THE UNIVERSITY OF CHICAGO

LIMITATIONS OF DATA REUSE IN STREAMING ITERATIVE ALGORITHMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER’S

DEPARTMENT OF COMPUTER SCIENCE

BY
ERIK BODZSÁR

CHICAGO, ILLINOIS
2013
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABSTRACT</strong></td>
<td>iv</td>
</tr>
<tr>
<td><strong>1 BACKGROUND</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Big data computation systems</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 MapReduce</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 PrIter</td>
<td>5</td>
</tr>
<tr>
<td>1.2.3 Spark</td>
<td>6</td>
</tr>
<tr>
<td>1.2.4 Presto</td>
<td>6</td>
</tr>
<tr>
<td>1.2.5 MadLINQ</td>
<td>7</td>
</tr>
<tr>
<td>1.2.6 GraphLab, GraphChi</td>
<td>7</td>
</tr>
<tr>
<td>1.2.7 Pregel</td>
<td>8</td>
</tr>
<tr>
<td>1.2.8 MPI, OpenMP</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Efficient stable storage utilization and SSDs</td>
<td>8</td>
</tr>
<tr>
<td>1.3.1 Out-of-core algorithms</td>
<td>8</td>
</tr>
<tr>
<td>1.3.2 Solid-state drives</td>
<td>9</td>
</tr>
<tr>
<td>1.3.3 SSD management systems</td>
<td>10</td>
</tr>
<tr>
<td><strong>2 THE PRESTO PROGRAMMING MODEL</strong></td>
<td>12</td>
</tr>
<tr>
<td>2.1 Distributed arrays</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Parallelism</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Examples</td>
<td>14</td>
</tr>
<tr>
<td><strong>3 DESIGN AND IMPLEMENTATION</strong></td>
<td>16</td>
</tr>
<tr>
<td>3.1 Target architecture and assumptions</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Blockus architecture overview</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Master</td>
<td>17</td>
</tr>
<tr>
<td>3.3.1 Scheduler</td>
<td>18</td>
</tr>
<tr>
<td>3.4 Worker</td>
<td>20</td>
</tr>
<tr>
<td>3.4.1 Shared memory system</td>
<td>21</td>
</tr>
<tr>
<td>3.4.2 Task execution</td>
<td>22</td>
</tr>
<tr>
<td>3.5 Executor</td>
<td>22</td>
</tr>
<tr>
<td>3.5.1 Importing and creating shared data</td>
<td>22</td>
</tr>
<tr>
<td>3.5.2 Mitigating the effects of garbage collection</td>
<td>25</td>
</tr>
<tr>
<td>3.6 Matrix serialization</td>
<td>26</td>
</tr>
<tr>
<td>3.7 Sparse matrix storage formats</td>
<td>27</td>
</tr>
<tr>
<td><strong>4 SCHEDULING POLICIES</strong></td>
<td>28</td>
</tr>
<tr>
<td>4.1 Task scheduling</td>
<td>28</td>
</tr>
<tr>
<td>4.2 Performance model</td>
<td>28</td>
</tr>
<tr>
<td>4.3 Scheduling policies</td>
<td>29</td>
</tr>
<tr>
<td>4.3.1 Streaming scheduling</td>
<td>30</td>
</tr>
<tr>
<td>4.3.2 Reversing scheduling</td>
<td>30</td>
</tr>
</tbody>
</table>
ABSTRACT

It is well-known that the MapReduce programming model is not expressive enough for many applications. However, more expressive big data computation systems are in-memory and therefore have limited scalability. We propose a scale up model for big data processing, using SSDs to eliminate the memory limitation and work on data sets bigger than memory.

We explore how parallelism and data reuse can be used by an out-of-core computation system to decrease I/O. We focus on iterative streaming algorithms and how computation reordering (based on knowledge about parallelism) can be used to reduce their total I/O and therefore running time in an out-of-core setting.

We propose two task schedulers that exploit the data reuse of streaming iterative algorithms. One exploits knowledge about memory contents to greedily execute tasks that require the least amount of I/O. The other exploits iterative algorithm structure by reversing execution order in every iteration to maximize data reuse between iterations. We evaluate the proposed schedulers using Blockus, a single-machine system that performs transparent out-of-core computation. Blockus is built on top of Presto, a parallel programming model and distributed execution engine for R.

The proposed schedulers can achieve 20–50% speedups over naive schedulers for simple iterative streaming algorithms on data sets that are 2–5× bigger than memory size. However, the proposed methods do not scale to data sizes orders of magnitude bigger than memory size, because the speedup is roughly inversely proportional to the size of the data set. This means that computation reordering is an ineffective scale-up technique for streaming iterative algorithms; the excess parallelism and data reuse cannot be exploited to scale up to data set sizes that are multiple times bigger than memory.
CHAPTER 1
BACKGROUND

In this chapter, we provide background for the work presented in the thesis. We provide motivation for increasing data capacity by scaling up instead of scaling out, and discuss current solutions for handling big data in computation systems.

We also briefly discuss the most important characteristics of SSDs, current work on their efficient utilization, and why they are easier to use for scaling up than hard disks.

1.1 Motivation

We are living in the age of “big data” [14]: data set sizes are increasing exponentially, and there is high scientific and financial value in these data sets. Scientists, corporations and governments seek to gain insight from large data sets using a wide variety of algorithms. However, many would-be users of big data do not have the expertise or the resources to implement programs that run on the hundreds or thousands of machines that are required to simply store these data sets. Big data computation systems aim to provide easy-to-use big data analytics by providing high-level programming abstractions and taking care of the low-level details (data and computation distribution, fault tolerance, etc.) of managing hundreds of machines.

The prevailing approach of big data computation systems is scale-out: they focus on increasing the number of nodes to increase data capacity. The most well-known and widely used such system is MapReduce [7] (and its open source implementation Hadoop [28]). MapReduce was designed to scale to thousands of machines to process web-scale data. This was achieved by providing a programming model built on the map and reduce primitives known from functional languages. This simple programming model enables the system to execute every step of the computation with stream and sort operations, which can be executed efficiently on hard disks. However, streaming all data through hard disks leads to low
system efficiency [8, 4, 30], and the programming model itself is not expressive enough for many algorithms [15, 16].

Due to the aforementioned limitations of MapReduce, many modifications [8, 4, 30, 3] and many new systems [15, 16, 20, 29] have been proposed for computing over web-scale data. However, these are in-memory systems and therefore do not scale as well as MapReduce. While the amount of RAM in a single server can reach a few terabytes, these systems are prohibitively expensive; most servers have a few tens of gigabytes of RAM. This means that to handle large data sets, these distributed computation systems require many machines. There are two problems with this. First, increasing the number of machines increases cost (which can be initial investment, maintenance or leasing cost, energy, etc.). Second, the probability of failure increases with the number of machines, and these in-memory systems typically recover from failures slower than MapReduce, because most of the live data is only present in memory.

The problem of providing a system that is more expressive and efficient than MapReduce and scales well is clearly unsolved. We try to solve this problem by focusing on scaling up, increasing the data capacity of compute nodes and eliminating the memory limitation. Blockus is a prototype system that seeks to work efficiently and transparently with data that are bigger than memory. To provide a flexible programming model, Blockus is built on top of Presto [27], a distributed computation system designed for matrix computations expressed over matrix blocks.

A successful scale-up system would provide new capabilities on multiple platforms. It would enable big data analysis on laptops and desktop machines [12] and increase the data capacity of small clusters. For large and extreme scale systems, using SSDs instead of additional DRAM can lead to energy savings [26].

Solid-state drives (and similar future storage technologies) provide a new opportunity for out-of-core computation: their low latency (compared to traditional hard disks) and high random-access performance means that we do not have to worry about sequential access in
order to achieve high I/O bandwidth. However, the latency of SSDs is still too high to use them as a drop-in memory extension relying on operating system paging [2, 12].

In order to achieve high system efficiency, a big data computation system relying on SSDs must exploit some application-specific knowledge, for example data access and reuse patterns, or asynchronicity in the computation. The research challenge in building such a system is identifying application characteristics that can be exploited to increase I/O performance (by decreasing I/O and latency and increasing bandwidth) and therefore overall system performance, and designing techniques to exploit them.

1.2 Big data computation systems

This section provides a brief overview of systems used for big data computations. There is a large number of such systems because they all make different trade-offs between expressiveness, ease of use, efficiency and scalability. We organize the systems in order of how much control the user has over data access in an iterative setting: some systems (e.g. MapReduce) process the whole data set in every iteration, some allow computation to be specified over blocks of data (e.g. Presto), and some allow efficient fine-grained data access (e.g. Pregel).

The difficulty of efficient out-of-core computation increases as data access granularity increases. This motivates our choice to study out-of-core computation on top of Presto: it is the next step up from MapReduce in terms of data access flexibility.

1.2.1 MapReduce

MapReduce [7] is the most popular distributed computation system for big data processing. It can scale to thousands of machines and petabytes of data fault-tolerantly and transparently. However, good scaling as achieved at the cost of expressiveness. Even simple algorithms need to be expressed as a sequence of MapReduce jobs. Each job processes the whole input data set, and there is a synchronization point at the end of every job.
Good scaling is achieved (in part) by storing all intermediate data on disks. In order to use disks efficiently, the system can only do sequential I/O, which limits the types of operations that can be done significantly. MapReduce relies on a special programming model to ensure that every I/O operation is sequential.

The programming model is built on two primitives that are well-known in functional programming languages: map and reduce. Map applies some function to all records of a data set, and reduce combines these results to obtain the final result of the computation.

More precisely, every MapReduce computation consists of two stages. The map stage applies some function $f$ to all records. The output of $f$ is a set of key-value pairs. The reduce stage applies a function $g$ to all keys and their associated values. The output of $g$ is also a set of key-value pairs.

The MapReduce runtime consists of many worker processes that execute tasks in parallel. The execution of a MapReduce program consists of the following stages:

1. Map: workers iterate over records of the input data set and apply the user-specified map function to each record, emitting key-value pairs that are stored locally on the worker.

2. Shuffle: each worker is assigned a subset of the keys emitted in the previous stage. The workers obtain the corresponding key-value pairs from other workers, and sort all the obtained values.

3. Reduce: Each worker iterates over its sorted input data, grouping together values that belong to the same key. Emitted key-value pairs are the final result of the computation.

All stages of a MapReduce computation consist of either iterating over data, or sorting. Both of these operations can be implemented efficiently on hard disks, with large sequential data accesses. This means that all input and intermediate data can reside on disk and the system is not limited by main memory size.
The main limitation of MapReduce is its programming model: many algorithms (e.g. graph algorithms, machine learning algorithms) are hard or inefficient to express [16, 15]. Every algorithm has to be expressed as a series of map and reduce steps over the entire data set. This makes MapReduce unsuitable for algorithms that rely on compute dependencies in the data [15]. Iterative algorithms in general are also inefficient because there is no data reuse between iterations; all data has to be written to and read from disks in every map-reduce stage.

There are many research variants that try to increase the efficiency of MapReduce for some specific kinds of computation. Some systems focus on iterative computations, for example by caching [4], eliminating synchronization between iterations [30], or doing incremental updates [3]. However, these systems retain the inexpressive programming model that requires streaming over the whole data set in every iteration.

1.2.2 PrIter

PrIter [31] is a MapReduce variant that focuses on prioritized (or selective) iteration. The programming model consists of iterative map-reduce jobs, where each record also has a priority associated to it. However, computation actually proceeds in subpasses: in each subpass, every compute node executes the map and reduce functions only on its top-\(k\) records, based on priority (\(k\) and the data distribution among nodes are decided by the runtime).

From the point of view of data access, PrIter provides more flexibility than MapReduce: the system accesses only a subset of the data set in every iteration, as defined by the priorities. However the runtime has significant influence over what is actually executed.

In order to utilize prioritized iteration while maintaining correctness, the algorithms used have to be asynchronous, limiting the applicability of PrIter. The good scaling properties of MapReduce are also lost, because PrIter relies on keeping its state (of size proportional to the input data) in memory for efficient execution.
1.2.3 Spark

Spark [29] is an in-memory distributed computation system where computation is expressed over resilient distributed data sets (RDDs) to ensure fast failure recovery. Data are kept in memory to ensure low latency and high performance for interactive computations.

Data sets can be manipulated using coarse-grained transformations (such as maps, filters, reductions, etc.). These transformations are tracked by the system, and in case of a failure the lost data sets can be recomputed from the most recent available ancestors. The dependencies between RDDs are also used for DAG-based execution. The set of available transformations is rich enough that other models of computation (such as those of MapReduce and Pregel) can be expressed in Spark.

Spark allows data access at the RDD level. The programming model can express a wider set of transformations than MapReduce, but there is no way to access data at higher than RDD granularity.

1.2.4 Presto

Presto [27] is a distributed computation system designed for computations on matrices (which can be used to express a large number of algorithms used for big data processing [21, 27]). It is implemented as a distributed execution engine for R [22], an array-based language and environment for statistical computation. R is well known for its ease of use and extensibility (as of 2013, there are more than 4000 packages available), and is used by a large number of researchers.

Distributed data are defined as partitioned matrices (and vectors). Computation is expressed as a sequence of parallel operations on arbitrary subsets of the matrix partitions, allowing the programming greater flexibility than systems that only enable full data set access. There is an implicit synchronization point after each set of parallel operations. Chapter 2 provides a detailed description of the Presto programming model.
1.2.5 MadLINQ

MadLINQ [21] is a matrix-oriented distributed computation system based on the LINQ technology. Operations are expressed over tiles (matrix blocks) and are executed in a distributed, parallel fashion. The system keeps track of block-level data dependencies and uses this information to execute computation as a DAG, avoiding unnecessary synchronization and performing pipelining between sequential operations.

1.2.6 GraphLab, GraphChi

GraphLab [15] is a computation system designed for executing graph and machine learning algorithms. It has a vertex-centric programming model, where some iterative computation is expressed from the point of view of the vertices of a graph. In each iteration, every vertex can modify its neighborhood in the graph by changing the state of adjacent edges and vertices. The vertex update functions are executed asynchronously: the exact execution order is undefined, but there are various execution engines that avoid race conditions (for example by avoiding concurrent updates to neighboring vertices).

GraphLab provides vertex-level access granularity, but the programmer does not have complete flexibility in what data to access. The access pattern is constrained by the graph structure and currently accessed data (i.e. a vertex can only trigger updates of its neighbors), and the runtime also has influence over exactly what data are accessed.

Executing algorithms with fine-grained data access is challenging in an out-of-core system. GraphChi [12] is an out-of-core execution engine for the GraphLab programming model that exploits asynchronicity to increase I/O efficiency. Vertex updates are executed in an order that ensures sequential access to data stored on disk. GraphChi on a single machine can provide running times that are comparable even to distributed systems [12]. However, its applicability is limited to algorithms that can be expressed in the vertex-centric programming model. It also oblivious to data reuse patterns of the executed algorithms, and streams over the input data set in every iteration in a fixed order.
1.2.7 Pregel

Pregel [16] is a distributed computation systems designed with a vertex-centric programming model similar to GraphLab. One of the key differences is that execution in Pregel is bulk synchronous parallel: there is a synchronization at the end of every iteration. Control over data access is similar to GraphLab: the granularity is vertex-level, but the exact access pattern is constrained by the graph structure. However, the graph structure can be mutated during execution.

Because of the fine-grained data access and bulk synchronous execution guarantees, it is extremely challenging to implement an efficient out-of-core execution engine for the Pregel model, and the system is entirely in-memory.

1.2.8 MPI, OpenMP

Systems such as MPI and OpenMP provide low-level parallelization and communication primitives. Due to their generality, they allow very high flexibility in terms of data access granularity and synchronization. This makes it possible to implement highly specialized and optimized algorithms. However, these systems are challenging to program [15] and therefore less widely used than systems that provide higher level abstractions.

1.3 Efficient stable storage utilization and SSDs

1.3.1 Out-of-core algorithms

Out-of-core algorithms are algorithms designed to efficiently operate on data that do not fit in main memory. They are designed to perform large sequential data accesses to ensure efficient I/O on hard disk drives. Such algorithms exist for problems in (typically dense) linear algebra [11, 25], visualization and graphics [6], graph theory [1], and many other fields. Designing such algorithms is difficult and requires deep application knowledge.
Cache-oblivious algorithms [9] are algorithms designed to execute efficiently on architectures with multiple levels of caching, without any knowledge about cache hardware parameters. They generally use a divide-and-conquer approach. Although originally designed for efficient use of CPU caches, these algorithms can be also efficiently executed out-of-core. The design patterns of these algorithms are useful for designing other out-of-core algorithms, but their applicability is limited.

1.3.2 Solid-state drives

The most important advantage of solid-state drives over traditional hard disk drives is the ability to perform small (on the order of kilobytes) I/O operations efficiently. This is a consequence of two important properties of SSDs: low latency and the ability to do multiple operations in parallel.

HDDs consist of a number of platters and a head that reads and writes data on them. For each I/O operation, there is a significant overhead: the platters need to rotate in the correct position and the head needs to be positioned over the right track. All of these are mechanical operations and their speed is limited by mechanical precision. This introduces a latency of about $5-15\, \text{ms}$ for every random I/O operation.

To ensure high efficiency, hard disk I/O needs to be sequential (in other words, all operations should read/write relatively big blocks) in order to amortize the effect of high latency [25]. For example, in hard disk-based storage systems, the I/O block size used is usually at least 8 megabytes [17]. For this reason, writing efficient out-of-core algorithms for hard disks is challenging.

On the other hand, SSDs have no moving parts and have very low latency: typical read and write latencies are $0.05\, \text{ms}$ and $0.25\, \text{ms}$, respectively [10]. They can serve multiple I/O requests in parallel, which can be key to providing good performance for multi-threaded workloads [19]. Their cost per GB is higher than disks’, but their maximum number of I/O operations per second is orders of magnitude higher [13].
We used the \texttt{fio} benchmarking program to measure how random read bandwidth depends on the block size and queue length (degree of parallelism) for our test SSD (OCZ Vertex 4 256GB) and a hard disk drive (WD Green 500GB) in the same machine (see figure 1.1). On the SSD, 64KB reads can utilize about 90% of the full read bandwidth, while the HDD needs to do at least 4MB random reads to get close to its full bandwidth. This means that SSDs are well-suited for workloads that do many small I/O operations, and out-of-core computations running on SSDs do not need to worry about sequential access (as much) to achieve high performance. SSDs are also capable of higher bandwidth; for example, our test SSD has a maximum read bandwidth of about 500MB/s, while commercial hard drives typically have read bandwidths in the 50 – 150MB/s range.

\subsection{SSD management systems}

SSDAlloc

SSDAlloc [2] is a memory management system that uses SSDs as RAM extensions. It provides allocation functions that are very similar to malloc. After replacing memory allocations in a program with these special allocators, the system works transparently, storing data on the SSD and using RAM as a cache.
SSDAlloc can extract almost all of the SSD’s performance by doing caching and I/O at object-level granularity. This can be an order of magnitude faster than using the SSD as operating system swap space because of the relative inefficiency of page-level caching and I/O.

FlashVM

FlashVM [24] is a virtual memory management system for SSDs. It is implemented as an enhancement of the Linux memory subsystem. Among other optimizations, it increases virtual memory paging performance on SSDs by exploiting fast random reads. When the virtual memory system prefetches pages after a page fault, FlashVM is capable of skipping over free/bad pages. It also supports stride prefetching, loading possibly distant pages based on the addresses of the previous two faults (assuming a linear, large-stride traversal of data).
CHAPTER 2
THE PRESTO PROGRAMMING MODEL

In this chapter, we describe the Presto programming model used by the Blockus system.

Presto has two main abstractions. Distributed arrays are used to define data as partitioned (blocked) matrices and vectors, and the foreach function is used to express parallel operations on subsets of the partitions.

2.1 Distributed arrays

The basic data structure of the programming model is the distributed array (darray). A distributed array is a (dense or sparse) matrix partitioned into rectangular sub-matrices (blocks). Note that row- and column-partitioning are also allowed. The user defines the size of the array, the size of the blocks, and whether it is sparse or dense. If the specified block size does not divide the total size evenly, some blocks will be smaller than the specified size (i.e. if a darray has 8 rows and the specified blocks have 3 rows, the last block will only have 2 rows). For example, the R code

\[ A \leftarrow \text{darray}(\text{dim} = c(100, 50), \text{blocks} = c(10, 10), \text{sparse} = \text{FALSE}, \text{data} = 1) \]

creates a distributed array representing a dense matrix with 100 rows and 50 columns, made up of square blocks of size (10,10), and stores a handle to the darray in the variable \( A \). \text{data} is an optional argument that can be used to fill up the darray with an initial value.

Distributed arrays are used for data distribution (and parallelism, see section 2.2) in Presto. In Blockus, the partitioning provided by distributed arrays is used for I/O blocking and parallelism.

Tasks can access single blocks of a darray, or materialize the whole darray as a single array, called a composite array. The blocks of a darray are referred to using the \text{splits} function: \text{splits}(d, i) refers to the block \( i \) of darray \( d \), and \text{splits}(d, i, j) can be used to access blocks of a 2D-partitioned darray conveniently. Calling the function
without index arguments refers to the composite array. \( \text{length}(\text{splits}(x)) \) is a special overload of the \text{length} function in R and returns the total number of blocks in darray \( x \).

2.2 Parallelism

Parallelism is provided by the foreach construct. A foreach loop is a collection of tasks that can be executed in parallel. Each task operates on blocks of distributed arrays. A call to a foreach does not return any data; instead, it can update some of the blocks it is using with the update function. The syntax for the foreach function is the following:

\[
\text{foreach}(\text{index}, \text{range}, \text{function}(x = \text{splits}(a, f(\text{index}))), \\
... \{ \\
\quad \# \text{ computation and updates} \\
\})
\]

For each value in the vector range, a task is created that executes the specified function, with index = value. Not only darray blocks, but any R object (e.g. scalars, vectors, functions) can be supplied as an argument to the function inside the foreach. The end of each foreach is an implicit synchronization point.

For example, the following code implements \( a <- a + b \) on two darrays (assuming that they have the same block structure):

\[
\text{foreach}(i, 1:\text{length}(\text{splits}(a)), \text{function}(x = \text{splits}(a, i), \\
y = \text{splits}(b, i)) \{ \\
x <- x + y \\
\text{update}(x) \\
})
\]

The additions for different pairs of blocks happen in parallel.

Note that a foreach does not have to operate on all blocks of a darray. The range of the index variable can be any arbitrary integer vector, so a foreach can operate on any subset of
blocks of a darray.

Multiple updates to the same block in a foreach result in undefined behavior, so the programmer has to make sure that in a given foreach no two tasks update the same block. Updates are only visible after all tasks of the foreach have finished, and any later references to the block will refer to the latest version.

2.3 Examples

The following code shows a Blockus implementation of the pagerank algorithm (see section 5.3.3):

```r
n <- 685230  # number of vertices
s <- 10000    # number of vertices per partition
G <- darray(c(n, n), c(n, s), sparse=TRUE)  # transition matrix
... # load data

pageranks <- darray(c(1, n), c(1, s), sparse=FALSE, data=1/n)
for (iter in 1:numiterations) {
  foreach(i, 1:numsplits(G),
    pagerank <- function(g = splits(G,i),
      pr = splits(pageranks,i),
      prall = splits(pageranks),
      N = n) {
      # matrix multiplication and vector addition
      pr <- .15/N + .85 * prall %*% g
      update(pr)
    })
}
```

The pagerank vector is stored as a $1 \times n$ darray, and the matrix is column partitioned. In each iteration of the algorithm, we use foreach to perform the pagerank step (a matrix-vector multiplication and a vector addition) in parallel for each partition. Note that `pr = splits(pageranks,i)` refers to a single partition, and `prall = splits(pageranks)` refers to the whole pagerank vector. The `N = n` argument is necessary because of the scopes: variables outside a foreach are not directly accessible from the inside, and variables have to be passed explicitly. The implementation relies on the fact that updates are only
visible after the foreach (so `update(pr)` does not affect the value of the `prall` argument for other tasks).

The following code shows an implementation using 2D matrix partitioning:

```r
n <- 685230  # number of vertices
s <- 10000  # number of vertices per partition
G <- darray(c(n, n), c(s, s), sparse=TRUE)  # transition matrix
numpartitions <- ceiling(n/s)
... # load data

prevpageranks <- darray(c(1, n), c(1, s), sparse=FALSE, data=1/n)
for (iter in 1:numiterations) {
  pageranks <- darray(c(1, n), c(1, s), sparse=FALSE, data=1/n)
  for (row in 1:numpartitions) {
    foreach(i, 1:numpartitions,
      pagerank <- function(g = splits(G,row,i),
        pr = splits(pageranks,i),
        prprev = splits(prevpageranks,row)) {
          # matrix multiplication and vector addition
          pr <- pr + .85 * prprev %*% g
          update(pr)
      })
    prevpageranks <- pageranks
  }
}
```

This implementation is more complicated (an extra `darray` and a for loop) than the one using row partitioning. The reason for this is the restriction of the programming model that requires that every partition is only updated at most once in every foreach. Any partition $i$ of the pagerank vector depends on partitions $(j,i)$ $1 \leq j \leq p$. This means that we need $p$ separate foreshades to perform all the updates without conflicts.
CHAPTER 3
DESIGN AND IMPLEMENTATION

In this chapter, we describe the architecture of the Blockus system and its components, and present some implementation details.

Since Blockus is built on top of Presto, the two systems are very similar. The main difference is that Presto adds big data support to R by scaling out, while Blockus focuses on scaling up. Currently, Blockus only supports single-machine execution. As in the case of Presto, one of the design goals is to not modify R itself. Blockus is implemented as an R package that can be used by unmodified R installations.

3.1 Target architecture and assumptions

Blockus is designed for a single-machine architecture. The components that are relevant are the CPU cores, main memory and stable storage.

Main memory size is assumed to be limited (i.e. smaller than the working sets of the programs being run by the system), but storage space is assumed to be abundant.

I/O operations are assumed to require no CPU resources, and to overlap perfectly with computation. Other levels of the memory hierarchy (such as CPU caches) are ignored.
3.2 Blockus architecture overview

The single-machine Blockus system consists of three different kinds of processes (see figure 3.2):

- a master process, which executes a Blockus program and takes care of data and task distribution and scheduling;
- a worker process, which acts as a unified IO and communication layer for the executors;
- a number of executor processes, which execute the tasks defined by the foreach calls of the Blockus program.

![Diagram of Blockus system processes](image)

Figure 3.2: Processes of the Blockus system and their main components.

Note that this is similar to the Presto architecture. The only difference is that since Presto is distributed, there is one worker per physical machine, and each worker has its own set of executors.

3.3 Master

The master is an R process that is responsible for reading a Blockus program and coordinating its execution. The only difference compared to a normal R process is that an extra
package is loaded to provide support for the Presto primitives (darray, foreach).

3.3.1 Scheduler

The scheduler keeps track of the state (configuration, block metadata, ongoing operations) of the Blockus system. Based on this information, it makes decisions about I/O and computation scheduling. These decisions can have a significant effect on system performance through optimization techniques such as computation reordering, prefetching, caching, etc.

When a foreach statement is encountered at the master, it is submitted to the scheduler as a set of tasks. Each task consists of a list of its input arguments (handles for darray blocks and serialized values for other R objects), and the function to be executed.

The scheduler maintains all information necessary for scheduling decisions:

- system configuration (number of executors, memory limit),
- currently scheduled and currently executing tasks,
- block sizes and locations,
- ongoing IO operations.

The scheduler is event-driven. Each command to the worker has a unique task ID. When the worker is done executing a command, it sends a notification to the scheduler that contains the ID of the corresponding task. In the case of foreach tasks, the notification also contains the details (name, size, location) of newly created data blocks.

Locking

Because data in Presto (and other distributed in-memory systems) are always present in memory, and are only deleted through garbage collection, the Presto scheduler is relatively simple: for each task, first it needs to fetch any non-local input data from remote locations, and then it can safely execute the task.
The Blockus scheduler is significantly more complicated because of a fundamental difference: memory is no longer assumed to be abundant. As a consequence, data are no longer always in memory, and the scheduler has to avoid multiple failure scenarios, such as deleting a block from memory while it is being used in a computation or I/O operation, over-allocating memory, etc.

To solve these problems, the scheduler uses the concept of locking. A block being locked means that its in-memory space is protected from flushing: the space associated with the lock can only be freed after the block is unlocked. Note that this is different from locking virtual memory with the `mlock` system call, and it is only an implementation technique internal to the scheduler. A block can be locked even before its contents are physically present in memory: if a block is only present on disk, the system needs to allocate space for it in memory, and then lock it to make sure that the allocated space remains available while the I/O operation is in progress.

The issue of locking also makes scheduling tasks more subtle than in the case of Presto. In Presto, since memory is abundant, once a task is assigned to a worker node, the system can begin fetching all necessary input data, and the task can be executed once all data arrive. In Blockus, a task can have 3 states: waiting, scheduled, and executing. A task is initially in the `waiting` state when it is submitted to the scheduler. When it gets `scheduled`, the scheduler locks its input blocks, and starts loading any missing blocks. After all missing input blocks are loaded, the task is `executed`.

**Basic scheduler**

The most important branch of the scheduler’s event handler is the one that handles task completion, because this is the branch that does task (and consequently, I/O) scheduling. This branch is described in pseudocode in algorithm 1.

Note that the final loading operation might not be able to load all necessary blocks, if there is not enough space and other blocks need to be flushed. In this case, these loads will
Algorithm 1 Task completion event handling

\[ T: \text{completed task} \]
\[ M_L: \text{total size of locked (but not necessarily present) blocks} \]
\[ M_U: \text{total size of unlocked blocks present in memory} \]
\[ M: \text{memory limit} \]

Unlock all blocks of \( T \)
Find a task \( T' \) to schedule, such that after locking input blocks of \( T' \), \( M_L < M \)
Lock all inputs of \( T' \)
\[ \text{if } M_L + M_U \geq M \text{ then} \]
\[ \quad \text{Flush at least } M - M_U \text{ worth of (unlocked) blocks} \]
\[ \text{end if} \]
\[ \text{if Some input blocks are missing then} \]
\[ \quad \text{Try loading missing blocks} \]
\[ \text{else} \]
\[ \quad \text{Execute } T' \]
\[ \text{end if} \]

be retried in the event handler for flush completions. If there were missing blocks, the task will be executed from the event handler for load completions when the final block is loaded.

Prefetching is implicit in the scheduler: required blocks are loaded as soon as there is available memory, regardless of how many tasks are executing at the time.

Deciding which task to schedule and which blocks to flush (if necessary) can be controlled by custom scheduling policies to optimize execution. Blockus currently uses the least recently used block eviction policy. Sections 4.3 describes the implemented task scheduling policies.

3.4 Worker

The worker process provides a unified I/O and communication layer for the executors. The worker is responsible for task execution (by forwarding it to an executor), darray block I/O and delete operations, and composite array creation.

To ensure that computation and I/O can always overlap, multiple different thread pools are used to process incoming requests. For example, if the thread pool responsible for I/O is exhausted, it does not affect the worker’s ability to execute tasks, because they are served
by threads from a different pool.

### 3.4.1 Shared memory system

Because of the single-threaded implementation of R, a process can only have a single R environment running in it. This means that the worker and the executors are all separate processes that do not share a virtual address space. To avoid having to copy data between the different processes (and avoid having to keep track of what data are present in which processes), all in-memory darray data is stored in shared memory that is mapped into the virtual address space of the worker and all executors.

UNIX shared memory objects are identified by their name, and the objects are stored as files on a special RAM disk. Opening shared memory objects yields a file descriptor that can be used to `mmap` the contents, just like with a file descriptor that belongs to a regular file.

In early implementations, each darray block was a separate shared memory object that was opened and mapped into the executor virtual address spaces. However, this increased task overheads because of the necessary open, close and `mmap` calls for each task. To avoid this, the worker allocates a single, large shared memory objects at system startup, which is mapped into each executor’s space at the beginning. Storage for darray blocks is allocated from this pre-allocated shared memory space, and for each task to be executed the worker only sends the offsets of the necessary darray blocks, eliminating all expensive open, close and `mmap` system calls from the process. The worker maintains a lookup table that maps the names of the in-memory blocks to their locations in the shared memory block. When the allocated shared memory is full, the scheduler is responsible for choosing a block to flush.

Not all available main memory is allocated as shared memory. The fluctuations in the memory usage of R instances can be quite large, so some memory has to remain unreserved to accommodate temporary objects created by R. As a rule of thumb, we leave 500MB of free space for each R instance (the exact memory usage depends on the workload and block
3.4.2 Task execution

At startup, the worker starts a set of executor processes (using traditional fork-exec). These processes form a (constant sized) pool similar to a thread pool. When the worker has to execute a task, it simply waits for an executor to become available, and sends the task to it. When the executor is done with the task, it notifies the worker and sends back information about newly created blocks.

All communication between the executors and the worker is done through UNIX pipes.

3.5 Executor

Executors are responsible for executing the individual tasks of a foreach call. Each executor has an embedded R instance that handles R code execution. At startup, the R instance is initialized and necessary R packages are loaded. Then the executor goes into a loop, waiting for tasks to execute.

Each task is encoded as a set of input variables and a function. The input variables are \((name, value)\) pairs, where \(name\) is the name of the variable in the R environment, and \(value\) is a serialized R object for regular R object arguments, and a shared memory offset for darray blocks (see section 3.4.1). The function is passed as a string, and is parsed into a language object by the R environment. Because each executor is expected to run many tasks from the same foreach, all input variables and the task function are cached.

3.5.1 Importing and creating shared data

As mentioned in section 3.4.1, all darray blocks are stored in shared memory. We want to use blocks transparently, in their native R matrix representations, and avoid adding an extra interface that manipulates matrices residing in shared memory. However, because of the way
R objects are laid out in memory, this is not straightforward to do.

Every R object consists of two parts: a fixed-size header at the beginning, and a variable-size data part following the header. The header contains data specific to the R instance (for example, garbage collection information), and information about the object itself (for example, the length of the data part). The data part is a C array of characters, integers or doubles, depending on the R object. Variables are passed between R and C code as pointers to these objects.

Because multiple executors might use the same block at the same time, we cannot simply include the R header in the shared memory representation of a block and then use a pointer to this structure as an R variable. The different R instances would write their own instance-specific data in the header, leading to header corruption. However, for large objects we would also like to avoid creating a local R object and then copying the block contents into it from shared memory; this would increase memory footprint and running time.

The system uses memory mapping (using the \texttt{mmap} system call) to avoid R header corruption and data copying. Note that because \texttt{mmap} works by manipulating the page table, it puts some restrictions on the addresses involved in this process. More precisely, the base address of the mapping has to be at the beginning of a page.

For each block of size $S$ used by the executor, a block of virtual memory of size $S + \text{pagesize}$ is allocated. The R header is placed at the end of the first page (allocating one block of physical memory in the process), and the data part is mapped from shared memory using \texttt{mmap} (see figure 3.3). This way, each R instance has its own private header for the object, but the data part is shared efficiently. Only one page of physical memory is used (and $S + \text{pagesize}$ bytes worth of virtual memory pages, but these are abundant on a 64-bit system). Objects in R are read-only (write-once), so there are no race conditions with the shared data.

R has its own object allocation functions that have to be used to allocate vectors and matrices. These allocation functions allocate necessary space using malloc, set up the object
Figure 3.3: Virtual memory layout of R objects containing shared data.

header, and register it in the garbage collector. Instead of modifying R to use the new allocation scheme described here, the calls to malloc are intercepted using the \texttt{\_malloc\_hook} functionality of glibc. Note that this is only necessary at the beginning of each task, when the system initializes the input blocks. When R tries to allocate space for $S$ bytes for an input block, the special malloc override function allocates $S + \text{pagesize}$ bytes instead, and returns a pointer to where the header needs to be placed. This ensures that the alignment restrictions for \texttt{mmap} are satisfied. The behavior of the free function is also overridden to make sure that these specially allocated objects are freed safely. After the input arguments of the task are initialized, the glibc \texttt{malloc} is restored (to be used by R for allocating temporary variables during task execution).

Newly created blocks are not immediately allocated in shared memory. They are simply allocated by the (unmodified) allocation functions of R. The Blockus system has no way to tell whether some new R object will end up being a block of a darray, or simply a temporary R object. Once the \texttt{update} function (see section 2.2) is called on an object, it is copied into shared memory.

If there is not enough free space in shared memory to accommodate the new object, it is copied immediately to disk (and the task completion message sent to the scheduler will reflect this). Another possible solution is to notify the scheduler that there is not enough free shared memory for the new object, and then wait until something is flushed and enough
shared memory is available. We chose the former solution due to its simplicity.

### 3.5.2 Mitigating the effects of garbage collection

R is a garbage collected language. This can have a negative impact on running time if the system has to run a large number of tasks. Each task has input variables (and presumably some temporary variables) that are only used in that task. This means that many short-lived objects are allocated in the R instances of the executors, which can trigger many garbage collections, introducing a significant overhead. Also, because most inputs are mmap-ed from shared memory, the memory usage is actually lower than what the R garbage collector thinks.

Let $V$ denote the R heap size. A garbage collection is triggered when the total memory used by R objects is greater than $V$. The garbage collector keeps track of three generations of objects. First, it tries to collect objects in the first generation; if memory usage is still greater than $V$, it tries collecting objects from the second generation, etc. Collecting from later generations is more expensive. Let $S'$ denote the total used memory size after garbage collection (of as many generations as necessary). If $S' \leq 0.3V$, then $V$ is decreased to $0.8V$. If $S' \geq 0.7V$, then $V$ is increased to $\max(S', 1.05V)$.

Because the default value for $V$ is small, temporary variables in each task are likely to cause $V$ to be raised. However, these variables quickly become unused, and a future garbage collection will collect them and reclaim enough space to decrease $V$ close to its original value. $V$ will never increase beyond a certain limit and frequent garbage collections will be triggered.

To increase $V$ permanently, we use a technique similar to the one used for mmap-ing blocks (described in section 3.5.1). We call an R allocation function to allocate an object of size $S_0$, but intercept the underlying call to `malloc` and do not allocate $S_0$ bytes of memory. However, the R garbage collector thinks that an object of size $S_0$ was allocated. $V$ will keep getting increased until $V \geq \frac{S_0}{0.7}$. Garbage collections will be triggered when the total size $S$ of new objects is at least $0.3V$, which is at least $\frac{0.3S_0}{0.7} \approx 0.43S_0$. For example,
if we want the garbage collection limit for actual objects \((S)\) to be at least 200\(MB\), then we need to allocate a dummy object of size \(S_0 = \frac{S}{0.43} \approx 465MB\).

Using this garbage collection mitigation technique decreases time spent in garbage collection significantly. Table 3.1 shows the effects on one run of the K-means algorithm on a 32\(GB\) data set. Overall running time is decreased by 25\%, and relative time spent doing garbage collection is reduced from 36\% to 4\%. The total number of garbage collections is reduced by a factor of 4, but more importantly, the number of level 2 garbage collections (when all generations are tried to be collected) is reduced by a factor of 10.

<table>
<thead>
<tr>
<th></th>
<th>Execution time (s)</th>
<th>Garbage collection time (s)</th>
<th>Number of garbage collections</th>
<th>Number of level 2 garbage collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without GC mitigation</td>
<td>778</td>
<td>283</td>
<td>8517</td>
<td>2607</td>
</tr>
<tr>
<td>With GC mitigation</td>
<td>587</td>
<td>24</td>
<td>2078</td>
<td>242</td>
</tr>
<tr>
<td>Relative reduction</td>
<td>24%</td>
<td>91%</td>
<td>75%</td>
<td>90%</td>
</tr>
</tbody>
</table>

Table 3.1: Effects of garbage collection mitigation on the K-means algorithm on a 32\(GB\) data set.

### 3.6 Matrix serialization

R stores vectors and dense matrices as simple R vector objects: the R header contains information about the dimensions, and the contents are stored in the data part as a (column-major ordered) C array.

The representation of sparse matrices is more complicated. The R object representing a sparse matrix contains pointers to 3 other R objects, which are vectors describing the contents of the matrix, depending on the format (the C++ analogy of this is a class containing 3 pointers to arrays). The two most commonly used formats are the compressed sparse column and coordinate format [23] (see section 3.7).

Vectors and dense matrices are stored in shared memory in a straightforward manner: an initial page contains dimension information, and following pages store the contents as a
regular C array. Sparse matrices are stored similarly: an initial page stores information about
the dimension, encoding (compressed sparse column or coordinate format), and number of
nonzeros of the object. The following pages are C arrays storing the 3 vectors encoding the
contents of the matrix. Note that each vector has to start at a page boundary to ensure that
the mapping described in section 3.5.1 can be done. When the sparse matrix is loaded into
an executor from shared memory, 3 mappings need to be done to construct the object.

### 3.7 Sparse matrix storage formats

The default sparse matrix storage format in R is the compressed sparse column (CSC)
format. A sparse matrix is represented using 3 vectors, commonly denoted by $i$, $p$ and $x$.
For a matrix with $c$ columns and $n$ nonzeros, $i$ and $x$ have length $n$ and contain the row-
indices and values of the nonzero matrix elements (in column-major order), respectively. $p$
has length $c + 1$, and $p(i)$ is the index (in $i$ and $x$) of the first element after column $i - 1$. 
The last element of $p$ is defined to be $n$. Assuming that the elements of the three vectors
have equal size, the storage size of a matrix in this format is $c + 2n$.

The other sparse matrix format in R that is used by Blockus is the coordinate format.
The encoding vectors are $i$, $j$, and $x$. They simply store the row- and column-indices and
values of the non-zero entries, respectively. The values do not have to be in column-major
order. The storage size of this format is $3n$.

For $c < n$ (which is true if we assume that the matrix does not have any empty columns),
CSC format leads to better compression. It also supports random access to the contents of a
specific column. However, there is one common case in Blockus when the coordinate format
can lead to significantly smaller storage requirements. If a sparse matrix is partitioned into
2D blocks, such that each column contains $k$ blocks, the total space used by the blocks is
still only $3n$ using coordinate format, but it is $kc + 2n$ using CSC format. Therefore, if
$k \gg \frac{n}{c}$, CSC format will use significantly more space, and (in most cases) the blocks of a
2D-partitioned matrix should be encoded in the coordinate format.
CHAPTER 4
SCHEDULING POLICIES

This chapter describes the problem of scheduling tasks of a foreach. It presents a cost model that expresses the running time of a foreach as a function of the scheduling policy. Four schedulers are described and their costs (according to the presented model) are calculated. Table 4.1 contains a summary of the schedulers and their performance according to the model.

4.1 Task scheduling

In systems with out-of-core computation capability (such as MapReduce [7], GraphChi [12], or operating system paging), tasks are executed in some pre-determined order and I/O is done on-demand, as a result of trying to execute the scheduled tasks.

Blockus allows implementations of task scheduling policies that are I/O aware: tasks can be scheduled in a way that tries to minimize I/O and exploit data reuse. More precisely, every foreach is submitted to the scheduler as a set of tasks. Every task consists of the function to be executed, and a list of its input blocks. Tasks in the same foreach can be executed in any order, and the goal of the scheduler is to find an order that results in the lowest execution time. The scheduler has no knowledge (beyond speculation) of future foreaches. Therefore, the task scheduling problem is always constrained to tasks in a single foreach. However, the system could use knowledge obtained from previous foreaches.

4.2 Performance model

We assume that all schedulers are computation preserving: because the executors are stateless, the order of the tasks does not affect the total computation time. On the other hand, the order of the tasks has an effect on the I/O performed during execution. The system can save I/O by executing tasks that already have some of their inputs in memory.
To build a simple performance model for foreach execution, assume that the rate of computation (i.e. how many bytes of input data are processed per unit time) and the rate of I/O are constant. Let \( R_C \) and \( R_I \) denote these values, respectively. Let \( D \) denote the total input data size, and let \( M \) denote the main memory size. Let \( I_S(M, D) \) denote the total I/O done in the foreach by scheduling policy \( S \) on a data set of size \( D \) and a machine with main memory size \( M \). Let \( T_C, T_I, T_E \) denote the computation, I/O and execution time of the foreach, respectively. Note that \( R_C \) and \( I_S(M, D) \) also depend on the algorithm, this is not noted for the sake of simplicity. In order to simplify expressions, we will use variables \( \alpha = \frac{M}{D} \) and \( \beta = \frac{R_I}{R_C} \).

It is easy to see that \( \max(T_C, T_I) \leq T_E \leq T_C + T_I (\leq 2 \max(T_C, T_I)) \), and \( \max(T_C, T_I) \approx T_E \) if computation and I/O almost completely overlap. For the sake of simplicity we will assume that computation and I/O overlap completely.

Computation time only depends on the algorithm and the input data. Let us assume that the algorithm itself is linear and execution also scales linearly: \( T_C = \frac{D}{R_C} \) (in reality, the scaling is most likely worse than linear because of overheads).

We assume that I/O time only depends on I/O size and I/O bandwidth: \( T_I = \frac{I_S(M, D)}{R_I} \).

Schedulers can decrease execution time by decreasing the amount of I/O during execution.

### 4.3 Scheduling policies

In this section, we describe the implemented schedulers and calculate their I/O cost according to the model using \( D \) (data set size) and \( M \) (memory size). Table 4.1 contains a summary of the schedulers and their performance according to the model.

With an input size of \( D \) and memory size of \( M \), the size of data not present in memory is at least \( D - M \). All of these data have to be read at least once, so the lower bound on I/O is \( D - M \).

For the sake of simplicity, we assume that all contents of the main memory are reusable in the current foreach. In realistic scenarios this is usually not the case: some memory
is usually unused because it was recently used for temporary storage, and some blocks in memory are not needed in the current foreach. Because of these, it is more accurate to think of $M$ as the “total reusable in-memory data size”.

We also assume that for streaming iterative algorithms, an iteration consists of a single foreach. For the algorithms studied in this thesis, a single foreach dominates the running time of every iteration; this assumption simply ignores the other foreaches.

4.3.1 Streaming scheduling

The simplest thing to do is to execute the tasks in the order defined by the index variable of the foreach. This corresponds to executing the specified tasks in a regular for loop, iterating over the data set in some fixed order. This schedule is oblivious to the underlying I/O operations. It is sufficient for in-memory systems, and the Presto scheduler uses it. GraphChi [12] also executes its operations similarly, in some pre-determined order, because it relies on sequential iteration to provide high disk throughput.

For streaming iterative algorithms, this policy does not provide any data reuse between iterations, because the previous iteration ended with tasks using blocks at the end of the data set, and the current iteration starts with blocks at the beginning. This means that the full data set has to be read from stable storage in every iteration: $I_{\text{streaming}}(M, D) = D$.

4.3.2 Reversing scheduling

Many big data algorithms are streaming iterative (i.e. they process the whole data set in every iteration). The aforementioned streaming scheduling is not I/O-aware and ignores the fact that some there might be some useful data left in memory from the previous iteration.

A simple approach to exploit this is to execute the tasks of the second iteration in reverse order. This way, execution starts with the last tasks, whose input data are possibly still in memory from the end of the previous iteration. The third iteration should execute tasks in the natural order, to reuse some blocks from the end of the previous, “reversed” iteration,
etc. In general, odd iterations should be executed in order, and even operations in reverse order.

In Blockus we can apply the above heuristic to for eaches. We assume that if the algorithm uses some blocks of a darray in a foreach, it uses all of them, and the index variable goes from 1 to $n$. For each darray $d$, we maintain a boolean $o(d)$ that stores whether the darray was last accessed in reverse order. When a new foreach is executed, the scheduler has to decide whether to execute the tasks in normal or reverse order. Let $s(d)$ denote the total size of darray $d$. Then the order of task execution is determined by $r = \arg\min_{r \in \{\text{true, false}\}} \sum_d (o(d) == r) s(d)$ (the tasks are executed in reverse order if $r$ is true). This means that the scheduler is trying to maximize the size of the darrays that are processed in the opposite order as they were the previous time.

The biggest disadvantage of reversing scheduling is that it is ineffective for algorithms with more complex data access patterns (i.e. algorithms that access the blocks of a darray in non-sequential order, or only access some blocks).

Assuming that the heuristic task order reversal is successful (i.e. the input data were accessed in reverse order during their previous use), all in-memory data can be reused, and the I/O done by the reversing scheduler is $I_{\text{reversing}}(M, D) = D - M$, which is the lower bound on I/O.

4.3.3 Random scheduling

The previous two scheduling policies relied on some pre-determined ordering of tasks. However, conceptually there should not be any ordering of the tasks of a foreach. One way to capture this is to execute the tasks in a random order.

Random scheduling is easy to implement and execute. Data reuse depends on chance. Tasks that have a chance to reuse data from the previous iteration are the ones scheduled in the beginning of execution.

The expected amount of data reused can be approximated by considering the reuse
probability of each block separately (and then using the additive property of expected value) and assuming that all blocks have the same size. Let $b_1, \ldots, b_M$ denote the $M$ blocks that are present in memory from the previous iteration, in order of access time (i.e. $b_1$ is the first block to be flushed). Let $D$ denote the total number of blocks. Let $s(i)$ denote the index of the task that uses $b_i$ (for example if $s(i) = 1$, then $b_i$ is used by the task that is executed first).

$$
P(b_i \text{ is reused}) = P(b_i \text{ is reused} | s(i) \leq i)P(s(i) \leq i) + P(b_i \text{ is reused} | s(i) > i)P(s(i) > i)
$$

Because of the random scheduling, $P(s(i) \leq x) = \frac{x}{D}$ and $P(s(i) > x) = 1 - \frac{x}{D}$. $P(b_i \text{ is reused} | s(i) \leq i) = 1$ because even if all of the first $s(i) - 1$ tasks result in a block eviction, $b_i$ is still not evicted before it is used. On the other hand, $P(b_i \text{ is reused} | s(i) > i) \leq P(s(i) \leq 2M | s(i) > i)$, because a block can only be reused in the first $2M$ tasks; after $2M$ tasks, there are at least $M$ block evictions, meaning that all blocks from the previous iterations have been evicted. Using these, we can establish the following bounds for $P(b_i \text{ is reused})$:

$$
\frac{i}{D} \leq P(b_i \text{ is reused}) \leq \frac{i}{D} + \frac{2M - i}{D} = \frac{2M}{D}.
$$

These lead to the lower and upper bounds for the expected number of blocks reused:

$$
E \geq \sum_{i=1}^{M} \frac{i}{D} = \frac{(M)}{2D} \geq \frac{M^2}{2D}
$$

$$
E \leq \sum_{i=1}^{M} \frac{2M}{D} = \frac{2M^2}{D}.
$$

For the sake of simplicity, we will use the approximation $E(I_{\text{random}}(M, D)) = \frac{M^2}{D}$. 
4.3.4 Greedy scheduling

Greedy scheduling aims to minimize I/O greedily, on a per-task basis. When a task has to be scheduled for execution, the one that requires the least amount of I/O is chosen (tasks that do not require any I/O are always chosen first).

This is possible to do in Blockus because the scheduler keeps track of all blocks sizes and locations. However, the implementation is more complex than for other policies: the tasks have to be stored in a min-heap that has to be modified on every block I/O.

The main drawbacks of greedy scheduling are its complexity, and its reliance on full block residence knowledge. This can limit its applicability in other systems (or in future, distributed versions of Blockus), where this information is not available in a central scheduler.

Because the greedy scheduler can reuse all in-memory data, the I/O done is $I \text{_{greedy}}(M, D) = D - M$, which is the lower bound on I/O.

4.4 Performance of schedulers according to the model

In order to simplify expressions, let us use the variables $\alpha = \frac{M}{D}$ and $\beta = \frac{R_I}{R_C}$. $\alpha$ and $\beta$ can be thought of as variables measuring the memory- and I/O-limitations of the systems, respectively. Assuming an I/O limited system, $0 < \alpha, \beta < 1$, and bigger $\alpha$ and $\beta$ are “better” (i.e. more of the data set fits in main memory, and the system is less I/O limited).

The system is not I/O limited if $T_C \geq T_I$, which is equivalent to $\frac{D}{R_C} \geq \frac{I_S(M, D)}{R_I}$. A different way to formulate this is to consider the maximal size of a data set that can be processed without having to wait for I/O. With streaming scheduling, the data set has to fit completely in memory, but the other policies can process larger-than-memory data sets without being I/O limited. The maximal size (relative to main memory size) is $\frac{1}{\alpha}$, which can be obtained by solving the above inequality.

In the I/O limited case ($T_C < T_I$), execution time is equal to I/O time, which is $\frac{I_S(M, D)}{R_I}$. The I/O times can be expressed using only $\alpha$ and $\beta$ if we consider them relative to the I/O
time of the streaming scheduler.

Table 4.1 contains a summary of the schedulers and when they are compute or I/O limited.

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Task execution order</th>
<th>Compute limited when</th>
<th>Maximal compute limited data size (relative to $M$)</th>
<th>I/O limited relative execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming</td>
<td>Fixed order defined by index variable</td>
<td>$\alpha \geq 1$ or $\beta \geq 1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Random</td>
<td>Random</td>
<td>$\beta \geq 1 - \alpha^2$</td>
<td>$\sqrt{\frac{1}{1-\beta}}$</td>
<td>$1 - \alpha^2$</td>
</tr>
<tr>
<td>Reversing</td>
<td>Reverse of previous order</td>
<td>$\beta \geq 1 - \alpha$</td>
<td>$\frac{1}{1-\beta}$</td>
<td>$1 - \alpha$</td>
</tr>
<tr>
<td>Greedy</td>
<td>Greedily minimize required I/O</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Schedulers and their performance according to the model.
CHAPTER 5
EVALUATION

In this chapter, we evaluate the performance of the schedulers described in chapter 4 on a set of streaming iterative workloads. In order to do this, we describe the experimental systems, methodology and workloads. To understand experimental results on the workloads, we also measure system overheads using microbenchmarks.

5.1 Experiment setup

For all benchmarks except k-means, we use a machine with a 2.9GHz quad-core AMD A8-3850 processor, 16GB main memory and an OCZ Vertex 4 256GB SSD (with a maximum read bandwidth of about 500MB/s). For all experiments, memory size is limited to 8GB. For the k-means benchmark, we use a machine with two 2.6GHz six-core Intel Xeon X5650 processors, with 96GB main memory and an OCZ Deneva 2 128GB SSD (with a maximum read bandwidth of about 230MB/s). During experiments, memory size is limited to 16GB.

To simulate a memory size smaller than the physical memory size of the machine, we use a simple C program that allocates and locks (using the mlock call) the amount of memory that we do not want to use.

Swap partitions are turned off in all experiments. This would not be recommended during real use of the system, because the memory usage of the executors is unpredictable (as it depends on user code and R behavior), and a big spike in executor memory usage can make the (operating) system run out of memory and cause Blockus to be killed. However, the experiments are easier to understand if there is no swapping done by the operating system.

CPU usage measurements are done using the mpstat tool, which reports average CPU utilization percentages over 2 second (configurable) intervals. I/O-related measurements are done using the iostat tool, which reports averages of various I/O metrics over 2 second (configurable) intervals. iostat can also report the total number of bytes written and read.
(for a given disk) since system startup. Using this, the total number of bytes written and read through a given period can also be obtained (by subtracting the totals at the beginning from the totals at the end).

Computation times are obtained from executor logs. Each executor measures the time spent in the R environment executing each task (using the `gettimeofday` function). To obtain the computation time for a benchmark, we simply sum up the execution times in the log (starting from the execution of the first task of the algorithm to ignore time spent loading and initializing data). Because the executors run in parallel, the log of the first executor is used to estimate the wall clock time required to do the parallel computation.

## 5.2 Microbenchmarks

### 5.2.1 Task overheads

We measure the total system overhead per task by running `foreaches` that execute an empty function with no input arguments. This overhead is made up of many components: parsing the input arguments of the function and evaluating them for all values of the index variable, submitting tasks to the scheduler, scheduling, master ↔ worker ↔ executor communication, worker overhead (finding a free executor and submitting the task, and sending the results back to the master), executor overhead (parsing function and input arguments, calling R code, processing results). Figure 5.1 shows that the total overhead scales linearly with the number of tasks, and so it makes sense to talk of a per-task overhead, which is approximately 0.58\( ms\) with a single executor. The parallelism provided by multiple executors can reduce this overhead significantly, for example using 4 executors reduces the overhead per task to 0.16\( ms\).
Section 3.5.1 describes how Blockus uses mmap to initialize task input variables without copying. We use foreaches of 1000 tasks with different numbers of input variables to measure the variable initialization overhead with and without using the mmap technique described in section 3.5.1. Figure 5.2a shows that the overhead of mmap is bigger: creating an 8-byte variable using mmap takes approximately 0.12\textit{ms}, while creating it by copying data takes 0.04\textit{ms}. However, as the size of the variable increases, the overhead of the operation is dominated by the amount of time it takes to copy the data. Figure 5.2b shows that using mmap, variable creation time does not increase as the size of the variable increases. On the other hand, the time it takes to initialize using copying is proportional to the size of the variable. Even at a few kilobytes, the difference is negligible compared to the system overhead per task.
5.2.3 I/O performance

To measure the total overhead of I/O in the system, we run two sets of experiments. The first one consists of foreaches with an empty function with input blocks of size 1MB, running entirely in memory. The second set has the same foreaches, but with an artificial memory limit of 10MB, triggering I/O decisions in the scheduler and I/O operations in the worker. However, the I/O operations are not executed at the device level (thus the experiment does not measure the drive overhead). Figure 5.3a shows the differences between running times of the two sets using 1 and 4 executors. With a single executor, the average I/O overhead per task is about 0.6ms. However, the overhead decreases to 0.15ms with 4 executors; although the executors are not involved in the I/O operations, the empty tasks are executed faster, thus the blocks can be moved in and out of memory at a higher rate, and these I/O operations can also happen simultaneously.

To measure how overheads affect the I/O (read) bandwidth the system can achieve, we run foreaches of empty functions, varying the size of input blocks. Figure 5.3b shows that block size and parallelism (number of executors) have a big impact on I/O bandwidth. Maximal bandwidth (approximately 400MB/s) is achieved with block sizes of at least 512KB, but with enough parallelism, a block size of 256KB already provides a bandwidth of 350MB/s.
This section describes the streaming iterative workloads and their Blockus implementations. A streaming iterative algorithm is an iterative algorithm that processes its whole input data set in every iteration. Table 5.1 contains a summary of the workloads.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Main operations</th>
<th>Input data</th>
<th>Input data sizes (GB)</th>
<th>Memory size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense matrix multiplication</td>
<td>Dense matrix multiplication</td>
<td>2 dense, square matrices</td>
<td>5.96</td>
<td>4</td>
</tr>
<tr>
<td>K-means</td>
<td>Find closest cluster center for each data point</td>
<td>D dimensional input points and cluster centers</td>
<td>26.8, 40.2, 53.6</td>
<td>16</td>
</tr>
<tr>
<td>Pagerank</td>
<td>Sparse matrix-vector multiplication</td>
<td>Transition matrix, pagerank vector</td>
<td>13.3, 19.5, 27.6</td>
<td>8</td>
</tr>
<tr>
<td>Conjugate gradient method</td>
<td>Sparse matrix-vector multiplication, vector dot products, linear combinations</td>
<td>Symmetric, pos. def. matrix, 5 vectors</td>
<td>13, 18.8, 26.2</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.1: Workloads used for evaluation.
5.3.1 Dense matrix multiplication

Dense matrix multiplication computes the product $C$ of matrices $A$ and $B$ using the formula $c_{ij} = \sum_k a_{ik} b_{kj}$. For the sake of simplicity let us assume that $A$ and $B$ are $n \times n$. Using the above formula, computing all $c_{ij}$ takes $\Theta(n^3)$ operations (asymptotically faster algorithms exist, such as the Strassen algorithm and the Coppersmith-Winograd algorithm). For a machine with main memory size $M$ and matrices bigger than $2M$, the naive implementation of the formula (three nested loops iterating over $i,j,k$) uses $\Theta(n^3)$ I/O operations, but blocked matrix multiplication using blocks of size $\sqrt{M}$ uses only $\Theta\left(\frac{n^3}{\sqrt{M}}\right)$ I/O operations, and this is asymptotically optimal (using the “conventional algorithm”) [25]. The blocked matrix multiplication algorithm is straightforward to express in the Blockus programming model.

5.3.2 K-means

The goal of k-means clustering is to partition $N$ $D$-dimensional points into $K$ clusters, minimizing the sum of the squared distances of the points and their cluster centers:

$$\{c_1, \ldots, c_K\} = \arg\min_{\{c_1, \ldots, c_K\}} \sum_i \sum_{p \in c_i} \|p - m_i\|^2,$$

where $m_i = \sum_{p \in c_i} p$. Finding the optimal clustering is NP-hard.

Lloyd’s algorithm (sometimes simply called the k-means algorithm) is a widely used, easy to implement heuristic. It is an iterative algorithm where each iteration consists of two steps.

1. Assign each point to the cluster with the closest cluster center.

2. Update each cluster center (to be mean of the point assigned to the cluster).

The initial cluster centers are chosen randomly from the data points.

In the Blockus implementation, the input data points are stored in a $D \times N$ matrix that is column-partitioned into $p$ partitions. A $D \times K$ matrix contains the cluster centers.
Note that $K$ is small, so $D \times K$ is a small matrix. In the assignment step, the points of each partition are assigned to the closest cluster center. For each cluster, there are also $p$ variables, containing the contributions of the various partitions to the given cluster’s mean and number of points. In the update step, the mean of each cluster is computed from the aforementioned $p$ variables.

The input data sets are synthetic. The input points are generated randomly, with each coordinate chosen uniformly randomly in the $(0, 1)$ interval. We fix $D = 6$, $K = 3$, and we use $N = 600 \times 10^6, 900 \times 10^6, 1200 \times 10^6$ to generate data sets of size $26.8 \text{GB}, 40.2 \text{GB}, 53.6 \text{GB}$. The partition size is fixed at 500000 points, which means each block has a size of $22.8 \text{MB}$.

### 5.3.3 Pagerank

The pagerank algorithm [18] was designed to compute a ranking of web pages. Let $p_i^{(k)}$ denote the pagerank in step $k$ and let $d_i$ denote the out-degree of page $i$. The algorithm iteratively updates all pageranks using the following formula:

$$p_i^{(k+1)} = \frac{1 - d}{N} + d \sum_{\exists j \rightarrow i} p_j^{(k)},$$

where $d$ is called the damping factor, and its value is usually 0.85.

The algorithm computes the stationary distribution of a random walk on the web graph. In each step, the random walk either (with probability $d$) goes to a random neighbor of the current vertex, or goes to a random vertex of the graph (with probability $1 - d$). This computation is straightforward to express using linear algebra operations. The update step is

$$p^{(k+1)} = \frac{1 - d}{N} \mathbb{1}_N + d T p^{(k)},$$

where $T$ is the web transition matrix (appropriately normalized adjacency matrix), and $\mathbb{1}_N$ is an all-one vector of length $N$.

In the Blockus implementation, the matrix is column-partitioned, and the pagerank vec-
tor is partitioned appropriately. In the graph formulation, this corresponds to partitioning the vertices, and storing the in-edges of each partition together. If \( p_i \) and \( T_i \) denote partition \( i \), then the update step is a foreach over the partitions, where each task updates a partition of the pagerank vector:

\[
p_i = \frac{1 - d}{N} \mathbf{1}_N + dT_ip
\]

The exact implementation is shown in section 2.3.

The test input data are R-MAT random graphs [5]. These graphs have degree distributions similar to real-life power law graphs. We generate 3 input graphs, with \(|V|, |E| = \{2^{26}, 16 \times 2^{26}\}, \{2^{26}, 24 \times 2^{26}\}, \{2^{27}, 16 \times 2^{27}\} \). The sizes of these graphs are 12.8GB, 19.2GB and 26.6GB respectively. We convert the graphs into transition matrices for the pagerank algorithm, which involves adding self-loops to vertices that do not have any out-edges. This is a small change and has no significant effect on the data set sizes. To obtain the total input data size, we need to add the size of the pagerank vector (0.5GB and 1GB, depending on the number of vertices) to the graph size. The number of vertices per partition is \( 2^{18} \), which results in average partition sizes of 51MB, 75MB and 51MB for the three data sets.

5.3.4 Conjugate gradient method

The conjugate gradient method is used to solve a system of linear equations \( Ax = b \), where \( A \) is a symmetric, positive definite matrix.

The algorithm iteratively updates a candidate solution \( (x_i) \) until convergence. Algorithm 2 shows the steps of the algorithm. Each iteration consists of one matrix-vector multiplication \( (Ap_k) \) and vector operations (linear combinations and dot products).

The matrix-vector multiplication is performed the same way as in the pagerank implementation. Vector linear combination is performed partition-wise. Dot product is also performed partition-wise, and the resulting sub-dot-products are summed at the master.

Similarly to the pagerank benchmark, the test input data are R-MAT random graphs
Algorithm 2 Conjugate gradient method

\[ r_0 \leftarrow b - Ax_0 \]
\[ p_0 \leftarrow r_0 \]
\[ k \leftarrow 0 \]
\[ \text{while } |r_k| > \varepsilon \text{ do} \]
\[ \alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T Ap_k} \]
\[ x_{k+1} \leftarrow x_k + \alpha_k p_k \]
\[ r_{k+1} \leftarrow r_k - \alpha_k Ap_k \]
\[ \beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \]
\[ p_{k+1} \leftarrow r_{k+1} + \beta_k p_k \]
\[ k \leftarrow k + 1 \]
\[ \text{end while} \]
return \( x_k \)

[5]. However, the conjugate gradient method only works on symmetric matrices, so we symmetrize the generated matrices. We generate 3 input graphs, with \(|V|, |E| = \{2^{25}, 16 \times 2^{25}\}, \{2^{25}, 24 \times 2^{25}\}, \{2^{26}, 16 \times 2^{26}\}\). After symmetrization, the sizes of these graphs are 11.8GB, 17.5GB and 23.7GB respectively. To obtain the total input data size, we need to add the size of all the vectors used by the implementation. The size of the vectors is 256MB or 512MB, depending on the number of vertices, and there are 5 such vectors. The total input data sizes are 13GB, 18.8GB and 26.2GB. The number of vertices per partition is \(2^{18}\), which results in average partition sizes of 94MB, 140MB and 95MB for the three data sets.

5.4 System evaluation using workloads

In this section we evaluate the system on the workloads using the scheduling policies presented in section 4.3. We compare the results to the predictions of the performance model presented in section 4.2. We also compare to a version of Blockus that uses the streaming scheduler and has been modified to allocate memory by mmaping files, providing out-of-core computation capability solely by relying on the operating system to do paging.

Throughout this section, we refer to the optimal execution time as the minimum of
computation time and I/O time (required for reading input data not present in memory). This is the same as the predictions of the performance model for the execution times of the reversing and greedy schedulers (see section 4.3). Computation time is read from executor logs, and I/O time is estimated by dividing the lower bound on I/O (the difference between the data set size and the total memory size) by $400\text{MB/s}$, the maximum read speed achieved by Blockus as measured in section 5.2.3.

### 5.4.1 Dense matrix multiplication

Dense matrix multiplication is an example of a workload where I/O optimizations do not matter. The computation takes so much more time than I/O that any improvements to the I/O time are negligible. For example, running the algorithm on two matrices with a total size of $6GB$, with memory size limited to $4GB$ takes 5520 seconds using mmap. The total I/O (measured by performance counters) is $36GB$, which takes 92 seconds if we assume an I/O bandwidth of $400\text{MB/s}$; this is 1.6% of the computation time.

Execution time using explicit I/O is 5517 seconds.

### 5.4.2 K-means

![Figure 5.4: Blockus running times for the k-means benchmark. Horizontal lines denote computation time.](image)

(a) 4 executors

(b) 12 executors

Figure 5.4: Blockus running times for the k-means benchmark. Horizontal lines denote computation time.
Figure 5.4 shows Blockus system performance on the k-means benchmark.

Comparison with mmap-based system

Even with the streaming scheduler, the explicit I/O management of Blockus is faster than mmap. Using 4 executors, the speedup is $14 - 19\%$, while using 12 executors it is $5 - 7\%$. This is because mmap with 4 executors is not aggressive enough to utilize the full read bandwidth of the SSD. However, the I/O load increases as we increase parallelism (and therefore the number of simultaneous mmaps). Figure 5.5 shows (for the 64GB data set) how the average read bandwidth increases with the number of executors when using mmap, but it stays constant (and close to maximum bandwidth) when using explicit I/O.

![Figure 5.5: Average read speed as a function of the number of executors.](image)

Comparison of schedulers

For the k-means benchmark, random scheduling can provide $2 - 18\%$ speedups, while reversing & greedy scheduling can provide $20 - 38\%$ speedups over streaming scheduling.

Figure 5.4 shows that the reversing & greedy schedulers achieve compute limited execution with 4 executors, while all other schedulers are I/O limited. In this case, $\beta = 0.8,$
which means that (according to the model) these schedulers can process data sets up to $5 \times$ memory size (up to 60GB).

On the other hand, $\beta = 0.58$ with 12 executors. This leads to compute limited execution (for reversing & greedy schedulers) only on the small data set. On the larger data sets, execution is I/O limited for all schedulers. As the data set size increases, the relative benefits provided by the two schedulers decrease in accordance with the performance model.

The relative errors of the performance model for reversing & greedy scheduling are 0.5 – 11%, and 7 – 15% for random scheduling. This validates the assumptions made in the model for the k-means benchmark.

### 5.4.3 Pagerank

Figure 5.6 shows Blockus system performance on the pagerank benchmark.

![Pagerank Figure 5.6](image)

Figure 5.6: Blockus running times for the pagerank benchmark. Horizontal lines denote computation time.

**Comparison with mmap-based system**

Figure 5.6 shows that the pagerank algorithm is compute limited when using 2 executors, but becomes I/O limited when using 4. In the compute limited case, Blockus only provides a 5 – 15% speedup. However, using 4 executors, I/O becomes the bottleneck, leading to a
20% speedup.

Comparison of schedulers

Using 2 executors, $\beta > 1$, so all schedulers (even the streaming scheduler) should be compute limited. In this case, the model does not predict any improvement when using the more advanced schedulers. However, the results show some improvement. The reason for this is the (false) assumption that I/O and computation overlap perfectly. This is false at the beginning of every iteration: the streaming scheduler has to wait for the first few blocks to be present in memory before it can start any computation.

Using 4 executors, all schedulers become I/O limited. The random scheduler provides speedups of $8 - 11\%$, and the reversing & greedy schedulers provide speedups of $8 - 32\%$.

With 4 executors, the execution times for the reversing & greedy schedulers are significantly higher than predicted by the performance model (leading to model relative errors of $13 - 14\%$ for the small and large data sets). The main reason for these errors is that the model assumes that all of the memory contents are reusable between iterations. However, this is not the case for the Blockus pagerank implementation: the full pagerank vector has to be constructed from its blocks at the beginning of every iteration. This requires $0.5 - 1GB$ of memory (the size of the pagerank vector), which is significant compared to the shared memory size ($6GB$). We can try to model this aspect of the algorithm by assuming that we have a smaller memory size available for caching. Modifying $\alpha$ accordingly decreases the relative errors of the model for the reversing & greedy schedulers to $5.7\%$ (for the large data set) and $11.7\%$ (for the small data set).

5.4.4 Conjugate gradient method

Figure 5.7 shows Blockus system performance on the conjugate gradient method benchmark.
Comparison with mmap-based system

The conjugate gradient method benchmark is less compute intensive than the pagerank benchmark. Computation for conjugate gradient with 2 executors takes 27 seconds on a 13GB data set, while for pagerank it takes 36 seconds on a 13.3GB data set.

Faster computation leads to faster I/O for mmap, because there is less time spent doing computation between the I/O requests. This explains the results seen in figure 5.7: Blockus with the streaming scheduler only provides a 5—8% speedup over mmap, which is significantly less than the 20% speedup in the case of pagerank. The measured average I/O bandwidth during execution (with 2 executors over the large data sets) also confirms that the reason for the improvement difference between pagerank and conjugate gradient method is the I/O bandwidth difference: the average I/O bandwidth achieved by mmap during pagerank is about 280MB/s, while it is about 330MB/s during conjugate gradient method.

Comparison of schedulers

The random scheduler achieves speedups of 0 – 8%, and the reversing & greedy schedulers achieve speedups of 5 – 25%. The measured execution times are significantly higher than predicted by the performance model; model prediction relative errors are 19 – 29%.
The main reason for the high prediction errors is the violation of the model assumption that all computation in an iteration can be reordered (i.e. that all memory contents are reusable at the beginning of an iteration). This is not the case for the conjugate gradient method: each iteration starts with a matrix-vector multiplication, and then does some vector operations. This means that at the end of an iteration, the vectors will take up most of the memory, but the algorithm requires the matrix (and one vector) at the beginning of an iteration. The scheduler log contains details about the exact memory contents. The log shows that for the small data set, only about $3.5\text{GB}$ of the matrix is present in memory at the beginning of each iteration. For the large data set, only $2.1\text{GB}$ is present. Because the model assumes that all $6\text{GB}$ of memory contains reusable data, the model overestimates I/O reduction (for the reversing & greedy schedulers) by about a factor of 2.

As in the case of pagerank, we can update the $\alpha$ model parameter with the a posteriori knowledge of exact data reuse between iterations. Simply changing the “memory size” involved in the computation of $\alpha$ from $6\text{GB}$ to $3.5\text{GB}$ and $2.1\text{GB}$ (for the small and large data sets, respectively) to account for the extra vector operations decreases the model relative errors to 5.1% and 4.8%.

The results clearly show the sensitivity of computation reordering to the data access pattern and parallelism of the application. The reversing & greedy schedulers provide smaller relative improvements (than for other algorithms) because iterations of the conjugate gradient method algorithm consist of a sequence of parallel operations that do not share all input data, and the system is unable to perform computation reordering across these operations.

### 5.5 Summary and discussion

#### 5.5.1 System performance

Figure 5.8 shows that the system utilizes the underlying hardware with high efficiency. When the system does not need to wait for I/O (typically at the beginning of every iteration
with the greedy scheduler), the CPU is utilized at 90% – 100%. CPU utilization is highly variable for the conjugate gradient method benchmark because each iteration consists of many for eaches, and there is less data reuse between iterations than in the other algorithms (shortening the high CPU utilization period at the beginning of iterations). The SSDs are utilized at around 75% – 100%. Although the SSD utilization is not perfect, the system (with streaming scheduling) can achieve 5% – 20% speedups over a naive solution that relies on the operating system through mmap to perform I/O. The overall bandwidth provided by mmap increases with parallelism, and also increases as computation density decreases. However, mmap performance remains worse than Blockus performance in all benchmarks.

5.5.2 Scheduling benefits and data reuse of applications

The results in section 5.4 show that the reversing and greedy schedulers can provide significant benefits (depending on the application and data size) over streaming and random scheduling. Figure 5.9 shows the execution time improvements provided by the different schedulers on the benchmarks. Figure 5.10 shows the improvements of the reversing & greedy schedulers over the streaming scheduler across benchmarks. In line with the performance model, we see that the improvements decrease as the data set sizes increase.

For the k-means algorithm, the execution time achieved by the reversing & greedy schedulers is only 1% – 13% higher than the optimal execution time. The reason for this is that the
structure of the k-means algorithm is very simple: every iteration consists of two foreaches, and one of them only work on a small amount (a few kilobytes) of data, having negligible impact on memory contents. The other foreach processes the whole data set. This pattern leads to high data reuse between iterations that can be exploited by the schedulers.

For the pagerank and conjugate gradient method benchmarks, advanced scheduling still provides benefits, but the execution times are significantly higher (as high as 52%) than the optimal execution time predicted by the model. The reason for this is that these algorithms have more complicated computation structures. In pagerank, the full pagerank vector has to be constructed from its blocks at the beginning of every iteration. In conjugate gradient method, every iteration consists of many foreaches that do not use the same input blocks, decreasing data reuse. Reordering across foreaches could improve the data reuse of these applications.

5.5.3 Model validity

In this discussion, we focus on the model predictions for the reversing & greedy schedulers. Predictions for the random scheduler are less accurate not only due to randomness, but because of the difficulty of computing the exact expected value and using an approximation instead (see section 4.3.3).

The performance model is accurate for the k-means benchmark because of its simplicity.
Figure 5.10: Speedup of the reversing & greedy schedulers across benchmarks, using the maximum number of executors for each benchmark (higher is better).

It is less accurate for pagerank, and highly inaccurate for conjugate gradient method, because these algorithms violate some of the model’s assumptions about data reuse. However, adjusting the $\alpha$ parameter of the model using a posteriori information about data reuse increases accuracy (see sections 5.4.3 and 5.4.4).

The execution times of the k-means benchmarks are close to the corresponding predictions of the performance model because the assumptions made by the model hold: there is a single big foreach in every iteration that can use all in-memory data. Therefore, computation reordering can reuse all in-memory data.

The predictions have higher error rates for the pagerank and conjugate gradient benchmarks because a critical assumption of the model is violated: not all in-memory data is reusable. In the case of pagerank, some of the memory is used up to reconstruct the full pagerank vector from its blocks. In the case of conjugate gradient method, each iteration consists of multiple for-foreaches that use 5 different vectors total, and data reuse is not possible across all for-foreaches.
Note that when the assumptions are violated, the measured execution time always increases compared to the model prediction. The negative predictions of the model about large data sets still hold, because the model provides a lower bound on execution time.

5.5.4 Scaling to big data sets

According to the model, computation reordering does not scale well to big data sets. Keeping the memory size \((M)\) fixed and increasing the data set size \((D)\) is equivalent to \(\alpha \to 0\). In this case, the system only remains efficient (i.e. compute limited) if \(\beta \to 1\), which means that the algorithm itself has to be compute limited on the given hardware. Another way of looking at this is considering the maximum data set size that can be processed for a given \(\beta\), without being I/O limited (see figure 5.11). Again, scaling to big data sets is only possible if \(\beta \approx 1\).

In the I/O limited case, the execution time (relative to the streaming scheduler) with computation reordering is \(1 - \alpha\) in the best case, which goes to 1 as \(\alpha \to 0\), meaning that

Figure 5.11: Maximum data set size that can be processed without waiting for I/O, as a function of \(\beta\).
there is no significant improvement gained from computation reordering (see figure 5.12).

For streaming iterative algorithms, the effectiveness of computation reordering schedulers is limited to the case when the data set size is comparable to the memory size. When data sets are large, the extremely limited data reuse of streaming iterative algorithms means that computation reordering provides negligible benefits, and execution time is close to the time it takes to read the whole data set from stable storage.
CHAPTER 6

SUMMARY AND FUTURE WORK

6.1 Research question and methodology

Our goal was to evaluate how exploiting excess parallelism in streaming iterative algorithms can increase their I/O-efficiency for out-of-core execution. We sought to understand the computation reordering techniques and system configurations that enable a system to execute these algorithms on data sets bigger than memory without being I/O limited, and the speedups that can be achieved in the I/O limited case.

The motivation for this question is to evaluate the viability of scale-up computing systems that use stable storage (SSDs) to increase data capacity. We focus on streaming iterative algorithms because of their simplicity: MapReduce does not exploit their inter-iteration data reuse patterns, but other (in-memory) scale-out systems can execute them with high efficiency.

We implemented Blockus, a modified Presto system with an execution engine that transparently manages I/O for out-of-core execution on a single machine. The baseline system in our evaluation is Blockus with a task scheduler that does not do any I/O-aware computation reordering, and simply traverses data blocks in a fixed order.

We proposed and evaluated two task scheduling policies that minimize I/O by exploiting parallelism and data reuse. The first policy explicitly targets iterative streaming algorithms using a simple heuristic that involves reversing the direction of data traversal to maximize in-memory data reuse. The second policy exploits the system’s knowledge about memory contents to greedily schedule tasks that require the least amount of I/O to execute. We used three streaming iterative algorithms (k-means, pagerank, conjugate gradient method) for our benchmarks.
6.2 Results

We characterize configurations (of system, algorithm and data set) using two numbers: $\alpha$ is the ratio of reusable in-memory data size (at the beginning of every iteration) to data set size, and $\beta$ is the ratio of computation time and I/O time over a block of data (assuming that computation and I/O time scale linearly with block size). For the k-means algorithm, the reusable in-memory data size is simply the size of the memory. For other algorithms, not all contents of memory might be reusable because of their data access patterns.

We have found that for our set of iterative streaming benchmarks, the two proposed schedulers have similar performance. They can avoid being I/O limited if $\alpha + \beta \geq 1$ (as opposed to $\max(\alpha, \beta) \geq 1$ for the baseline system). If they are not I/O limited, they can still improve performance, providing an execution time that is approximately $1 - \alpha$ times the execution time on the baseline system.

On our experimental systems and benchmarks, $\beta$ is always at least 0.55, which means that the proposed schedulers can avoid being I/O limited even on data set that are up to $2.2 \times$ bigger than memory size. On larger data sets, speedups decrease proportionally with data set size.

Although we can achieve speedups on smaller data sets, it is clear that this method does not scale to large data set sizes: as $\alpha \to 0$, all speedups converge to 0. This means that the extra parallelism in streaming iterative algorithms is not enough to hide I/O latency: as the data set size increases, execution time converges to the time it takes to read the whole data set. Despite speedups from improved scheduling on smaller data sets, streaming iterative is not a good algorithmic formulation for out-of-core computation.

6.3 Future work

The following list contains future work that could provide better understanding of what kinds of applications are well-suited for out-of-core computation, and what kinds of application
knowledge an out-of-core computation system could exploit to achieve high efficiency.

- Only being able to schedule tasks from a single foreach at a time is a serious limitation. Using a system that has a more flexible (e.g. DAG-based) execution model and better dependency tracking, we could study the effects of computation reordering on more complex algorithms.

- Studying the data access patterns of non-streaming algorithms, and the underlying computation reordering opportunities. We should consider algorithms with more complicated, but fixed data access patterns (such as matrix factorizations) first, and then algorithms with data dependent access patterns (such as graph traversals).

- Caching policies can have a big impact on data reuse. Throughout this thesis, we assumed an LRU eviction policy. Studying other policies could enable much better memory utilization. For example, algorithms with non-uniform data access could benefit from some kind of caching that keeps frequently accessed data in memory.

- We assumed a single SSD for stable storage. Studying I/O scheduling and data placement strategies on heterogeneous storage systems (for example a disk array with high latency and high bandwidth, and an SSD with low latency and medium bandwidth) could lead to a more flexible system.

- Understanding how out-of-core computation interacts with a distributed storage hierarchy (with storage in local DRAM, local SSDs, remote DRAM and remote SSDs) could lead to more efficient distributed out-of-core scheduling or even data center design, significantly increasing the data capacity of distributed computation systems.

- Asynchronous algorithms could be exploited to reorder computation more aggressively. There are trade-offs between prioritizing computation that minimizes I/O and prioritizing computation that speeds up convergence. Understanding these trade-offs could lead to highly efficient out-of-core execution for asynchronous algorithms.
• Exploring the trade-offs involved in choosing a storage block size. Larger block size increases I/O bandwidth, but decreases access granularity and therefore increases the amount of unnecessary I/O.

• Understanding the data access patterns of algorithms with fine-grained data access could lead to more efficient storage formats and I/O: for example, vertices of a graph could be reorganized to make sure that vertices frequently accessed together (temporally) are stored in the same storage block, increasing access locality.
REFERENCES


