




SilverChunk: An Efficient In-Memory Parallel Graph Processing System

Tianqi Zheng^{1,2} , Zhibin Zhang¹, and Xueqi Cheng^{1,2}

¹ CAS Key Laboratory of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
zhengtianqi@ict.ac.cn

² University of Chinese Academy of Sciences, Beijing, China

Abstract. One of the main constructs of graph processing is the two-level nested loop structure. Parallelizing nested loops is notoriously unfriendly to both CPU and memory access when dealing with real graph data due to its skewed distribution. To address this problem, we present **SilverChunk**, a high performance graph processing system. **SilverChunk** builds edge chunks of equal size from original graphs and unfolds nested loops statically in pull-based executions (**VR-Chunk**) and dynamically in push-based executions (**D-Chunk**). **VR-Chunk** slices the entire graph into several chunks. A virtual vertex is generated pointing to the first half of each sliced edge list so that no edge list lives in more than one chunk. **D-Chunk** builds its chunk list via binary searching over the prefix degree sum array of the active vertices. Each chunk has a local buffer for conflict-free maintenance of the next frontier. By changing the units of scheduling from edges to chunks, **SilverChunk** achieves better CPU and memory utilization. **SilverChunk** provides a high level programming interface combined with multiple optimization techniques to help developing efficient graph processing applications. Our evaluation results reveal that **SilverChunk** outperforms state-of-the-art shared-memory graph processing systems by up to 4×, including Gemini, Grazelle, etc. Moreover, it has lower memory overheads and nearly zero pre-processing time.

Keywords: Graph processing · Parallel scheduling · Chunking

1 Introduction

1.1 Background

Graphs are commonly used to represent interactions between real world entities. Graph analytics are algorithms that extract information from a graph, which are widely used in social network analytics, transportation, ad and e-commerce recommendation systems. As a result, a large number of graph processing systems are proposed to facilitate graph analytics. Recently there is a rising interest of building multi-core shared memory graph processing systems on a single machine because (1) distributed graph systems incur a lot of communication overheads;

(2) real world graphs, e.g., Twitter’s follower graph, despite its billions of edges, can still fit into main memory; and (3) memory capacity and bandwidth are increasing and will keep increasing in the near future. These systems [5, 6, 8–14] process a big graph in main memory of a single high-end server with large RAM space. They provide high level interfaces for programming simplicity and aim at full utilization of all CPU and memory resources without manual tweaking. For example, Ligra [9] provides two simple primitives, `EdgeMap` and `VertexMap`, for iterating over edges and vertices respectively in parallel. These simple primitives can be applied to various graph algorithms which operate on a subset of vertices during each iteration.

1.2 Problems

Parallel graph processing is nontrivial due to complex data dependencies in graphs, however, it is essential for efficient graph analytics. In this paper we discuss two problems of building an high-performance in-memory graph processing system.

Preliminaries. In-memory graph processing systems often organize outgoing edges in the Compressed Sparse Row (CSR) format and incoming ones in the Compressed Sparse Column (CSC) format, as shown in Fig. 1. A frontier is a subset of the vertices which are active in the current iteration, as shown in Fig. 2. Graph algorithms visit the destination vertices of the active edges and apply an algorithm-specific function to propagate the value from each edge’s source to its destination. This operation is repeated until the current frontier is empty or user defined condition is met. We refer to this process as frontier-based computing.

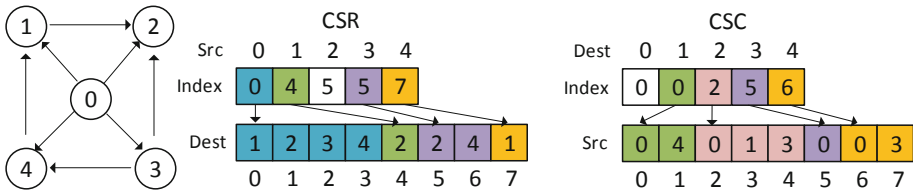


Fig. 1. Compressed Sparse Row/Column format

The frontier structure may be implemented either as a bitmap (dense format) or as an array directly storing the vertex IDs (sparse format). Which one is better depends on the density of the frontier. Frontier-based computing can have two different execution modes, namely push and pull. Both modes contains a two-level nested loop. In push mode, frontiers are used in the outer loop and updates are propagated from active vertices to their neighbors, while in pull mode, the outer loop is the entire vertex list and each vertex receives updates from its

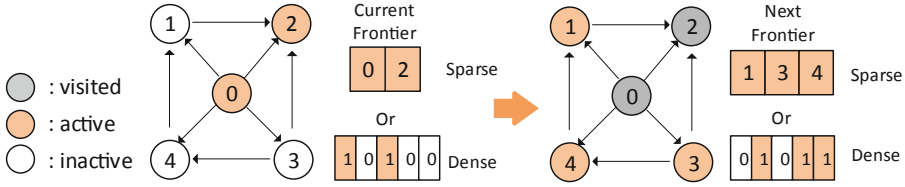


Fig. 2. Frontiers in a simple BFS algorithm

in-bound edges by checking if the source vertex is inside the current frontier or not. There are active researches [1, 7] studying whether to push or pull. The basic principle is to push when the frontier is sparse and to pull if otherwise. As a result, graph processing engines like Ligra [9] automatically switches between these two execution modes based on the density of the current frontier.

Problems. We discuss the following two problems:

- In both execution modes, the outer loop is parallelized in order to leverage the multiple cores of modern processor chips. Unfortunately, due to the power-law nature of real world social graphs, only a small fraction of vertices has a significant large number of neighbors while a major fraction of vertices has relatively few neighbors. As a result, parallelizing only the outer loop is insufficient as it can lead to significant load imbalance. One naive approach is to use traditional parallel schedulers such as Cilk [2] or OpenMP [3] to parallelize the inner loop. However, this approach can lead to numerous conflicting writes and scheduling overhead which completely negates the benefits of the pull execution mode. Grazelle [5] solves this problem by introducing a scheduler-aware interface that allows programmers to directly operate on the internal structure of the execution unit of the underlying scheduler. It provides thread local storage for local updates and merge buffers for global updates in order to achieve conflict-free parallelization. However the implementation is architecture-specific and requires additional efforts to implement even a simple graph algorithm.
- In push mode, due to the sparsity of the frontier, there is a high probability that the next frontier will also be sparse, hence building the next frontier as a sparse array instead of a bitmap is more efficient. However, building sparse frontiers in parallel is nontrivial. Ligra [9] does this by first allocating a scratch buffer that is large enough to hold all possible vertices in the next frontier, and then computing an offset array via parallel prefix summing over the active vertices' degrees in the current frontier. When a vertex successfully updates one of its neighbor, Ligra puts the neighbor into the scratch place pointed by its corresponding offset and atomically adds one to the offset. Finally it gathers all the valid vertices inside the scratch buffer into the next frontier. This process is both CPU and memory unfriendly. It scatters the vertices in the scratch buffer with random writes and relies on atomic instructions to synchronize the updates of the offset values.

1.3 Our Solutions and Contributions

To address these problems, we present **SilverChunk**, a graph processing system that enables balanced execution of parallel nested loop and conflict-free frontier maintenance. **SilverChunk** consists of two different chunking schemes, namely **VR-Chunk** for pull mode and **D-Chunk** for push mode. It also provides a high level programming interface with additional optimizations. The main contributions of our work are summarized as follows:

- **VR-Chunk.** We show that our **VR-Chunk** solves the first problem in a clean way. Instead of tuning the parallel scheduler, we change the scheduling unit directly from vertices to chunks. **VR-Chunk** splits the edge list statically into small chunks and generates additional virtual vertices to ensure conflict-free updates.
- **D-Chunk.** To tackle the second problem, we propose **D-Chunk**, a dynamic chunking scheme that applies to sparse frontiers. Since the vertices in a sparse frontier is discrete in memory, we build a list of virtual chunks that contains the information to help iterate over the edge list one piece of at a time. A virtual chunk provides a scratch space to aggregate vertices for the next iteration, which alleviates concurrent conflicts when building sparse frontiers.
- **Hybrid Polymorphic Interface and Optimizations.** We propose a new programming interface addressing different execution modes and graph algorithm properties for further optimizations. We design a new execution mode: **AllPull** mode, which optimizes the execution when the current frontiers are very dense.
- **Extensive Experiments.** We carry out extensive experiments using both large-scale real-world graphs and synthetic graphs to validate the performance of **SilverChunk**. Our experiments look into the key performance factors to all in-memory systems including the pre-processing time, the computational time and the effectiveness of main memory utilization. The results reveal that **SilverChunk** outperforms the state-of-the-art graph processing systems in most test cases by up to 4 \times .

The rest of this paper is organized as follows. Section 2 describes the main constructs of **SilverChunk**. Section 3 shows the high level programming interface and additional optimizations. Section 4 contains experimental results. Finally, Sect. 5 discusses the related works and Sect. 6 gives the concluding remarks.

2 Constructs

The main constructs of **SilverChunk** are the two chunking schemes: **VR-Chunk** and **D-Chunk**. Both schemes output similar chunk structures which are used to iterate over the input graphs. As a result, we unfold the nested loop into one flat loop which is efficient for parallel scheduling.

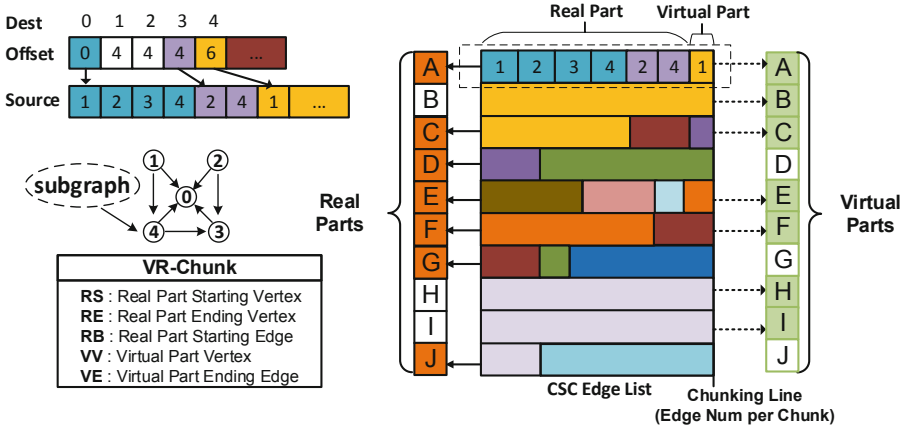


Fig. 3. VR-Chunk

2.1 VR-Chunk

In pull-based execution, we always iterate over the entire edge list to pull updates from the active vertices, thus the chunking scheme is static. Figure 3 shows how chunks are built from the original CSC array. Due to the dense feature of the frontier in pull mode, we assume that every edge requires the same amount of computation. Hence we slice the edge list into several chunks with equal number of edges, and assign each thread the same number of chunks to process.

Each chunk only needs to maintain five data fields: the starting and the ending destination vertices, the first edge, the virtual vertex and the last edge. The first two fields are obvious. As VR-Chunk might break the edge list, we need to maintain the first edge at each boundary. These fields form the real part of a chunk. The interesting one is the virtual vertex field, which stores the virtual vertex’s ID, referring to the virtual part of a chunk. A different approach of dealing skewed distribution would be directly slicing the giant vertices into small virtual vertices. However, it cannot generate balanced chunks with respect to the edge number. VR-Chunk always slices giant vertices if its neighbor size is greater than the chunk size. Virtual vertices are used as delegates to the real vertices so that each vertex is assigned to exact one chunk. Virtual vertices are appended at the end of the vertex array to enlarge the vertex space so that the application data such as the PageRank value array gets transparently expanded too. Therefore, every application data gets a dedicated merge buffer which is appended at the end and there is no need to explicitly maintain a separate one.

2.2 D-Chunk

In push-based execution, since the active frontier is known only at runtime, VR-Chunk cannot be applied directly. Also the push execution always incurs random writes, synchronization is unavoidable. However, we can still benefit from

chunking because it allows the destination vertices be collected in a conflict-free manner, therefore improving the sparse frontier’s maintenance.

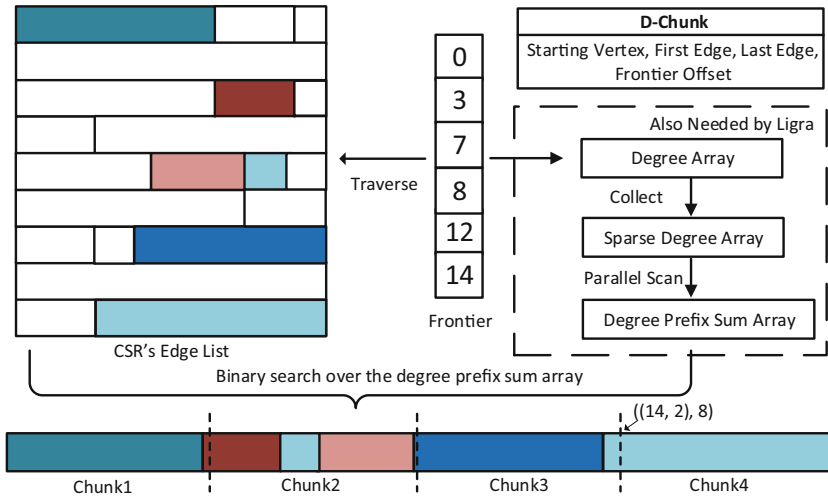


Fig. 4. D-Chunk

To build a chunk list dynamically in push-based execution, we extend the sparse frontier construction process used in Ligra [9], which requires calculating the prefix sum of the degree array. Figure 4 shows the building process of D-Chunk. An astute reader might notice that we need to rebuild the chunk list every time when entering push mode. This might sound problematic but actually building a chunk list for sparse frontier is very fast. Since we already have the prefix sum of the vertices’ degrees in the current frontier during the original construction process, the running time of building the chunk list is proportional to the logarithm of the frontier’s size. The additional work that D-Chunk does is a binary search to generate chunks with equal number of edges.

Each chunk only needs to maintain four data fields: the starting source vertex, the first and last edges, and the frontier offset. The first three data fields are used together with the current sparse frontier to iterate over the active edges. The frontier offset is a variable that helps collecting the vertices into the next sparse frontier. Since it is local to each chunk and there is no inter-chunk parallelism, the collecting process is conflict-free. Moreover, it generates sequential writes for each chunk. Hence the frontier maintenance is both CPU and memory friendly. Note that by using chunking in push mode, we can reuse the parallel scheduler in pull mode, which leads to better thread locality too. The actual scheduler is a simple thread pool implemented using a user-space thread barrier. Each thread is bound to a unique CPU core and the scheduler does round-robin work-stealing over the chunk list.

3 Implementations and Optimizations

Both `VR-Chunk` and `D-Chunk` are computational efficient but may require some amount of work to implement an actual graph algorithm based on them. As a result, we provide abstractions to hide the implementation details of the chunk internals. In this section we discuss the high level API design of `SilverChunk` and its optimizations.

3.1 Programming Interface

There are two commonly used APIs for graph processing systems: edge-based and list-based. `Ligra` [9] uses an edge-based API which allows users to only implement edge updating logic without caring about frontier maintenance. However, it also prevents the application to do customized optimizations since the actual execution context is limited to only one edge. On the other hand, `Gemini` [15] exposes a list-based API for the end users which allows application based optimizations, such as merging application states locally and doing vectorized processing. However, it requires the end users to maintain the next frontier in application code which is nontrivial for sparse frontiers. Therefore `Gemini` only uses dense frontiers. Moreover, a direct implementation of list-based API can lead to workload imbalance due to the skewed distribution of a input graph.

As a result, we adopt these two API styles into `SilverChunk` and propose a hybrid interface. For push mode, we use the edge-based API similar to `Ligra`. The main reason is, since we are already doing random writes in push mode, there is little chance for a list-based API to provide further optimizations. Instead, we can hide the nontrivial frontier maintenance from the end users. An actual implementation of graph algorithms in push mode is instantiated as a `push` operator. A `push` operator accepts a source vertex and a destination vertex. It requires synchronization when updating to the destination vertex. A `push` operator can return a boolean value indicating whether the destination vertex should be put into the next frontier. It can also return nothing so that any sane compilers will get rid of unnecessary instructions of the frontier maintenance.

For pull mode, we use the list-based API similar to `Gemini`. Thanks to our `VR-Chunk` scheme, giant vertices are already sliced, so workload balance is guaranteed. The running instance is called the `pull` operator. A `pull` operator accepts the starting and ending pointers of a source edge list, a real destination vertex and a destination vertex that might be real or virtual. Every update is guaranteed to be conflict-free when the `pull` operator is executed in parallel. The destination vertex is equal to the real destination vertex unless the vertex has its source edge list sliced by `VR-Chunk`. In that case, it is equal to the corresponding virtual vertex. In additional, pull mode also requires a `pull reduce` operator to be specified so that at the end of each iteration, all virtual vertices' states are merged to their corresponding real ones.

Listing 1 shows a vanilla implementation of the PageRank algorithm using the `SilverChunk`'s API. The `graph` argument contains the input graph data and

is able to run a graph algorithm. The `Algorithm` class is instantiated with the aforementioned three operators, written as C++ lambdas.

```
void PageRankFunction::run(Graph & graph) {
    ... // initialization code
    Algorithm algo(
        [&](UInt32 s, UInt32 d) { // push
            atomicAdd(pr_new[d], pr[s]); },
        [&](UInt32* b, UInt32* e, UInt32 rd, UInt32 d) { // pull
            Float y = 0;
            while (b < e) y += pr[*b++];
            pr_new[d] = y; },
        [&](UInt32 rd, UInt32 d) { // pull reduce
            atomicAdd(pr_new[rd], pr_new[d]); }
    );
    while (!finish) {
        graph.run(algo);
        ... /* other related code */ }
}
```

Listing 1. Page Rank Implementation

3.2 Optimizations

In the previous section we briefly described the polymorphism of the `push` operator, which enables optimizations when returning nothing. We call algorithms having this kind of operators `Immutable` since the frontier does not change after each iteration. We also identify other properties of graph algorithms for potential optimizations, as shown in Table 1. When all vertices are activated, the code path of propagating updates can be further optimized by removing unnecessary checks. We refer to this execution mode as `AllPull`.

Table 1. Algorithm properties

Algorithm	Immutable	Bypassable	Idempotent
PageRank	✓		
BFS		✓	✓
Components			✓
BellmanFord			✓

An algorithm is `Bypassable` if every vertex is supposed to be activated only once. An example is the simple breadth first search algorithm which finds any one traversing tree from the starting vertex. As shown in Listing 2, the `Algorithm` class accepts a `Bypassable` flag that checks if a vertex is already activated and can be bypassed for any further updates. When `Bypassable` is specified, the frontier maintenance does not interact with the application, hence it can be optimized statically. Note that the `pull reduce` operator is not needed in this algorithm.


```

void SimpleBFSFunction::run(Graph & graph) {
... // initialization code
Algorithm algo(
  [&](UInt32 s, UInt32 d) { // push
    parent[d] = s; },
  [&](UInt32* b, UInt32* e, UInt32 rd, UInt32 d) { // pull
    while (b < e)
      if (graph.isActive(*b)) { parent[rd] = *b; return; },
    Bypassable();
while (!finish) {
  graph.run(algo);
... /* other related code */ }}

```

Listing 2. Simple BFS Implementation

An algorithm is **Idempotent** if algorithm correctness is not affected by propagating updates from inactive vertices to their neighbors. An example is the label propagation algorithm for computing connected components. As shown in Listing 3, the **Algorithm** class accepts a **Idempotent** threshold that switches to **AllPull** execution when current frontier’s density is greater than the threshold. The reason of specializing this property is because when frontiers are near full, **AllPull** is faster than normal pull mode.

```

void LabelPropagationFunction::run(Graph & graph) {
... // initialization code
Algorithm algo(
  [&](UInt32 s, UInt32 d) { // push
    return writeMin(id[d], id[s]); },
  [&](UInt32* b, UInt32* e, UInt32 rd, UInt32 d) { // pull
    UInt32 m = MAX_UINT32;
    while (b < e) if (graph.isActive(*b)) m = min(m, *b);
    if (m < id[rd]) { id[d] = m; return true; }
    return false; },
  [&](UInt32 rd, UInt32 d) { // pull reduce
    writeMin(id[rd], id[d]); },
  Idempotent(0.5));
while (!finish) {
  graph.run(algo);
... /* other related code */ }}

```

Listing 3. Label Propagation Implementation

4 Experiments

In this section, we evaluate **SilverChunk**’s performance using a physical server with four applications (PageRank, BFS, WCC and BellmanFord) and five datasets (RMat24, RMat27, Twitter, Powerlaw and USARoad). The physical server contains two Intel Xeon E5-2640v4 CPUs with 128 GB memory. We synthesized graphs using the R-MAT generator, following the same configuration used by

the graph500 benchmark. The synthetic power-law graph (PowerLaw) with fixed power-law constant 2.0 was generated using the tool in PowerGraph [4], which randomly samples the degree of each vertex from a Zipf distribution and then adds edges. We also use two types of real-world datasets, a social network graph (`twitter-2010`¹) and a geometric graph (`USARoad`²). All graphs are unweighted except `USARoad`. To provide a weighted input for the SSSP algorithm, we add a random edge weight in the range [1, 100] to each edge. Following Table 2 shows the basic information of used datasets .

Table 2. Data set

Dataset	Vertex Num	Edge Num	Avg Deg	Max Indeg	Max Outdeg	Size (CSV)
RMat24	16M	0.3B	16.0	18.0K	17.3K	4.0 GB
RMat27	134M	2.1B	15.8	0.90M	0.86M	34 GB
Twitter	42M	1.5B	35.3	0.77M	3.0M	25 GB
Powerlaw	10M	0.1B	9.2	10	2.1M	1.4 GB
USARoad	23M	58M	2.4	9	9	1.3 GB

We compare `SilverChunk` to a number of different in-memory graph engines. Primarily, we compare `SilverChunk` with Ligra [9], Polymer [13], Gemini [15], Grazelle [5] and Galois [8] as these systems achieves state-of-the-art performance on a single-machine environment using in-memory storage. We run these systems with four graph algorithms on five different data sets using two different configuration of one commodity machine (Dell PowerEdge R730xd). We run iterative algorithms like Pagerank (PR) as well as traversal algorithms such as Bellman-Ford (BF) algorithm on these engines. This allows a comparison on how well a graph engine can handle different kinds of graph algorithms with different graph data distributions. The detailed information of the evaluated graph algorithms are as follow:

PageRank (PR) computes the rank of each vertex based on the ranks of its neighbors. We use the synchronous, pull-based PageRank in all cases and apply the division elimination optimization to all applications except Grazelle.

Breadth-first search (BFS) traverses an unweighted graph by visiting the sibling vertices before visiting the child vertices. The source is vertex one for this test.

Connected components (CC) calculates a maximal set of vertices that are reachable from each other for a directed graph. All systems adopt label propagation algorithm except Galois, which provides a topology-driven algorithm based on a concurrent union-find data structure.

¹ <http://law.di.unimi.it/datasets.php>.

² <http://www.dis.uniroma1.it/challenge9/>.

Table 3. Running times (in seconds) of algorithms over various data sets

System	Data set	PR (5 iterations)		BFS		CC		SSSP	
		one cpu	two cpus	one cpu	two cpus	one cpu	two cpus	one cpu	two cpus
SilverChunk	R-Mat24	<u>1.35</u>	<u>0.84</u>	<u>0.13</u>	<u>0.10</u>	<u>0.79</u>	<u>0.48</u>	<u>3.50</u>	<u>2.49</u>
	R-Mat27	<u>9.52</u>	<u>5.86</u>	<u>0.62</u>	<u>0.42</u>	<u>5.56</u>	<u>2.69</u>	<u>7.66</u>	<u>4.63</u>
	Twitter	<u>4.55</u>	<u>2.64</u>	<u>0.41</u>	<u>0.30</u>	<u>4.49</u>	<u>2.35</u>	<u>7.54</u>	<u>4.59</u>
	Powerlaw	<u>0.34</u>	<u>0.21</u>	<u>0.13</u>	<u>0.10</u>	<u>0.53</u>	<u>0.27</u>	<u>0.93</u>	<u>0.64</u>
	US Road	<u>0.36</u>	<u>0.23</u>	<u>0.55</u>	<u>0.80</u>	<u>23.18</u>	<u>15.01</u>	<u>117.29</u>	<u>70.19</u>
Ligra	R-Mat24	2.78	1.81	0.23	0.20	1.93	1.03	<u>3.84</u>	<u>2.51</u>
	R-Mat27	19.13	14.80	1.07	1.06	13.32	7.62	7.84	5.08
	Twitter	9.18	6.69	0.68	0.61	10.97	6.75	<u>7.65</u>	<u>5.03</u>
	Powerlaw	0.94	0.72	0.18	<u>0.12</u>	1.44	0.98	<u>1.26</u>	0.93
	US Road	0.88	0.65	1.46	1.57	62.42	40.12	169.23	87.26
Polymer	R-Mat24	4.71	1.88	0.26	0.22	1.64	0.80	4.23	2.53
	R-Mat27	43.98	19.08	1.36	1.04	13.82	6.58	9.48	4.91
	Twitter	28.82	12.02	0.79	0.65	16.51	8.61	7.69	5.15
	Powerlaw	1.54	0.71	0.18	0.20	1.58	1.02	1.29	0.73
	US Road	0.61	0.52	1.21	1.25	82.94	45.59	258.03	180.71
Gemini	R-Mat24	<u>1.52</u>	<u>0.85</u>	0.18	0.14	3.12	1.35	7.06	3.55
	R-Mat27	<u>9.64</u>	<u>6.14</u>	0.86	0.76	18.28	8.77	16.21	8.14
	Twitter	<u>4.88</u>	<u>2.56</u>	0.56	0.74	19.06	9.84	12.66	6.51
	Powerlaw	0.46	0.41	0.15	0.23	1.25	0.54	1.39	<u>0.72</u>
	US Road	0.61	0.31	20.42	21.64	176.23	123.24	533.04	379.54
Grazelle	R-Mat24	2.18	1.42	<u>0.14</u>	<u>0.13</u>	<u>1.02</u>	<u>0.63</u>	No Impl	
	R-Mat27	13.30	9.05	<u>0.69</u>	<u>0.70</u>	<u>7.67</u>	<u>4.36</u>		
	Twitter	6.27	3.81	<u>0.54</u>	<u>0.44</u>	6.27	4.47		
	Powerlaw	<u>0.45</u>	<u>0.33</u>	<u>0.14</u>	0.13	0.88	0.43		
	US Road	<u>0.39</u>	<u>0.22</u>	2.91	1.85	26.23	15.66		
Galois	R-Mat24	5.09	2.72	0.61	0.32	1.04	0.64	4.40	4.18
	R-Mat27	36.93	20.48	4.14	2.41	7.87	4.90	<u>7.40</u>	<u>4.48</u>
	Twitter	10.47	6.12	2.19	1.55	<u>3.86</u>	<u>2.48</u>	90.92	69.74
	Powerlaw	1.71	0.86	0.38	0.22	<u>0.51</u>	<u>0.35</u>	3.88	4.27
	US Road	4.40	2.15	<u>0.33</u>	<u>0.30</u>	<u>0.81</u>	<u>0.45</u>	<u>0.90</u>	<u>1.02</u>

Fastest time is denoted as underline. Second fastest time is denoted as underwave.

Single-source shortest-paths (SSSP) computes the distance of the shortest path from a given source vertex to other vertices. The source is vertex one for this test. All systems implement SSSP based on the Bellman-Ford algorithm with synchronously data-driven scheduling, while Galois uses a data-driven and asynchronously scheduled delta-stepping algorithm.

4.1 Graph Algorithm Test

Table 3 gives a complete runtime comparison. Of all the test cases, we report the execution time of their five runs. For PageRank algorithm, **SilverChunk** achieves optimal performance against other systems using only one CPU. Gemini and Grazelle are the second best. With two CPUs enabled, systems like Polymer, Gemini and Grazelle scales better than **SilverChunk**, however **SilverChunk** still holds three best results out of five. On the other hand, 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, resulting in fewer memory accesses. Therefore, **SilverChunk** outperforms all other systems except Galois, which either adopts different algorithms for the problem or uses specialized scheduler for asynchronous execution. In most test cases, **SilverChunk** takes a leading position, except the **USRoad** graph. For high-diameter graphs like **USRoad**, the asynchronous scheduling and special implementations in Galois are able to exploit more parallelism for the graph traversal algorithms, such as CC and SSSP. In general, our graph chunking technique achieves 99% of CPU usage without any dynamic coordination in pull mode. It also gives consistent load balance in push mode.

4.2 VR-Chunk Test

As can be seen from Fig. 5, compared to other systems, **VR-Chunk** does not introduce pre-processing overheads, while still achieves the best performance. Figure 6 compares the running time of the PageRank algorithm on the twitter graph with three different implementations: Cilk [2], **VR-Chunk** and **VR-Chunk** with work-stealing. The static execution of **VR-Chunk** already excels the Cilk scheduler. Adding a simple chunk-based work-stealing mechanism gives another 10% performance gain.

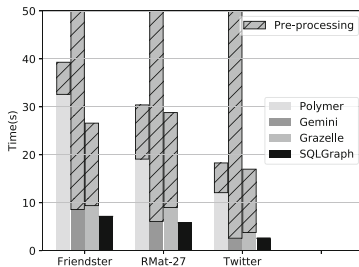


Fig. 5. Comparison among different systems

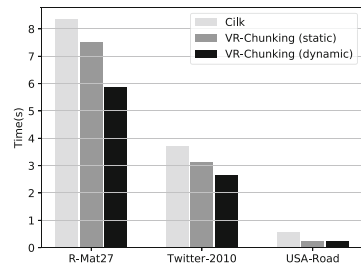


Fig. 6. Comparison with hand-written code

4.3 AllPull Test

We test different thresholds of **AllPull** execution combined with adaptive Push-Pull switching. Figure 7 shows the test result of running the Connected Components algorithm. With **AllPull** mode enabled, we get 30% performance gain. All three different data set achieve the best running time when the threshold is between 0.3 and 0.5. Therefore it can serve as a proper reference value for optimizing idempotent algorithms.

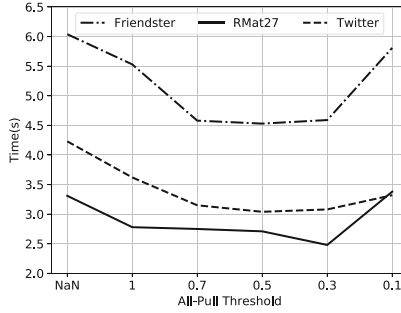


Fig. 7. Connected components execution time with different **AllPull** thresholds

4.4 NUMA and Cache Optimization Test

Since NUMA based engine Polymer [13] does not reveal proper performance, and cache based engine Cagra [14] does not open source their code, we implement both optimization schemes in order to complete our testing. We also combine NUMA and cache optimizations along with the optimizations used in **SilverChunk**. As can be seen from Table 4, both NUMA or cache optimizations can effectively improve the performance. The last column lists the memory consumption with values related to the lowest one. Cache optimization gives better running time than NUMA optimization but it introduces a huge amount of memory consumption and pre-processing time. **SilverChunk** gives further improvements in all optimization combinations, and it is more effective when there is no NUMA or cache optimization applied, which suggests that **SilverChunk** not only balances workloads, but also optimizes memory accesses. Notice that both NUMA and cache optimizations in this test have their pre-processing time longer than the actual running time. As a result, Whether to enable such optimization needs further considerations.

Table 4. PageRank (5 iters) over Twitter-2010

	Nested loop	VR-Chunk	Pre-processing	Peak memory
No NUMA, No Cache	3.13 s	2.64 s	0 s	1.0
NUMA, No Cache	2.28 s	2.08 s	3.83 s	1.05
No NUMA, Cache	1.91 s	1.68 s	6.52 s	1.56
NUMA, Cache	1.67 s	1.55 s	11.84 s	1.64

5 Related Works

The field of single machine graph processing in main memory has seen efforts in both parallel scheduling and graph partitioning. Ligra [9] proposes an EdgeMap

interface to hide the inner loop parallelism, however it does not solve the actual workload imbalance issue. Grazelle [5] adopts a schedule-aware to achieve workload balance which however makes graph applications hard to implement. Polymer [13], Gemini [15] and Grazelle [5] are exponents in NUMA optimizations. They partition graph into subgraphs for each NUMA node, trying to reduce remote memory access. However it takes more time in pre-processing and its effectiveness is related to the graph data distribution and the actual running modes. For sparse frontiers, pre-partitioned graphs are less effective. Systems like GRACE [12] and Cagra [14] partition the input graph even further, at the CPU cache level. Cagra manually partitions the graph in order to make sure one batch of concurrent workload would end up only reading data from CPU's LLC. However, this adds a lot of complexity to the initialization process, and similar to NUMA-aware partitioning, it barely helps when the frontiers are sparse. Graph-Grind [10] uses partition-based optimization only when the frontier's density exceeds certain threshold, which is 50% in their experiments, while still keeps the vanilla CSR/CSC formats for sparse and medium-dense frontiers. However, they add one additional copy of the graph data to store the partitioned graph, resulting in 50% more memory consumption.

6 Conclusion

We present **SilverChunk**, an efficient in-memory parallel graph processing system running on a single machine. **SilverChunk** solves the workload imbalance issue of frontier-based computing by unfolding the nested loop into a flat loop over a chunk list. We extend the chunking scheme to support both pull and push modes and provide a unified high level API for implementing graph applications. In addition, we address new optimization opportunities based on different execution modes and algorithm properties, and use a policy based API to automatically apply the corresponding optimizations. Currently **SilverChunk** cannot handle graphs too big to fit into main memory. We plan to extend the ideas presented in this paper to external memory and distributed environment in near future.

Acknowledgements. We thank anonymous reviewers whose comments helped improve and clarify this manuscript. This work is funded by the National Key Research and Development Program of China.

References

1. Besta, M., Podstawski, M., Groner, L., Solomonik, E., Hoefler, T.: To push or to pull: on reducing communication and synchronization in graph computations. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp. 93–104. HPDC 2017, ACM, New York (2017). <https://doi.org/10.1145/3078597.3078616>

2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 1995, pp. 207–216. ACM, New York (1995). <https://doi.org/10.1145/209936.209958>
3. Dagum, L., Menon, R.: OpenMP: an industry standard api for shared-memory programming. *IEEE Comput. Sci. Eng.* **5**(1), 46–55 (1998). <https://doi.org/10.1109/99.660313>
4. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), pp. 17–30. USENIX, Hollywood (2012)
5. Grossman, S., Litz, H., Kozyrakis, C.: Making pull-based graph processing performant, pp. 246–260. ACM Press (2018). <https://doi.org/10.1145/3178487.3178506>
6. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In: Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pp. 349–362. ACM, New York (2012). <https://doi.org/10.1145/2150976.2151013>
7. Malicevic, J., Lepers, B., Zwaenepoel, W.: Everything you always wanted to know about multicore graph processing but were afraid to ask. In: 2017 USENIX Annual Technical Conference (USENIX ATC 2017), pp. 631–643. USENIX Association, Santa Clara (2017)
8. Pingali, K., et al.: The tao of parallelism in algorithms. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 12–25. ACM, New York (2011). <https://doi.org/10.1145/1993498.1993501>
9. Shun, J., Blleloch, G.E.: Ligra: a lightweight graph processing framework for shared memory, p. 135. ACM Press (2013). <https://doi.org/10.1145/2442516.2442530>
10. Sun, J., Vandierendonck, H., Nikolopoulos, D.S.: GraphGrind: addressing load imbalance of graph partitioning. In: Proceedings of the International Conference on Supercomputing, ICS 2017, pp. 16:1–16:10. ACM, New York (2017). <https://doi.org/10.1145/3079079.3079097>
11. Sundaram, N., et al.: GraphMat: high performance graph analytics made productive. *Proc. VLDB Endowment* **8**(11), 1214–1225 (2015). <https://doi.org/10.14778/2809974.2809983>
12. Wang, G., Xie, W., Demers, A.J., Gehrke, J.: Asynchronous large-scale graph processing made easy. In: CIDR, vol. 13, pp. 3–6 (2013)
13. Zhang, K., Chen, R., Chen, H.: NUMA-aware graph-structured analytics, pp. 183–193. ACM Press (2015). <https://doi.org/10.1145/2688500.2688507>
14. Zhang, Y., Kiriansky, V., Mendis, C., Amarasinghe, S., Zaharia, M.: Making caches work for graph analytics. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 293–302, December 2017. <https://doi.org/10.1109/BigData.2017.8257937>
15. Zhu, X., Chen, W., Zheng, W., Ma, X.: Gemini: a computation-centric distributed graph processing system. In: OSDI 2016, pp. 301–316. USENIX Association (2016)