G, A, PRVertF)
        Swap(D_curr, D_next)
        iter ++
    end
end
```

NUMA-aware graph partitioning: Polymer treats a NUMA machine as a distributed system and partitions graph data structures across nodes. For a NUMA-machine with N memory nodes, Polymer splits its in-memory data structures P into N disjoint partitions $P_1, P_2 ... P_N$, where $\bigcup_{i=1}^{N} P_i = P$. All vertices are indexed from 0 to $|V| - 1$ and evenly assigned to N disjoint vertex sets $V_1, V_2 ... V_N$, where the V_i belongs to P_i. All edges are also assigned to N disjoint out-edge or in-edge sets by their *target* vertex or *source* vertex respectively, where the E_i also belongs to P_i.

One approach is co-locating all edges with its source vertices to a single node [43]. However, as we discussed in Section 3.2 (Figure 2), this will lead to excessive *random* remote accesses of application-defined data (such as rank) and runtime states. Hence, Polymer only co-locates edges with their target vertices in a NUMA node in the push mode. All other application-defined data and runtime states are stored together with its owning vertices.

Further, to eliminate remote accesses due to accesses to graph topology data, Polymer borrows the idea of vertex replication from distributed graph systems [13, 22], and introduces *lightweight vertex replicas*, namely **agents**, for the vertices owned by other partitions. An agent is immutable and only maintains partial topology data, such as the start of neighboring edges and the degree of the vertex. The sole purpose of the agent is used to initiate computation over its master vertex within the node. This essentially eliminates random remote accesses when accessing application-defined data and runtime states. Since the number of vertices is much smaller than that of edges for most graphs, the agents will not cause much memory pressure (see Section 6.5).

Figure 6 shows an example of partitioning and placing graph data structures for the sample graph in Figure 2 for both push and pull modes. Vertices 1 to 3 are assigned to node 1 and vertices 4 to 6 are assigned to node 2. In push mode, Polymer co-locates out-edges with their target vertices (vertices 1 to 3). For such edges, Polymer also creates agents for edges whose source vertices are not located in this node. As a result, Polymer creates agents for vertices 4 to 6 accordingly.

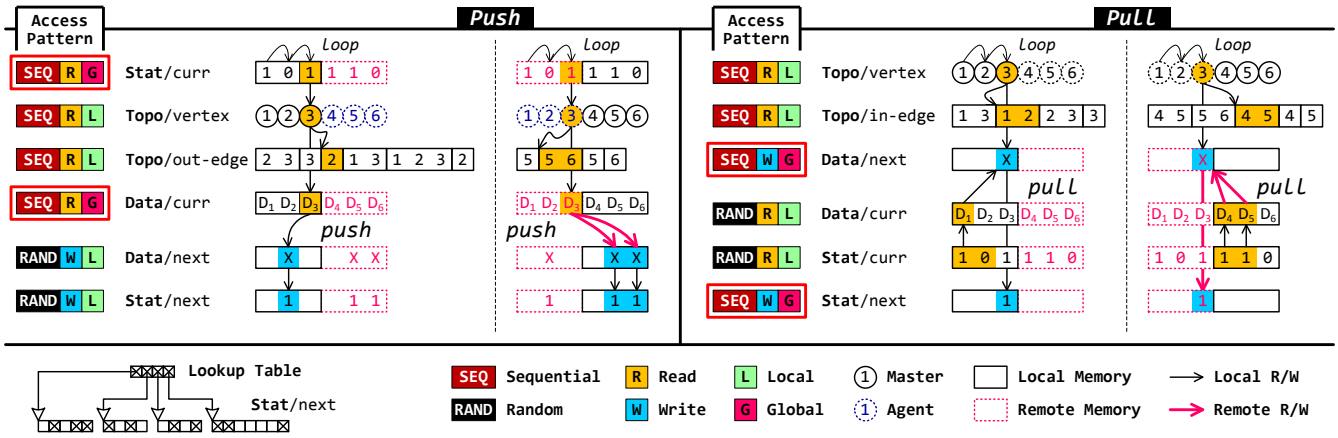


Figure 6. The in-memory data layout and execution flow of Polymer for two modes on the sample graph in Figure 2. The pattern to access in-memory data is also labeled by SEQ/RAND, R/W and L/G. Yellow indicates source vertex, and blue indicates target vertex.

NUMA-aware allocation of graph data: To overcome the random or interleaved allocation of graph data due to Linux's "first touch" policy, Polymer tackles the mismatch between the allocating threads and the computing threads by letting threads on the ith NUMA-node to allocate and initiate all of data in partition P_i. However, simply using co-location may cause the physical addresses of in-memory data *discrete*, which may result in additional cost for indirect accesses.

To this end, Polymer differentiates the placement policy of virtual address for these data according to their memory access behavior. First, since the graph topology data is immutable and only accessed in local, Polymer simply uses multiple worker threads bound on different NUMA-nodes to allocate and initiate them, which results in discrete virtual addresses. Second, the application-defined data is mutable and globally accessed in sequential order (see Section 4.3). Fortunately, we observe that such data is only allocated once at the construction stage and then repeatedly accessed at the computation stage. Hence, Polymer maps all partitions of application-defined data on discrete physical pages to a contiguous virtual address space, to avoid indirect accesses.

Finally, the runtime states are mutable. They are also globally read in sequential order, but are re-allocated and locally updated in each iteration. Hence, it may not be worthwhile to repeatedly construct a contiguous virtual address space for them due to high overhead of doing this. Instead, Polymer uses a lock-less tree-structure lookup table to collect all partitions of the runtime states allocated on different NUMA nodes (see lower-left corner of Figure 6). Each partition is concurrently allocated and linked to an indirect router array without contention.

Table 1 summarizes the data layout of various in-memory data in Polymer. Compared to existing systems, Polymer exploits co-located accesses for all in-memory data and selectively constructs the contiguous virtual address space for them. This may significantly reduce remote accesses.

4.3 NUMA-Aware Graph Computation

Even under co-located data layout, Polymer can still not thoroughly eliminate remote accesses as well as random accesses when accessing neighboring vertices owned by other partitions. One approach would use an additional phase like global shuffling in X-Stream after each iteration, which, however, would incur non-trivial overhead. To this end, Polymer borrows the idea from distributed graph systems [22] to distribute the computations on a singe vertex over multiple machines. However unlike distributed graph systems, Polymer only replicates the immutable topology data by using agent. For application-defined data, all updates are still applied into a single copy of them by shard memory rather than using replication and explicit messages for synchronization.

Table 1. A summary of the data layout of for various systems. P and V represent physical and virtual address, while D and C represent discrete and contiguous memory address. CE, IL and CO represent centralized, interleaved and co-located memory access respectively.

Data Layout	Existing Systems		Polymer	
	Alloc(P,V)	Access	Alloc(P,V)	Access
Topo	D, C	IL	D, D	CO
Data	D, C	IL	D, C	CO
Stat	C, C	CE	D, D	CO

By factoring computations, the worker threads of each NUMA node only perform *part* of computations for *all* of vertices, instead of *all* of computations for *part* of vertices. For example, in the push mode of Figure 6, the computation on three out-edges of vertex 3 are distributed to different threads on two partitions. The first thread will only push the value of vertex 3 to its local neighboring vertex 2, and the second thread will also push the value of vertex 3 to its two local neighboring vertices 5 and 6. In contrasts to Ligra, where all computations are performed by the first thread (see Figure 2). This neat change of strategy drastically converts the combination of access patterns. For example, in push mode, compared to the combination of local sequential read (`SEQ|R|L`) and global random write (`RAND|W|G`) in Ligra, Polymer adopts global sequential read for the data of source vertices (`SEQ|R|G`) and local random write for the data of target vertices (`RAND|W|L`). This access pattern during computation essentially conforms to our observations on NUMA characteristics

Unfortunately, in the pull mode, the access pattern is first local random read from source vertices (`RAND|R|L`, and then global sequential write for the target vertices (`SEQ|W|G`). For example, in Figure 6, both worker threads will pull the data from local in-neighbors to the vertex 3 in parallel. Due to the same sequential order of updating, the same vertex may be updated simultaneously or closely by multiple worker threads on different NUMA-nodes, which may cause heavy contention and frequent cache invalidation. Similarly, in the push mode, the simultaneous read operations on nearby vertices owned by the same NUMA-node may cause congestion on interconnects and memory controllers. Inspired by solutions to the similar problem for handling messages in distributed graph systems [12], Polymer makes worker threads on different NUMA-nodes process vertices in a *rolling* order, starting with the own vertices. For example, the worker thread in the second partition will start with vertex 4.

5. Optimization on Polymer

This section describes three optimizations that aim at improving scheduling on NUMA machines and leveraging different properties of graphs.

Hierarchical and Efficient Barrier: As a result of the hierarchical structure of cache and memory in NUMA machines, the cost of synchronization among threads on different cores rapidly increases with the growing of involved sockets. This leads to the necessity of a topology-aware synchronization mechanism. The default *pthread_barrier* uses a uniform "flat" model to make all participants on different cores wait on a globally shared variable and trap into the kernel, which can lead to one order of magnitude performance degradation from intra-node to inter-node synchronization (30μs vs. 570μs).

Inspired by hierarchical scheduling in Tiled-MapReduce [11], Polymer designs a hierarchical barrier. The worker threads on the same NUMA-node are grouped to share a partition of data and computation tasks. All worker threads are synchronized within a group first, and then the last thread of each group will further synchronize across the groups. With the increase of involved sockets, the hierarchical barrier can obviously decrease cache coherence broadcasts.

However, the hierarchical implementation based on *pthread_barrier* still suffers from the penalty of trapping into kernel. Polymer implements a *sense_reversing* centralized barrier [36] based on the atomic fetch-and-add instruction, which is used as a building block for our hierarchical NUMA-aware barrier.

Balanced Partitioning: For synchronous execution, it is crucial to balance the load among worker threads, especially with hierarchical scheduling. First, it is relatively simple to balance the load among threads within a group on the same NUMA node, since all of them share the same partition and the memory access is uniformed. Polymer adopts dynamic task scheduling among worker threads within a NUMA node, in which each worker thread dynamically fetch a portion of tasks after finished its previous tasks.

Second, Polymer still has to balance the load across NUMA nodes. However, due to the co-location of data and computation, Polymer requires a balanced partitioning to evenly split the entire workload into multiple groups, and separately processes them on different NUMA nodes. A natural approach is evenly assigning vertices to multiple nodes, whereas the computation complexity and memory access frequency of the scatter and gather phases is linear to the number of edges.

However, for skewed graphs, such partitioning can lead to substantial work imbalance. Inspired by vertex-cut [22] in distributed graph systems, Polymer treats edges as first-class citizen in partitioning, and evenly assigns edges to groups on different NUMA-nodes. For a graph $G = (V, E)$, the edge-oriented balanced partitioning splits vertices into disjoint sets $V_1, V_2 ... V_N$ to minimize the deviation of $\{\sum_{v \in V_i} |N_{in/out}(v)| \mid 1 \leq i \leq N\}$. Polymer can efficiently implement such balanced partitioning due to little communication cost in single machine. In addition, it should be noted that it is hard to evenly partition both in-edges and out-edges at same time. Fortunately, in most cases, Polymer only uses one type of edges in either push or pull mode, so that it is enough to evenly partition edges in one direction.

Adaptive Data Structures: As shown in Figure 6, Polymer uses a lock-less tree-structure lookup table to represent the runtime states. Each leaf partition of the table is a bitmap, which is efficient for a large proportion of active vertices. However, as most graph algorithms converge asymmetrically, the proportion of active vertices will sustainedly decline and reach zero. The overhead of traversing through sparse bitmaps is non-trivial in each iteration, leading to non-trivial performance slowdown, especially for traversal algorithms with high-diameter graphs.

Table 2. A collection of real-world and synthetic graphs.

Graph	Num. Vertices	Num. Edges
twitter	41.7 M	1.47 B
rMat24	16.8 M	268 M
rMat27	134.2 M	2.14 B
powerlaw	10.0 M	105 M
roadUS	23.9 M	58 M

Inspired by the solution in Ligra [43], Polymer uses adaptive data structures for runtime states within the tree-structure table and automatically switches data structure for leaf partitions according to the proportion of active vertices. Unlike the bitmap shared by all worker threads within a NUMA-node, each thread on different cores will allocate a private queue and append active vertex ID to it without contention. The queues within a NUMA-node can be merged when de-duplication of vertex ID if necessary or linked to a local indirect router array to form a two-level tree-structure. Finally, Polymer uses the total degrees of active vertices and the application-defined threshold to decide whether to switch data structures.

6. Evaluation

Polymer is implemented in C++ using the Pthreads library. It currently supports both push and pull mode using a synchronous scheduler. Polymer and its applications are approximately 5,300 LOC, and are compiled using gcc version 4.8.1.

We evaluate Polymer against other three state-of-the-art graph analytics systems on a single machine: Ligra, X-Stream v0.9 and Galois v2.2. We omit comparison with GraphChi, which were shown to have inferior performance than others when the input graphs fit in memory [39, 43]. Unless otherwise mentioned, all experiments were performed on the 80-core Intel machine (without hyper-threading), which consists of eight 2.0GHz Intel Xeon E7-8850 processors connected with QPI. These form a twisted hypercube, maximizing the distance between two nodes to two hops. Each NUMA socket has 10 cores and a 128GB local DRAM. We also ran all experiments on our 64-core AMD machine, which consists of four multi-chip modules connected with HT. Each module has two 8-core die with independent memory controllers, thereby the system comprises eight memory nodes (i.e., sockets). Due to the space restriction, we only report representative results for scalability and skip the rest similar results on the AMD machine.

6.1 Algorithms and Graphs

We use six popular graph algorithms to evaluate Polymer.

PageRank (PR) computes the rank of each vertex based on the ranks of its neighbors [7]. We use the synchronous, push-based PageRank for Polymer, Ligra and X-Stream in all cases because it is relatively faster. Galois chooses a synchronously pull-based implementation to reduce synchronization overhead.

Sparse matrix-vector multiplication (SpMV) multiplies the sparse adjacency matrix of a directed graph with a dense vector of values, one per vertex.

Bayesian belief propagation (BP) estimates the probabilities of vertices by iterative message passing between vertices along weighted edges [25]. Since only X-Stream provides SpMV and BP applications, we implement the algorithms for other three systems.

Breadth-first search (BFS) traverses an unweighted graph by visiting the sibling vertices before visiting the child vertices. Ligra follows a data-driven hybrid implementation [6] by switching between sparse and dense representation of runtime states. Galois mixes synchronous and asynchronous scheduling to implement the BFS application.

Connected components (CC) calculates a maximal set of vertices that are reachable from each other for a directed graph. Polymer, Ligra and X-Stream all adopt label propagation [49] to im-

Table 3. Runtimes (in seconds) of algorithms over various datasets with 80 threads on the 80-core Intel machine. Red times are the best for each input and graph problem pair. (†) Galois adopts different algorithms.

Algo.	Graph	Polymer	Ligra	X-Stream	Galois
PR	twitter	5.28	15.03	28.91	11.55
	rMat24	1.84	4.10	4.80	2.89
	rMat27	9.63	28.00	18.20	19.61
	powerlaw	1.61	30.50	6.06	6.62
	roadUS	1.21	2.32	2.79	1.38
SpMV	twitter	7.55	29.00	59.57	11.68
	rMat24	1.86	4.30	5.24	6.02
	rMat27	19.15	54.25	52.54	41.86
	powerlaw	1.80	31.00	5.53	6.21
	roadUS	1.29	2.83	2.98	3.55
BP	twitter	38.00	63.10	2017.29	57.06
	rMat24	7.73	9.88	44.27	12.20
	rMat27	58.30	92.80	736.62	74.98
	powerlaw	8.08	30.70	38.26	8.58
	roadUS	5.18	2.59	19.99	7.05
BFS	twitter	0.90	1.13	28.70	2.67
	rMat24	0.53	0.50	4.30	0.40
	rMat27	1.56	1.86	30.18	2.54
	powerlaw	0.36	0.39	2.58	0.36
	roadUS	1.16	6.93	557.68	5.01
CC	twitter	4.60	5.51	54.80	31.91
	rMat24	1.11	0.98	11.01	11.55
	rMat27	8.72	7.74	39.95	33.86
	powerlaw	1.23	2.56	5.13	3.51
	roadUS	57.50	63.20	985.15	†1.18
SSSP	twitter	2.26	3.17	165.15	26.29
	rMat24	1.04	1.25	17.86	1.95
	rMat27	5.78	5.26	126.38	28.50
	powerlaw	0.85	1.12	12.36	26.58
	roadUS	341	338	1225	†0.33

plement this algorithm, while Galois provides a topology-driven algorithm based on a concurrent union-find data structure [39].

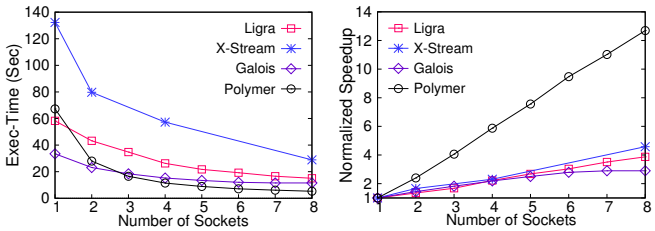
Single-source shortest-paths (SSSP) computes the distance of the shortest path from a given source vertex to each vertex. The SSSP implementation on Polymer, Ligra and X-Stream is based on the Bellman-Ford algorithm [16] with synchronously data-driven scheduling, while Galois uses a data-driven and asynchronously scheduled delta-stepping algorithm [37].

The input graphs used in our experiments are shown in Table 2. *twitter* is a real-world social follower graph [27]. The synthetic scale-free graphs, *rMat24* and *rMat27*, are generated by the R-MAT generator [9] in Graph500 [2]. The synthetic power-law graph (*powerlaw*) with fixed power-law constant 2.0 was generated by tools in PowerGraph [22], which randomly sample the degree of each vertex from a Zipf distribution [4] and then add edges. Finally, the road network of the United States (*roadUS*) is from the 9th DIMACS shortest paths challenge [1] with much high diameter. All graphs are unweighted except roadUSA. To provide a weighted input for the SpMV and SSSP algorithms, we add a random edge weight in the range $(0, 100]$ to each edge. For the other applications, we simply ignore the weights of roadUSA.

6.2 Overall Performance

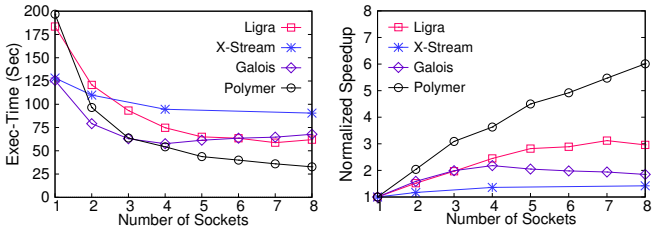
Table 3 gives a complete runtime comparison between Polymer and other three state-of-the-art graph-analytics systems on a single machine: Ligra, X-Stream and Galois. We report the execution time of first *five* iterations for PageRank, SpMV and BP, as other systems.

For three sparse matrix multiplication algorithms (PR, SpMV and BP), Polymer achieves optimal performance against other systems in all cases using 80 threads on multiple NUMA memory nodes, except BP on the roadUS graph. The largest improvement is



(a) Execution Time (Intel) (b) Normalized Speedup (Intel)

Figure 7. The execution time and normalized speedup for PageRank with the increasing number of sockets (using full cores) on the Intel machines.



(a) Execution Time (AMD) (b) Normalized Speedup (AMD)

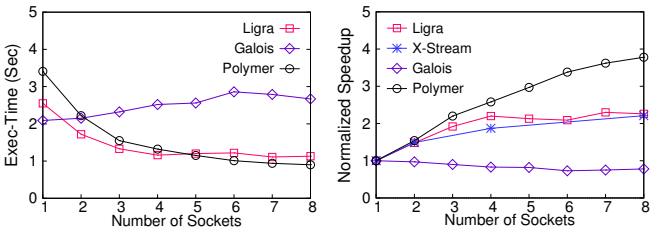
Figure 8. The execution time and normalized speedup for PageRank with the increasing number of sockets (using full cores) on the AMD machines.

from BP on the twitter graph with respect of X-Stream by 53.09X. For PR and SpMV, Polymer still outperforms X-Stream by up to 5.48X and 7.89X accordingly. Compared with Ligra, the largest improvements for three algorithms are all on the powerlaw graph by 18.9X, 17.3X and 3.80X respectively, benefiting from load balancing. Polymer also outperforms Galois by up to 4.11X, 3.46X and 1.58X respectively.

The graph traversal algorithms, including BFS, CC and SSSP, are not sensitive to the memory accesses of NUMA systems, since they have much fewer active vertices in each iteration, and then resulting in fewer memory accesses. Polymer still can provide optimal or close performance due to several optimizations, such as balance partitioning for power-law graphs and adaptive data structure for high-diameter graphs.

The performance of Ligra is with a similar pace as Polymer, due to the same execution modes (i.e., push and pull) and similar optimizations like automatic mode switching. However, Polymer outperforms Ligra in most cases due to its NUMA-aware designs. The edge-centric system such X-Stream has extremely poor performance for traversal algorithms, especially for high-diameter graphs like road networks, due to excessive accesses to edges and inefficient data structure for the runtime states. All edges (e.g., 58M for roadUS) must be identified whether to participant computation by accessing their state of source vertex in each iteration, even there are just several active vertices. Further, the extremely slow convergence (e.g. 6237 iterations for BFS with roadUS) of high-diameter graphs amplify the problem[6]. For high-diameter graphs like *roadUS*, the asynchronous scheduling and special implementations in Galois are able to exploit more parallelism for the graph traversal algorithms, such as CC and SSSP. Unfortunately, they do not work well with other graphs, due to their relative higher average degrees and lower diameter.

[6] Vertex-centric systems (e.g., Ligra, Polymer and Galois) can avoid such overhead by adaptively using queue-based data structure to maintain active vertices. For example, the average overhead in each iteration for BFS with roadUS are 0.032ms, 0.043ms and 92ms for Polymer, Ligra and X-Stream respectively (Galois uses asynchronous scheduling).



(a) Execution Time (Intel) (b) Normalized Speedup (Intel)

Figure 9. The execution time and normalized speedup for BFS with the increasing number of sockets (using full cores) on the Intel machines.

6.3 Scalability

Since all of existing systems have scaled well in terms of number of cores and Polymer mainly improves the performance and scalability on NUMA machines, we concentrate on the study of the scalability in terms of number of sockets using PageRank and BFS algorithms for the twitter graph on two NUMA machines.

On our 80-core Intel NUMA machine, as shown in Figure 7, the scalability of Polymer for PageRank is much better than all of existing systems. The scalability ratio even exceeds the number of sockets (i.e., 12.1 vs. 8). As the number of sockets increases, the size of total caches increases and the size of partitions for each socket decreases, which lead to less cache misses and thus a super-linear speedup.

Even if the performance of Polymer is just close or even worse on single NUMA-node compared with other systems, with the increasing number of sockets, Polymer can enjoys much larger speedup and outperforms Ligra, X-Stream and Galois by 2.84X, 5.45X and 2.19X. This conforms the effectiveness of NUMA-aware implementation and associated optimizations.

Figure 8 illustrates the performance and scalability results for PageRank on our 64-core AMD machine. Polymer exhibits a similar trend with the increase of sockets from 1 to 8, however, the scalability ratio on the AMD machine is 6.01X, which is lower than that on the Intel machine with the same 8 sockets, probably due to relative small last level cache (i.e. 16 MB vs. 24 MB) and different interconnect. Two sockets within multi-chip model share more bandwidth, restricting the scalability.

As shown in Figure 9, the scalability for BFS is relatively poor in all of systems, due to fewer active vertices in each iteration. However, Polymer still exhibits much better scalability and outperforms other existing systems using 8 sockets. Note that missing the execution time for X-Stream is due to out of range (from 69.4s to 28.7s).

Table 4. A comparison of the remote access rate, the number of remote accesses and the LLC miss rate due to remote accesses.

	Polymer	Ligra	X-Stream	Galois
Access Rate/R	37.5%	83.3%	47.4%	83.6%
Num. Accesses/R	3,090M	6,116M	5,016M	7,887M
LLC Miss Rate/R	3.94%	9.47%	8.67%	13.17%

(a) PageRank

	Polymer	Ligra	X-Stream	Galois
Access Rate/R	28.6%	37.5%	33.3%	47.4%
Num. Accesses/R	340M	591M	374M	709M
LLC Miss Rate/R	1.70%	2.27%	1.72%	7.10%

(b) BFS

6.4 Reduced Remote and Random Accesses

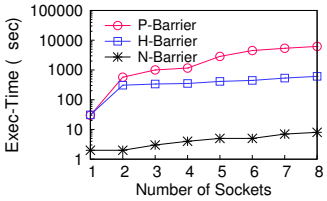
To understand the source of the improvement in Polymer, we evaluate various systems using PageRank and BFS algorithms with the twitter graph, to compare the remote access rate, the total number of remote accesses and the LLC miss rate due to remote access. For PageRank, as shown in Table 4(a), Polymer has much fewer remote accesses in both the rate and the number due to co-locating graph data and computation. Further, the remote accesses in Polymer is sequential, reducing up to 70% (from 55%) LLC miss rate due to remote accesses. For BFS, which leaves limited room for improvement due to fewer memory accesses, the results in Table 4(b) still confirms the better performance of Polymer.

Table 5. The peak memory usage (in GB) on Polymer and existing systems over various graphs for PageRank using 80 threads. The memory usage for agents is shown in brackets.

Graph	Polymer(agent)	Ligra	X-Stream	Galois
twitter	39.2(2.95)	37.0	39.9	25.1
rMat24	13.1(1.68)	10.2	17.4	10.2
rMat27	71.1(4.37)	66.6	75.4	64.0
powerlaw	7.0(1.13)	5.8	10.1	5.3
roadUS	11.2(1.52)	8.1	17.0	7.3

6.5 Memory Consumption

Polymer introduces lightweight replication of vertices across NUMA-nodes to factor computation and reduce remote memory accesses, but increasing the memory consumption. In Table 5, we investigate the memory consumption of Polymer and existing systems over various graphs. Since the number of replicas increases with the increasing number of sockets, we evaluate the peak memory consumption for PageRank algorithm using all of the eight NUMA-nodes. Since only Galois uses its own optimized memory allocator and carefully reuses memory between iterations, it is no surprise that it obtains the best results. X-Stream consumes the most memory due to additional buffers in the shuffle phase. The memory consumption of Polymer and Ligra is close, both of which use the default memory allocator. The increase of memory consumption is lower than 30%, except the roadUS graph (38.3%) due to much lower ration of edges to vertices (2.43X). The major extra usage (around 80%) of Polymer is from the lightweight replication.



(a) Synchronization Time

Algo.	w/o	w/
PR	1.31	1.21
SpMV	1.32	1.29
BP	5.62	5.18
BFS	68	1.16
CC	317	57.5
SSSP	436	341

(b) Execution Time

Figure 10. (a) The synchronization time with the increasing number of sockets using various barriers. (b) A comparison of execution time w/ and w/o NUMA-aware barrier using various algorithms on the roadUS graph.

6.6 Hierarchical and Efficient Barrier

To demonstrate the benefit of hierarchical and user-level synchronization, we compare our NUMA-aware barrier (N-Barrier) with default pthread_barrier (P-Barrier). To breakdown the improvement, we further implement a hierarchical_barrier (H-Barrier) using pthread_barrier with hierarchical mechanism.

Figure 10(a) presents absolute performance of three barriers with the increasing number of sockets. We bind 10 threads on each sockets, one thread per core. The hierarchical mechanism in H-Barrier can obviously improve the scalability by decreasing cache coherence broadcasts between NUMA-nodes. H-Barrier outperforms P-Barrier by almost one order of magnitude on eight sockets (6182μs vs. 612μs). However, the performance of H-Barrier still suffers from frequently trapping into the kernel. Based on H-Barrier, the N-Barrier further replace the pthread_barrier with atomic memory access, and achieves additional two order of magnitude improvement (612μs vs. 8μs).

Table 6. A comparison of execution time w/ and w/o (a) adaptive data structure and (b) balanced partitioning using various algorithms.

Algo.	w/o	w/
PR	1.32	1.21
SpMV	1.30	1.29
BP	5.69	5.18
BFS	827	1.16
CC	868	57.5
SSSP	1720	341

(a) Adaptive data structure

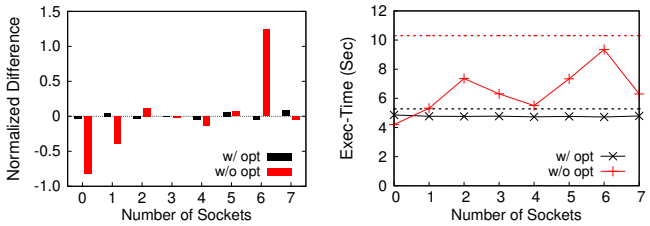
Algo.	w/o	w/
PR	10.30	5.28
SpMV	13.05	7.55
BP	49.20	38.00
BFS	3.30	0.90
CC	8.85	4.60
SSSP	5.32	2.26

(b) Balanced Partitioning

We further compare the performance of various algorithms on Polymer with and without the optimization for the roadUS graph. As shown in Figure 10(b), NUMA-aware barrier can only provide a limited improvement on PageRank, SpMV and BP by up to 8% (7%, 2% and 8%). While for three traversal algorithms, NUMA-aware barrier can improve the performance by 58.6X, 5.51X and 1.28X, due to high proportional synchronization overhead.

6.7 Adaptive Data Structure

To study the benefit of adaptive data structure, Table 6(a) compares the performance of various algorithms on Polymer with and without the optimization for roadUS graph. As the major improvement is from reducing the performance cost to check runtime states when there are few active vertices. For sparse matrix multiplication algorithms (PR, SpMV and BP), the active number of vertices is quite stable. The switching of data structure occurs much later or even no switch at all, thereby the improvement is limited, up to 9%. In contrast, the traversal algorithms, including BFS, CC and SSSP, can significantly benefit from more efficient data structure.



(a) Execution Time (b) Normalized Speedup

Figure 11. (a) The normalized difference of edges in each partition for the twitter graph. (b) The execution time w/ and w/o balance partitioning on each socket for PageRank with the twitter graph.

6.8 Balanced Partitioning

We first compare the performance of Polymer with and without balance partitioning using the twitter graph. The out-degree power-law constant (α) of the twitter graph is close to 2.0. Table 6(a) shows that the speedup ranges from 1.29X to 3.67X. To reveal the benefit of balanced partitioning, we first estimate the number of edges in each partition for the twitter graph. As shown in Figure 11(a), the default partitioning will evenly assign vertices among NUMA-nodes and incur the imbalance of edges, while the balance partitioning can prominently narrow the fluctuation into a range from -0.5% to 0.8%. We further collect the pure execution time of the threads on each socket using the PageRank algorithm. Due to synchronous scheduling, the overall execution time is decided by the slowest worker thread. With default partitioning, the imbalance of edges causes the difference of execution time, which will be amplified by the congestion on some interconnects and memory controllers. In Figure 11(b), the black and red dotted lines indicate the whole execution time with and without balanced partitioning. The execution time on each socket without optimization ranges from 4.16 to 9.32 seconds. In contrast, the range for balanced execution is from 4.72 to 4.86 seconds.

7. Other Related Work

Polymer directly departs from prior graph analytics systems such as Ligra [43], X-Stream [42] and Galois [39], but differs with them by adopting NUMA- and graph-aware data layout and access strategy.

Other single-machine graph-analytics systems: There are several efforts aiming at leveraging multicore platforms for graph processing [28, 41, 48]. For example, GraphChi [28] targets at disk-based graph computation by using parallel sliding windows to preserve access locality for graph data. Medusa [48] provides users with a simple interface to write graph-parallel code on GPUs. Such techniques should be useful when extending Polymer for disk-based processing, CPU-GPU co-processing and streaming processing. However, none of them focus on leveraging NUMA characteristics. Polymer borrows some designs from prior systems, and our techniques may be helpful to boost performance of such systems on NUMA machines.

Distributed graph-analytics systems: The popularity of graph analytics is also embodied in distributed graph analytics systems, such as Pregel [35], GraphLab [22, 32], Cyclops [13] and GraphX [23]. Mizan [26] leverages vertex migration for dynamic load balancing. PowerSwitch [46] proposes a hybrid execution mode that adaptively switches a graph-parallel program between synchronous and asynchronous execution modes for optimal performance. Imitator [45] reuses computational replication for fault tolerance in large-scale graph processing to provide low-overhead normal execution and fast crash recovery. There are also a few systems considering streaming processing [15, 38] or graph properties [12, 14, 24]. Many techniques aimed at NUMA systems in Polymer are borrowed from these distributed systems.

Other NUMA-aware computation systems: Liu et. al. [30] recently developed a tool to analyze and quantify the bottlenecks of multithreaded program on NUMA platforms. David et. al. [18] presented an exhaustive study of synchronization on various multicore and NUMA systems. MemProf [29] is a memory profiler aimed at NUMA systems for optimizing multithread programs Carrefour [17] is a memory placement algorithm for data-intensive applications, which aims at eliminating the congestion on memory controllers and interconnects. N-MASS [34] is a scheduling algorithm that simultaneously considers data locality and cache contention for NUMA systems. Gaud et. al. [21] discovered that large pages may hurt performance on NUMA systems and proposed a memory placement to recover the performance. Lock cohorting [19] can transform general spin-lock algorithms into scalable NUMA-aware versions. Calciu et. al. [8] further extended the lock cohorting technique to tailor reader-writer locks in a NUMA-friendly fashion. Ostrich [11] improved the performance of MapReduce on NUMA machines with tiling. Polymer is inspired by prior work on NUMA platforms, such as reducing remote memory accesses, but is specially designed to leverage graph-specific characteristics to boost performance.

8. Conclusion

This paper described Polymer, a NUMA-aware graph analytics system, which is motivated by a detailed study of NUMA characteristics. The key of Polymer's performance is using graph-aware graph data allocation, data layout and access strategy that reduces remote memory accesses as much as possible and turns inevitable remote accesses from random to sequential ones. This significantly boosted the performance of Polymer on NUMA platforms.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the Doctoral Fund of Ministry of Education of China (No. 20130073120040), the Program for New

Century Excellent Talents in University, Ministry of Education of China (No. ZXZY037003), a foundation for the Author of National Excellent Doctoral Dissertation of PR China (No. TS0220103006), the National Natural Science Foundation of China (No. 61303011 and 61402284), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2014A05), the Shanghai Science and Technology Development Fund for high-tech achievement translation (No. 14511100902), a research grant from Intel and the Singapore NRF (CREATE E2S2).

References

[1] The 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/.

[2] Graph 500. http://www.graph500.org.

[3] numactl. http://oss.sgi.com/projects/libnuma/.

[4] L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *SOSP*, 2009.

[6] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3):137–148, 2013.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[8] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. Numa-aware reader-writer locks. In *PPoPP*, 2013.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[10] R. Chen and H. Chen. Tiled-mapreduce: Efficient and flexible mapreduce processing on multicore with tiling. *ACM TACO*, 10, 2013.

[11] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *PACT*, 2010.

[12] R. Chen, J. Shi, Y. Chen, H. Guan, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. Technical Report 2013-11-001, IPADS, SJTU, 2013.

[13] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC*, 2014.

[14] R. Chen, J. Shi, B. Zang, and H. Guan. Bipartite-oriented distributed graph partitioning for big learning. In *APSys*, 2014.

[15] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *EuroSys*, pages 85–98, 2012.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[17] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on numa systems. In *ASPLOS*, 2013.

[18] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, 2013.

[19] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing numa locks. In *PPoPP*, pages 247–256, 2012.

[20] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM*, pages 251–262, 1999.

[21] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, V. Quema, and I. Grenoble. Large pages may be harmful on numa systems. In *USENIX ATC*, 2014.

[22] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[23] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, 2014.

[24] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, 2014.

[25] U. Kang, D. Horng, et al. Inference of beliefs on billion-scale graphs. In *SIGKDD-LDMTA*, 2010.

[26] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *EuroSys*, pages 169–182, 2013.

[27] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[28] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *OSDI*, 2012.

[29] R. Lachaize, B. Lepers, V. Quéma, et al. Memprof: A memory profiler for numa multicore systems. In *USENIX ATC*, 2012.

[30] X. Liu and J. Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *PPoPP*, 2014.

[31] Y. Liu, B. Wu, H. Wang, and P. Ma. Bpgm: A big graph mining tool. *Tsinghua Science and Technology*, 19(1):33–38, 2014.

[32] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *VLDB Endowment*, 5(8):716–727, 2012.

[33] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[34] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *ISMM*, 2011.

[35] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[36] J. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1):21–65, 1991.

[37] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[38] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[39] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, 2013.

[40] B. Panda, J. Herbach, S. Basu, and R. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. In *VLDB*, 2009.

[41] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Usenix ATC*, 2012.

[42] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.

[43] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *PPoPP*, 2013.

[44] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *VLDB*, 2010.

[45] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. Replication-based fault-tolerance for large-scale graph processing. In *DSN*, 2014.

[46] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *PPoPP*, 2015.

[47] X. Zhao, A. Chang, A. D. Sarma, H. Zheng, and B. Y. Zhao. On the embeddability of random walk distances. In *VLDB*, 2013.

[48] J. Zhong and B. He. Medusa: Simplified Graph Processing on GPUs. *TPDS*, 2013.

[49] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.