View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Apollo

PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal

Karthik Nilakant
University of Cambridge
karthik.nilakant@cl.cam.ac.uk

Valentin Dalibard
University of Cambridge
valentin.dalibard@cl.cam.ac.uk

Amitabha Roy
EPFL
amitabha.roy@epfl.ch

Eiko Yoneki
University of Cambridge
eiko.yoneki@cl.cam.ac.uk

ABSTRACT

Mining large graphs has now become an important aspect of multiple diverse applications and a number of computer systems have been proposed to provide runtime support. Recent interest in this area has led to the construction of single machine graph computation systems that use solid state drives (SSDs) to store the graph. This approach reduces the cost and simplifies the implementation of graph algorithms, making computations on large graphs available to the average user. However, SSDs are slower than main memory, and making full use of their bandwidth is crucial for executing graph algorithms in a reasonable amount of time. In this paper, we present PrefEdge, a prefetcher for graph algorithms that parallelises requests to derive maximum throughput from SSDs. PrefEdge combines a judicious distribution of graph state between main memory and SSDs with an innovative read-ahead algorithm to prefetch needed data in parallel. This is in contrast to existing approaches that depend on multi-threading the graph algorithms to saturate available bandwidth. Our experiments on graph algorithms using random access show that PrefEdge not only is capable of maximising the throughput from SSDs but is also able to almost hide the effect of I/O latency. The improvements in runtime for graph algorithms is up to 14$\times$ when compared to a single threaded baseline. When compared to multi-threaded implementations, PrefEdge performs up to 80% faster without the program complexity and the programmer effort needed for multi-threaded graph algorithms.

1. INTRODUCTION

Mining graph structured data is becoming increasingly important for numerous applications, ranging across the domains of bioinformatics, social networks, computer security and many others. However, the growing scale of real world graphs, which is expected to reach billions of vertices in the near future, has made efficient execution of graph algorithms particularly challenging. Traditional computer systems for computation on large data sets, such as MapReduce [14] and Hadoop [1], are usually inefficient at performing graph computations due to the large amount of data propagation needed when compared to actual computations [27]. This lack of locality in the data has lead to the assumption that processing large graphs necessarily requires them to be loaded entirely in main memory. A number of distributed graph processing platforms [28, 15] have been designed around this idea, offering a high performance solution to the problem.

A recent trend in research has been the development of graph computation systems on single computers with limited amounts of main memory. This is an attractive proposition for mainstream graph mining on low budgets, provided that the cost benefit outweighs any performance impact caused by switching to the use of external storage. A key performance driver with respect to graph programs is the ability to respond to random-access I/O requests, which is not a strength of traditional hard disk drives. On the other hand, Solid state drives (SSDs) are still far cheaper (an order of magnitude) than main memory and can provide higher random-access throughput than traditional magnetic media. However, deriving maximum throughput from such devices is contingent on the ability to apply parallelism to the I/O access pattern of a process.

There are two existing approaches of particular interest, which make use of external storage for processing graphs. Pearce et al. [33] advocate the parallelisation of graph algorithms, allowing the generation of multiple parallel I/O requests to external storage, boosting the throughput of the external storage device. Kyrola et al's approach [22] is to alter the structure of external data and prescribe a processing pattern that allows storage device's data throughput capability to be saturated, by exclusively employing a sequential access pattern.

We present an alternative to both these approaches: a graph prefetcher called PrefEdge that is able to saturate an SSD using random accesses from a single thread of execution. We are concerned with traversing large graphs in various vertex orders: sequential, breadth-first, shortest-path, and other graph algorithms on a single machine. During traversal of a graph with vertex set V and edge set E, we place $O(|V|)$ amount of data in expensive main memory together with a small constant sized cache; leaving $O(|E|)$ amount of data on the cheaper SSD. Although this needs more memory than a purely external memory algorithm, we believe the mix of resource requirements is practical, given the capabilities of current commodity hardware. For example, we run a single source shortest path traversal over a dataset from Twitter [2] containing approximately 52 million vertices and 1.6 billion edges on a single machine. We placed 2.74 GB of vertex map data in main memory and left an additional 2 GB of main memory as cache. This 4.74 GB of required main memory is in contrast to approximately 33 GB of edge related data on the SSD.

While the basic premise of prefetching is not new [32, 26], our system does not require developers to modify their algorithms to explicitly send hints to the prefetcher. Instead, PrefEdge interacts with three common types of graph iterator, allowing programs that use these iterators to benefit from the prefetcher. Our experiments show that PrefEdge is capable of hiding the effect of I/O latency in single threaded graph traversals, such as breadth-first and single-source shortest paths, resulting in an improvement ranging up to 14 times faster than an unassisted algorithm. When compared with multi-threaded graph algorithms, our prefetcher offers comparable performance, and in some cases operates up to 80% faster. We demonstrate that PrefEdge is able to deliver these benefits for a variety of different graph types and is therefore relatively insensitive

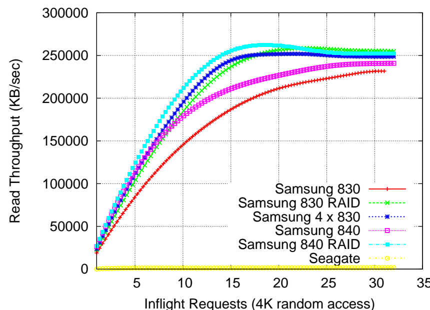


Figure 1: SSD I/O throughput: Random Read

to the structure of the graph.

We begin by describing the characteristics of general graph processing models in Section 2. We then review the properties of the SSD as an I/O device in Section 2.2. PrefEdge's architecture is presented in Section 3, including the format we used to represent graphs in Section 3.1. The results of our evaluation of PrefEdge is detailed in Section 4, followed by a discussion on its cost and limitations in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. GRAPH PROCESSING WITH SOLID-STATE STORAGE

In this section we give a brief summary of the properties of graphs and graph algorithms, and how using SSDs to store graph data relates to these properties.

2.1 Graph Algorithms

We distinguish between three main graph algorithm classes: traversal algorithms, fixed point iterations and characterisation algorithms.

Traversal algorithms involve iterating through vertices of the graph in a graph dependent ordering. Vertices can be traversed multiple times. This class includes search algorithms (such as breadth-first, depth-first and heuristic search), single source shortest paths, connectivity algorithms, maximum flow algorithms, minimum spanning tree algorithms and many more.

Fixed Point Iteration algorithms involve iterating over the entire graph multiple times until a convergence condition is reached. They can be implemented efficiently using the Bulk Synchronous Parallel model. This includes a number of graph analytics algorithms such as PageRank, triangle counting and diameter calculation.

Finally, characterisation algorithms perform a single computation over the entire graph and are usually more complex than the two classes above. They include linear algebra type algorithms such as spectral graph algorithms.

Our focus is on the first class. Parallelising traversal algorithms can be difficult as the next vertex to be traversed is often dependent on the current computation. Parallel implementations of such algorithms typically require some form of concurrency control, workload partitioning, or thread-level optimisations for optimum performance. We note that in a semi-external memory scenario, the dominant component of running time will be I/O latency. In our system, rather than attempting to parallelise the entire computation, we focus on parallelising I/O access.

2.2 SSD I/O Characteristics

Unlike magnetic media, SSDs can service multiple random access requests in parallel without suffering degradation in the latency of servicing an individual request. We refer to the number of requests that can be handled concurrently in the paper as the number of "inflight" requests (also referred to in literature as the queue depth).

To illustrate the impact of varying the inflight target, we compare the performance characteristics of some commercially available storage devices: a Samsung 830 series SSD, a Samsung 840 SSD, a RAID-0 array containing 4 Samsung 830 SSDs, and a Seagate hard disk drive. We used the Flexible I/O (`fio` [3]) benchmarking tool to obtain the response curve for throughput (measured in kilobytes of data read per second) with a varying number of inflight requests for that device. Figure 1 shows how throughput varies in a manner consistent with Little's Law [24], as the number of inflight requests for 4KB pages is increased.

As described earlier, the I/O access pattern exhibited by sequential traversal algorithms is not conducive to producing a large number of inflight requests. Typically a single I/O request will be made for each vertex processed. Our evaluation shows that this leads to very large I/O stalling rates. The aim of our system is to reduce this by identifying future vertices and retrieving the associated data from external storage ahead of time.

2.3 Processing Models

With PrefEdge, our processing model is relatively straightforward. We couple sequential (single-threaded) graph processing algorithms with a background prefetcher. The prefetcher ensures that edge data (which is stored on the SSD) is available in memory when the processing algorithm needs it. This reduces I/O stalling time, allowing the algorithm to complete more quickly.

An alternate approach would be to employ parallelism within the graph processing algorithm itself. Rather than stalling every time an I/O request occurs, this would allow other processing threads to progress while a request is being serviced. Furthermore, multiple threads can concurrently request blocks from external storage, thereby increasing the number of inflight requests to the storage device and increasing throughput. However, developing an efficient multi-threaded implementation of an algorithm can be challenging: one must handle concurrent accesses to shared data in a consistent manner, which necessitates the use of various synchronisation mechanisms. Often, architecture-specific optimisations may need to be employed to implement efficient locking or atomic operations on shared data.

Our evaluation of PrefEdge shows that prefetching offers comparable performance to algorithmic parallelism, in some cases outperforming the multi-threaded alternative. However, for algorithms where the prefetcher is able to eliminate I/O waiting times completely, employing parallelism within the processing algorithm may be the only way to further increase performance. For example, our evaluation of the PageRank algorithm revealed that I/O stalling times could be eliminated almost completely by the prefetcher, meaning that the single-threaded computation could be easily outperformed by a parallelised implementation.

2.4 OS support for I/O latency optimisation

Modern operating systems contain a number of optimisations for mitigating the operational latency of external storage. Filesystem caching and storage scheduling techniques such as request re-ordering and coalescing are now mature concepts that are in widespread use. Indeed, the basic concept of filesystem readahead is not novel [10]; many operating systems and storage controllers implement sequential prefetching as a standard feature.

Unfortunately, these existing techniques do not always suit the purposes of our problem area. In graph traversal, data is rarely reused,

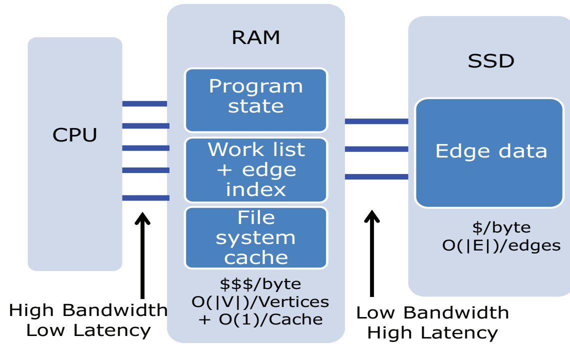


Figure 2: Semi-external data layout

which hinders the effectiveness of caching, especially when cache capacity is much smaller that the size of external data. Furthermore, graph traversal algorithms often exhibit a random access pattern that cannot be optimised using sequential prefetching. In the end, OS support techniques usually affect performance negatively, as system resources are spent retrieving and storing useless data, and possibly evicting useful data from the cache. Our objective with PrefEdge is to provide an application-driven framework for *random access prefetching*, identifying and retrieving data from an external device before it is processed by the application. When stalling is unavoidable, we aim to maximise the throughput of the SSD by issuing multiple requests in parallel.

3. PREFEDGE SYSTEM ARCHITECTURE

There are two basic design aspects that make up the PrefEdge system: we economise on resource costs by adopting a semi-external storage model, whereby bulky edge data is stored on one or more SSDs; secondly, we attempt to offset the performance impact of using external storage by employing a prefetcher that can interact with a variety of graph programs.

3.1 Data Layout and Storage Format

A conceptual diagram of memory usage in our system is shown in Figure 2. The most relevant components of our system are a CPU, RAM and persistent storage in the form of an SSD. We lock program state and the adjacency list index in memory, and leave all edge data on external storage. We process graphs stored in the Compressed Sparse Row (CSR) format. The CSR format is aimed at efficient storage of sparse matrices, in our case we use it to store the sparse adjacency matrix of the graph. This approach is consistent with GraphChi [22] and the work of Pearce et al.[33]. The basic principles of our system could be applied to other storage formats, however we chose CSR for the sake of comparability. The CSR format consists of two components: the row index and column index. The row index of A_G is a vector R_G of size $|V|$ with $R_G[i]$ being the index of the first non-zero element of row i in the column index. The column index of A_G is a vector C_G of size $|E|$ which is a row-wise listing of the column numbers of those elements in A_G, which are non-zero. We note that the traditional CSR format includes an auxiliary value vector paired with the column index that actually stores the values in the sparse matrix. Since the adjacency matrix is binary, we dispense with the value vector in the representation as the value is implicitly always `1`.

Figure 3 illustrates how a directed graph of 3 nodes is stored in CSR format. It should be evident that we can recover the original adjacency matrix from the CSR format. It should also be evident that CSR permits direct access to the set of neighbours of a vertex through the row index and iteration over that set. The termination point of the iterator is determined by looking up the start of the next row from R_G.

We process graphs by storing the $O(V)$ sized index R_G in main memory while leaving the $|E|$ sized C_G on the SSD. The objective of PrefEdge is to minimise the cost of I/O to move edges related information in C_G from the SSD into main memory for processing. For the rest of this paper we refer to C_G using the better known and functionally equivalent term 'adjacency list.'

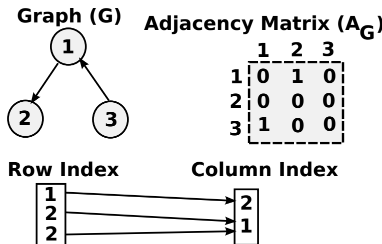


Figure 3: Example of CSR format

Algorithm 1 PrefEdge Prefetcher

Require: v is the current position of the vertex iterator
Require: IFT is the inflight-target
Require: `Issued` is a constant sized hash of issued requests
  `Needed` := Read-ahead(IFT)
  for all x in `Needed` considered in order **do**
    if $x \notin$ `Issued` **then**
      Issue prefetch request for CSR page of x
      Add x to Issued

3.2 Prefetcher Design

PrefEdge is based on the observation that a large majority of graph algorithms are built around different kinds of iterators. Informally an iterator for a graph consists of a current vertex(v), internal state (S) and the graph being iterated on (G). An iterator therefore may be represented by the triple $< v, S, G >$. The current vertex is simply the projection $\pi_{\texttt{vertex}}(< x, S, G >) = x$.

An iterator supports the function $\texttt{Next}(< v, S, G >) =< v', S', G >$. PrefEdge depends on the iterator being separable from the graph. This means that there exists a read-ahead function R and an integer $k(S) > 0$ such that $\forall i \leq k(S) : R^i(< v, S >) = \pi_{\text{vertex}}(\text{Next}^i(< v, S, G >))$. The read-ahead function is therefore able to determine the next k vertices accessed, without reference to the graph G.

PrefEdge contains two components:

1. Prefetch: Given a sequence of vertices to be accessed issue a set of requests for the corresponding filesystem pages to the SSD to achieve the optimum number of inflight requests. The prefetcher invokes the read-ahead component with a target T.
2. Read-ahead: The read-ahead component computes the result of an iterator-specific read-ahead function to determine at most the next T vertices: $\{R^{min(k(S),T)}(< v, S >)\}$. It returns an ordered list.

3.3 Prefetch

The prefetcher is responsible for issuing requests to the SSD based on the Read-ahead function and is parameterised by the inflight target (IFT). We identify the sequential and random inflight targets from the response curve described in the previous section. The prefetching algorithm used in PrefEdge is shown in Algorithm 1.

The PrefEdge prefetcher is based on the simple observation that

execution of the graph algorithm is synchronous and therefore stalls on an I/O. Hence, starting from the current position of the vertex iterator (where the algorithm is likely stalled) the prefetcher issues requests for the next `IFT` number of needed vertices from the CSR file.

3.4 Read-ahead

In this paper, we discuss read-ahead functions for three types of iterators. We use the example of discovering connected components using breadth-first search shown in Figure 4 for illustration. That example contains two iterators: the first is a simple sequential iterator over the vertices at Line `2` while the second is a breadth-first iterator at Lines `7 -8`. We discuss each of these in turn followed by a simple extension to the breadth-first search iterator, the priority queue iterator.

3.4.1 Sequential Iterator:

The sequential iterator iterates over the vertices in turn. Therefore, given the current vertex number v the next vertex is simply $v + 1$. The read-ahead function therefore is simply $R^i(< v, S >) = I(v) + i$ (where $I(v)$ is the vertex number corresponding to vertex v). $k(S)$ is just the remaining number of vertices to iterate over.

3.4.2 Breadth-first Iterator:

The breadth-first iterator state is maintained in a queue of vertices to be visited and this queue is accessed in First In First Out (FIFO) order. If the size of the queue is k then the next k vertices are simply the next k elements of the queue in order. Therefore $k(S)$ is the size of the FIFO queue encapsulated by the state S and $R^i(< v, S >)$ is simply the ith element of the FIFO queue encapsulated in S.

3.4.3 Priority Queue Iterator:

Workloads such as single-source shortest path (SSSP) [12] are essentially like breadth-first search but replace the FIFO queue with a priority queue of vertices. Each vertex is assigned a (changing) weight from the time when it is discovered and put in the priority queue to when it is finally removed at the point where it has the minimum weight among all elements in the priority queue. The fact that weight can change during execution renders it difficult to come up with $R^i(< v, S >)$ without reference to the underlying graph G. However a reasonable approximation is to ignore changes to the weights in the priority queue as we iterate over it. We then pick the top k elements from the priority queue where k is much smaller than the number of elements in the priority queue. This approximation depends on the top k elements not changing during the next k steps of the iterator, an assumption that tends to hold in practise as weight updates are bounded below by the weight of the vertex that is removed from the priority queue.

There are a number of priority queue data structures that allow selection of the top k elements. A simple solution is to fix k and then use a sorted array for the top k vertices while the remaining vertices are maintained in a standard heap, keeping asymptotic bounds the same as a heap for all the elements. Since PrefEdge is tolerant of incorrect vertex selection (as incorrectly predicted pages are simply left unused in cache), we use an even simpler approximation in our implementation: we use a binary heap (of dynamic size $|S|$) stored in an array (of size $|V|$) and consider the top $min(T, |S|)$ elements of the array for each request for T vertices from the prefetch component. We show in our evaluation that in spite of these approximations, PrefEdge provides good speedups for SSSP.

```
components = 0; for(i=0;i<vertices;i++) {      1
  if(!vertices[i].visited) {                     2
    components++;                                3
    bfs_queue.push_end(i);                       4
    visited[i]=TRUE;                             5
    while(!bfs_queue.empty()) {                  6
      v = bfs_queue.pop_front();                 7
      CSR_FOREACH_NEIGHBOUR(v, x) {              8
        if(!visited(x)) {                        9
          bfs_queue.push_end(x);                 10
          visited[x] = TRUE;                     11
}}}}}                                            12
                                                 13
```

Figure 4: Connected components for an undirected graph

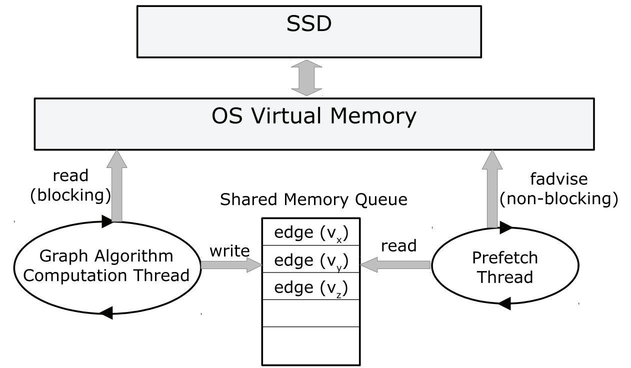


Figure 5: PrefEdge Architecture

3.5 Handling Multiple Iterators

We use two prefetchers, one for sequential iterators and the other for random access iterators. This is driven by the observation that SSDs demonstrate very different I/O characteristics for sequential reads as compared to random reads and therefore we use a different prefetcher for each type, with a different setting for the inflight target (IFT) for each. We allow the operating system scheduler to divide available I/O bandwidth between these two types of request streams.

We handle multiple instances of the same type of iterator by partitioning the quota of inflight requests between them. The allocation depends on the nesting of iterators. We allocate the entire IFT budget first to the innermost iterator and then move up allocating leftover inflight request budgets to the next outer iterator. For two iterators at the same level, we allocate equally to the two, interleaving prefetch requests from each iterator. The rationale behind this allocation strategy is to prefetch pages from the CSR file in the order in which they are required by the synchronously executing graph algorithm.

Multiple iterators of the same type occur in this paper with SSSP. We store edge weights in a separate file and therefore we have two iterators at the same level with SSSP, one reading the structure from the CSR file and the other reading edge weights from the property file. Following the strategy outlined above, we allocate half of the IFT budget for the random access prefetcher to the CSR file and the other half to the property file.

3.6 Implementation

The core components of our current implementation of PrefEdge are illustrated in Figure 5. We have implemented PrefEdge as C libraries that are used by algorithms and iterators written in C. At startup, vertex metadata is initialised and locked in memory, including the CSR index which is loaded from storage. The CSR adjacency list file and auxiliary edge weights are accessed using memory-mapped I/O; attempts to read pages that have not been fetched already will result in a major page fault and stalling.

In a separate thread, the prefetcher invokes any registered callbacks, which access the current state of the main program's iterator. The callback converts vertex IDs to page addresses, returning batches of pages for the prefetcher to retrieve. Asynchronous page load requests to the operating system are issued via the standard `posix_fadvise` interface using the `WILLNEED` hint. Such requests immediately initiate a fetch of the referenced page by the virtual memory subsystem. This process is repeated continuously, ensuring that pages containing future data move quickly to the active LRU list. After the data from a page has been used, its eventual eviction from the cache is managed by the OS.

4. EVALUATION

We now present an evaluation of PrefEdge to analyse its performance in the light of its design goal: accelerating SSD-based graph traversal through prefetching. Our evaluation highlights the following aspects of PrefEdge's performance:

- Prefetching allows sequential algorithms to run up to 14 times faster. The resultant processing rate approaches that of completely in-memory execution. The prefetcher continues to deliver these benefits even in memory constrained environments.
- PrefEdge is applicable across a range of graph types and algorithms.
- When compared to the alternative of multi-threading, PrefEdge can offer superior performance without the program complexity and programmer effort needed for multithreaded graph algorithms.
- We show that PrefEdge's performance can be superior to GraphChi [22], a similar external memory graph engine that makes use of sequential access.

We have used PrefEdge to process a graph with 1 billion vertices with 40 billion edges on a single industry standard server illustrating the usefulness and scalability of our approach.

4.1 PrefEdge-Assisted Algorithms

As described earlier, we developed read-ahead functions for three different types of iterators: FIFO queue, priority queue, and sequential. For the priority queue iterator, we added a property file with random floating-point edge weights in the range $[0, |V|)$ for the graph $G = (V, E)$. We used these iterators in turn to implement the following textbook single-threaded algorithms:

Page-Rank: We compute the page-rank of a graph: the probability for each vertex that random walk will reach that vertex. This is a well known metric [31] that we compute using an iterative approach where each vertex propagates its current probability to all its neighbours. We implement page-rank using a sequential iterator that is repeatedly run till convergence.

Weakly Connected Components (WCC): Treating every edge as undirected, perform a breadth-first traversal (i.e. BFS) of every component in the graph. Repeat this process for every unvisited vertex in the graph, to discover all components.

Strongly Connected Components (SCC): Computes the strongly connected components of the graph using a label passing approach. Breadth-first label propagation occurs both in the direction and reverse-direction of the edges. If two vertices have different labels, they must belong in different SCCs. The algorithm eventually converges to have one label per SCC.

Single source shortest path (SSSP): Compute the length of the shortest path from a randomly chosen source vertex to every other node in the graph, using the weight data associated with each directed edge. SSSP is implemented using the priority queue iterator.

KCore Decomposition: Identifies nested subset of vertices, called cores, to discover the central components of the graph. The kth core of a graph is the maximal subgraph in which each vertex has degree at least k. This is implemented using the algorithm proposed in [9], in which the lowest degree vertices are removed one by one from the current maximal subgraph. Here, we use the BFS iterator over the re-ordered adjacency list index.

In addition to these five algorithms we also implemented solutions to the problems of maximal independent set, computing conductance over graphs, minimum cost spanning trees and the A* heuristic search algorithm. We found that in each case, their performance was comparable to one of the algorithms above. This is unsurprising as they consist of the same basic iterators and are I/O bound. We therefore report only the basic algorithms listed above.

A Intel DX79A Desktop Board
- 2× Quad Core Xeon
- 64GB RAM
- LSI MegaRAID 2008 SATA3 Controller
- Storage:
  - **A1** Samsung 830 512GB SSD
  - **A2** 4× Seagate 1TB HDD, RAID0

B Dell PowerEdge R420
- 2× Hex Core Xeon E5-2420
- 128GB RAM
- Dell PERC H310 SAS
- Storage:
  - **B1** 4× Samsung 830 512GB SSD, RAID 0
  - **B2** 2× Samsung 840 512GB SSD, RAID 0

Table 3: Test environment hardware platform.

4.2 Graph Datasets

We evaluated the PrefEdge on a set of graphs with differing characteristics. This included synthetic random graphs, scale-free graphs generated using the Graph500 reference code [4], and a real-world graph extracted from the Twitter follower network [5]. We converted the only undirected graph, Watts-Strogatz, into a directed one. The largest connected component in this converted graph traversed by BFS and SSSP is 18,954,432 vertices out of 20,000,000, ensuring we remained close to the undirected structure.

Table 1 shows the runtime memory footprints for all the combinations of graphs and benchmarks that we have run. Unless otherwise specified, we limit the available memory for the OS cache (that caches pages from the SSD) to exactly 2 GB. This includes space for operating system data structures, the kernel and application images. The sum of the numbers in any row, with the exception of the last two entries is therefore the precise amount of RAM used during the execution. The table underlines a key design philosophy of this paper: more bulky $O(|E|)$ data resides on the SSD while smaller $O(|V|)$ data is placed in main memory.

4.3 Methodology

Our testing environment consisted of two machines, each with a set of storage devices – these configurations are detailed in Table 3. We used machine A to generate and process the medium scale graph data in Table 1 (TW, ER, WS, SF). Machine B was used for processing the larger graphs (SF27-30). The large amount of memory on machine A allowed us to evaluate the performance of running each algorithm after pre-loading an entire graph in memory. As described earlier, the current implementation of PrefEdge relies on the vertex map and metadata being pinned in memory, and utilises leftover memory for as a cache for adjacency list and auxiliary data. To measure performance, we compare the runtime of PrefEdge against

Graph	RAM					Cache + SSD	
	WCC	SSSP	Pagerank	SCC	K-Cores	Adjacency List	Edge Weights
Twitter (TW)	1.18	2.74	1.57	2.10	1.68	10.40	24.65
Erdõs-Réyni (ER)	0.45	1.04	0.60	0.80	0.64	12.41	30.54
Watts-Strogatz (WS)	0.45	1.04	0.60	0.80	0.64	12.41	30.54
Scale-free (SF)	1.10	1.05	1.00	1.34	1.07	13.07	32.18
Scale-free (SF27)	3.00	7.00	4.00	–	–	46.56	101.61
Scale-free (SF28)	6.00	14.00	8.00	–	–	73.84	162.96
Scale-free (SF29)	12.00	28.00	8.00	–	–	152.98	326.37
Scale-free (SF30)	24.00	56.00	16.00	–	–	316.80	654.04

Table 1: Memory footprint in gigabytes.

Graph	Vertices (M)	Edges (B)	Cache (20%)	Total Runtime (minutes)			
				WCC	BFC(GC)	SSSP	Pagerank
Scale-free (SF27)	134	6.71	9.3GB	42	32	88	29
Scale-free (SF28)	268	10.73	14.8GB	79	60	164	51
Scale-free (SF29)	536	21.47	30.4GB	164	126	329	112
Scale-free (SF30)	1073	42.95	63.3GB	319	306	607	230

Table 2: Large Graph Runtime Environment

two benchmarks: the ‘baseline’ refers to runtime performance of a cache-restricted unassisted version of each algorithm. The ‘optimal’ case involves pre-loading the entire graph in memory prior to running the unassisted algorithm. In order to change the available cache memory in each trial dynamically, we made use of Linux control groups (`cgroups`), which allow resource limits to be imposed on groups of processes. We cleared the filesystem cache before each trial.

Both of the machines used in our test platform were configured with a standard Ubuntu 12.04 64-bit system image. Each storage device was formatted with the ext4 filesystem. The default configuration of block devices on each machine was changed, to disable readahead on those devices, due to the reasons described in Section 2.4. We exclude the time taken to create in-memory structures from our runtime figures, since such setup time is either constant or not comparable among the versions.

4.4 Performance Comparison with Baseline

We first consider the performance benefit that PrefEdge brings to single-threaded graph algorithms. We ran each algorithm on the four medium-scale graphs with the filesystem cache size restricted to 2000 megabytes for the baseline and PrefEdge trials. This corresponds to approximately 15–20% of the adjacency list size for each graph. Figures 6a, 6b, 6c, 6d and 6e show the total runtime for each algorithm (WCC, SSSP, PageRank, SCC, and K-Cores respectively), on each graph, in each test case (baseline, PrefEdge and optimal). The figures show that each algorithm runs 3 to 14 times faster than the baseline. For those algorithms with a lower speedup compared to the baseline (e.g. PageRank), this is usually due to the fact that the algorithm’s runtime is close to optimal with prefetching. In most cases, using PrefEdge delivers performance that is comparable with the optimal case, despite using approximately 80–85% less memory.

The prefetcher improves the running rate of an algorithm in two ways: (1) data to be traversed next is continuously moved into memory by the prefetcher before it is needed, reducing the percentage of time spent by the main thread waiting for I/O; and (2) the prefetcher issues requests for multiple pages from the SSD at once, boosting throughput from the SSD. To illustrate this difference, Table 4 compares the average throughput from the SSD and percentage of time spent waiting for external storage by CPU, by PrefEdge against the baseline. The rate of improvement delivered by PrefEdge on each type of graph structure is related to the difference in these figures.

4.5 Impact of Cache Restrictions

One of the strengths of PrefEdge is that it performs well even with a small amount of cache memory. The prefetcher tries to ensure that the edge data that the main thread needs to progress is always available in memory – as long as enough edge data is available to keep the main thread busy, I/O stalling will be eliminated.

Figure 7 illustrates the runtime performance of WCC on the Twitter dataset, with respect to maximum cache size. With a cache size of 100 megabytes, the algorithm completes in under half the time of the unassisted BFS run presented above (where the cache size was 2000 megabytes). Conversely, increasing the cache size to 4000 megabytes results in a runtime that is approximately $1.5\times$ the optimal case (which requires all data to be preloaded in memory prior to running). In Section 4.7, we show that PrefEdge outperforms multi-threaded algorithms as the cache size is reduced.

Since the prefetcher is able to function with only a small proportion of all edge data in cache is, it is possible to process datasets with edge data that far exceeds memory capacity on a machine, as shown in the following section.

4.6 Processing Large Graphs

The larger scale-free datasets in Table 2 were processed using machine B, which contained enough storage capacity to hold each of the graphs. Figure 8 illustrates the runtime performance of PrefEdge-assisted WCC, PageRank and SSSP on these graphs. The non-PrefEdge baseline results are omitted in this scenario; for the largest graph, unassisted runtime would run to several days. We restricted available cache memory in two different ways for this

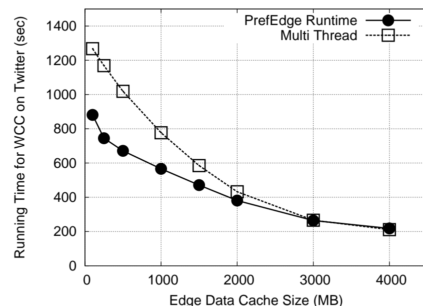
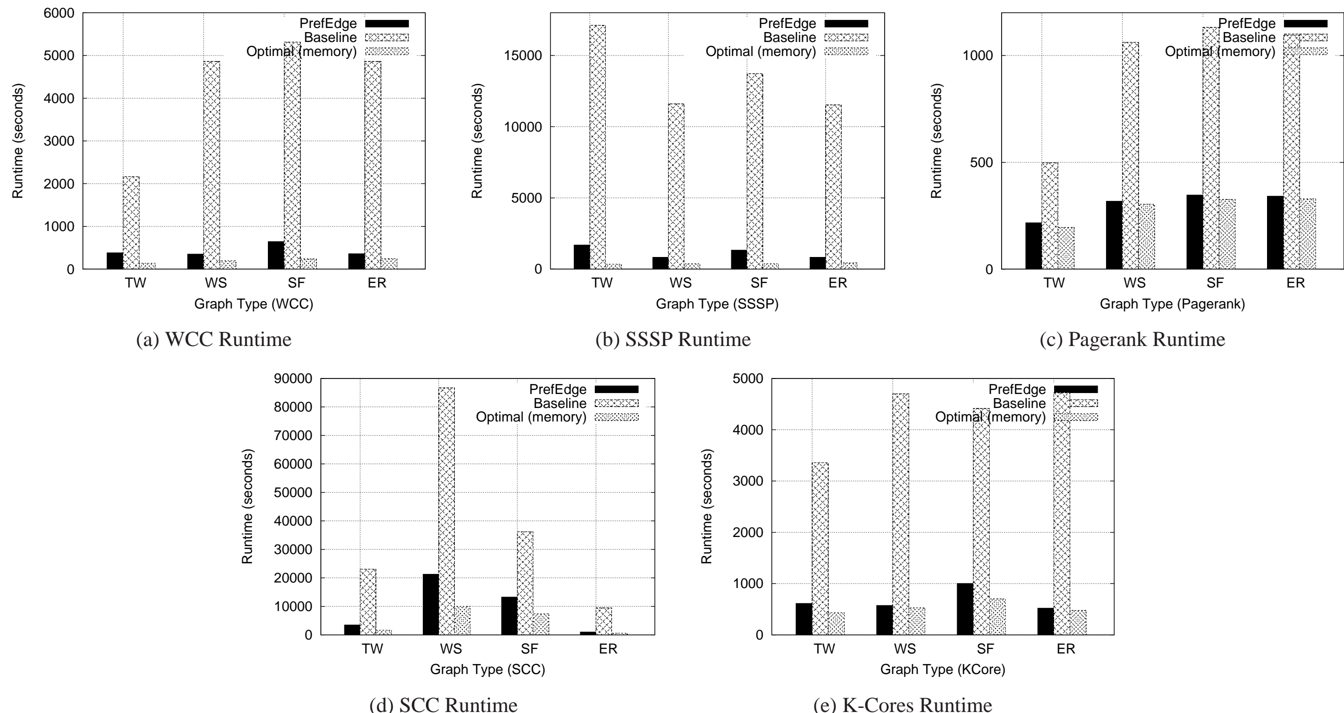


Figure 7: Cache Size Sensitivity (WCC on Twitter)



(a) WCC Runtime

(b) SSSP Runtime

(c) Pagerank Runtime

(d) SCC Runtime

(e) K-Cores Runtime

Figure 6: Runtimes: Comparison to Single Thread Processing and Memory-based operation

Graph	Throughput (MB/s) Baseline	Throughput (MB/s) PrefEdge	CPU I/O Wait % Baseline	CPU I/O Wait % PrefEdge
TW	18.6	96.1	74	31
WS	16.6	224.7	80	18
SF25	17.4	140.1	78	28
ER	17.0	219.2	78	12

Table 4: SSD Throughput and I/O Stalling (WCC)

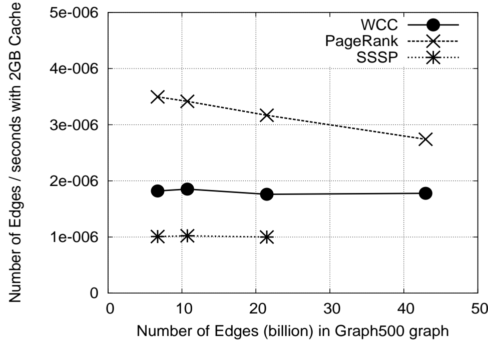
experiment: (1) as in the prior experiment, the cache size was fixed at 2 gigabytes; or (2) the cache size was fixed at a proportion of 20% of the adjacency list size for each graph. In Figures 8a and 8b, we compare the processing rate (in terms of edges visited per second), in each cache scenario. For the random access algorithms (SSSP and WCC), fixing the cache at 2GB appears to reduce the traversal rate by a constant factor. The degradation in PageRank's performance at higher scale may be due to increased OS memory management activity (since PageRank is CPU-bound). The data point for the SSSP on the SF30 graph is missing in the 2GB figure; one vertex in that graph has edge and weight data that exceeds 2GB, causing errant behaviour. The SF30 graph was able to run with a 4GB cache at a similar rate. We note that GraphChi's minimum memory requirement is also equal to the size of the largest edge data for any vertex in the graph.

The unassisted running time of each algorithm on the medium size scale-free graph (SF) lies between the PrefEdge-assisted running time on graphs SF28 and SF29. In other words, PrefEdge allows a graph to be processed in the same time that an unassisted algorithm would take to process a graph that is possibly an order of magnitude smaller.

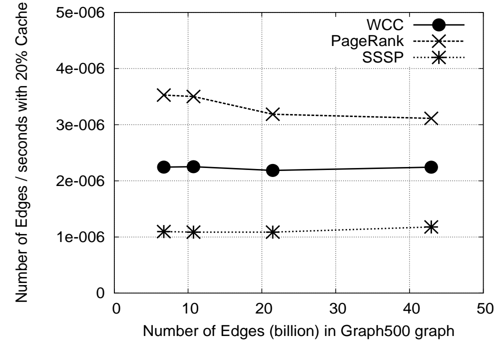
4.7 Comparison to Multi-Threading

Although our focus is on providing performance enhancements for single-threaded programs, we also present a comparison of PrefEdge against a multi-threaded algorithm. For this task, we chose to parallelise the weakly connected components algorithm. From a conceptual level, there are two approaches to parallelising WCC. Recall (from Figure 4) that the sequential WCC algorithm contains two nested loops; the inner loop iterates through the BFS queue, while the outer loop iterates through unvisited components. The first parallel WCC approach entails parallelising the BFS traversal loop, while leaving the outer loop sequential. The other approach involves traversing all components in the graph in parallel, using a label propagation technique. The latter approach is used by GraphChi – we present a comparison in the next section. To evaluate the former approach, we modified the existing WCC implementation to perform the BFS traversal with multiple threads reading from the queue. We used OpenMP [13] for this purpose. Our modified implementation closely resembles the shared queue random access algorithm described by Hong et al. [20]. The runtime performance results from running the multithreaded implementation on each medium scale graph are presented in Figure 9a. The trials were conducted using 8, 32 and 128 processing threads. Note that PrefEdge's runtime is lower than the multithreaded WCC algorithm on each of the graphs.

Figure 9b however can be construed to be unfair on the multithreaded algorithms due the lack of parallelism available in small sub-components of the graph, all of which must be explored by WCC. We therefore also measured the runtime for traversing the

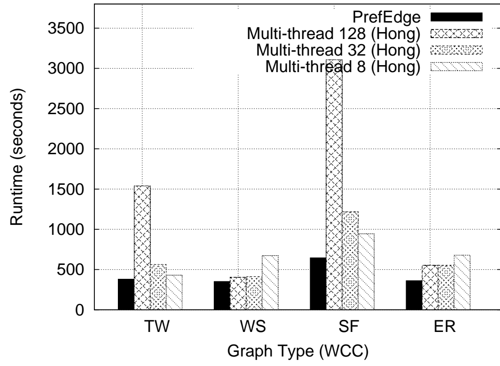


(a) Edges/sec with 2GB as Memory Cache for Edge Data

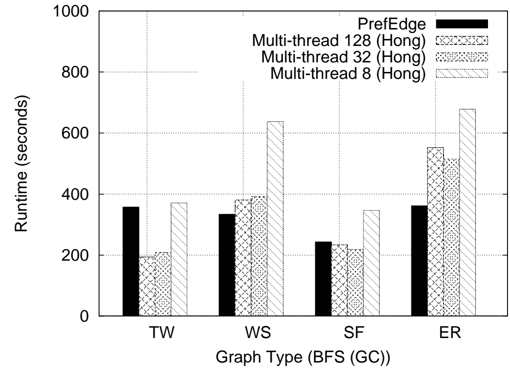


(b) Edges/sec with 20% of Total Edge Data Size as Memory Cache

Figure 8: Large Graph - Scalability



(a) WCC runtime



(b) BFS on Giant Component runtime

Figure 9: Comparison to Multithread Processing

largest component found by WCC (since each component is visited separately, this can be timed separately). This is essentially the time taken to perform a parallel breadth-first search of the giant component in each graph. On the Twitter and scale-free datasets, when operating with a cache capacity of 2GB, the multi-threaded algorithm is then able to outperform the PrefEdge-enhanced algorithm but only at high thread counts. This indicates that a large number of threads are better at handling the large BFS queue during traversal of the giant component in those graphs, but are encumbered by the lack of parallelism when processing smaller components. Further to this, PrefEdge consistently outperforms the multi-threaded algorithm on all types of graph, when the amount of cache capacity is reduced. Figure 7 compares the multi-threaded WCC algorithm with 8 threads on the Twitter dataset with PrefEdge, with successively smaller cache capacities. The multi-threaded approach may result in destructive competition between the threads for limited cache, whereas the prefetcher is able to ensure that pages remain in cache until the data in the pages are no longer required.

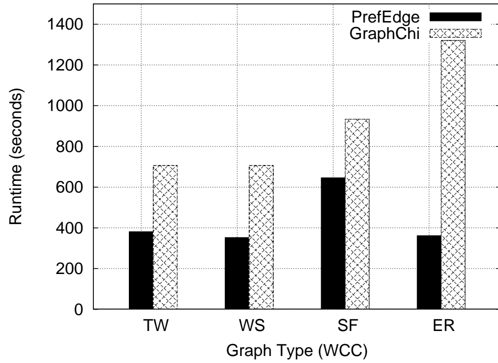
4.8 Comparison to GraphChi

GraphChi is an existing system with similar target applications to PrefEdge. However, there are several key differences between PrefEdge and GraphChi that we summarise here, prior to presenting an indicative performance comparison.

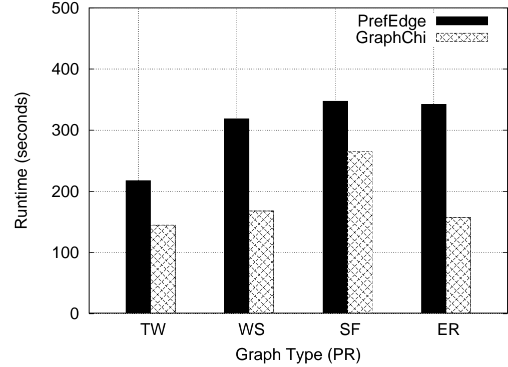
The major architectural difference is that GraphChi does not require vertex metadata to be pinned in memory. As a result, GraphChi programs do not have direct access to remote vertex metadata – instead, updates must be communicated to neighbouring vertices by employing a BSP-style programming pattern. Secondly, GraphChi rearranges edge data such that it is always accessed in sequence in each iteration superstep. As a result, GraphChi saturates the SSD, and I/O stalling is almost nonexistent. However, to support this mode of operation, the graph must be "sharded" into chunks of edge data that can fit into available memory on the machine. In contrast to PrefEdge, in order to take advantage of additional memory, GraphChi would need to reshard an entire graph to support the new memory capacity. The same is also true if a new graph processing program requires a different amount of metadata per edge.

Our aim in this section is to provide some insight into effects of the different approach taken by the two systems. We tested version 0.2.1 of GraphChi on machine A, using the default configuration parameters. Each of the medium-scale graphs was converted into the GraphChi compressed format. GraphChi includes implementations of WCC and PageRank as example applications, which we tested on each dataset. The results are shown in Figure 10.

As described above, BSP-based WCC implementation employed by GraphChi works by labelling all components of the graph concurrently. The number of iterations required to traverse each graph depends on its diameter, but the program only loads vertex and edge data for vertices that have been scheduled for visitation in each iteration. GraphChi's WCC ran approximately 1.5 to 4 times slower than the PrefEdge-assisted single threaded algorithm on each graph. Note that we did not restrict the cache for GraphChi, however this is largely irrelevant because GraphChi programs do not experience I/O stalling. In contrast, PageRank completed significantly faster with GraphChi. PrefEdge almost eliminates I/O stalling for PageRank, which means that the only mechanism for



(a) WCC Runtime



(b) PageRank Runtime

Figure 10: Comparison to GraphChi

further performance improvement is by employing algorithmic parallelism. Since GraphChi employs multiple execution threads, it is able to take advantage of more cores on the machine to process the graph faster.

5. COSTS AND LIMITATIONS

In order for PrefEdge to successfully hide I/O latency it is important that the internal state of the iterator can be maintained entirely in main memory. We note that the iterator state in all the examples given above are of size $O(|V|)$ and our underlying assumption in this paper is that we are willing to pay the cost of storing such data structures in main memory. However, a possible extension would be to reduce the space necessary to store the iterator. The $O(|V|)$ data structure is the row index part of the CSR encoding, we show that by storing a summary of this index along with the full index on the SSD, we can reduce the memory needed with little cost on performance.

The row index contains the positions of vertices within the adjacency list, storing it on the SSD implies two sequential fetches are needed to retrieve a vertex's neighbours: one for its position the adjacency list and an other for its edges. We propose to store in memory a summary of the row index that can be used to get the approximate position of a vertex on the adjacency list. This way, when a vertex needs to be prefetched, we can in parallel fetch its position on the adjacency list from the row index, and a block of memory from the adjacency list that contains its edges as shown in Figure 11. Since sequential reads on SSDs are cheap, fetching a block instead of a single address should come at a low cost. Finding this approximation also requires extra computations, but they come almost for free as during graph computations the CPU is often unused and waiting for I/O. In very general term, *we move the bottleneck away from memory and latency, and trade them off for computations and bandwidth*. A simple estimation done on the degree distribution of the Twitter graph showed that for a graph of 4 billion nodes, it would be reasonable to obtain a summary of 100MB.

A fundamental limitation is the dependency on being able to predict future work. Those algorithms where the iterator is not separable from the graph preclude the construction of a read-ahead function for the iterator. An example of such an algorithm is Depth First Search (DFS). DFS uses a stack of vertices, with the next vertex to be explored being some neighbour of the vertex at the top of the stack (unless it is possible to store the next `IFT` neighbours along with a vertex in the stack, which can be used to compute the read-ahead). In practice, this limitation is somewhat consistent with the ability to parallelise an algorithm (i.e. take a block of future work and distribute it amongst multiple workers).

Another important limitation is the need for concurrent access to iterator state. There are a number of well known techniques for concurrent access to data structures that can be used for safely interleaving access for both threads. In our case the iterators have statically allocated $O(|V|)$ sized structures (such as the queue in the BFS example of Figure 4). We appropriately initialise these structures and allow the read-ahead to traverse it without synchronisation. This leads to the read-ahead working on an inconsistent snapshot. However PrefEdge tolerates approximations to the read-ahead function and therefore our design trades synchronisation overheads for wasted prefetch bandwidth.

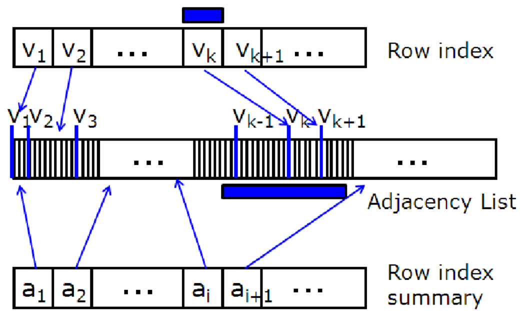


Figure 11: Illustration of the row index summary: The row index and the adjacency list are stored on the SSD and the summary in memory. When vertex v_k is needed, the two areas shaded along the row index and adjacency list are prefetched. The area for the adjacency list is deduced by linearly interpolating between the two pages pointed to by anchors a_i and a_{i+1}.

6. RELATED WORK

Single-machine graph processing. A key feature of current SSD hardware is that the device must service multiple requests in parallel to achieve maximum throughput. The approach taken by Pearce et al. [33] is to employ an asynchronous processing model that ensures the SSD is kept busy, by issuing requests from multiple processing threads. They propose a suite of parallelised graph algorithms for this purpose. However, the design and implementation of efficient multithreaded graph algorithms is is a difficult research problem [35, 39, 30, 18]. In contrast, PrefEdge does not require multithreading but delivers comparable performance. Furthermore,

multithreading could be combined with prefetching to improve performance on certain graph structures and algorithms. In the context of broader research on pointer-based prefetching [32, 26], our work differs by taking advantage of future state stored within iterators, rather than dereferencing pointers in the adjacency list structure directly.

We have presented a comparison of our system with GraphChi [22]. Their on-disk representation results in sequential access when executing vertex-centric algorithms. However this results in their having to move large chunks of data into memory even when all of it is not required. A good example of this is breadth-first traversal (BFS), which is the basis of many popular graph algorithms [7]. In contrast, PrefEdge's random access prefetching ensures only necessary data is retrieved during traversal.

Somewhat similar to the BSP model, the Parallel Boost Graph Library [16] is a contains a number of key generic graph algorithms with an implementation based on MPI. In contrast, the Green-Marl platform provides a domain-specific language for creating graph processing programs, and optimises breadth-first traversal in such programs [19]. Our intention is to integrate with suites such as these, augmenting their performance on external memory. A recent system with similar aims to PrefEdge is TurboGraph [17], which focuses on problems that can be represented as matrix-vector multiplication, using both thread-level parallelism and I/O optimisation mechanisms. The main difference is that PrefEdge is designed to work with existing graph programs with minimal modification. X-Stream [36] is another recent system which takes a very different approach, by attempting to avoid random-access I/O completely, relying on sequential access to optimise throughput. One disadvantage of that approach is the number of wasted edges processed in each iteration, similar to the pattern seen in our comparison with GraphChi. However such systems provide compelling performance on non-traversal algorithms.

Distributed Graph Processing. A large amount of recent research has focused on analysing large scale data with distributed computing resources. However general purpose frameworks such as MapReduce [14] have been found to be inadequate for graph processing for a number of reasons: complex data dependencies within the data result in poor locality of access; a variety of processing patterns exist that do not translate easily to the map/fold model. For this reason, a number of systems specifically for graph processing have arisen. Most of them provide a "think as a vertex" abstraction similar to the one proposed in the Bulk Synchronous Parallel (BSP) [38] model. Pregel [28] is a distributed system based on the BSP model. GraphLab [25] implements a similar model but using a distributed shared memory view instead of message passing, and allows for asynchronous operations. PowerGraph [15] builds on GraphLab and uses a very similar computational model but with an implementation tuned for natural graphs with degrees following a power law distribution.

A number of systems have proposed distributed matrix computation as a platform for graph algorithms. Pegasus [21] implements a generalised iterated sparse matrix multiplication. MadLinq [34] implements a distributed computation library for .NET products. Kineograph [11] uses a time evolving approach, producing consistent snapshots of the graph data at regular intervals. Naiad [29] presents a general incremental computation model, differential dataflow, which can be used for graph computation. Finally, graph databases have seen a substantial amount of work recently, both with distributed implementations such as Trinity [37] and single machine implementation such as Neo4j [6]. The basic techniques described here are applicable to transactional and analytical workloads.

Applications. There has been a recent surge of interest in the area of extracting information from graphs due to the rapidly growing amount of graph structured data available. Applications range across the domain of biology [40], social and information networks [23], and many others.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel prefetcher, PrefEdge, designed to reduce I/O latency in graph traversals for graphs stored on SSDs. The prefetcher almost eliminates the difference between operating from external memory and operating in-memory, for a range of algorithms and graph structures. Prefetcher-assisted algorithms can achieve these benefits even with proportionally small memory caches. Furthermore, PrefEdge offers comparable (and in some cases, superior) performance to parallel algorithms. We have shown the system offers a practical alternative approach to GraphChi that is more easily able to scale with changing memory conditions.

Our main aims for further development of the platform include: addressing the performance of the prefetcher on low-degree vertices; generalising the prefetcher to allow integration with a larger class of algorithms, in addition to providing fine-grained cache management; and relaxing the requirement to lock vertex metadata in memory, by utilising the index summarisation methodology described earlier.

It is interesting to contrast other results in the single machine category with ours. Bader et al. [8] report on breadth-first search on a scale free graph with 1 billion edges in 2.5 seconds on a Cray MTA-2. In comparison our system performs breadth-first search of 1 billion edges in 0.21 hours or about 300 times slower. On the other hand our system costs under $2500 while a Cray MTA-2 would be in the region of millions of dollars: about 1000 times more expensive.

8. REFERENCES

[1] http://hadoop.apache.org/.

[2] http://www.twitter.com/.

[3] http://freecode.com/projects/fio.

[4] http://www.graph500.org/.

[5] http://twitter.mpi-sws.org/.

[6] http://neo4j.org/.

[7] Agarwal, V., Petrini, F., Pasetto, D., and Bader, D. A. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[8] Bader, D. A., and Madduri, K. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *Proceedings of the 2006 International Conference on Parallel Processing* (2006), IEEE Computer Society, pp. 523–530.

[9] Batagelj, V., and Zaversnik, M. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049* (2003).

[10] Cao, P., Felten, E. W., Karlin, A. R., and Li, K. A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (New York, NY, USA, 1995), SIGMETRICS '95/PERFORMANCE '95, ACM,

pp. 188–197.

[11] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 85–98.

[12] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*. MIT Press, 2001.

[13] DAGUM, L., AND MENON, R. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE 5*, 1 (1998), 46–55.

[14] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6* (2004), USENIX Association, pp. 10–10.

[15] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 17–30.

[16] GREGOR, D., AND LUMSDAINE, A. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)* (2005).

[17] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (New York, NY, USA, 2013), KDD '13, ACM, pp. 77–85.

[18] HAN, Y., AND WAGNER, R. A. An efficient and fast parallel-connected component algorithm. *J. ACM 37*, 3 (July 1990), 626–642.

[19] HONG, S., CHAFI, H., SEDLAR, E., AND 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* (2012), ACM, pp. 349–362.

[20] HONG, S., OGUNTEBI, T., AND OLUKOTUN, K. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), IEEE, pp. 78–88.

[21] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on* (2009), IEEE, pp. 229–238.

[22] KYROLA, A., AND BLELLOCH, G. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th conference on Symposium on Opearting Systems Design & Implementation* (2012), USENIX Association.

[23] LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD) 1*, 1 (2007), 2.

[24] LITTLE, J. D. A proof for the queuing formula: $L = \lambda W$. *Operations Research 9*, 3 (May 1961), 383–387.

[25] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990* (2010).

[26] LUK, C.-K., AND MOWRY, T. C. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1996), ASPLOS VII, ACM, pp. 222–233.

[27] LUMSDAINE, A., GREGOR, D., HENDRICKSON, B., AND BERRY, J. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 1 (2007 2007), 5–20.

[28] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[29] MCSHERRY, F., ISAACS, R., ISARD, M., AND MURRAY, D. G. Composable incremental and iterative data-parallel computation with naiad. Tech. Rep. MSR-TR-2012-105, 2012.

[30] MEYER, U. *Design and analysis of sequential and parallel single-source shortest-paths algorithms*. PhD thesis, 2002.

[31] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[32] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. Tech. rep., New York, NY, USA, 1995.

[33] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[34] QIAN, Z., CHEN, X., KANG, N., CHEN, M., YU, Y., MOSCIBRODA, T., AND ZHANG, Z. Madlinq: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 197–210.

[35] REGHBATI, E., AND CORNEIL, D. G. Parallel computations in graph theory. *SIAM J. Comput. 7*, 2 (1978), 230–237.

[36] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[37] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. Tech. rep., New York, NY, USA, 2013.

[38] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.

[39] YOO, A., CHOW, E., HENDERSON, K., MCLENDON, W., HENDRICKSON, B., AND CATALYUREK, U. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *ACM/IEEE conference on Supercomputing* (2005).

[40] ZERBINO, D. R., AND BIRNEY, E. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research 18*, 5 (2008), 821–829.