Parallel Processing for Data Deduplication

Peter Sobe, Denny Pazak, Martin Stiehr
Faculty of Computer Science and Mathematics
Dresden University of Applied Sciences Dresden, Germany
Corresponding Author Email: sobe@htw-dresden.de

Abstract: Data deduplication is a technique for detection and elimination of duplicated data blocks in storage systems. It creates a set of unique data blocks and places references accordingly, which allows to access the original data within a reduced amount of data blocks. For deduplication, hashes of data blocks are calculated and compared in order to detect and remove duplicates. It can be seen as an alternative to data compression that allows to save storage capacity in large storage systems. A storage capacity saving is reached at the cost of additional computational effort that originates when data blocks are written and updated. This computational effort increases with the size of the storage system. On a single processor system, deduplication influences the performance in a negative way, particularly the write and update rates drop. The utilization of parallelism is a rewarding task to compensate this performance drop, particularly for hash value calculations and comparisons of hashes. In this paper we explain in which parts of a deduplication system it is worth to parallelize and how. Exemplarily, we show the performance results of two deduplication algorithms and their parallel implementations, based on multithreading and on parallel GPU computations.

1 Introduction

In this paper, parallelism concepts for data deduplication algorithms are presented. Deduplication first appeared in the field of backup systems and repositories for source code and technical documents (such as git) that contain many versions of files with little alterations. Meanwhile, general purpose file systems and database systems got equipped with deduplication. Examples for the use of deduplication are ZFS, lessfs [les15], opendedup [ope15] and OracleDB for storage of large data objects in databases.

Deduplication improves the storage utilization and is capable to reduce the storage operation cost for large systems. This benefit comes with extra computational effort for write and update operations. Deduplication gets more effective with increasing storage size. Unfortunately, the computational cost increases as well with the amount of stored data. This computational cost indirectly reduces the storage access performance, but can be compensated by parallel computing, particularly by using multiple cores or by utilizing the GPU of a computer system.

Recently, research started to evaluate several possibilities of parallel computations for deduplication, such as [XJF+12] for exploiting thread-level parallelism in multi core systems to reach acceptable high I/O throughput for data deduplication. Besides of parallelism for the deduplication task, distributed storage systems are in the focus of research as well.

In [KMBE12] a cluster solution using data deduplication is presented.

The contribution of this paper is the description of two algorithms for deduplication. These algorithms are called (i) Backward Referencing and (ii) Hashmap-based Referencing. These two algorithms reflect state-of-the-art principles that were found in existing deduplication solutions. The description is followed by the identification of algorithm phases for parallel execution and performance reports about two different approaches for parallel processing. One approach is thread-level parallelism utilized by task-parallel programming with the Intel TBB class library. Another approach consists in offloading of the search for duplicates to the GPU. The GPU is capable to perform a huge number of comparisons in parallel.

The remainder of the paper is organized as follows. The principle of data deduplication and two algorithmic approaches are described in Section 2. In Section 3 the algorithms are revisited with respect to parallelism. Last, in Section 4 we present the performance gain of parallel execution for the two algorithms. A summary concludes the paper.

2 Data Deduplication

2.1 Original Data Layout

For the description of deduplication we start with a small example of 8 data blocks that contain repetitions. The sequence of blocks is shown in Fig. 1 where equal letters represent equal block content. These can be equal files, duplicates of emails or equal blocks in different versions of files.

sequence of blocks

A	B	B	C	D	A	D	C

Figure 1: Example of a sequence of blocks taken as original data layout.

The objective of deduplication is to remove double blocks from the storage and to represent their existence by references to unique blocks.

In the following two different deduplication algorithms with different data structures are compared. One algorithm we call Backward Referencing (see 2.2) and the other Hashmap-based Referencing (see 2.3)

2.2 Backward Referencing

Fig. 2 illustrates one possible deduplicated data layout. Data blocks are indirectly referenced via an index that contains pointers to data block descriptors. A data descriptor holds a reference to the data block X and in addition the hash value h_X and a reference counter.

To store the hash value is optional, but beneficial for the integration of new blocks.

In file systems such index structures to blocks are already present. Thus, the integration of deduplication does not cause noticeable extra access cost for read operations.

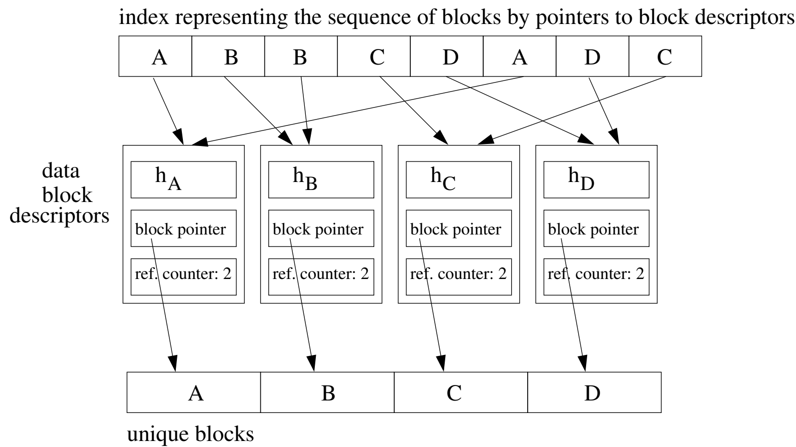

Figure 2: Deduplicated data layout.

The transformation from the original to the deduplicated data layout is done stepwise as described below. For each step we give the time complexity of the sequential algorithm step, depending on the number of blocks in the system n.

1. **Chunking and Hashing:** Original data is divided into blocks (a so called chunking). For each block a hash value is calculated. In order to keep the probability of hash collisions considerably low, strong cryptographic hash functions are used, such as MD5. At the end of this step, a sequence of block references and corresponding hash values is obtained. The computational complexity of this step is $O(n)$.
2. **Indexing:** In a first walk through the sequence of hash values, multiple occurrences of hash values are tagged. For every hash value at position i, all hashes from position $i + 1$ to the end of the sequence are compared. In case of equal hash values, a backward reference is added. At the end of this step, the sequence of hash values got extended by duplicate identification tags and backward references. This step involves $n(n - 1)/2$ comparisons for n blocks, and is characterized by $O(n^2)$.
3. **Re-indexing:** To obtain structures for fast access to data blocks, the references have to reflect the concentration of data blocks to a sequence of unique blocks. Thus the backward references have to be corrected to the block numbers in the sequence of unique blocks. For all blocks references, all other references that possibly point back to it have to be corrected. At the end of this step, a sequence of references to unique blocks is obtained. Equally to the the Indexing step, $n(n - 1)/2$ references must be visited and a complexity of $O(n^2)$ is present. This last step can include a

reorganisation of data blocks to a set of unique blocks and references that correctly address unique data blocks.

This last step can include a reorganisation of data blocks to a set of unique blocks and references that correctly address unique data blocks.

Updates within a deduplication storage system work similar to block write operations. An altered block is handled like a completely newly written one. In case that the new block content is a new unique block, this block is stored and referenced via a new block descriptor. Otherwise a reference to another already existing block is included and the reference to the old block descriptor is removed. With the help of the reference counter, the decision can be taken, whether a block can be finally deleted or not.

Reading blocks from a deduplicated storage layout is done by accessing the index, following the link to a data block descriptor and finally fetching the entire data block. There is no extra computational overhead apart from referencing blocks via an index which is already present in most file systems.

2.3 Hashmap-based Referencing

Another algorithm for deduplication appeared that uses a hashmap data structure for the detection of duplicated blocks. The first phase of hashing is similar to backward referencing and consumes a compute time according $O(n)$. A difference compared to the backward referencing algorithm is that a sequence of hash values is kept as initial reference to the the blocks, and to represent the order of original blocks.

In a second step, tuples consisting of a reference counter and a block pointer are inserted into a hashmap data structure that places entries with the key h_X according to a hash function h(h_X) at a defined place. The hash values from the first phase are taken as keys for the tuples.

In the case that the entry h_X is a new one, it gets newly placed in the hashmap. The related data block X is moved to the set of unique blocks and referenced from the hashmap entry. The reference counter is set to 1. In case that the entry for h_X already exists, solely the reference counter is increased. The corresponding data block is a duplicate and can be discarded. At the end of step 2, a data layout as depicted in Fig. 3 is obtained. The time of hashmap insertion for n blocks corresponds to the complexity order of $O(n)$, because of $O(1)$ for a single hashmap access operation.

Updates of blocks first cause a lookup of the old hashmap entry and its removal when the reference counter is 1, otherwise the reference counter is decreased. For the new content, an according hash value is inserted in the sequence of hash values. The rest is done according to a write operation of the block.

The read operation first reads the hash from the sequence of hashes and then accesses the corresponding data block through the hash map.

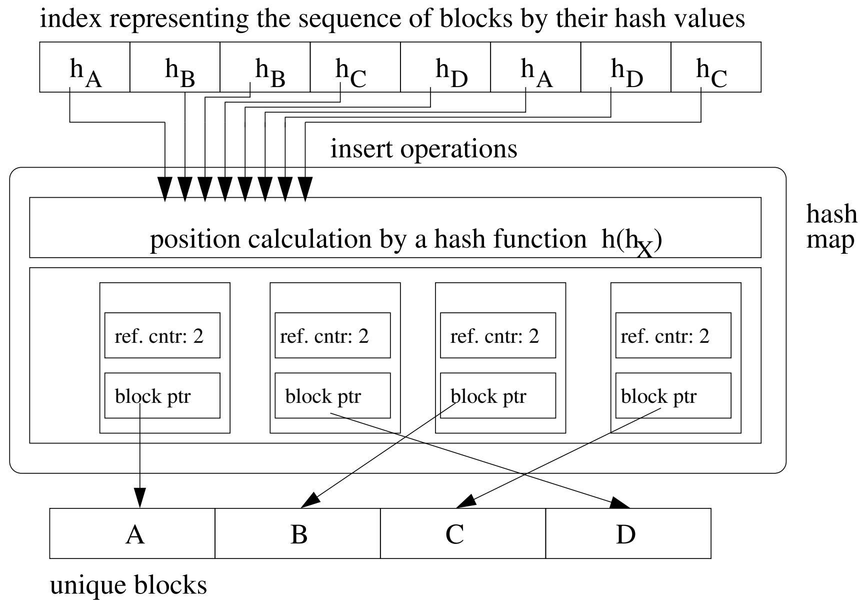

Figure 3: Hashmap-based Referencing: Deduplicated data layout.

3 Parallelism Concepts

For both algorithms, the hashing phase can be executed independently on different blocks. With p processor cores, a single core at most has to calculate $\lceil n/p \rceil$ hash values. This reduces the time consumption of this phase by the factor $1/p$.

3.1 Parallelism of the Backward Referencing algorithm

The phases Indexing and Re-indexing offer room for parallel execution. Every phase requires that for a sequence of entries, all entries must be selected sequentially, and all entries positioned right of the selected one must be visited for hash comparison or for a correction of the reference pointer. For a data set of n blocks, this requires $n(n-1)/2$ steps. With p processors the number of steps can be obviously reduced to approximately $n(\lceil (n-1)/p \rceil)/2$ because of the independence of the operations that relate to the same selected entry. Ideally, a speedup of p can be reached for a large n.

3.2 Parallelism of the Hashmap-based Indexing algorithm

The hashmap-based algorithm works differently and delegates the duplicate detection to the insert operation of the hashmap data structure. The insert operation signals a second colliding entry with the same key. From this perspective, parallel tasks can only be utilized in the way of concurrent insert operations.

It strongly depends on the implementation of the hashmap access operations whether parallel inserts are possible at all. Insert operations are allowed to run parallel in the case that the insert positions do not collide. In the case of a collision, typically locking is applied. In the case that the first write operation to a specific memory place wins, locking can be mapped to the placement of an entry itself.

When the hashmap supports concurrent inserts, the workload can be distributed to p processors. A number of n insert operations can be executed in $\lceil n/p \rceil$ steps and a speedup of approximately p can be reached.

3.3 Comparison

A comparison of the estimated number of steps of both algorithms is shown in Table 1.

	Backward Referencing	Hashmap-based Referencing
Hashing	n	n
Indexing	$n(n-1)/2$	-
Re-indexing	$n(n-1)/2$	-
Hashmap insert	-	n
total steps / sequential	$n + 2(n(n-1)/2)$	$2n$
total steps / parallel	$\lceil \frac{n}{p} \rceil + 2(n(\lceil (n-1)/p \rceil)/2)$	$2\lceil \frac{n}{p} \rceil$
memory	low memory requirements	highly memory intensive
our implementation	@GPU, CUDA	@CPU, Intel TBB

Table 1: Comparison of the algorithms regarding the number of computation steps and memory consumption.

The comparison shows that both algorithms are well suited for parallel execution and ideally utilize the processors or cores that are available. Another insight is that the Hashmap-based Referencing algorithm generally is less costly in terms of computation steps. This result is based on the assumption that the hashmap allows insert operations in $O(1)$ which is true for sparely filled hashmaps and a large memory that can be used potentially. The benefit in terms of computation cost is combined with a higher memory consumption of the hashmap.

4 Performance Evaluation

In this section, we report first on the parallel implementation of the Backward Referencing algorithm on a GPU using CUDA (4.1) and second on the parallel implementation of the Hashmap-based Referencing algorithm using multithreading and Intel TBB data structures (4.2).

4.1 Backward Referencing using GPU computing

The CUDA version of the backward referencing algorithm searches a specific hash value on different positions in parallel, utilizing a huge number of GPU threads. When a specific hash value has to be compared with a number of x other hash values, the one hash is transferred in the constant memory of the GPU, and the other x hash values for comparison are copied to the GPU global memory. Values are compared by parallel threads. Identical values are found faster compared to a sequential iterative execution. The GPU kernel thread execute according actions in parallel (tagging of duplicates, setting or correction of backward references). The implementation revealed performance results as depicted in Fig. 4. The bars show the relative speedup compared to a sequential CPU implementation on an AMD Phenom II, X4, 840, 3.2 GHz. The GPU used is a consumer model (Type NVidia Quadro 600) with 96 CUDA cores and 1.28 GHz clock rate. The reached speedup does not fully utilize the parallelism of the GPU, due to the interplay of the sequential CPU execution and the parallel GPU execution.

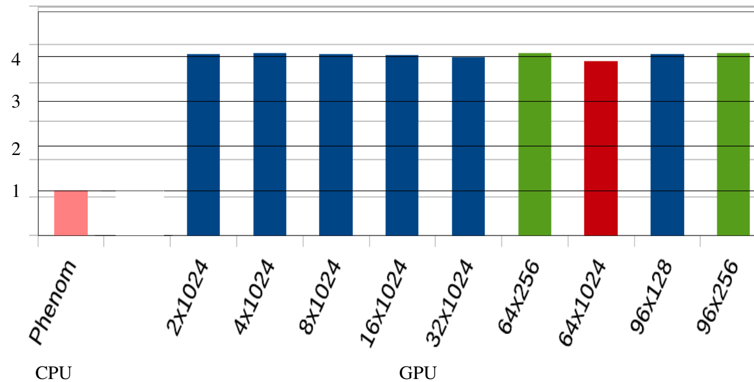

Figure 4: Deduplication throughput using a CPU and a GPU implementation using CUDA. The numbers (e.g. 2x1024) identify the number of blocks and treads for the CUDA kernel invocation.

4.2 Multithreaded Hashmap-based Referencing using Intel Task Building Blocks

A parallel implementation of the Hashmap-based Referencing algorithm was created using C++, a fast MD5-hash implementation[Thi91] and the Intel TBB library (TBB: task building blocks) [Sof15]. This task-parallelism library does not only provide mechanisms to manage several tasks on a multithreaded system, it also provides data structures that can be accessed by parallel tasks. Specifically, the TBB concurrent hashmap [Gue12] was used. This data structure implementation fulfills the above stated requirement of concurrent insert operations that do not block.

The experiments run on an Intel Core i5-750, 2.7 GHz system and a 64-Bit gcc compiler was used. The measurements cover the deduplication of a 1.07 GByte ISO image. The execution time for sequential execution and parallel execution with 2,3, and 4 tasks is shown in Fig. 5.

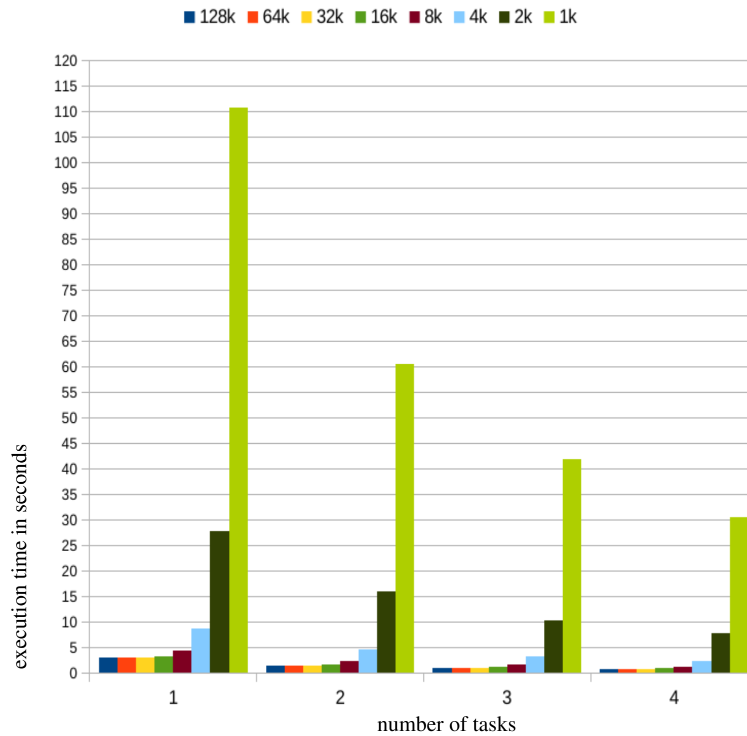

Figure 5: Deduplication execution times, parallel Hashmap-based Referencing algorithm

The plots represent different runs of the deduplication algorithm with specific minimal blocks sizes (128k to 1k). Actually, the implementation covers deduplication with adaptive block lengths and first tries to find equal blocks of a maximum length (128 kByte). For all remaining differing blocks the blocksize for deduplication is divided by 2 until a

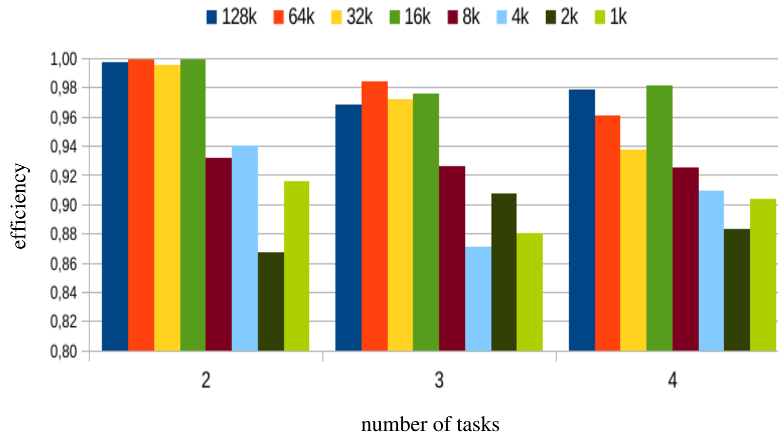

Figure 6: Efficiency of the parallel Hashmap-based Referencing algorithm using Intel TBB

minimum block length is reached. A smaller minimum block size increases the computational cost, because more hash values are generated and have to be processed. These block size variations confirm the general observation of efficient parallel execution.

In general the hashmap-based algorithm turned out as well suited for parallel execution, provided the hashmap allows parallel insert operations. A notable performance could be reached, e.g. roughly the deduplication of 1 GByte could be done in 30 seconds, down to a small block size of 1 kByte. This is a data throughput of 33 MByte/s for a very fine-granular deduplication. The efficiency of the parallel Hashmap-based Referencing algorithm execution is shown in Fig 6.

5 Summary

It could be shown that deduplication is well suited for parallel execution. The two algorithms are different regarding their strategy to compare blocks. Backward referencing works iteratively, but independently on blocks. It compares a number of block pairs, which is the source of parallelism. This can be seen as data parallelism generated from the control flow of a program.
Another approach is based on a hashmap data structure and maps the comparison to a placement problem. Equal entries collide and different entries get placed at different positions. This saves one iteration stage of the sequential algorithm. Even this principle could be modified for parallel execution, where the parallel non-blocking access to the data structure turned out to to be central point for the parallelisation. The Intel task building block library provided the task management and the concurrent hashmap implementation.

References

[Gue12] P. Guermonperez. Parallel Programming Course - Threading Building Blocks (TBB). Intel Software, www.Intel-Software-Academic-Program.com, 2012.

[KMBE12] J. Kaiser, D. Meister, A. Brinkmann, and S. Effert. Design of an Exact Data Deduplication Cluster. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[les15] lessfs. http://www.lessfs.com, accessed 2015, February 9, 2015.

[ope15] opendedup. http://opendedup.org, accessed 2015, February 9, 2015.

[Sof15] Intel Software. Intel Threading Builiding Blocks. www.threadingbuildingblocks.org, accessed: March 20, 2015.

[Thi91] F. Thielo. C++ MD5 Implementation. http://www.zedwood.com/article/cpp-md5-function, accessed: 2015, March 20, 1991.

[XJF+12] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Z. Wang. P-Dedupe: Exploiting Parallelism in Data Deduplication System. In *IEEE 7th Intl. Conference on Networking, Architecture and Storage*, 2012.