Size Oblivious Programming with *InfiniMem*[⋆]

Sai Charan Koduru, Rajiv Gupta, Iulian Neamtiu

Department of Computer Science and Engineering
University of California, Riverside
`{scharan,gupta,neamtiu}@cs.ucr.edu`

Abstract. Many recently proposed BigData processing frameworks make programming easier, but typically expect the datasets to fit in the memory of either a single multicore machine or a cluster of multicore machines. When this assumption does not hold, these frameworks fail. We introduce the *InfiniMem* framework that enables *size oblivious processing* of large collections of objects that do not fit in memory by making them *disk-resident*. *InfiniMem* is easy to program with: the user just indicates the large collections of objects that are to be made disk-resident, while *InfiniMem* transparently handles their I/O management. The *InfiniMem* library can manage a very large number of objects in a uniform manner, even though the objects have different characteristics and relationships which, when processed, give rise to a wide range of access patterns requiring different organizations of data on the disk. We demonstrate the ease of programming and versatility of *InfiniMem* with 3 different probabilistic analytics algorithms, 3 different graph processing *size oblivious frameworks*; they require minimal effort, 6–9 additional lines of code. We show that *InfiniMem* can successfully generate a mesh with 7.5 million nodes and 300 million edges (4.5 GB on disk) in 40 minutes and it performs the PageRank computation on a 14GB graph with 134 million vertices and 805 million edges at 14 minutes per iteration on an 8-core machine with 8 GB RAM. Many graph generators and processing frameworks cannot handle such large graphs. We also exploit *InfiniMem* on a cluster to scale-up an object-based DSM.

1 Introduction

BigData processing frameworks are an important part of today's data science research and development. Much research has been devoted to scale-out performance via distributed processing [8,12,13,17] and some recent research explores scale-up [1,6,11,15,16,21]. However, these scale-up solutions typically assume that the input dataset fits in memory. When this assumption does not hold, they simply fail. For example, experiments by Bu et al. [4] show that different open-source Big Data computing systems like Giraph [1], Spark [21], and Mahout [19] often crash on various input graphs. Particularly, in one of the experiments, a 70GB web graph dataset was partitioned across 180 machines (each with 16 GB RAM) to perform the PageRank computation. However, all the systems crashed with `java.lang.OutOfMemoryError`, even though there was less

[⋆] This work was supported by NSF Grant CCF-1524852, CCF-1318103, CNS-1157377, CCF-0963996, CCF-0905509, and a Google Research Award.

than 500MB of data to be processed per machine. In our experiments we also found that GTgraph's popular R-MAT generator [2], a tool commonly used to generate power-law graphs, crashed immediately with a *Segmentation Fault* from memory allocation failure when we tried to generate a graph with 1M vertices and 400M edges on a machine with 8GB RAM.

Motivated by the above observations, in this paper, we develop *InfiniMem*, a system that enables *Size Oblivious Programming* – the programmer develops the applications without concern for the input sizes involved and *InfiniMem* ensures that these applications do not run out of memory. Specifically, the *InfiniMem* library provides interfaces for transparently managing a large number of objects stored in files on disk. For efficiency, *InfiniMem* implements different read and write policies to handle objects that have different characteristics (fixed size vs. variable size) and require different handling strategies (sequential vs. random access I/O). We demonstrate the ease of programming with *InfiniMem* by programming BigData analysis applications like frequency estimation, exact membership query, and Bloom filters. We further demonstrate the *versatility* of *InfiniMem* by developing size oblivious graph processing frameworks with three different graph data representations: vertex data and edges in a single data structure; decoupled vertex data and edges; and the shard representation used by GraphChi [11]. One advantage of *InfiniMem* is that it allows researchers and programmers to easily experiment with different data representations with minimal additional programming effort. We evaluate various graph applications with three different representations. For example, a quick and simple shard implementation of PageRank with *InfiniMem* performs within ∼30% of GraphChi.

The remainder of the paper is organized as follows: Section 2 motivates the problem and presents the requirements expected from a *size oblivious programming* system. Section 3 introduces the programming interface for size oblivious programming. Section 4 describes the object representation used by *InfiniMem* in detail. Section 5 describes the experimental setup and results of our evaluation. Related work and conclusions are presented in Sections 6 and 7, respectively.

2 Size Oblivious Programming

The need to program processing of very large data sets is fairly common today. Typically a processing task involves representing the data set as a large collection of objects and then performing analysis on them. When this large collection of objects does not fit in memory, the programmer must spend considerable effort on writing code to make use of disk storage to manage the large number of objects. In this work we free the programmer from this burden by developing a system that allows the programmer to write *size oblivious* programs, i.e., programs where the user need not explicitly deal with the complexity of using disk storage to manage large collections of objects that cannot be held in available memory. To enable the successful execution of size oblivious programs, we propose a general-purpose programming interface along with an *I/O efficient* representation of objects on disk. We now introduce a few motivating applications and identify requirements to achieve I/O efficiency for our *size oblivious programming* system.

Motivating applications: Consider an application that is reading continuously streaming input into a Hash Table in heap memory (lines 1–3, Algorithm 1); a website analytics data stream is an excellent example of this scenario. When the memory gets full, the `insert` on line 3 could fail, resulting in an application failure. Similarly, consider the GTGraph [2] graph generator which fails to generate a graph with 1M edges and 400M vertices. Consider a common approach to graph generation which assumes that the entire graph can be held in memory during generation, as illustrated by lines 8–15 in Algorithm 1. First, memory for `NUM-VERTICES` is allocated (line 8) and then the undirected edges are generated (lines 11-13). Note that the program can crash *as early as line 8* when memory allocation fails due to a large number of vertices. Finally, consider the problem of graph processing, using SSSP as a proxy for a large class of graph processing applications. Typically, such applications have three phases: (1) input, (2) compute, and (3) output. The pseudocode for SSSP is outlined in lines 16–31 in Algorithm 1, highlighting these three phases. *Note that if the input graph does not fit in memory, this program will not even begin execution.*

Algorithm 1: Motivating applications: Membership Query, Mesh Generation and Graph Processing.

```
1  HashTable ht;
2  while read(value) do
3      ht.insert(value);
4  while more items do
5      if ht.find(item) then
6          print("Item found");
7  ————————————————————————
8  Mesh m(NUM-VERTICES)
9  foreach node n in Mesh m do
10     i ← rand(0, MAX);
11     for j=0; j < i; j++ do
12         n.addNeighbor(m[j]);
13         m[j].addNeighbor(n);
14 foreach Node n in Mesh m do
15     Write(n)
16 Graph g;
17 while not end of input file do
18     read next;
19     g.Add( α(next) );
20 repeat
21     termCondition ← true;
22     forall the Vertices v in g do
23         for int i=0; i<v.nbrs(); i++ do
24             Vertex n = v.neighbors[i];
25             if v.dst>n.dst+v.wt[i] then
26                 v.dst←(n.dst+v.wt[i]);
27     if NOT converged then
28         termCondition ← false;
29 until termCondition is true;
30 foreach Node n in Graph g do
31     Write(n);
```

Our solution: We focus on supporting size oblivious programming for C++ programs via the *InfiniMem* C++ library and runtime. Examples in Algorithm 1 indicate that the data structures that can grow very large are represented as *collections* of objects. Size Oblivious Programming with *InfiniMem* simply requires programmers to *identify potentially large collections of objects* using very simple abstractions provided by the library and these collections are transparently made *disk-resident* and can be efficiently and concurrent accessed. We now analyze these representative applications to tease out the requirements for *size oblivious programming* that have influenced the architecture of *InfiniMem*.

Let us reconsider the pseudocode in Algorithm 1, mindful of the requirement of efficient I/O. Lines 5–6 will execute for *every* key in the input; similarly, lines 9 and 14 indicate that lines 10–13 and line 15 will be executed for *every* node in the mesh. Similarly, line 22 indicates that lines 23–26 will be performed on *every* vertex in the graph. It is natural to read a contiguous *block* of data so that no additional I/O is required for lines 24–26 for the vertices and is an efficient disk I/O property. Moreover, this would be useful for any application in general, by way of decreasing I/O requests and batching as much I/O as possible. Therefore, we have our first requirement:

Support for efficient block-based IO.

Consider next, the example of the hash table where the input data is *not* sorted; then, line 3 of Algorithm 1 motivates need for random access for indexing into the hash table. As another example, observe that line 24 in Algorithm 1 fetches every neighbor of the current vertex. When part of this graph is disk-resident, we need a way of efficiently fetching the neighbors, much like *random access* in memory. This is important because any vertex in a graph serves two roles: (1) vertex and (2) neighbor. For the role (1), if vertices are contiguously stored on disk block-based I/O can be used. However, when the vertex is accessed as a neighbor, the neighbor could be stored anywhere on disk, and thus requires an imitation of random access on the disk. Hence our next requirement is:

Support for efficient, random access to data on disk.

To make the case for our final requirement, consider a typical definition of the HashTable shown in Figure 1a. Each `key` can store multiple values to support *chaining*. Clearly, each `HashTableEntry` is a variable sized entity, as it can hold multiple `values` by chaining. As another example, consider the definition for a *Vertex* shown in Figure 1b: the size of `neighbors` array can vary; and with the exception of the `neighbors` member, the size of a *Vertex* can be viewed as a fixed-size object. When reading/writing data from/to the disk, one can devise very fast block-based I/O for fixed-size data (Section 4). However, reading variable-sized data requires remembering the size of the data and performing n reads of appropriate sizes; this is particularly wasteful in terms of disk I/O bandwidth utilization. For example, if the average number of neighbors is 10, every time a distance value is needed, we will incur a 10x overhead in read but useless data. As a final example, Figure 1c is an example of an arbitrary container that showcases the need for both fixed and variable sized data. Hence we arrive at our final requirement from *InfiniMem*:

Support to speed up I/O for variable-sized data.

```
template <typename K, typename V>
struct HashTableEntry {
  K key;
  V* values; /* for chaining */
};
```

(a) Hash Table

```
struct Vertex {
   int distance;
   int* weights; /*Edge weights*/
   Vertex* neighbors;   /*Array*/
};
```

(b) Graph Vertex

```
template<typename T>
struct Container{
  T stackObjects[96]; /* Fixed */
  T *heapObjects;  /* Variable */
};
```

(c) Arbitrary container

Fig. 1: Common data structure declarations to motivate the need for explicit support for fixed and variable sized data, block based and random IO.

The goal of *InfiniMem* is to transparently support disk-resident versions of object collections so that they can grow to large sizes without causing programs to crash. *InfiniMem*'s design allows size oblivious programming with little effort as the programmer merely identifies the presence and processing of potentially large object collections via *InfiniMem*'s simple programming interface. The details of how *InfiniMem* manages I/O (i.e., uses block-based I/O, random access I/O, and I/O for fixed and variable sized data) during processing of a disk-resident data structure are hidden from the programmer.

3 The *InfiniMem* Programming Interface

InfiniMem is a C++ template library that allows programmers to identify size oblivious versions of fixed- and variable-sized data collections and enables transparent, efficient processing of these collections. We now describe *InfiniMem*'s simple application programming interface (API) that powers *size oblivious programming*. *InfiniMem* provides a *high-level* API with a default *processing* strategy that hides I/O details from the programmer; however the programmer has the flexibility to use the *low-level* API to implement any customized processing.

```
template<typename T>
struct Container: public Box<T> { // or Bag<T>
  T data;
  void update() {  /* for each T */
     ...
  }

  void process();
};

typedef Container<int> intData;

typedef Container<MyObject> objData;

int main() {
   Infinimem<intData> idata;
   idata.read("/input/file");
   idata.process();

   Infinimem<objData> odata;
   odata.read("/input/data/");
   odata.process();
}
```

```
template<typename T>
T Box::fetch(ID);

template<typename T>
T* Box::fetchBatch(ID, count);

template<typename T>
void Box::store(ID, const T*);

template<typename T>
void Box::storeBatch(ID, count);

template<typename T>
T Bag::fetch(ID);

template<typename T>
T* Bag::fetchBatch(ID, count);

template<typename T>
void Bag::store(ID, const T*);

template<typename T>
void Bag::storeBatch(ID, count);
```

Fig. 2: Programming with *InfiniMem*: the `Box` and `Bag` interfaces are used for *fixed size* and *variable sized* objects; `process` drives the computation using the user-defined `update()` methods and the low-level `fetch()` and `store()` API.

Identifying Large Collection of Objects: In *InfiniMem*, the programmer identifies object collections that potentially grow large and need to be made disk-resident. In addition, the programmer classifies them as fixed or variable sized. This is achieved by using the `Box` and `Bag` abstractions respectively. The `Box` abstraction can be used to hold fixed-size data, while the `Bag` holds flexible-sized data. Figure 2 illustrates an example and lists the interface. The programmer uses the `Box` or `Bag` interface by simply inheriting from the `Box` (or `Bag`) type and provides an implementation for the `update()` method to process each object in the container. Here, `Container` is the collection that can potentially grow large, as identified by the programmer. `Infinimem` is the default processing engine; *InfiniMem*'s `process()` function hides the details of I/O and fetches objects as needed by the `update()` method, thereby enabling *size oblivious programming.*

Processing Data: The `process()` method is the execution engine: it implements the low-level details of efficiently fetching objects from the disk, applies the user-defined `update()` method and efficiently spills the updated objects to disk. Figure 3 details the default `process()`. By default, the `process()`-ing engine `fetch`es, `processes` and `store`-es data in batches of size `BATCH_SIZE` which is automatically determined from available free memory such that the entire batch fits and can be processed in memory.

```
// SZ = SIZEOF_INPUT; BSZ = BATCH_SIZE;
Box<T>::process() { // or Bag<T>
  for(i=0; i<SZ; i+=BSZ) {
    // fetch a portion of Box<T> or Bag<T>
    cache = fetchBatch(ID(i), BSZ);
    for(j=0; j<BSZ; j++)
      cache[j].update();
  }
}
```

Fig. 3: *InfiniMem*'s generic batch `process()`-ing.

While *InfiniMem* provides the default implementation for `process()` shown in Figure 3, this method can be overridden: programmers can use the accessors and mutators exposed by *InfiniMem* (Figure 2) to write their own processing frameworks. Notice that *InfiniMem* natively supports both sequential/block-based and random accessors and mutators, satisfying each of the requirements formulated earlier. For block-based and random access, *InfiniMem* provides the following intuitively named fetch and store APIs: `fetch()`, `fetchBatch()`, `store()` and `storeBatch()`.

Illustration of* InfiniMem *for graph processing: We next demonstrate *InfiniMem*'s versatility and ease of use by programming graph applications using *three* different graph representations. We start with the standard declaration of a `Vertex` as seen in Figure 1b. An alternate definition of `Vertex` separates the fixed sized data from variable sized edgelist for IO efficiency, and used in many vertex centric frameworks [12,11]. Finally, we program GraphChi's [11] shards.

Figure 4a declares the `Graph` to be a `Bag` of vertices, using the declaration from Figure 1b. With this declaration, the programmer has identified that the collection of vertices is the potentially large collection that can benefit from size oblivious programming. The `preprocess()` phase partitions the input graph into disjoint intervals of vertices to allow for parallel processing. These examples use a vertex-centric graph processing approach where the `update()` method of `Vertex` defines the algorithm to process each vertex in the graph. The `process()` method of `Graph` uses the accessors and mutators from Figure 2 to provide a *size oblivious programming* experience to the programmer. Figure 4b declares a `Graph` as the composition of a `Box` of `Vertex` and a `Bag` of EdgeLists, where `EdgeList` is an implicitly defined list of neighbors. Finally, Figure 4c uses a similar graph declaration, with the simple tweak of creating an array of `N` `shard` partitions; a `shard` imposes additional constraints on the vertices that are in the shard: vertices are partitioned into intervals such that all vertices with neighbors in a given vertex interval are all available in the same shard [11], enabling fewer random accesses by having all vertices' neighbors available before processing each shard. Note that representing shards in *InfiniMem* is very simple.

```
void Vertex::update() {
  foreach(neighbor n)
    distance = f(n.distance);
}

template <typename V>
class Graph {
  Bag<V> vertices;

public:
  void process();
};

int main() {
   Graph<Vertex> g;
   g.read("/path/to/graph");
   g.preprocess(); //Partition
   g.process();
}
```

(a) Graph for Vertex in Figure 1b.

```
void Vertex::update() {
  foreach(neighbor n)
    distance = f(n.distance);
}

template <typnam V,typnam E>
class Graph {
  Box<V> vertices;
  Bag<E> edgeLists;

public:
  void process();
};

int main() {
 Graph<Vertex, EdgeList> g;
 g.read("/path/to/graph");
 g.preprocess(); //Partition
 g.process();
}
```

(b) Decoupling Vertices from Edgelists.

```
void Vertex::update() {
  foreach(neighbor n)
    distance = f(n.distance);
}

template <typename V,typename E>
class Graph {
  Box<V> vertexShard[N];
  Bag<E> edgeShard[N];

public:
  void processShard(int);
};

int main() {
 Graph<Vertex, EdgeList> g;
 g.read("/path/to/graph");
 g.createShards(N);//Preprocess
 for(int i=0; i<N; i++)
     g.processShard(i);
}
```

(c) Using Shard representation of graphs.

Fig. 4: Variations of graph programming, showcasing the ease and versatility of programming with *InfiniMem*, using its high-level API.

```
// SZ = SIZEOF_INPUT;
// BSZ = BATCH_SIZE;
// vb = vertices;

Graph<V>::process() {
 for(i=0; i<SZ; i+=BSZ) {
  // fetch a batch
  vb=fetchBatch(ID(i), BSZ);

  for(j=0; j<BSZ; j++)
    vb[j].update();

  storeBatch(vb, BSZ);
 }
}
```

(a) `process()`-ing graph in Figure 1b.

```
// SZ = SIZEOF_INPUT;
// BSZ = BATCH_SIZE;
// v = vertices;
// e = edgeLists;

Graph<V, E>::process() {
 for(i=0; i<SZ; i+=BSZ) {
  // fetch a batch
  vb=v.fetchBatch(ID(i), BSZ);

  // fetch corr. edgelist
  eb=e.fetchBatch(ID(i), BSZ);

  for(j=0; j<BSZ; j++)
    vb[j].update(eb[j]);

  storeBatch(vb, BSZ);
 }
}
```

(b) `process()` for decoupled Vertex.

```
// NS = NUM_SHARDS;
// SS = SIZEOF_SHARD;
// vs = vertexShard;

Graph<V, E>::process() {
 for(i=0; i<NS; i++) {
  // fetch entire memory shard
  mshrd = vs[i].fetchBatch(..,SS);

  // fetch sliding shards
  for(j=0; j<NS; j++)
   sshrd += vs[j].fetchBatch(.,.);

  sg = buildSubGraph(mshrd,sshrd);

  foreach(v in sg)
    v.update();

  storeBatch(mshrd, SS);
 }
}
```

(c) Custom `process()` for shards.

Fig. 5: Default and custom overrides for `process()` via low-level *InfiniMem* API.

Figure 5a illustrates the default `process()`: objects in the `Box` or `Bag` are read in batches and processed one at a time. For graphs with vertices decoupled from edgelists, vertices and edgelists are read in batches and processed one vertex at a time (Figure 5b); batches are concurrently processed. Figure 5c illustrates custom shard processing: each memory shard and corresponding sliding shards build the subgraph in memory; then each vertex in the subgraph is processed [11].

4 *InfiniMem*'s I/O Efficient Object Representation

We now discuss the I/O efficient representation provided by *InfiniMem*. Specifically, we propose an *Implicitly Indexed* representation for fixed-sized data (`Box`); and an *Explicitly Indexed* representation for variable-sized data (`Bag`).

As the number of objects grows beyond what can be accommodated in main memory, the frequency of object I/O to/from disk storage will increase. This warrants an organization of the disk storage that reduces I/O latency. To allow an object to be addressed regardless of where it resides, it is assigned a unique numeric `ID` from a stream of non-negative, monotonically increasing integers. Figure 6 shows the access mechanism for objects using their IDs: fixed-sized data is stored at a location determined by its `ID` and its fixed size: `FILE_START + (sizeof(Object)*ID)`. For variable-sized data, we use a *metafile* whose fixed-sized address entries store the offset of the variable-sized data into the *datafile*. The `Vertex` declared in Figure 4a for example, would only use the *explicitly indexed* `Bag` notation to store data, while the representations in Figure 4b and Figure 4c use both the `Box` and `Bag` for the fixed size `Vertex` and the variable sized `EdgeList` respectively. Thus, fixed-sized data can be fetched/stored in a *single logical* disk seek and variable-sized data in *two logical* seeks. This ensures fetch and store times are nearly constant with *InfiniMem* and independent of the number of objects in the file (like random memory access), and enabling:

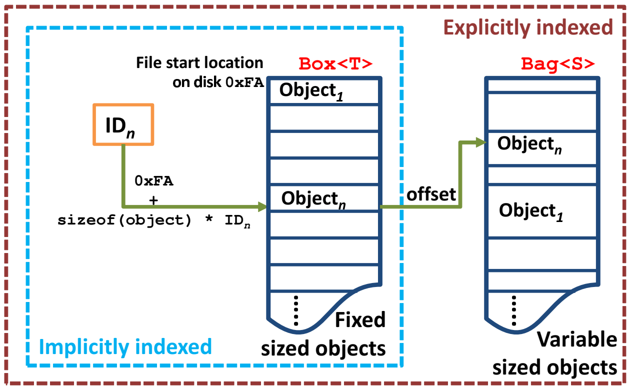


Fig. 6: Indexed disk representation of fixed- and variable-sized objects.

– **Efficient access for Fixed-Sized objects:** Using the object `ID` to index into the datafile, *InfiniMem* gives fast access to fixed-sized objects in 1 logical seek.
– **Efficient access for Variable-Sized objects:** The metafile enables fast, random-access access to objects in the datafile, in at most 2 logical seeks.
– **Random Access Disk I/O:** The indexing mechanism provides an imitation of random access to both fixed and variable sized objects on disk.
– **Sequential/Batch Disk I/O:** To read `n` consecutive objects, we seek to the start of the first object. We then read `sizeof(obj)*n` bytes and up to the end of the last object in the sequence for fixed- and variable-sized objects, respectively.
– **Concurrent I/O:** For parallel processing, different objects in the datafile must be concurrently and safely accessed. Given the large number of objects, individual locks for each object would be impractical. Instead, *InfiniMem* provides

locks for groups of objects: to decrease lock conflicts, we group non-contiguous objects using modulo `ID` modulo a `MAX_CONCURRENCY` parameter set at 25.

5 Evaluation

We now evaluate the programmability and performance of *InfiniMem*. This evaluation is based upon three class of applications: probabilistic web analytics, graph/mesh generation, and graph processing. We also study the scalability of size oblivious applications written using *InfiniMem* with degree of parallelism and input sizes. We programmed size oblivious versions of several applications using *InfiniMem* and are listed in Table 1. We begin with data analytics benchmarks: frequency counting using *arrays*, membership query using *hash tables*, and probabilistic membership query using *Bloom filters*. Then, in addition to mesh generation, in this evaluation, we use a variety of graph processing algorithms from diverse domains like graph mining, machine learning, etc. The *Connected Components* (CC) algorithm finds nodes in a graph that are connected to each other by at least one path, with applications in graph theory. *Graph Coloring* (GC) assigns a color to a vertex such that it is distinct from those of all its neighboring vertices with applications in register allocation etc. In a web graph, *PageRank* (PR) [14] iteratively ranks a page based on the ranks of pages with inbound links to the page and is used to rank web search results. *NumPaths* (NP) counts the number of paths between a source and other vertices. From a source node in a graph, *Single Source Shortest Path* (SSSP) finds the shortest path to all other nodes in the graph with applications in logistics and transportation.

5.1 Programmability

Application	Additional LoC	Application	Additional LoC
Probabilistic Web Analytics		Graph Processing	
Freq. Counting	$2 + 3 + 3 = 8$	Graph Coloring	
Member Query	$2 + 3 + 3 = 8$	PageRank	
Bloom Filter	$2 + 4 + 3 = 9$	SSSP	$1 + 3 + 2 = 6$
Graph/Mesh Generation		Num Paths	
Mesh Generation	$2 + 2 + 2 = 6$	Conn. Components	

Table 1: Between 6 and 9 *additional* lines of code are needed to make these applications *size oblivious*. Graph processing uses decoupled version (Figure 4b).

Writing size oblivious programs with *InfiniMem* is simple. The programmer needs to only: (a) initialize the *InfiniMem* library, (b) identify the large collections and `Box` or `Bag` them as necessary, and (c) use the default `process()`-ing engine or provide a custom engine. Table 1 quantifies the ease of programming with *InfiniMem* by listing the number of additional lines of code for these tasks to make the program size oblivious using the default processing engine. At most 9 lines of code are needed in this case and *InfiniMem* does all the heavy lifting with about 700 lines for the I/O subsystem, and about 900 lines for the runtime, all of which hides the complexity of making data structures disk-resident from the user. Even programming the shard processing framework was relatively easy: about 100 lines for simplistic shard generation and another 200 lines for rest of the processing including loading memory and corresponding sliding shards, building the subgraph in memory and processing the subgraph; rest of the complexity of handling the I/O etc., are handled by *InfiniMem*.

5.2 Performance

We now present the runtime performance of applications programmed with *InfiniMem*. We evaluated *InfiniMem* on a Dell Inspiron machine with 8 cores and 8GB RAM with a commodity 500GB, 7200RPM SATA 3.0 Hitachi `HUA722050CLA330` hard drive. For consistency, the disk cache is fully flushed before each run.

Input Graph	$\|V\|$	$\|E\|$	Size
Pokec	1,632,804	61,245,128	497M
Live Journal	4,847,571	68,993,773	1.2G
Orkut-2007	3,072,627	223,534,301	3.2G
Delicious-UI	33,778,221	151,772,443	4.2G
RMAT-536-67	67,108,864	536,870,912	8.8G
RMAT-805-134	134,217,728	805,306,368	14G

Table 2: Inputs used in this evaluation.

Size Oblivious *Graph Processing:* We begin with the evaluation of graph processing applications using input graph datasets with varying number of vertices and edges, listed in Table 2. Orkut, Pokec, and LiveJournal graphs are directed graphs representing friend relationships. Vertices in the Amazon graph represent products, while edges represent purchases. The largest input in this evaluation is rmat-805-134 at 14GB on disk, 805M edges and 134M vertices.

We first discuss the benefits of decoupling edges from vertices. When vertex data and edgelists are in the same data structure, line 22 in Algorithm 1 requires fetching the edgelists for the vertices even though they are not used in this phase of the computation. Decoupling the edgelists from vertex data has the benefit of avoiding wasteful I/O as seen in Table 3. The very large decrease in running time is due to the extremely wasteful I/O that reads the variable sized edgelists along with the vertex data even though only the vertex data is needed.

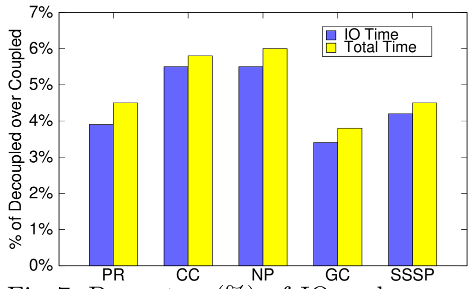


Fig. 7: Percentage(%) of IO and execution time for decoupled over coupled representations for various applications on the 'Delicious-UI' input.

Figure 7 shows the I/O breakdowns for various benchmarks on the moderately sized Delicious-UI input. While the programming effort with *InfiniMem* is already minimal, switching between representations for the same program can be easier too: with as little as *a single change* to data structure definition (figures 4a-4b), the programmer can evaluate different representations.

Tables 4 and 5 show the frequencies and percentage of total execution time spent in various I/O operations for processing the *decoupled* graph representation with *InfiniMem*, as illustrated in Figure 4b. Observe that the number of batched vertex reads and writes is the same in Table 4 since both vertices and edgelists are read together in batches. There are no individual vertex writes since *InfiniMem* only writes vertices in batches. Moreover, the number of batched vertex writes is less than the reads since we write only updated vertices and as the algorithm converges, in some batches, there are no updates. Observe in Table 5 that as described earlier, the maximum time is spent in random vertex reads.

Input Graph	PageRank		Conn Comp		Numpaths		Graph Coloring		SSSP	
	Co	DeCo	Co	DeCo	Co	DeCo	Co	DeCo	Co	DeCo
Pokec	2,228	172	352	60	37	8	277	28	48	7
Live Journal	8,975	409	1,316	122	106	14	602	58	133	70
Orkut	3,323	81	3,750	277	459	11	3,046	140	660	154
Delicious-UI	32,743	1,484	15,404	904	1,112	67	9,524	365	1,453	65
rmat-536-67	23,588	3,233	12,118	2,545	1,499	861	5,783	1,167	1,853	584
rmat-805-134	25,698	3,391	>8h	3,380	3,069	1,482	11,332	2,071	>8h	2,882

Table 3: Decoupling vertices and edgelists avoids wasteful I/O (runntime time shown is in seconds). 'Co' and 'DeCo' refer to coupled and decoupled respectively.

I/O Operation	LiveJournal	Orkut	Delicious-UI	rmat-536-67	rmat-805-134
Vertex Batched Reads	7,891	421	40,578	12,481	24,052
Edge Batched Reads	7,891	421	40,578	12,481	24,052
Vertex Individual Reads	865e+6	188e+6	2.8e+9	1.8e+9	2.5e+9
Vertex Batched Writes	7,883	413	40,570	12,473	24,044

Table 4: Frequencies of operations for various inputs for PageRank.

I/O Operation	LiveJournal	Orkut	Delicious-UI	rmat-536-67	rmat-805-134
Vertex Batched Reads	0.05%	0.02%	0.31%	0.12%	0.13%
Edge Batched Reads	8.48%	2.75%	11.25%	7.75%	9.72%
Vertex Individual Reads	54.80%	71.59%	76.96%	86.47%	81.73%
Vertex Batched Writes	0.12%	0.03%	0.37%	0.04%	0.10%
Total IO	63.45%	74.39%	88.89%	94.38%	91.68%

Table 5: Percentage of time for I/O operations for various inputs for PageRank.

Sharding with* InfiniMem*: In the rest of this discussion, we always use the decoupled versions of Vertex and EdgeLists. We now compare various versions of graph processing using *InfiniMem*. Table 6 compares the performance of the two simple graph processing frameworks we built on top of *InfiniMem* with that of GraphChi-provided implementations in their 8 thread configuration. *InfiniShard* refers to the shard processing framework based on *InfiniMem*. In general, the slowdown observed with *InfiniMem* is due to the large number of random reads generated, which is O($|E|$). For PageRank with Orkut, however, we see speedup for the following reason: as the iterations progress, the set of changed vertices becomes considerably small: ∼50. So, the number of random reads generated also goes down considerably, speeding up PageRank on the Orkut input. With Connected Components, our *InfiniMem* runs slower primarily because the GraphChi converges in less than half as many iterations on most inputs. Table 6 also presents the data for PageRank that processes shards with our *InfiniMem* library as compared to the very fine-tuned GraphChi library. The speedup observed in Table 6 from *InfiniMem* to *InfiniShard* is from eliminating random reads enabled by the shard format. Notice that even with our quick, unoptimized ∼350 line implementation of sharding, the average slowdown we see is only 18.7% for PageRank and 22.7% for Connected Components compared to the highly tuned and hand-optimized GraphChi implementation. Therefore, we have shown that *InfiniMem* can be used to easily and quickly provide a *size oblivious programming* experience along with I/O efficiency for quickly evaluating various representations of the same data.

Input Graph	PageRank Time (sec)			Conn. Comp. Time (sec)		
	InfiniMem (speedup)	*InfiniShard* (speedup)	GraphChi	*InfiniMem* (speedup)	*InfiniShard* (speedup)	GraphChi
Pokec	172 (0.72)	121 (1.02)	124	60 (0.40)	26 (0.92)	24
LiveJournal	409 (0.90)	488 (0.76)	371	122 (0.49)	80 (0.75)	60
Orkut	81 (1.91)	190 (0.82)	156	277 (0.44)	142 (0.87)	123
Delicious-UI	1,484 (0.43)	730 (0.89)	652	904 (0.17)	191 (0.78)	149
rmat-536-67	3,233 (0.36)	1,637 (0.70)	1,146	2,545 (0.21)	746 (0.71)	529
rmat-805-134	3,391 (0.44)	2,162 (0.69)	1,492	3,380 (0.30)	1,662 (0.61)	1,016

Table 6: *InfiniMem* (decoupled) vs. *InfiniShard*; Speedups over GraphChi.

Size-Oblivious Programming *of Probabilistic Apps:* Here, we present the throughput numbers for the probabilistic applications in Table 7. We evaluated these applications by generating uniformly random numeric input. Frequency counting is evaluated by counting frequencies of random inserts while membership query and Bloom filter are evaluated using uniformly generated random queries on the previously generated uniformly random input. Jenkins hashes are used in Bloom filter. Bloom filter achieves about half the throughput of Frequency Counting since Bloom filter generates twice as many writes.

Application	Throughput (qps)
Frequency Counting	635,031
Membership Query	446,536
Bloom Filter	369,726

Table 7: QPS for the probabilistic apps.

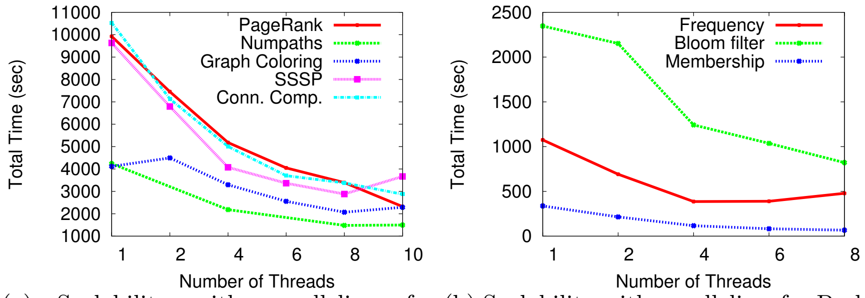
We also experimented with querying. We searched for entries using the Orkut input file (3.2GB on disk) as an input file. Using a naive, sequential scan and search took 67 seconds. Using *InfiniMem* with 1 thread took 15 seconds, while using 4 threads took 5 seconds for the *same* naive implementation. The highly optimized GNU Regular Expressions utility took an average of 4.5 seconds for the same search. This shows that in addition to ease of programming, *InfiniMem* performs well even with very simple implementations.

5.3 Scalability

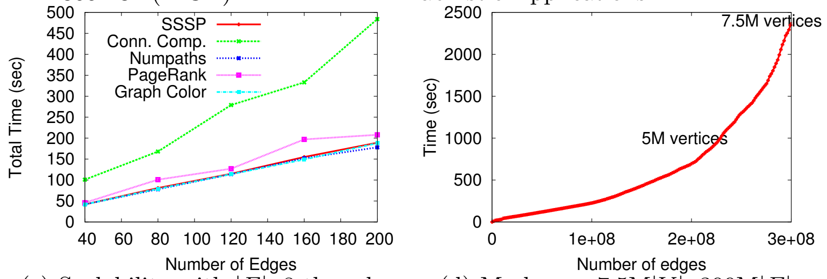
Next, we present data to show that *InfiniMem* scales with increasing parallelism. Figure 8a shows the total running times for various applications on the 14GB rmat-805-134 input: for most applications *InfiniMem* scales well up to 8 threads.

However, given that the performance of applications is determined by the data representation and the number of random accesses that result in disk I/O, we want to study how well *InfiniMem* scales with increasing input size. To objectively study the scalability with increasing number of edges with fixed vertices and controlling for variations in distribution of vertex degrees and other input graph characteristics, we perform a controlled experiment where we resort to synthetic inputs with 4M vertices and 40M, 80M, 120M, 160M and 200M edges. Figure 8c shows the time for each of the for these inputs. We see that with increasing parallelism, *InfiniMem* scales well for increasing number of edges in the graph. This shows that *InfiniMem* effectively manages the limited memory resource by orchestrating seamless offloading to disk as required by the application. The performance on real-world graphs is determined by specific characteristics of the graph like distribution of degrees of the vertices etc. But for a graph of a specified size, Figure 8c can be viewed as a practical upper bound.

Figure 8b illustrates the scalability achievable with programming with *InfiniMem* with parallelism for the Frequency counting, Exact membership query and Probabilistic membership query using Bloom filters. Notice that these applications scale well with increasing number of threads as well as increasing input sizes. The execution time for Bloom filter is significantly larger since Bloom filter generates more random writes, depending on the number of hash functions utilized by the filter; our implementation uses two independent hashes.



(a) Scalability with parallelism for RMAT-805-134 (14GB) (b) Scalability with parallelism for Probabilistic Applications



(c) Scalability with $|E|$; 8 threads. (d) Mesh gen: 7.5M$|V|$, 300M$|E|$.

Fig. 8: Scalability of *InfiniMem* with parallelism and input size.

Figure 8d illustrates that very large graph generation is feasible with *InfiniMem* by showing the generation of a Mesh with 7.5M vertices and 300M edges which takes about 40 minutes (2400 seconds). We observe that up to 5M vertices and 200M edges, the time for generation increases nearly linearly with the number of edges generated after which the generation begins to slow down. This slowdown is not due to the inherent complexity of generating larger graphs: the number of type of disk operations needed to add edges is independent of the size of the graph – edge addition entails adding the vertex as the neighbor's neighbor and accessing the desired data in *InfiniMem* requires a maximum of 2 logical seeks. The reason for the observed slowdown is as follows: modifications of variable sized data structures in *InfiniMem* are appended to the datafile on disk; this data file, therefore, grows very large over time and the disk caching mechanisms begin to get less effective. Compare this with the fact that GTGraph crashed immediately for a graph with just 1M vertices and 400M edges.

5.4 Integration with Distributed Shared Memory (DSM)

Next we demonstrate the applicability of Size Oblivious Programming in the context of Distributed Shared Memory. While clusters are easy to *scale out*, multi-tenant environments can restrict memory available to user processes or certain inputs may not fit in the distributed memory. In either case, it would be beneficial to have the programs run successfully without rewrites. We applied the *InfiniMem* framework to seamlessly make our object based DSM [9] size oblivious. When the data allocated to the node does not fit in available memory, the DSM system spills data to local disk and fetches it back to local memory as demanded by the application. When running distributed software speculation with 75% of the input in memory and the rest spilt to disk, *InfiniMem* has much lower overhead as compared to an alternative solution based upon RocksDB [7]: Figure 9 shows that RocksDB based programs run up to ∼20.5% slower than using *InfiniMem*. Compared to when all the data fits in memory, *InfiniMem* introduces a small overhead of 5% over our baseline DSM, i.e. at this small cost, *InfiniMem* makes our DSM size oblivious.

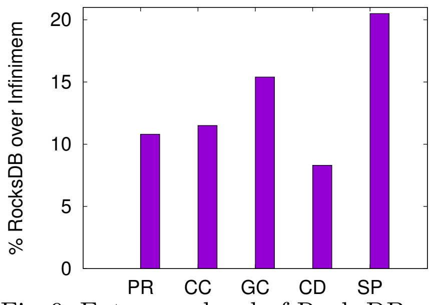


Fig. 9: Extra overhead of RocksDB over *InfiniMem* in our DSM.

6 Related Work

The closest file organization to that used by *InfiniMem* and illustrated in Figure 6 is the B+ tree representation used in database systems. The primary differences in our design are the following: (1) *InfiniMem* uses a flat organization, with at most one level index for variable sized data. (2) *InfiniMem* provides O(1) time I/O operations for random access while the B+ trees require O(log n) time.

Out-of-core Computations– In this paper, we enable applications with very large input data sets to efficiently run on a single multicore machine, with minimal programming effort. The design of the *InfiniMem* transparently enables large datasets become disk-resident while common out-of-core algorithms [5,10,20] *explicitly* do this. As demonstrated with shards, it should be easy to program these techniques with *InfiniMem*.

Processing on a Single Machine– Traditional approaches to large-scale data processing on a single machine involve using machines with very large amounts of memory, while *InfiniMem* does not have that limitation. Examples include Ligra [16], Galois [15], BGL [18], MTGL [3], Spark [21] etc. FlashGraph [6] is a semi-external memory graph processing framework and requires enough memory to hold all the edgelists; *InfiniMem* has no such memory requirements.

GraphChi [11] recently proposed the Parallel Sliding Window model based on *sharded* inputs. Shard format enables a complete subgraph to be loaded in memory, thus avoiding random accesses. GraphChi is designed for and works very well with algorithms that depend on static scheduling. *InfiniMem* is general-purpose and recognizes the need for sequential/batched *and* random input for fixed *and* variable sized data and provides simple APIs for rapid prototyping.

7 Conclusion

We have presented the *InfiniMem* system for enabling *size oblivious programming.* The techniques developed in this paper are incorporated in the versatile general purpose *InfiniMem* library. In addition to various general purpose programs, we also built two more graph processing frameworks on top of *InfiniMem*: (1) with a simple data format and (2) to process GraphChi-style shards. We have shown that *InfiniMem* performance scales well with parallelism, increasing input size and highlight the necessity of concurrent I/O design in a parallel set up. Our experiments show that *InfiniMem* can successfully generate a graph with 7.5 million vertices and 300 million edges (4.5 GB on disk) in 40 minutes and it performs the PageRank computation on an RMAT graph with 134M vertices and 805M edges (14GB on disk) an 8-core machine in about 54 minutes.

References

1. Avery, C.: Giraph: large-scale graph processing infrastruction on hadoop. Proceedings of Hadoop Summit. Santa Clara, USA:[sn] (2011)
2. Bader, D.A., Madduri, K.: Gtgraph: A synthetic graph generator suite. Atlanta, GA, February (2006)
3. Berry, J., Mackey, G.: The multithreaded graph library (2014)
4. Bu, Y., Borkar, V., Xu, G., Carey, M.J.: A bloat-aware design for big data applications. In: Proc. of ISMM 2013. pp. 119–130. ACM (2013)
5. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proc. of SODA '95. pp. 139–149
6. Da Zheng, D.M., Burns, R., Vogelstein, J., Priebe, C.E., Szalay, A.S.: Flashgraph: processing billion-node graphs on an array of commodity ssds
7. Facebook: RocksDB Project, `RocksDB.org`
8. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: OSDI '12. pp. 17–30
9. Koduru, S-C., Vora, K., Gupta, R.: Optimizing Caching DSM for Distributed Software Speculation. In: Proc. Cluster 2015.
10. Kundeti, V.K., et al.: Efficient parallel and out of core algorithms for constructing large bi-directed de bruijn graphs. BMC bioinformatics 11(1), 560 (2010)
11. Kyrola, A., Blelloch, G., Guestrin, C.: Graphchi: Large-scale graph computation on just a pc. In: Proc. of the 10th USENIX Symposium on OSDI. pp. 31–46 (2012)
12. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. arXiv:1006.4990 (2010)
13. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: Proc. of the 2010 ACM SIGMOD ICMD. pp. 135–146. ACM (2010)
14. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. (1999)
15. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., et al.: The tao of parallelism in algorithms. In: ACM SIGPLAN Notices. vol. 46, pp. 12–25 (2011)
16. Shun, J., Blelloch, G.E.: Ligra: a lightweight graph processing framework for shared memory. In: Proc. PPoPP 2013. pp. 135–146. ACM (2013)
17. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: IEEE MSST, 2010. pp. 1–10 (2010)
18. Siek, J., Lee, L., Lumsdaine, A.: The boost graph library (bgl) (2000)
19. Team, T., et al.: Apache mahout project (2014), `https://mahout.apace.org`
20. Toledo, S.: A survey of out-of-core algorithms in numerical linear algebra. External Memory Algorithms and Visualization 50, 161–179 (1999)
21. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proc. HotCloud 2010. pp. 10–10 (2010)