Technical Report

UCAM-CL-TR-823
ISSN 1476-2986

Number 823

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Mitigating I/O latency in SSD-based graph traversal

Amitabha Roy, Karthik Nilakant, Valentin Dalibard, Eiko Yoneki

November 2012

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

Mitigating I/O latency in SSD-based Graph Traversal

Amitabha Roy, Karthik Nilakant, Valentin Dalibard, and Eiko Yoneki
University of Cambridge Computer Laboratory
Cambridge, United Kingdom

Abstract

Mining large graphs has now become an important aspect of many applications. Recent interest in low cost graph traversal on single machines has lead to the construction of systems that use solid state drives (SSDs) to store the graph. An SSD can be accessed with far lower latency than magnetic media, while remaining cheaper than main memory. Unfortunately SSDs are slower than main memory and algorithms running on such systems are hampered by large IO latencies when accessing the SSD. In this paper we present two novel techniques to reduce the impact of SSD IO latency on semi-external memory graph traversal. We introduce a variant of the Compressed Sparse Row (CSR) format that we call Compressed Enumerated Encoded Sparse Offset Row (CEESOR). CEESOR is particularly efficient for graphs with hierarchical structure and can reduce the space required to represent connectivity information by amounts varying from 5% to as much as 76%. CEESOR allows a larger number of edges to be moved for each unit of IO transfer from the SSD to main memory and more effective use of operating system caches. Our second contribution is a runtime prefetching technique that exploits the ability of solid state drives to service multiple random access requests in parallel. We present a novel Run Along SSD Prefetcher (RASP). RASP is capable of hiding the effect of IO latency in single threaded graph traversal in breadth-first and shorted path order to the extent that it improves iteration time for large graphs by amounts varying from 2.6X-6X.

1 Introduction

Mining graph structured data is becoming increasingly important for numerous applications; ranging across the domains of social networks, bioinformatics, security and many more. A basic problem with mining graph structured data is that the lack of locality in such data coupled with their large size renders building systems that iterate over large graphs with reasonable performance a challenging task [1]. The lack of locality also means that traditional abstractions such as map-reduce [2] do not work well with graph structured data.

This lack of locality has lead to the assumption that processing large graphs necessarily requires them to be loaded entirely in main memory. Recently however, researchers are starting to explore the possibility of mining graphs on single computers with limited amounts of main memory. This is a very attractive proposition for mainstream graph mining without large budgets, a mode of thinking to which we subscribe. Existing systems [3, 4] that have been built to make graph mining practical on single machines advocate using solid-state drives to store graph data. Solid state drives (SSD) are far cheaper (an order of magnitude) than main memory and provide far lower latency than traditional magnetic media. Nevertheless IO latency remains a determinant of performance on such systems with the poor locality during graph traversal rendering main-memory caches of data stored on the SSD ineffective.

The approach taken by Pearce et. al. [3] uses the fact that a typical SSD can service multiple random access requests in parallel. This is exploited by multithreaded algorithms to issue a request from each thread with the aim of hiding IO latency and improving overall throughput. Unfortunately the design and implementation of multithreaded graph algorithms is difficult, necessitating complex solutions [5, 6, 7, 8]. On the other hand the approach of Kyrola et. al. [4] uses special on-disk representations and limits usage to sequential iteration over vertices to make lower latency sequential IO more dominant. This precludes running simple breadth-first traversal on their system. Breath-first traversal is the basis of many popular graph algorithms such as shortest paths, connected components and heuristic search and it is used in many important applications in analysing graphs [9]. This paper is motivated by the question of whether low cost machines can be combined with *simple single-threaded graph traversal*. We therefore explore the potential for mitigating IO latency in *single-threaded* graph traversal. Our techniques are easily extensible to multithreaded and asynchronous graph algorithms, which one may view as a composition of single threads issuing IO requests.

Our solution trades main memory capacity for IO latency. Given a graph consisting of a set of vertices V and edges E; we place $O(V)$ sized data structures in main memory while leaving $O(E)$ sized data on disk. Although this needs more memory than a purely external memory algorithm, the memory requirements are still capped at reasonable levels, while reducing IO latency considerably. For example, we run a single source shortest path traversal over a dataset from Twitter [10] 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. Further, even in the case where a single machine cannot accommodate this data in main memory, splitting the graph over a set of machines can easily bring the footprint of data structures within reach for even the largest of graphs, *while keeping the overall amount of main memory needed within a reasonable budget*. We also note that this assumption is explicitly present in Pearce et. al.'s work [3] where they term such a distribution of graph data as "semi-external memory". The same assumption is also implicit in Kyrola et. al.'s GraphChi [4] that requires enough main memory to hold all the neighbours of any vertex, this can easily be seen to translate to $O(V)$ main memory requirements for high degree vertices.

Our baseline system is shown in Figure 1. We are concerned with traversing large graphs in various vertex orders: sequential, breadth-first and shortest-path on a single machine. The most relevant components of our system are a CPU, RAM and persistent storage in the form of an SSD. During graph traversal, 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. In this context, our paper makes two novel contributions:

1. We propose a variant of the widely used Compressed Sparse Row (CSR) format that we call Compressed Enumerated Encoded Sparse Offset Row (CEESOR). CEESOR consists of an $O(V)$ sized row index that we place in memory and a compressed $O(E)$ offset

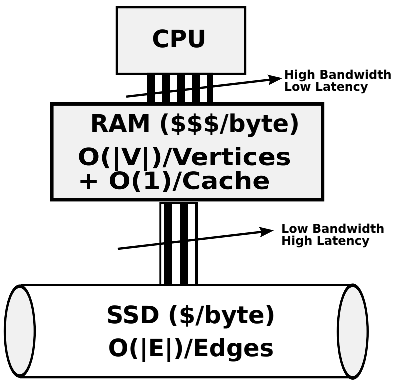

Figure 1: System Overview

vector that we place on the SSD. CEESOR is targeted at graphs with hierarchical structure and can reduce the space required to represent the edge-list compared to CSR by amounts varying from 5% to as much as 76%. Reducing the amount of space needed to represent each edge on the SSD means that each IO operation can bring in a larger number of edges and each cached page contains a larger number of graph edges, directly reducing the amount of IO needed.

2. Prefetch data from the edge list on the SSD before it is needed. This improves the hit rate seen by algorithms in the small cache and thereby reduces the amount of IO latency during graph traversal. This is *independent* of the structure of the graph making it applicable even when graphs lack hierarchical structure [11]. We present a novel Run Along SSD Prefetcher (RASP) that prefetches data for vertices before they are needed during traversal. RASP places key data structures (of size proportional to the number of vertices) from the underlying graph iterator in main memory. For example, we perform breadth-first search (BFS) on a power-law graph with 6 billion edges, using just 5 GB of RAM *including* a 2 GB OS cache. RASP improved the hit rate in the OS cache when accessing 41 GB of edge data placed on the SSD from 25% to 91%. This in turn reduced the run time of BFS from 6 hours to under 1.5 hours, an improvement of over 4X. In general, we are able to speed up a set of basic graph algorithms by amounts varying from 2.6X to 6X.

We now begin by describing CEESOR in Sections 2 to 4. We then characterise the SSD as an IO device in Section 5 before describing RASP in Section 6. We then individually evaluate CEESOR and RASP in Section 7 before discussing related work in Section 8 and concluding.

2 CEESOR

CEESOR is based on the Compressed Sparse Row (CSR) format. Originally designed for storing sparse matrices, CSR is also often used to store the adjacency matrix of sparse graphs. We consider a graph $G = (V, E)$ with an enumeration of the vertices $I : V \quad 1..V$ (I is bijective) that produces for each vertex v an identifying number $I(v)$. The adjacency matrix of the graph $G = (V, E)$ is a $V \quad V$ matrix $A_{\ G}$ with $A_G(i, j) = 1$ iff $(I^{-1}(i), I^{-1}(j)) \quad E$

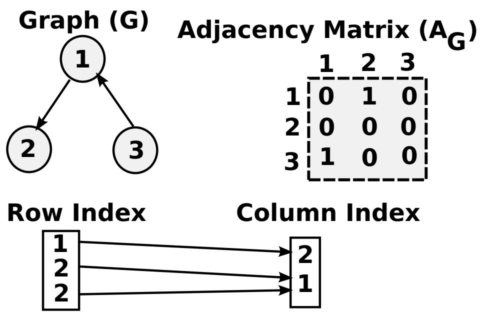

Figure 2: Example of CSR format

with $A_G(i, j) = 0$ otherwise. The adjacency matrix is also sparse and often encoded using the Compressed Sparse Row (CSR) format.

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 2 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 aim to replicate this functionality in CEESOR.

Our starting point for the design of CEESOR is the assumption that the $O(V)$ sized R_G fits in main memory. We therefore focus our efforts with CEESOR on reducing the size of the E sized C_G that is stored on the SSD.

2.1 Hierarchy and Clustering

A significant feature of many real world graphs is hierarchy and/or clustering as noted by Clauset et. al. [12]. A key feature of such graphs is that their set of vertices can be partitioned into subsets such that a majority of the edges are between vertices in the same subset. Clauset et. al. specifically point to ecological niches in foodwebs, modules in biochemical networks and communities in social networks. Figure 3 shows a synthetic example of a graph with both hierarchy and clustering. In the case of multilevel hierarchies or subclusters within a larger cluster each of these subsets may be further partitioned. CEESOR is a variant of CSR specifically designed to take advantage of such clustering and hierarchy.

CEESOR allows exploiting hierarchy and clustering to reduce the size of the column index that must be accessed from external storage. However, we aimed to also ensure that CEESOR remains competitive to CSR for graphs that do not exhibit structure, such as random graphs. It is therefore not exclusively suitable for graphs with hierarchy but rather provides the means to

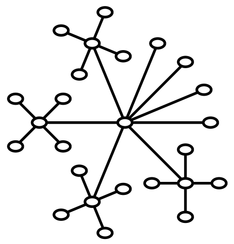

Figure 3: Hierarchy in Graphs (a synthetic example)

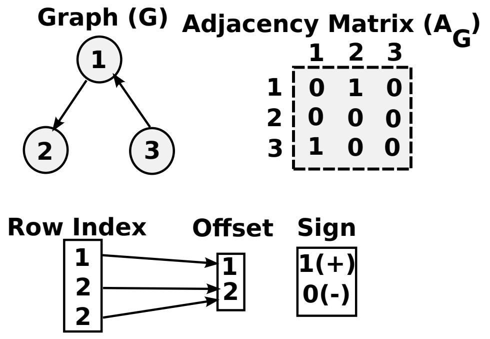

Figure 4: Example of CEESOR format

exploit hierarchy when it is available.

2.2 CEESOR

CEESOR replaces the column index C_G of the CSR representation with a sign vector S_G and an offset vector O_G. For any edge e we denote $C_G[e]$, $S_G[e]$ and $O_G[e]$ to be be the corresponding entries for that edge in each of those vectors. For every edge $e = (u, v)$, we set $O_G[e] = |I(v) \quad I(u)|$; and $S_G[i] = 1$ iff $(I(v) \quad I(u)) > 0$ or 0 otherwise. We therefore represent entries in the column index by the combination of an offset and a sign bit. Figure 4 shows how the graph from Figure 2 is represented in CEESOR.

CEESOR is designed with the goal of exploiting hierarchy and clustering. If vertices were to be assigned an enumeration that ensures that the partitions described in the previous section are *separately* enumerated (a problem we consider in Section 3), we are left with $|v \quad u|$ being a small quantity that we then appropriately encode at a lower cost. This allows us to effectively exploit graph structure with CEESOR.

2.3 Graphs Without Structure

Before proceeding further, it is important to consider the case where the graph does not have hierarchical structure or clusters. How badly might CEESOR perform in comparison to the widely used CSR format in this case ? We analyse this situation by considering the case of a random graph $G = (V, E)$ where every possible edge has an equal probability of being present (the Erdõs-Réyni model [13]). We consider the information-theoretic minimum cost of:

1. Encoding C_G considering vertices as symbols.
2. Encoding O_G considering the absolute offsets as symbols plus 1 bit for the corresponding entry in S_G.

For a random graph, the frequencies of all vertices in terms of their occurrence in C_G is a-priori the same and when normalised equals $\frac{1}{|V|}$. For a large number of such random graphs with the same number of vertices and fixed encoding for entries in C_G, the average cost of representing an entry is bounded below by Shannon's source coding theorem [14, 15] as (note: log to base 2 unless otherwise specified):

$$\sum_{i=1}^{|V|} \frac{1}{\ V\ }\log(\ V\ ) = \log(\ V\ ) \tag{1}$$

This means that we cannot represent entries in C_G using less than $log\ V\ $ bits on average.

In the case of CEESOR we need to consider the average cost of storing entries in O_G and the sign bit for edges. In the case of O_G *the critical observation here is that for an edge $e = (u, v)$ in a random graph, $O[e] = \ I(v)\quad I(u)\ $ is skewed in favour of smaller values.* Given that every edge is equiprobable: the number of edges (u, v) where $\ I(v)\quad I(u)\ = 1$ is $\ V\quad 1$, while the number of edges where $\ I(v)\quad I(u)\ = \ V\quad 1$ is exactly one. Generalising, the number of edges where $\ I(v)\quad I(u)\ = k,\ k\quad [1..(\ V\quad 1)]$ is $\ V\quad k$. *We note here that the number of edges with smaller offsets are more frequent even in random graphs.* Shannon's source coding theorem now gives the average cost of encoding an entry in O_G as (setting the i^{th} symbol to represent $\ V\quad i$ and adding one for the sign bit):

$$1 + \sum_{i=1}^{|V|-1} \frac{i}{\binom{|V|}{2}} \log\left[\frac{\binom{|V|}{2}}{i}\right] \tag{2}$$

$$= \log(\ V\ ) + \log(\ V\quad 1) \qquad \frac{1}{\binom{|V|}{2}} \sum_{i=1}^{|V|-1} i\log(i) \tag{3}$$

Comparing equation 3 to equation 1, the overhead per entry in O_G and S_G taken together is:

$$\log(\ V\quad 1) \qquad \frac{1}{\binom{|V|}{2}} \sum_{i=1}^{|V|-1} i\log(i)$$

Using the approximation of sums using integrals [16](e is the base of the natural logarithm):

$$\frac{1}{\binom{|V|}{2}} \sum_{i=1}^{|V|-1} i\log(i) \qquad \frac{1}{\binom{|V|}{2}} \int_0^{|V|-1} x\log(x)\,dx$$
$$= (1 \qquad \frac{1}{\ V\ })[\log(\ V\quad 1)\quad 0.5\log(e)]$$

V	Overhead
10	0.290
100	0.118
1,000	0.073
10,000	0.054
100,000	0.043
1,000,000	0.036

Table 1: Theoretical Overhead for CEESOR on random graphs

Hence, the overhead is bounded by:

$$\begin{aligned} &\log(V \quad 1) \quad (1 \qquad \frac{1}{V})[\log(V \quad 1) \quad 0.5\log(e)] \\ &= 0.5\log(e) + \frac{\log(V \quad 1) \quad 0.5\log(e)}{V} \end{aligned}$$

Hence, the overhead expressed as a fraction of the baseline CSR cost is bounded by:

$$\frac{0.5\log(e) + [\log(V \quad 1) \quad 0.5\log(e)]/ V}{\log(V)}$$

The bound on the overhead drops with an increasing number of vertices as shown in Table 1. For large random graphs therefore the overhead is bounded acceptably to under 5%.

2.4 Encoding CEESOR Column Indices

We encode entries in the CEESOR offset and sign vectors so as to exploit low-valued offsets. We use a variable-length encoding that is decodable without a dictionary. We assume that there are no self loops and hence no absolute offsets of zero. We describe later how self-loops may be handled in CEESOR.

We represent an absolute offset o using the minimum possible b bits where $(2^b \quad 1) \quad o$. Clearly the most significant bit cannot be zero. Given a list of offsets Algorithm 1 describes the encoding process. We arrange the offsets in increasing order and therefore also in order of *increasing bit count*. We emit these in big-endian order with additional `0` bits separating entries where the number of bits needed for representation increases.

Algorithm 2 describes the decoding process. Note that the algorithm represents one step in an *iterator* that is used to scan the edges connected to a vertex. We scan the bit stream corresponding to entries in O_G. At any instant we maintain the current size of entries, increasing by one whenever we see a leading zero.

Finally, R_G contains a *bit-level* index into O_G, which is now a bit-stream. For convenience we interleave the bits from S_G into O_G. We do this by placing $S_G[i]$ at the end of the bits representing $O_G[i]$. This extra bit is read by the decoder immediately after reading $O_G[i]$, before

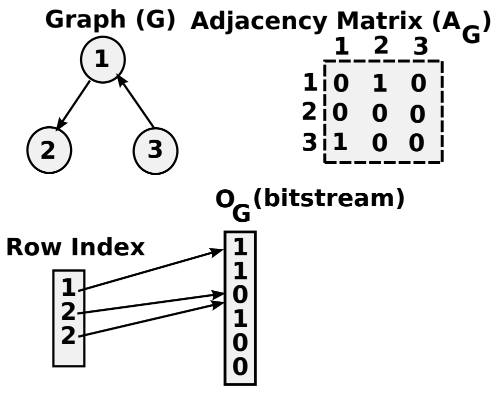

Figure 5: CEESOR examples after encoding

Algorithm 1 Encoding CEESOR entries

```
Require: L is the list of offsets to be emitted
  sort L
  bits := 1
  while L is not empty do
    entry := offset at head of L
    if $(2^{\text{bits}}\ \ 1) <$ offset then
      Emit 0
      bits := bits + 1
    else
      Emit entry in big-endian (leading bit first)
      remove entry from L
```

proceeding further. For illustration, we show the encoded form of the CEESOR representation in Figure 4 in Figure 5.

In the case of CSR (CEESOR), we put the E sized C_G (encoded O_G and S_G) in external memory while retaining the V sized R_G in main memory. Our implementation uses 64 bits for each entry in the row-index R_G. To support self-loops we reserve the topmost bit of the index to indicate a self loop. Hence O_G is limited to 2^{63} bits = 32 petabytes.

This encoding chosen for CEESOR further inflates the overheads over CSR shown in Table 1. However, we show in our evaluation in Section 7 that CEESOR is more optimal that CSR for

Algorithm 2 Decoding CEESOR entries

```
Require: L is the list of bits to be decoded
Require: size is the current decoding size
  while bit at head of L is 0 do
    size := size + 1
    remove bit at head of L
  entry := 'size' bits at head of L interpreted in big-endian
  remove 'size' bits from head of L
  return entry
```

all the graphs we have considered.

3 Vertex Enumeration in CEESOR

It is evident that the efficiency of CEESOR depends on the enumeration of vertices. Enumerations that place connected vertices close together lead to edges with smaller values for offsets. In this section we consider a heuristic for achieving better enumerations for graphs with clustering and hierarchy, the target for efficiency with CEESOR.

The cost we wish to minimise is that of offsets in O_G. For a graph $G = (V, E)$, the cost in bits of the entries in O_G, disregarding the encoding overhead of separator zero bits, and the interleaved sign bit is:

$$\Sigma_{(u,v)\in E}(\; log \; I(v) \quad I(u) \quad + 1)$$

We wish to minimise this cost by choosing an appropriate enumeration $I(v)$. We note that the closely related problem of minimising:

$$\Sigma_{(u,v)\in E} \; I(v) \quad I(u)$$

is known to be NP-hard (the minimum linear arrangement problem). Although we have not proved that minimising CEESOR is also NP-hard, we use a heuristic for minimising the cost rather than attempting to find provably efficient algorithm. Since our objective with CEESOR is graphs with hierarchy, a possible heuristic to do this is to apply a graph clustering algorithm and recursively enumerate each cluster separately. However, the input edge list is a large external memory file and this makes most clustering algorithms very expensive in terms of IO. We instead chose to design our own enumeration, which is based on the combination of a depth-first traversal of the graph and a heuristic to pick nodes for enumeration. The driving intuition is to try to enumerate connected vertices close together so as to minimise the cost of representing that edge in CEESOR.

Given a graph G we traverse it in depth first search (DFS) order *treating the graph as undirected*. This produces a forest of DFS spanning trees over the graph and takes $O(\; V \; + \; E \;)$ time and $O(\; V \;)$ space. During this DFS enumeration, we also store, for each vertex, the number of vertices covered by the DFS subtree rooted at it. We call this the weight of the subtree rooted at that vertex. We then enumerate each tree in the forest separately. Clearly there is nothing to be gained by enumerating them together, since there are no edges between trees whose representation cost could be minimised.

Consider a DFS tree T with root r and weight $w(r)$. An example of such a DFS tree is shown in Figure 6. The solid edges are part of the spanning tree while the dashed edges (each connected to some vertex within the respective spanning sub-trees) are not. It is important to note that the nature of depth first search results in there being *no edges present between nodes in different subtrees*. Therefore r is a bridge node removing which would cause subtrees below it to become disconnected (as can be seen from the example in Figure 6). We choose to enumerate each of

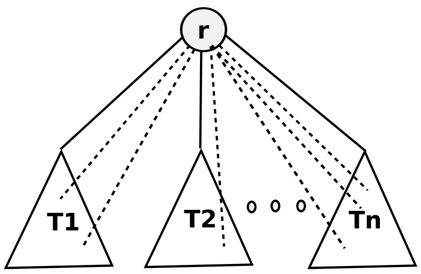

Figure 6: A DFS Spanning Tree

the subtrees completely and separately. This is in fact the best strategy if we ignore the presence of r. Our strategy of partitioning the graph around bridge nodes such as `r` is also driven by the observation that bridge nodes connect *relatively* densely connected clusters in many real-world graphs.

The next aspect is to choose the order in which we want to enumerate subtrees. Our strategy is to always consider the heaviest subtree first. Starting from the heaviest subtree ensures that any large connected components deeper in the DFS hierarchy and contained in a subtree are enumerated first. They do not then contribute to the representation cost of edges from r into other subtrees as these edges do not need to cross over the vertices in the first subtree enumerated.

The final aspect is choosing exactly when to enumerate the root r as there are possibly multiple edges from r into each of the subtrees. We attempt to place r at the mid-point of the enumeration. This minimises the cases where an edge e out of r has a higher value for the offset than half the number of vertices in the entire DFS tree.

In summary, our enumeration heuristic starting from the DFS tree in the example is as follows: we completely (recursively) enumerate subtrees below r (`T1` to `Tn`) one by one until the combined weight of the enumerated subtrees is greater than or equal to $w(r)/2$, We then enumerate r and then continue enumerating the remaining subtrees below it. We always consider the heaviest subtree first (this costs at most $O(E)$ time or $O(V)$ space) but consider all other subtrees in the order obtained from DFS over the existing graph.

4 Converting CSR to CEESOR

Our toolchain converts graphs from CSR to CEESOR. We require the input graph to be either undirected or available in bidirectional form so that we may treat it as undirected for building the DFS tree. This sometimes necessitated an extra step in conversion.

Two steps in the CSR to CEESOR conversion are of concern with regard to overall conversion time. The first is the generation of the DFS trees. This leads to random access over the edge list and is the most time consuming step. We perform this step keeping the column vector of the CSR file on SSD and the row index in memory. The slowest SSD we have worked with has a latency of 75us for a random read. This means that a depth-first search can process about a million edges in 75 seconds and a billion edges in 21 hours. We note that conversion is a one-time cost to pay and the costs would be significantly lower with a faster SSD or by loading the graph in the RAM of a large cluster for conversion. Also the CEESOR format allows the addition or deletion of vertices and edges, with representation efficiency degrading as a function of the number of added edges. The other step of concern in this conversion is the sorting of offsets in order to emit them in increasing order into CEESOR (Algorithm 1). We do this using counting sort [16] with $O(log\ V\ )$ space and $O(\ V\ log\ V\ )$ complexity overall. The overall complexity of the conversion from CSR to CEESOR is therefore $O(\ V\ log\ V\ +\ E\ )$ considering both the $O(\ V\ +\ E\ )$ enumeration and $O(\ V\ log\ V\ )$ sorting cost during encoding. It is accomplished using $O(\ V\ )$ space. The offsets, encoding and enumeration aspects of CEESOR are its essential differences from CSR and hence the name Compressed Enumerated Encoded Sparse Offset Row.

5 SSD IO Characteristics

Solid state drives are persistent storage devices that store data using non-volatile flash memory (usually NAND flash) rather than on magnetic media like traditional hard drives. SSDs contain no moving parts and this makes them fundamentally different in terms of IO characteristics from traditional magnetic drives. From the perspective of storage for graphs, two characteristics of SSDs are of particular interest:

- Like magnetic media, SSDs provide far better throughput and lower latency for servicing sequential requests as opposed to random access ones. This differential performance in fact is one of the key focus areas for closely related work such as GraphChi [4].
- 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 this in the paper as the number of inflight requests (also referred to in literature as the queue depth).

A useful model for understanding the IO characteristics of SSDs is Little's Law [17]. This views the SSD as a black box. If we then assume that N is the number of outstanding requests to the SSD, R is the average response time for servicing a request and λ is the throughput obtained in terms of requests per unit time, Little's Law says that at steady state:

$$N = \lambda R$$

The throughput (λ) from an SSD therefore depends on the number of inflight requests and the average latency of servicing a request. Our objective with RASP in the next section is to maximise the throughput from the SSD. In our current implementation, we have little control over latency as that is determined by the software stack and the SSD controller itself. We use Linux without tuning, depending on its virtual memory subsystem for buffering. However, we have direct control over the number of in-flight requests. Starting from a small number of inflight requests (N), throughput generally increases with an increasing value of N. In this situation latency holds roughly constant as the SSD is able to service requests in parallel and queuing delays in the software stack are minimal. The throughput reaches a maximum after which queuing delays primarily due to the inability of the SSD to respond to requests leads to

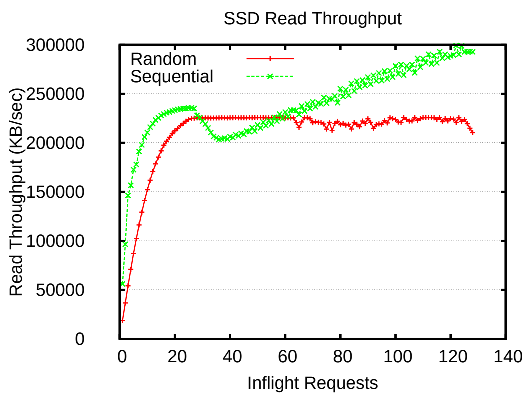

Figure 7: SSD IO throughput

latency growing with the number of inflight requests. This caps the available throughput from the SSD.

We have used a Samsung™512 GB SSD for the experiments in this paper. We used the `Flexible IO` [18] 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 SSD. Figure 7 shows how throughput varies in a manner consistent with Little's Law, as the number of inflight requests for 4KB pages is increased. That figure also shows that sequential accesses can be serviced with a higher throughput than random accesses.

There is therefore a certain number of in-flight requests that leads to maximum throughput from the SSD. This number is different for random IO as opposed to sequential IO (as Figure 7 shows). At this optimum point, the SSD is able to supply a peak of 300000 KB/s for sequential accesses and 225000 KB/s for random accesses. The objective of the (R)un (A)head (S)SD (P)refetcher (RASP) in the next section is to ensure that during the execution of graph algorithms, the number outstanding requests to the CEESOR file held on the SSD is maintained at this optimum number for each type of IO.

6 RASP

We now discuss our Run Ahead SSD Prefetcher (RASP). RASP takes advantage of SSDs to service multiple requests in parallel. It aims to reduce IO stalls during semi-external memory traversal of large graphs by prefetching data into main memory before it is needed. We begin this section by providing an overview of RASP. We then provide an implementation sketch detailing how the prefetcher works. Finally, we discuss how RASP can be integrated with different types of graph algorithms.

6.1 Overview

RASP 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),

Algorithm 3 RASP 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 := Runahead(IFT)
  for all $x$ in Needed considered in order do
    if $x$ /Issued then
      Issue prefetch request for CEESOR page of $x$
      Add $x$ to Issued
```

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 $\pi_{\text{vertex}}(< x, S, G >) = x$.

An iterator supports the next function Next$(< v, S, G >) =< v', S', G >$. RASP depends on the iterator being *separable from the graph*. This means that there exists a runahead function R and an integer $k(S) > 0$ such that $i \quad k(S) : R\ ^{i}(< v, S >) = \pi_{\text{vertex}}(\text{Next}^{i}(< v, S, G >))$. The runahead function is therefore able to determine the next k accessed, without reference to the graph G. A simple example of this is breadth-first search (BFS) iteration over the vertices of a graph. The set of vertices to be accessed next is held in a FIFO queue that can be consulted to determine the next set of vertices to be accessed without reference to the graph.

RASP contains two components:

1. Prefetch: Given a sequence of vertices to be accessed issue a set of requests for the corresponding CEESOR pages to the SSD to achieve the optimum (Section 5) number of inflight requests. The prefetcher calls the runahead component (described next) with a target T.

2. Runahead: The runahead component computes the result of the runahead function to determine *at most* the next T vertices: $R^{\,min(k(S),T)}(< v, S >)$. It returns an *ordered* list.

6.2 Prefetch

The prefetcher is responsible for issuing requests to the SSD based on the Runahead 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 RASP is shown in Algorithm 3.

The RASP prefetcher is based on the simple observation that execution of the graph algorithm is synchronous and therefore stalls on an IO. 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 CEESOR file. It maintains a hash for these issued requests. A new request is issued only if it does not match one of the last `IFT` requests. Since the iterator only makes progress when data is returned from the CEESOR file, this simple scheme ensures that the number of inflight requests is maintained at `IFT`.

The prefetcher calls the runahead component (described in the next subsection) to provide it

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

Figure 8: Connected components for an undirected graph

with enough vertices to issue `IFT` requests to the SSD. There are two key problems in deciding how many vertices of lookahead are needed to satisfy the need for `IFT` requests to the SSD:

1. Multiple vertices can map to the same page in the CEESOR file and hence they coalesce to one IFT request. This problem is easily handled by having the runahead component filter out vertices from the target that can be coalesced with other requests by virtue of mapping to the same page in the CEESOR file. We have implemented this filtering in our system.

2. Depending on the locality of the CEESOR file it is possible for requests in the IFT budget to hit the operating system cache. This results in underutilisation of SSD bandwidth. The only way to get around this problem is to further filter those vertices that are already in the cache, something that we have not implemented in this paper.

6.3 Run-ahead

We now discuss the run-ahead component, which is responsible for computing Lookahead. In this paper, we discuss lookaheads for three types of iterators. We use the example of discovering connected components using breadth-first search shown in Figure 8 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.

6.3.1 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 runahead 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.

6.3.2 Breadth-first Iterator:

The breadth-first iterator is somewhat more complicated from a runahead perspective. We note that the state of the iterator 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 i^{th} element of the FIFO queue encapsulated in S.

6.3.3 Priority Queue Iterator:

Workloads such as single-source shortest path (SSSP) [16] 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 *good 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 RASP is clearly *tolerant of incorrect vertex selection by the runahead component,* 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 inspite of these approximations, RASP provides good speedups for SSSP.

6.4 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 IO 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 IO 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 CEESOR 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 CEESOR 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 CEESOR file and the other half to the property file.

6.5 Costs and Limitations

In order for RASP to successfully hide IO 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.

RASP works for a large number of algorithms but is not a catch-all solution. The most serious limitation of RASP is that it is not applicable to those algorithms where the iterator is not separable from the graph thereby precluding the construction of a runahead 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*. There is therefore no way to determine the next set of vertices to be accessed without reference to the CEESOR file; rendering RASP ineffective for DFS. We note however that DFS can be restructured to work with RASP. It is possible to store the next `IFT` neighbours along with a vertex in the stack, which can be used to compute the runahead in RASP. This would of course inflate the main memory cost of the algorithm by a constant factor.

Another important limitation for RASP is the need for concurrent access to iterator state. We use a separate thread for prefetching in order that the main thread running the graph algorithm is not slowed down. Hence, the prefetch thread must read iterator state in order to compute runahead simultaneously with the main algorithm thread modifying it. 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 8). We appropriately initialise these structures and allow the runahead to traverse it *without synchronisation*. This leads to the runahead working on an inconsistent snapshot. However RASP tolerates approximations to the runahead function and therefore our design trades synchronisation overheads for wasted prefetch bandwidth.

7 Evaluation

We have implemented CEESOR and RASP as C libraries that are used by algorithms and iterators written in C. We use a system with a 3.2Ghz Intel™i7 CPU and the 512 GB Samsung™SSD characterised in Section 5. We describe the precise memory footprints of our benchmarks later in this section. The overall amount of main memory in our system was 32 GB, too small to fit all our graphs. In the interests of portability and simplicity, we chose to use the virtual memory subsystem of the stock Linux 3.2 kernel running on the system as a cache by memory mapping the CEESOR file. Prefetch requests were issued via the standard `posix_fadvise` interface using the `WILLNEED` hint. We did no tuning of the (Ubuntu 12.04) system other than disabling operating system readahead for the SSD as it caused the baseline system without RASP to perform poorly. We did so in the interests of a fair quantification of the speedups due to RASP.

7.1 CEESOR

We begin by evaluating CEESOR on a range of graphs shown in Table 2 to judge its efficiency at representing edge-list data. We tried to pick as many different graphs as possible to demonstrate the general applicability of CEESOR.

The first eight are the largest real-world graphs available in the online Stanford Large Network Dataset Collection [19] (*after eliminating isolated vertices*). In addition we also consider the DIMACS [20] road network of the entire United States. The Twitter data set was available online [21] and we used the connectivity information from that data set. The scale-free (power-law) graph was generated using the Graph500 tool [22] which itself is based on a recursive matrix generator [23]. We modified the generator slightly to output edges to external memory. It would otherwise buffer the entire graph in main memory, which is unfeasible for the size of the graphs we wished to consider. We also enabled a process for smoothing the degree distribution of vertices in the generated graph to ensure that we had a true scale-free graph. We wrote our own generator for generating random Erdõs-Réyni graphs, while the Watts-Strogatz graph was generated using SNAP [24].

We compare CEESOR to plain CSR using the following metrics.

1. Bits per edge with CEESOR [Bits(CEESOR)]: This is the actual number of bits per edge in the CEESOR representation after the enumeration using the algorithm of Section 3.

2. Bits per edge with CSR [Bits(CSR)]: We assume a fixed length representation using $\log(V)$ bits for each entry in the column index. Note that this is a somewhat unfair point of comparison as it restricts additions to the CSR format while there is no such problem with CEESOR.

Table 2 clearly illustrates the efficiency of CEESOR for the various graphs in our data set. *In all cases CEESOR is more efficient than CSR*. The reduction in the size of edge data (stored on external memory in our case) varies from 5% to as much as 76%. DIMACS shows the largest reduction in size. This is not surprising as the road network is a grid-like graph with a large amount of clustering, that is automatically detected and exploited by the enumeration algorithm. Geographically close road junctions (vertices) are assigned nearby enumerations leading to their connecting edges (roads) being represented with small offsets. Finally, CEESOR outperforms CSR even for random graphs (Erdõs-Réyni) confirming the analysis in Section 2.2.

Graph	V / E	Bits(CEESOR)	Bits(CSR)	Saving
amazon0601	403,394/244,308	13.04	19	31.36%
ca-HepPh	89,209/118,489	10.26	17	39.65%
cit-Patents	6,009,555/16,518,947	18.22	23	20.78%
p2p-Gnutella31	62,586/147,892	11.73	16	26.69%
soc-LiveJournal	4,847,571/42,851,237	18.45	23	19.78%
soc-sign-epinions	131,828/711,210	14.27	18	20.72%
web-Google	916,428/4,322,051	11.47	20	42.65%
wiki-Talk	2,394,385/4,659,565	17.61	22	19.95%
DIMACS-USA	23,947,347/28,854,312	5.84	25	76.64%
Twitter (TW)	52,579,682/1,614,106,187	23.3752	26	10.10%
Erdõs-Réyni (ER)	20,000,000/1,999,990,173	23.4907	25	6.04%
Scale-free (SF)	134,217,728/6,652,662,596	25.645	27	5.02%
Watts-Strogatz (WS)	20,000,000/1,999,995,087	20.3891	25	18.44%

Table 2: CEESOR Efficiency

7.2 RASP

We implemented breadth-first, SSSP and sequential iterators for evaluating RASP. For the SSSP 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 [25] 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.
- **Breadth-first Search (BFS)**: Visit every vertex in breadth first order starting from a randomly chosen source vertex. This is just the breadth-first iterator.
- **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. SSSP is implemented using the SSSP iterator.

In addition to these three 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 identical to one of the three algorithms above. This is unsurprising as they consist of the same basic iterators and are IO bound. We therefore report only the basic algorithms listed above.

We use the four largest graphs in our data set which are the last four graphs in Table 2 to evaluate RASP. Since our objective is to provide an IO workload we are careful to remove sources of variation in the runtime that are not attributable to IO performance. For BFS and SSSP we try every possible start vertex in an enclosing sequential iterator, reporting the overall runtime. This ensures that the entire CEESOR file on SSD is accessed regardless of the randomly chosen

Graph	Main Memory			Cache	CEESOR	Edge weight	Speedup over CSR		
	BFS	SSSP	Pagerank				BFS	SSSP	Pagerank
Twitter (TW)	1.18	2.74	1.57	2	9.16	24.05	1.40	1.08	1.15
Erdõs-Réyni (ER)	0.45	1.04	0.60	2	11.40	29.80	1.00	1.00	1.00
Scale-free (SF)	3.00	7.00	4.00	2	41.27	99.13	1.10	1.05	1.00
Watts-Strogatz (WS)	0.45	1.04	0.60	2	9.96	29.80	1.53	1.03	1.26

Table 3: Memory footprint in Gigabytes and speedup over CSR

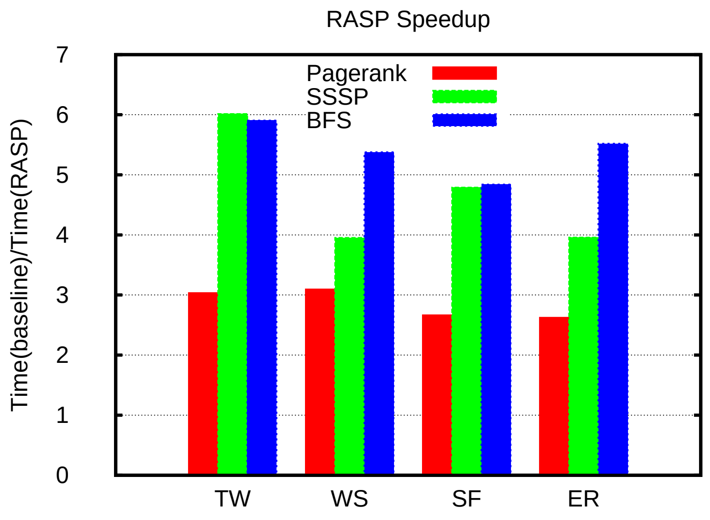

Figure 9: RASP Speedups for CEESOR

start vertex. For pagerank we fix the number of iterations at five to ensure a consistent IO workload. We converted the only undirected graph, Watts-Strogatz, into a directed one. The largest 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 3 shows the *runtime* memory footprints for all the combinations of graphs and benchmarks that we have run. In all cases, 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.

Switching to the CEESOR format from CSR provided both space savings as well as translating to reduced IO as each IO operation brings in a larger number of edges; effectively inflating SSD bandwidth. The last three columns of Table 3 show the speedups obtained by switching from CSR to CEESOR. The largest speedups are for the two graphs expected to have structure: the real world Twitter graph and the Watts-Strogatz graph.

We now focus on evaluating RASP by considering our baseline to be the algorithm running over the CEESOR file without RASP.

The results of running the algorithms with and without RASP are summarised in Figure 9. RASP speeds up the execution of single-threaded graph algorithms that are mostly IO bound

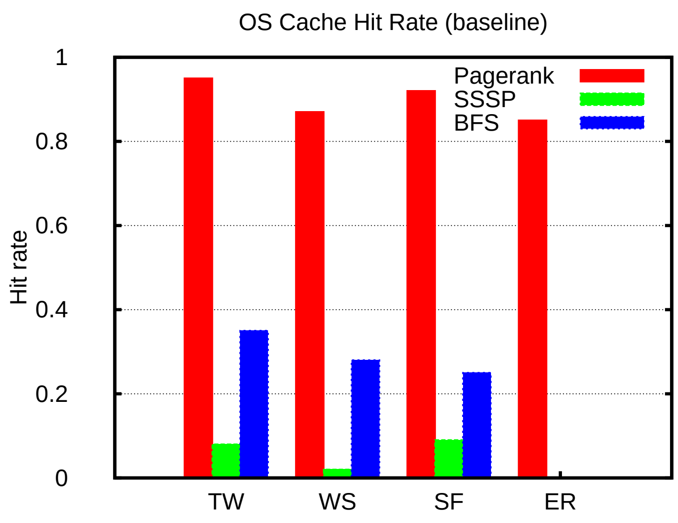

Figure 10: OS Cache Hit Rate CEESOR without RASP

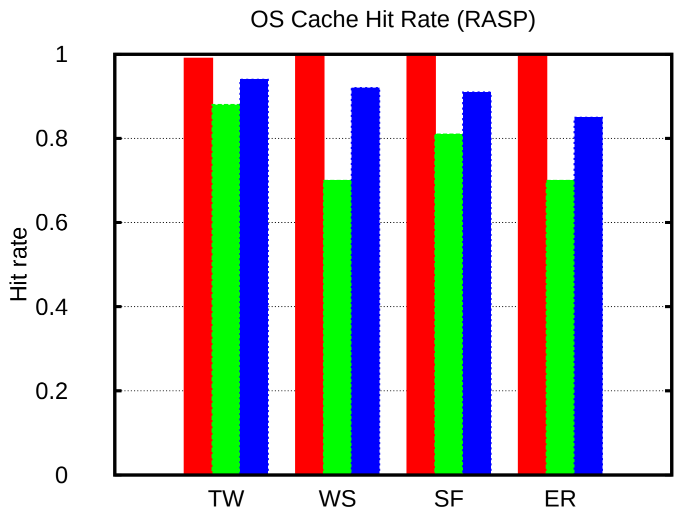

Figure 11: OS Cache Hit Rate CEESOR with RASP

by amounts varying from 2X to 6X. The speedup originates from reducing the IO latency due to improved utilisation of the SSD bandwidth, starting from *synchronous single-threaded* graph algorithms.

The basic driver for this improved performance is reduced IO latency. Accesses to the CEESOR file now see hits in the operating systems cache as necessary data has already been prefetched by RASP. This is illustrated in Figures 10 and11. We define iterating over all neighbours of a vertex as one access to the CEESOR file. We divide the number of major page faults by this metric to obtain the miss rate. The hit rate is improved considerably by prefetching. The lowest final hit rate is with SSSP. This is due to the lookahead being an approximation as described in Section 6.3.3, although this can be mitigated with data structure improvements described in that section. RASP uses the available bandwidth of the SSD for prefetching, this is shown in Figures 12 and 13. The increased read throughput from the SSD supplies the demand from the prefetcher. Finally, the high baseline hit rate of pagerank is due to its sequential nature that

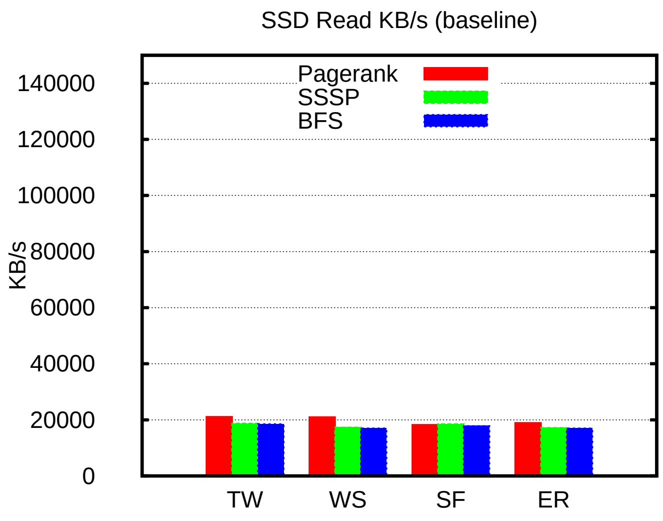

Figure 12: SSD Read Throughput CEESOR without RASP

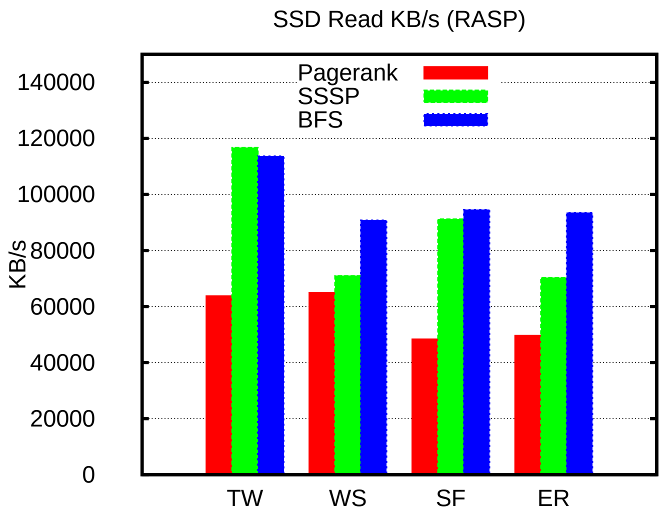

Figure 13: SSD Read Throughput CEESOR with RASP

completely uses all data on a fetched page before moving to the next. Table 4 shows absolute runtimes for the largest graph in our data set (6 billion edges).

It is interesting to contrast other results in the single machine category with ours. Bader et. al. [26] 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. Graphchi [4] achieves a single page rank iteration on the Twitter graph with an SSD in 158 seconds in comparison to our 180 seconds. We note however that they report main memory usage of 8 GB compared to only 3.57 GB in our case.

Our results also illustrate the limitations of our current implementation of RASP. Some of the requests from the prefetcher are already in cache leading to wasted SSD bandwidth. Conse-

Algorithm	Base (hr.)	RASP (hr.)	$k(S)\quad 1000$
BFS	6.32	1.28	75%
SSSP	15.67	3.75	75%
Pagerank	3.98	1.50	N/A

Table 4: Absolute Runtimes for Scale-free Graph

quently the hit rate does not touch 100% even with accurate lookahead in BFS.

There is however enough parallelism exposed by the lookahead. As Table 4 shows, for 75% of the time the runahead can access more than 1000 vertices. This means that there is enough scope for filtering out prefetch requests for pages already in cache. This is something that our current implementation does not include as Linux does not provide a low latency mechanism for querying the contents of the operating system cache from the user program. This can easily be remedied in a graph processing system with its own buffer manager.

8 Related Work

Researchers have also explored distributed graph processing as a way to scale graph processing algorithms for large data sets. Notable among these approaches is the Bulk Synchronous Processing (BSP) style of approaches adopted by systems such as Pregel [27]. These approaches partition the graph among multiple machines and synchronously update properties of vertices at each computation step. Also in the category of distributed approaches are graph databases and graph query platforms such as Trinity [28], Neo4j [29] and HyperGraphDB [30]. Many of these systems load the entire graph into main memory prior to processing.

Due to their generic nature both CEESOR and RASP can individually be incorporated into systems that store the graph in main memory and/or make use of multithreading. CEESOR can be used to reduce the footprint of graphs regardless of where they may be stored. RASP can also be used to prefetch graphs into shared last level caches common on many multicores today to alleviate the cost of cache misses even in main-memory multithreaded graph algorithms.

Another key area of related work is graph compression. CEESOR attempts to compress the edge list of graphs with structure. On a similar note, researchers have exploited specific structures of real-world graphs such as web graphs to design compression algorithms for them [31, 32]. We also exploit structure inherent in the graph to reduce the space needed to represent it. Unlike the other approaches though, *we can access the adjacency list of a vertex without reference to other vertices or dictionaries*. This is key to reducing IO latency; the other approaches would have required multiple references for decompression thereby increasing IO latency but possibly providing better compression than CEESOR. Further, CEESOR supports seamless updates as the in memory index consists of pointers and individual adjacency lists can be copied or updated.

9 Conclusion

This paper has presented two novel techniques: CEESOR and RASP. CEESOR is a variant of CSR that reduces the amount of space needed to represent the edge list of a graph; and RASP is a runtime SSD prefetcher for graphs that have hitherto been considered to have poor locality. We have demonstrated that CEESOR can significantly reduce the space required to represent graph edges and RASP can lead to large speedups and high hit-rates with even small memory caches. Both CEESOR and RASP are basic building blocks in low-cost graph processing systems that we currently use as well as in systems we are building [33]. We are investigating low cost graph partitioning algorithms [34] and adapting approximation algorithms for the related minimum linear arrangement problem [35] for even more efficient enumeration in CEESOR. We are also exploring automating RASP by integrating it into domain specific languages for graphs [36] as well as removing operating system related overheads currently present in our implementation.

Acknowledgment. The research is part funded by EU grant FP7-ICT-257756 and EPSRC grant EP/H003959. We would like to thank members of the Systems Research Group at Cambridge and Terence Kelly at HP for their feedback.

References

[1] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, pages 10–10. USENIX Association, 2004.

[3] Roger Pearce, Maya Gokhale, and Nancy M. Amato. 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*, pages 1–11. IEEE Computer Society, 2010.

[4] Aapo Kyrola and Guy Blelloch. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2012.

[5] Eshrat Reghbati and Derek G. Corneil. Parallel computations in graph theory. *SIAM J. Comput.*, 7(2):230–237, 1978.

[6] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 25–. IEEE Computer Society, 2005.

[7] Ulrich Meyer. *Design and analysis of sequential and parallel single-source shortest-paths algorithms*. PhD thesis, 2002.

[8] Yujie Han and Robert A. Wagner. An efficient and fast parallel-connected component algorithm. *J. ACM*, 37(3):626–642, July 1990.

[9] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[10] `http://www.twitter.com/`.

[11] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[12] Aaron Clauset, Cristopher Moore, and M EJ Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98–101, 2008.

[13] Paul Erdos and Alfred Renyi. On the evolution of random graphs. *Publ. Math. Inst. Hungary. Acad. Sci.*, 5:17–61, 1960.

[14] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001.

[15] Darrel Hankerson, Peter D. Johnson, and Greg A. Harris. *Introduction to Information Theory and Data Compression*. CRC Press, Inc., 1st edition, 1998.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

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

[18] `http://freecode.com/projects/fio`.

[19] `http://snap.stanford.edu/data/`.

[20] `http://dimacs.rutgers.edu/Challenges/`.

[21] `http://twitter.mpi-sws.org/`.

[22] `http://www.graph500.org/`.

[23] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[24] `http://snap.stanford.edu/snap/index.html`.

[25] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[26] David A. Bader and Kamesh Madduri. 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*, pages 523–530. IEEE Computer Society, 2006.

[27] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing.

In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[28] B. Shao, H. Wang, and Y. Li. The trinity graph engine. Technical report, Stanford University, 1998.

[29] `http://neo4j.org/`.

[30] `http://www.hypergraphdb.org/index`.

[31] Micah Adler and Michael Mitzenmacher. Towards compressing web graphs. In *Proceedings of the Data Compression Conference*, pages 203–. IEEE Computer Society, 2001.

[32] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228. ACM, 2009.

[33] Eiko Yoneki and Amitabha Roy. A Unified Graph Query Layer for Multiple Databases. Technical Report UCAM-CL-TR-820, University of Cambridge, Computer Laboratory, August 2012.

[34] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[35] Uriel Feige and James R. Lee. An improved approximation ratio for the minimum linear arrangement problem. *Information Processing Letters*, 101(1):26–29, January 2007.

[36] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 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*, pages 349–362. ACM, 2012.