CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–44
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Modeling and Optimizing MapReduce Programs

Jens Dörre, Sven Apel, Christian Lengauer
University of Passau
Germany

SUMMARY

MapReduce frameworks allow programmers to write distributed, data-parallel programs that operate on multisets. These frameworks offer considerable flexibility to support various kinds of programs and data. To understand the essence of the programming model better and to provide a rigorous foundation for optimizations, we present an abstract, functional model of MapReduce along with a number of customization options. We demonstrate that the MapReduce programming model can also represent programs that operate on lists, which differ from multisets in that the order of elements matters. Along with the functional model, we offer a cost model that allows programmers to estimate and compare the performance of MapReduce programs. Based on the cost model, we introduce two transformation rules aiming at performance optimization of MapReduce programs, which also demonstrates the usefulness of our model. In an exploratory study, we assess the impact of applying these rules to two applications. The functional model and the cost model provide insights at a proper level of abstraction into why the optimization works. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: MapReduce, Hadoop, distributed computing, list homomorphisms, multiset homomorphisms, program transformation, program optimization

1. INTRODUCTION

Since the advent of (cheap) cluster computing with Beowulf Linux clusters in the 1990s [1], Google's MapReduce programming model [2] has been one of the contributions with highest practical impact in the field of distributed computing. MapReduce is closely related to functional programming, especially to the algebraic theory of *list homomorphisms*: functions that preserve list structure [3]. List homomorphisms facilitate program composition, optimization of intermediate data, and parallelization. Basically, list algorithms that are parallelizable via the divide-and-conquer paradigm are homomorphisms or can be made so by simple pre- and postprocessing [4]. MapReduce is a special case.

To put these theoretical benefits to practical use, we strive for a combination of the formal basis of list homomorphisms with the scalability and industrial-strength distributed implementation of MapReduce.

MapReduce Programming Model Viewed at an abstract level, MapReduce is a simple data-parallel programming model enhanced with sorting, grouping, and reduction capabilities, and with the ability to scale to very large volumes of data. Looking more closely, MapReduce offers many customization options with many interdependences. For example, one can set the total amount of buffer memory to be used during sorting, as well as the total memory available to a task—a value by which the former parameter is *bounded*. Another example is the `SortComparator` function: the user can supply a custom function to specify the *order* in which the members of a group will be passed to the `Reducer` function of a MapReduce program, but the same parameter will also be used to specify *groups* that are passed to another function, the `Combiner` function. MapReduce programmers must keep many details in mind. Making the wrong choices can result in two kinds of bugs: correctness bugs and performance bugs. First, the program may be incorrect, which may be noticed only for larger inputs—a condition which makes testing difficult. Second, the program may be correct, but it may run not much faster than a sequential program, yet consume far more resources. The program may even fail due to a lack of resources. So, our foremost question is: *What is the essence of the MapReduce programming model?* Answering this question will help to avoid these bugs. To address it, we develop a *functional model*, an abstract view of the behavior of MapReduce computations.

Cost Model The functional model also allows us to extract the primitive operations that we have to consider in a corresponding cost model for MapReduce computations, which we develop on top. The *cost model* includes startup, computation, and I/O costs for the different phases of a MapReduce computation. It is parameterized with the input size of the problem to be solved, with selected properties of the MapReduce program executed, as well as with properties of the MapReduce cluster environment on which the program is being run. This helps the programmer to estimate the scaling behavior, and to compare different MapReduce programs, taking the underlying cluster platform into account. More importantly, the cost model is the foundation for the optimization rules that we develop further on.

Focus on Order We put a special focus on a class of MapReduce programs that operate on lists, in which elements are ordered by their position. Standard MapReduce programs work only on multisets, in which order is of no importance. This is explained by the fact that, in a distributed computation, it can become very costly to preserve (list) order, because this requires additional synchronization between distributed nodes. So, there are good reasons for the MapReduce programming model not to preserve order by default. Still, there are many practical uses of MapReduce programs that operate on lists, for example, the Maximum Common Subsequence problem, or the analysis of consecutive revisions of a file in version control systems for the accumulated effect of changes, or different analyses of financial time series (of stock prices), or many others that are formulated in a (sequential) way that requires list structure implicitly. We demonstrate that, with our functional model, it is possible—with reasonable extra effort—to write MapReduce programs that respect list order. To this end, we require that the input data be encoded as a sequence of key–value pairs with consecutive indices as keys. Furthermore, we describe which of the user-defined functions (with which the

DOI: 10.1002/cpe

MapReduce framework is parameterized) need to be made order-aware, and we give example implementations of them. We do not have to change the MapReduce framework itself.

Optimization Rules To demonstrate the expressiveness and applicability of the model, we propose optimization rules for MapReduce programs that are applicable to two important classes of parallel algorithms. These classes are

1. multiset homomorphisms that aggregate data to produce a small result, and
2. list homomorphisms that aggregate data to produce a small result.

We formulate the optimization rules based on our functional model. Using our cost model, we show that these rules are beneficial when applied to MapReduce computations on input sizes above a certain threshold. Furthermore, we validate these theoretical predictions in several experiments.

Experiments We have conducted a series of experiments to provide an initial evaluation of our cost model and optimization rules on a 16-node, 128-core Hadoop cluster. (Apache Hadoop MapReduce [5] is the most widely used MapReduce framework, written in Java.) To this end, we use the example problems described next, create different Hadoop Java programs per problem, and measure and compare their executions on the Hadoop cluster. We obtain speedups between 16 and more than 64 for larger input sizes. These results provide initial evidence that the functional and cost model are practical and that the optimization rules work.

Example Programs For experimentation, we have developed, for two canonical problems, different Hadoop MapReduce programs that vary in parallelism and performance. The two problems are the Maximum (Max) problem and the Maximum Segment Sum (MSS) problem [6, 7], of which the latter is a classical example of data-parallel programming, skeletal parallelism, and list homomorphisms [3]. These problems are canonical in the sense that they represent a whole class of useful and relevant applications. For example, the Max problem can easily be extended to record also the indices of the maximal elements in the input, or changed to perform a sum computation instead. Basically, all standard database aggregation operators are covered by the Max example, along with many customization possibilities not easily possible in standard database management systems.

For each of these two problems, we have created three MapReduce programs. Beginning with a sequential program, we use a first optimization rule to introduce parallelism without sacrificing correctness, which leads us to a two-step parallel program involving huge amounts of communication. Then, we use a second optimization rule to fuse both steps into a single-step parallel program with minimized communication. For the Max problem, these optimizations are just more formal and structured descriptions of best practices, whereas for the MSS problem, they involve intricate control of list order to preserve correctness.

Contributions Let us summarize our contributions:

- a formal model of MapReduce programs suitable for optimization (comprising a functional model and a cost model),

Prepared using cpeauth.cls
DOI: 10.1002/cpe

- an approach to design MapReduce programs that operate on lists instead of on multisets only,
- a total of four optimization rules for MapReduce programs formulated on top of our formal model,
- experiments to validate the model and the optimization rules.

Structure The rest of this article is structured as follows, In Section 2, we give some background on the relevant functional concepts of MapReduce and discuss related work. In Section 3, we introduce our functional model and cost model of MapReduce computations. Section 4 presents the optimization rules applicable to two classes of MapReduce algorithms: multiset homomorphisms and list homomorphisms. To be able to deal with list homomorphisms, we present a general approach for creating MapReduce programs that depend on proper handling of list order. In Section 5, we apply each rule to an example program. To this end, we start with a simple Maximum computation and continue with the Maximum Segment Sum problem—thereby outlining our approach to model MapReduce programs that operate on lists. We report on an exploratory performance evaluation, using these examples on a practical Hadoop cluster. In Section 7, we conclude and outline avenues of further work.

2. BACKGROUND

Let us discuss the background on Google's MapReduce as well as on universal algebra and list homomorphisms, on which our work is based.

2.1. MapReduce

MapReduce is a framework for large-scale, distributed, data-intensive batch processing, developed by Google. Google has promoted MapReduce in several publications [2, 8, 9], which has lead to the creation of multiple alternative implementations and their adoption in industry. In parallel, many research groups from different communities have pinpointed limitations and proposed improvements, extensions, and alternative frameworks in over a thousand publications to day that cite one of Google's MapReduce publications [10].

`Mapper` and `Reducer` Conceptually, MapReduce is an algorithmic template that leaves, in the simplest variant, two functions to be implemented by the user. The `Mapper` function transforms a key and a value to a list of key–value pairs; the type of keys may differ from the type of values. The `Reducer` function transforms its parameters—a key and a list of values—to a list of key–value pairs. The framework applies these functions in the following manner, yielding a useful template applicable to many problems: It applies the `Mapper` function to all key–value pairs in the input, groups the resulting intermediate data by key, applies the `Reducer` function to each group, and, finally, stores all the results. All this happens in a distributed fashion and, after a brief startup phase, without any sequencing.

DOI: 10.1002/cpe

Infrastructure A distributed implementation of the MapReduce framework requires an underlying distributed file system to access input data, giving preference to local access, and to store output and log data. Consequently, MapReduce normally runs as a set of server processes on each node in a cluster, and manages most of the available disk space.

Hadoop Apache *Hadoop*[*] [5] is an open-source Java implementation of Google's MapReduce and the distributed Google File System. Users can choose to run it in their own environment or on a virtual cluster in a cloud environment. Hadoop is the MapReduce implementation most widely used today, available in different distributions from different vendors. It was this maturity and popularity that let us chose to base our work on Hadoop MapReduce. From now on, we will mean Hadoop MapReduce when we speak of MapReduce, and few details may be specific to this implementation.

Further Functions In MapReduce, partitioning is done (usually by hashing) to form larger chunks (that is, partitions) of intermediate data to be grouped. Grouping and (optional) ordering of the data in each partition are achieved by an external sorting function. MapReduce operates on pairs of keys and values (although, theoretically, one could, in the `Mapper`, store all data in the keys, thus making the values obsolete). Between the execution of `Mapper` and `Reducer`, all intermediate data are re-distributed to the different nodes in the cluster, as specified by the partitioning. To reduce the amount of communication, an additional `Combiner` function can be used, which the framework can invoke on parts of intermediate data to reduce their volume. For example, MapReduce can be used with `Combiner` functions to count the number of occurrences of each word in a set of documents: The `Mapper` function extracts each word from a document, uses it as the key and associates it with the value 1 as the occurrence count. The `Reducer` function then accepts a word together with a long list of 1's and computes the sum. Finally, a `Combiner` function sum up all the local values of a node executing the `Mapper` function, before they are transmitted over the network and passed to the `Reducer` function.

Data Parallelism MapReduce aims at *data parallelism*, in which each constituting piece of data is (implicitly) processed by the same function in parallel. This is in contrast to *task parallelism*, in which possibly heterogeneous concurrent tasks (or threads) need to be created explicitly and synchronized, while avoiding deadlocks, starvation, or the corruption of shared data. In data parallelism, the mental model of a programmer can be sequential: there is no need to consider complex interactions between different parallel processes, because all interactions are made explicit via function parameters and return values. Despite the simplicity of this model of parallelism, there are many real applications.

Task Farming MapReduce employs the concept of *task farming*: a job is divided automatically into many tasks. (More exactly, the input data is divided into many chunks.) Each task is assumed to take the same amount of time to complete. If this is not the case,

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

Prepared using cpeauth.cls
DOI: 10.1002/cpe

we have probably encountered data skew. This problem is partly solved by creating multiple smaller tasks per processor (core) in a node, and by the use of dynamic scheduling: tasks are scheduled at run time, such that the scheduler can react to imbalances. Large differences in task completion time that are not due to inherent characteristics of the task input data, but rather stem from temporary differences in node performance in the distributed environment, are addressed by a specific latency optimization [2].

2.2. Foundations from Universal Algebra

When we talk about correctness and different classes of MapReduce programs in our functional model later on, we take an algebraic view of data structures. In universal algebra, data are represented by basic singleton (one-element) structures (of type `X`) and a binary operator $\oplus$ (of type $\texttt{X} \to \texttt{X} \to \texttt{X}$) on (non-empty, basic or complex) data. Assume an operator `S` from numbers to a singleton structure of type `X`, we can formulate the following example data structure (subsequently named `d1`): $(\texttt{S(0)} \oplus \texttt{S(7)}) \oplus \texttt{S(0)}$, which we will use in the remaining discussion.

Trees, Lists, Multisets, and Sets The data structure defined varies depending on the algebraic properties of the binary operator $\oplus$ used in our example. In particular, these properties specify which instances of the data structure are considered equal, that is, cannot be distinguished. This will be important later on when optimizing MapReduce programs (Section 4), because the optimizations require some of the following properties to hold for the data processed and the user-defined functions employed.

- If we know nothing about operator $\oplus$, we need to store all information of the operator tree defining an instance like `d1`: We have defined a *tree*, the simplest (easiest to define) data structure in algebra. Trees are only considered equal iff their syntactical representations are identical. For example, the following tree `t2` differs from tree `d1`, because it has a different structure: $\texttt{S(0)} \oplus (\texttt{S(7)} \oplus \texttt{S(0)})$.
- If operator $\oplus$ is known to be *associative* ($\forall\ \texttt{xs}, \texttt{ys}, \texttt{zs} : \texttt{xs}, \texttt{ys}, \texttt{zs} \in \texttt{X} : (\texttt{xs} \oplus \texttt{ys}) \oplus \texttt{zs} = \texttt{xs} \oplus (\texttt{ys} \oplus \texttt{zs})$), we can neglect the operator/tree structure and use a linear representation without parentheses: we are speaking of *lists*. As lists, both `d1` and `t2` are the same as the following `l1`: $\texttt{S(0)} \oplus \texttt{S(7)} \oplus \texttt{S(0)}$. Yet they are all different from this `l2`: $\texttt{S(0)} \oplus \texttt{S(0)} \oplus \texttt{S(7)}$.
- If, in addition to associativity, we also have the commutativity ($\forall\ \texttt{xs}, \texttt{ys} : \texttt{xs}, \texttt{ys} \in \texttt{X} : \texttt{xs} \oplus \texttt{ys} = \texttt{ys} \oplus \texttt{xs}$) of $\oplus$, we can also neglect the order of construction. We can, for example, choose some arbitrary order to define a normal form to represent the elements of this *multiset* (or *bag*), thereby grouping multiple identical elements. Alternatively, we may choose not to impose a specific order, but rather work with any existing ordering, which is very useful in a distributed context, in which any order, if imposed, would necessitate synchronization. As multisets, `l1` and `l2` and the following `m1`: $\texttt{S(7)} \oplus \texttt{S(0)} \oplus \texttt{S(0)}$ are equal, too. Still, they differ from the following `m2`: $\texttt{S(0)} \oplus \texttt{S(7)}$.

DOI: 10.1002/cpe

- Finally, if the operator $\oplus$ is also idempotent ($\forall$ `xs` : `xs` $\in$ `X` : `xs` $\oplus$ `xs` = `xs`), we do not even need to consider multiples of an element: we have defined *sets*. As sets, `m1` and `m2` and the following `s1`: `S(7)` $\oplus$ `S(0)` are also equal.
  Of course, the simplest data structure *to use* is the set, which is why it plays such an important role in mathematics and also, for example, in the semantics of relational databases.

As an aside, there are also other important properties, for example, the existence of a (left/right) neutral element to allow for empty data, and other combinations of the algebraic properties mentioned.

2.3. Correctness Conditions for `Combiner` Functions

A `Combiner` function is a `Reducer` function whose output type coincides with its input type and that is associative and commutative (when viewed as a binary operator $\oplus$ on individual key–value pairs (k_1, v_1)). When viewed as a unary function (for example, function `C` of type `X` $\rightarrow$ `X`) on multisets, it should also be idempotent ($\forall$ `xs` : `xs` $\in$ `X` : `C`(`C`(`xs`)) = `C`(`xs`)), because it may be applied multiple times (at least, in newer versions of Hadoop). The `Combiner` function may not even be applied at all by the framework. To ensure correctness in this case, the user-defined `Reducer` function should first apply the `Combiner` function on its input, and may only then conduct an arbitrary computation.

2.4. Combinators and List Homomorphisms

Important ingredients of functional programming are higher-order functions or *combinators*—functions that have other functions as parameters or results. In this sense, they are (usually very small) algorithmic templates.

Lists in Functional Programming Usually, the basic data structure in functional programming is the immutable linked list. This is why most standard combinators can be defined on lists. Such a list, if finite, corresponds largely to a stack, which only offers access to one end of the linked data structure. For example, in the syntax of the functional language Haskell, the type of homogeneous lists with elements of type a is denoted by $[a]$.

Basic List Combinators Two of the simplest higher-order functions on this kind of lists are `map` and `reduce`. Function `map` is of type $(a{\rightarrow}b){\rightarrow}[a]{\rightarrow}[b]$. It applies a user-defined function to all elements of an input list, and it returns an output list consisting of the results of all these applications. The type of `reduce` is $(a{\rightarrow}a{\rightarrow}a){\rightarrow}[a]{\rightarrow}a$. It applies a user-defined associative binary function successively to two neighboring elements of the list. The consideration of some special cases is instructive: Because `reduce` cannot possibly be written to return any concrete element of an arbitrary type a (a type selected at invocation time), it cannot handle the empty list. Thus, we will restrict its input to non-empty lists. If we wanted to allow

Prepared using cpeauth.cls
DOI: 10.1002/cpe

also non-associative functions as parameters of `reduce`, we would have to specify the order of iteration over the list.[†]

List and Multiset Homomorphisms A *list homomorphism* is a function that operates on lists, preserving their algebraic structure. Recall that, in universal algebra, unlike in functional programming, list structure is defined through a list concatenation operator, where a list with at least two elements is viewed as a concatenation of two non-empty segments; in the extreme case, at least one of the segments is a singleton list. All list homomorphisms can be written in Haskell as the composition of a `map` before a `reduce` combinator. The function `map` exposes (massive) independent data parallelism, because its application-specific function parameter (say `f`) can be applied to all list elements in parallel. In parallel functional programming, the function `reduce` is also assumed to be data-parallel, because, exploiting the associativity of its function parameter (say `g`), it can be implemented as a balanced tree of applications of `g`; in consequence, its execution requires a logarithmic number of parallel steps. Applications of list homomorphisms will often also use a post-processing function, which shall be a constant-time, sequential function. In summary, every list homomorphism can be formulated in Haskell as [11]:

```
listHomomorphism = (postProcess) .  reduce(g) .  map(f)
```

This code uses the right-to-left function composition operator `.`, the `map` and the `reduce` combinator, and three user-defined function parameters, each put between brackets.

A *multiset homomorphism* is a function that respects the algebraic structure of multisets. The only difference to a list homomorphism is that a multiset homomorphism may disregard the order of its input, because the algebraic multiset concatenation (or better: union) operator is commutative, as we have seen above. (It follows that the function parameter, say `g`, of the combinator `reduce` must also be commutative.)

MapReduce and Combinators MapReduce does not offer the combinators `map` and `reduce` directly. Rather, it offers a `map` (or, more exactly, a `concatMap`), followed by a `groupBy` on sorted lists and a second `map` (`concatMap`), where the second `map` is often parameterized by a user-implemented `reduce` [12]. This standard case is known as a *segmented reduction* in the MPI community [13] Consequently, although MapReduce is not simply a composition of a `reduce` after `map` (as the name would suggest), there is often a `reduce` involved in MapReduce programs. For details, refer to our functional model of MapReduce (Section 3.1).

Combinators in Practical Distributed Systems We have just seen that MapReduce is rooted in functional programming with combinators. One of the Google MapReduce papers [8] even cites earlier work on the parallelization of combinators [14]. Yet, unlike the combinators used in functional programming, Google's MapReduce is available as a robust, large-scale, distributed system that is used in practice by companies all over the world (in the form of its open-source clone Apache Hadoop). In this regard, it is comparable with the *Message Passing*

[†]In Haskell, a generalized `reduce`, iterating from left to right with an additionally supplied initial value is known as `foldl`.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Interface (MPI) [13], which offers functional combinators (called *collective operations*) on distributed data, which include, for example, a variant of `reduce` and a segmented reduction. Though one can use MPI without using its collective operations, it has been the distributed platform offering functional combinators with the most practical impact before the rise of Google's MapReduce. Of course, the C-based MPI has many downsides from the point of view of functional or high-level programming, one of which is its lack of abstraction and a type system: there are no data structures, no static types (everything is a `void*`), and in the C implementation, code is not even made type-safe using run-time checks.

3. A MODEL OF MAPREDUCE

In a first step, we present a functional model of MapReduce. Based on it, we proceed to develop a cost model. Both models are inspired by the semantics of Apache Hadoop.

3.1. Functional Model

We use the functional language Haskell [15] to represent our functional model of MapReduce in a concise way, expressing all transformations as state-less data flow.

3.1.1. Types In the functional model, a MapReduce program is expressed as a function (`mapReduce`) from input to output that is parameterized with user-defined functions that employ three user-defined types. Function `mapReduce` is a generic function, parametric in the types of its input (`m`), its intermediate data (`r`), and its final output (`o`). We begin with an explanation of the types of the six user-defined function parameters of function `mapReduce`.

```
mapReduce ::
          (m -> [r])            -- Mapper
  ->    ([r] -> [r])            -- Combiner
  ->    ([r] -> [o])            -- Reducer
  -> (r -> r -> Ordering)       -- Partitioner
  -> (r -> r -> Ordering)       -- SortComparator
  -> (r -> r -> Ordering)       -- GroupingComparator
  -> [[m]]                      -- input
  -> [[o]]                      -- output
```

The first three parameters of `mapReduce` (`Mapper`, `Combiner` and `Reducer`; see the comments) describe transformations from input type `m` to intermediate type `r`, within intermediate type `r`, and from `r` to output type `o`, respectively. All three functions can produce multiple results or no result at all; this is modeled with list types as result types. In the same vein, whereas the `Mapper` input is a single value, both `Combiner` and `Reducer` take a (non-empty) group of values as input, which is passed lazily as an iterator in Hadoop MapReduce; we represent this in the functional model with the Haskell list type. We make the model abstract enough to neglect the possible distinction of key and value, resulting in a simplification of all types involved. This does not preclude the possibility to *instantiate* the model with standard MapReduce key–value

Prepared using cpeauth.cls
DOI: 10.1002/cpe

pairs—or, alternatively, with a more database-oriented view in which all data is in the keys only, and part of these data are projected away when not needed.

Comparator Functions Since we do not distinguish between keys and values, we need the remaining three function parameters (`Partitioner`, `SortComparator`, and `GroupingComparator`): they are different comparison operators, each on two values of the intermediate type `r`, returning whether their first value parameter is less than, equal to, or greater than their second parameter, according to a user-defined order. They are used by function `mapReduce` to guarantee that intermediate data is sent to the correct partition (`Partitioner`), processed together with (only) the values in the correct group (`GroupingComparator`) in this partition, and that all groups in a partition are processed in the correct order (`SortComparator`). (Note that, in Hadoop, `Partitioner` is not defined using comparison, but rather using hashing, which enables slightly faster processing; nevertheless, we abstract from this detail, for consistency with sorting and grouping.) Finally, function `mapReduce` takes as a last parameter a distributed input of values of type `m`, and produces a distributed output of values of type `o`. Using nested lists (denoted `[[a]]` for elements of type `a`), we model the fact that there are multiple values, distributed in different partitions, and that the individual values in a partition are read or written sequentially.

3.1.2. Basic Syntax We assume basic knowledge of Haskell syntax and of some standard library functions. Additionally, we use `x0 $> f x1 ... xn` to denote left-to-right function application, similarily to object-oriented method calls `x0.f(x1, ..., xn)` in Java. (Think of data flowing in the direction of the arrow head to a parameterized function that will process it.) Furthermore, we use custom variants of the Haskell functions `map`, `concat`, `groupBy`, and `mergeBy`. Their names carry the prefix `mr` (for MapReduce) to mark non-standard behavior; extra prefixes of `d` (or `s`) denote distributed (or sequential) execution, whereby multiple prefixes describe, from left to right, the computation from the outermost list level to more deeply nested levels. For example, the prefix `ds` means outer distributed, inner sequential computation. So, function `dsMap` is the `map` combinator that applies its argument function to a two-level list, whereby only the outer level is executed in a distributed manner. All the remaining functions used are described next.

3.1.3. Data flow Our functional model of MapReduce describes the data flow from input to output in different steps.[‡] The six user-defined function parameters of function `mapReduce` all end in `F` (for function), and they use shortened names (for example, `mapF` for `Mapper` function).

Let us start with a general remark on the types of the data passed between the different steps. When the level of nesting of the lists that hold the data changes between any two consecutive steps, this change is indicated in the comment after the first step with the new type of the data just produced. To enable this form of explanation, we model explicitly—by means of an extra step in both `Mapper` and `Reducer` tasks—the flattening of the innermost list level using `dMap mrConcat`, although this happens implicitly in MapReduce.

[‡]In reality, all steps are fused, and the only global synchronization happens after the shuffling.

DOI: 10.1002/cpe

```
mapReduce mapF combF redF partF sortF groupF input =
  input                                  -- [[ m ]]

  -- in Mapper tasks
  $> dsMap mapF                          -- [[[r]]]
  $> dMap mrConcat                       -- [[ r ]]
  $> dMap (mrGroupBy partF)              -- [[[r]]]
  $> dMap (repeatUntilSorted (\ir -> ir
                            $> sMap combF
                            $> sMap (oneRunOfExternalSortBy sortF)
          ))

  -- pulled by all Reducer tasks from all Mapper tasks
  $> dmrShuffle partF

  -- in Reducer tasks
  $> dMap (mrMergeBy sortF)              -- [[ r ]]
  $> dMap (mrGroupBy groupF)             -- [[[r]]]
  $> dsMap redF
  $> dMap mrConcat                       -- [[ o ]]
```

Figure 1. Data-flow implementation of the `mapReduce` function in Haskell; changes of the nesting depth are indicated in comments.

`Mapper` execution We start with the `input`, which is modeled as a two-level list with inner elements of type `m` in Figure 1. The outer level is distributed across the cluster nodes that execute the `Mapper` function in parallel, and the nested lists represent the mostly local data that are read sequentially by each node. For each input value, the first four steps are executed together in a pipeline on the same `Mapper` node. Overall, this takes place in a distributed fashion in the `Mapper` tasks on many cluster nodes (all four functions are prefixed with `d`): This way, we can also take the *global view* of all the data being processed "at once" on all nodes, producing a single global, distributed result after each of the four steps:

1. The user-defined `Mapper` function `mapF` is applied to each inner element of the input, possibly producing multiple results (of different type `r`) per input element. Consequently, an additional third list level is introduced.
2. The new third list level is then fused (using `mrConcat`) at the second list level, denoting sequentially accessed data.
3. The result is grouped according to the user-defined `Partitioner` function `partF`, again producing an additional third list level. Each of the resulting third-level lists represents a partition that will be sent to a different `Reducer` node.
4. These third-level data are sorted according to the user-defined sort function `sortF`. An external sort algorithm is used to cater for intermediate data too big to fit into the main memory available to the `Mapper` task effecting the sort. Sorting possibly requires multiple runs across the data. These runs are interspersed with calls to the user-defined `Combiner` function `combF`, whose purpose is to combine a list of multiple elements (which will be equal according to `sortF`) into a smaller list, this way reducing the size of intermediate data to be further processed.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Shuffle execution The step in the middle of the computation (`dmrShuffle`) models the global communication and synchronization between the computation by the `Mapper` tasks just described, and the subsequent computation by the `Reducer` tasks. Here, we take again a global view, modeling a completely distributed computation as a single function call. In practice, each `Reducer` task pulls "its" partition of intermediate data from each `Mapper` task. Synchronization is implied by each `Reducer` task waiting for each `Mapper` task to supply all the data needed. This synchronization cannot be avoided, because the merging procedure done in the next step is a *pipeline breaker*, a computation that will not produce output before it has consumed *all* of its input. (For simplicity, our function `dmrShuffle` may use `partF`, although this is not needed in practical implementations.)

`Reducer` execution The last four steps are executed, again, in a distributed fashion but, this time, in the `Reducer` tasks on the different nodes. In the following description, we take a local view of one `Reducer` task.

1. All sorted partitions (one from each `Mapper` task) are merged into a single local list using `sortF`.
2. The single local list is divided into different groups according to the user-defined grouping function `groupF`, creating one more level of nested list structure.
3. Each of the groups (consisting of multiple elements) is passed in one function call to the user-defined `Reducer` function `redF`, which produces, for each group, a list of output elements of type `o`.
4. Finally, all the lists produced by one `Reducer` task (for different groups) are fused (using `mrConcat`) into a single local output list (per `Reducer` task).
   The distributed list containing all these local lists forms the output of the MapReduce job.

Overall, we have crafted a data-flow model of MapReduce that is customizable via a handful of function parameters. It is as complex as needed to represent different classes of MapReduce programs (see Section 4), and as simple as possible for this undertaking.

3.2. Modelling Performance

We start with some preliminaries about the platforms on which MapReduce programs are typically executed; then, we give a general overview of the kinds of resources and costs that our cost model addresses.

Cluster Environment MapReduce is designed for cluster-local execution on cost-effective homogeneous nodes with large local hard disks. As a consequence, network latency can largely be ignored. Considering network bandwidth, the situation is different. Each node's local bandwidth is limited. And, there is another bottleneck: for large installations, full network bisection width is very costly to realize, and thus rare. Because this is likely to change, we do not consider limits of bisection width.

DOI: 10.1002/cpe

Chunked I/O An important design goal of MapReduce has been to avoid random access patterns for input/output (I/O) operations. This is in line with its functional roots: Customary data structures, such as arrays or graphs, do not require random memory access, and existing data are never updated, new data are created instead. As a consequence, MapReduce shares many properties with data-flow and stream programming, although it is batch-oriented. For I/O, this means that data are read in large, contiguous chunks using sequential disk scans; this is also the goal when writing data. Hence, caches are effective in hiding disk and memory latency, both of which are ignored by our cost model.

Resources Ignoring caches, the following resources remain to be considered: CPUs (the number of CPU cores), memory and disks (both with bandwidth, without latency, and with size), and network (with local bandwidth, without latency). Because the location of data in a MapReduce cluster can hardly be controlled, we will assume sufficient disk space on every node in the cluster, and ignore disk size. For reasons of coherence, we will also exclude memory size from our considerations.

Cost We are interested in MapReduce job latency. This is opposed to other cost measures such as throughput, or utilization, in a shared cluster or efficiency of usage of individual resources, as achieved by low-level improvements in the implementation of the Hadoop framework. We define cost as the minimum latency of MapReduce job execution on an otherwise unused cluster.

Skew in Data The minimum in this cost measure corresponds to the best case, which is never attained in practice because of skew in the data: the division of the job into tasks is not perfect because it is driven only by the input size and not by the processing effort needed. Consequently, some tasks take longer to complete than others. Yet, in practice, there is often only a small deviation. This is partly due to an optimization that executes speculation a redundant copy of the slowest tasks that have yet to complete (see the discussion on "stragglers" [8]). But, often enough, there is sufficient skew in either the input or the intermediate data, to let the MapReduce job in question will never come close to the best case because some of the tasks in its execution will need considerably longer than the average to complete. We specifically do *not* model this aspect.

Performance Portability Furthermore, we assume the same, fixed processing speed for all CPU cores; we do *not* model the performance of programs ported to different hardware (despite the fact that porting a Hadoop program is easy because it is written in run-anywhere Java for a framework that hides most machine details).

In summary, we restrict ourselves in several dimensions. Nevertheless, the experiments confirm that our model of MapReduce performance is relevant, (see Section 5).

3.3. Cost Model

Given the considerations in Section 3.2, we propose the analytical model of the performance of MapReduce programs, as shown in Figure 2. It is a linear model for costing time, so the

Prepared using cpeauth.cls
DOI: 10.1002/cpe

unit of each summand (1a–h) is time in seconds. Each summand is a product of a per-unit cost factor and the number of units affected. The unit differs between startup costs (seconds) and processing costs (seconds per byte processed).

3.3.1. Basic Parameters Let us describe the three different kinds of basic parameters, used in the formula of Figure 2, before we explain the total cost of a MapReduce *job* (a run of a MapReduce program in a given environment) and each of its summands in detail.

Input and Dependent Data (size in bytes)

inputSize is the size of the job's input in bytes.

mapOutputSize is the total size of the output of all `Mapper` tasks, directly after applying the `Mapper` function only.

combineOutputSize is the total size of the output of all `Mapper` tasks, after application of the `Combiner` function (if applied; otherwise, it is equal to *mapOutputSize*).

outputSize is the total size of the job's output.

These sizes enter into calculations involving the size of data processed by different kinds of tasks.

Cluster Configuration Parameters (various units)

numCpuCores (unitless) is the number of processor cores in the cluster.

chunkSize (size in bytes) is the size of a chunk of data in an I/O operation (about 64 *MByte*).

These parameters will be used to calculate the number of tasks into which the MapReduce job is divided.

$$
\begin{aligned}
cost = & && \text{(1a)}\\
& cost_{\text{jobStartup}} && \text{(1b)}\\
+ \quad & cost_{\text{taskStartup}} * \frac{(\frac{inputSize}{chunkSize} + numReducers)}{numCpuCores} && \text{(1c)}\\
+ \quad & (cost_{\text{readDFS}} + cost_{\texttt{Mapper}}) * \frac{inputSize}{numCpuCores} && \text{(1d)}\\
+ \quad & (cost_{\text{sort}} + cost_{\texttt{Combiner}}) * \frac{mapOutputSize}{numCpuCores} && \text{(1e)}\\
+ \quad & cost_{\text{write}} * \frac{combineOutputSize}{numCpuCores} && \text{(1f)}\\
+ \quad & (cost_{\text{readNet}} + cost_{\text{merge}} + cost_{\texttt{Reducer}}) * \frac{mapOutputSize}{numReducers_{\text{Eff}}} && \text{(1g)}\\
+ \quad & cost_{\text{writeDFS}} * \frac{outputSize}{numReducers_{\text{Eff}}} && \text{(1h)}
\end{aligned}
$$

Figure 2. The MapReduce cost model.

DOI: 10.1002/cpe

Program Parameters (unitless)

numReducers is the number of `Reducer` tasks requested by the program.

$numReducers_{\text{Eff}}$ is the number of `Reducer` tasks used effectively: those that receive data to process; it can be, at most, *numReducers* (in the best case); it may be smaller, if there are `Reducer` tasks that receive little or no data groups to process. Its actual value depends on how well the user-defined `Partitioner` function divides the intermediate data into chunks of equal size.

Parameter *numReducers* is the most important tuning parameter for many MapReduce programs but, in practice, the number of `Reducer` tasks that do useful work during a job may be smaller, which is modelled by parameter $numReducers_{\text{Eff}}$.

3.3.2. Explanation of Each Cost Term Next, we explain the total cost of running a MapReduce program and each of the cost summands (1a–h) in detail, including the basic costs from which the cost summands are composed.

Absolute Performance (time in seconds)

cost (1a) is the estimated (minimum) latency of the execution of a MapReduce job, consisting of a MapReduce program (which may have been defined just-in-time before execution), a cluster configuration (or the default configuration for local single-threaded execution with one `Reducer` task only), and an input to be processed.

We define the total cost as the sum of individual startup costs and processing costs, as described below. This implies a sequencing and barrier synchronization between the different processing steps which, in practice, occurs only between some of these steps. As a consequence, we can model CPU-bound as well as I/O-bound jobs, but we will over-estimate the costs of jobs that need much processing and much I/O. We could model this over-estimation with some maximum operators, but decided that this would complicate our model unnecessarily.

Startup Costs (time in seconds)

The startup costs depend on the cluster configuration, especially, on whether execution is local or distributed.

$cost_{\text{jobStartup}}$ (1b) is the cost incurred when submitting a MapReduce job, including the time needed by the `Splitter` function (see Section 3.3.3 for more detail) to compute the split boundaries of all files in the input; it can be ignored for realistic input sizes.

$cost_{\text{taskStartup}}$ (1c) is the cost incurred when starting an additional task in a job. For local execution, is it negligible, whereas it amounts to some time for distributed execution; it can be ignored if each input file's size is larger than *chunkSize*, the input is divided with a standard Hadoop `Splitter` into large partitions, and thus the processing time of each task outweighs its startup costs. This cost must be multiplied with the (average) number of tasks (`Mapper` plus `Reducer` tasks) per CPU core (*numCpuCores*). The number of

Prepared using cpeauth.cls
DOI: 10.1002/cpe

`Reducer` tasks is specified explicitly by parameter *numReducers*, whereas the number of `Mapper` tasks depends on the `Splitter` used, which, by default, divides each file into chunks of size of, at most, *chunkSize*. This is an under-approximation, as we abstract slightly from the fact that the input consists often of multiple files, whose sizes may not be multiples of *chunkSize*. So, in practice, there may be one additional small chunk per input file.

Processing Costs (1/throughput in seconds/byte)

$cost_{\text{readDFS}}$ (1d) is the cost of reading a byte from the distributed file system (DFS). In most cases, the MapReduce framework schedules `Mapper` tasks to be executed on the node on which their input data are located; then, it is equal to the cost of a local read. Like all other read and write costs, it includes (de-)serialization overhead.

$cost_{\texttt{Mapper}}$ is the cost of executing the `Mapper` function on the byte of input just read. It is often negligible, but it can be arbitrarily large, depending on the `Mapper` code (for example, in the extreme case of running a complete, possibly non-terminating sequential program in a single `Mapper` task). The sum of these two costs has to be multiplied with the (average) size of the input (*inputSize*) that is processed in parallel, per CPU core (*numCpuCores*).

$cost_{\text{sort}}$ (1e) is the cost of sorting a byte externally (which is necessary because the entire task output may not fit into the main memory available to the `Mapper` task).

$cost_{\texttt{Combiner}}$ is the cost of executing the `Combiner` function on a byte of input. As for the `Mapper` function, this cost depends on the application code. The sum of the last two cost items has to be multiplied with the size of the data on which they operate (*mapOutputSize*), which is often similar to the size of the input; but, in some cases, it is an order of magnitude larger, and, in some cases, it is considerably smaller. Furthermore, the cost model makes the assumptions that sorting is linear and that it only occurs before the `Combiner` function sees the data. Both these assumptions are gross abstractions: Hadoop MapReduce uses external QuickSort and MergeSort, and the `Combiner` function is applied after each of the normally multiple rounds of external sorting in practice. So, in the worst case, in which the `Combiner` function is superfluous because it is the identity on its input, it will be executed in each round of external sorting on the entire intermediate data, wasting resources.

$cost_{\text{write}}$ (1f) is the cost of writing a byte to local disk. It is multiplied with (*combineOutputSize*), the size of the data on which it operates which, in most cases, will be considerably smaller than (*mapOutputSize*), and often a constant.

$cost_{\text{readNet}}$ (1g) is the cost of reading a byte that has to be transmitted across the cluster network.

$cost_{\text{merge}}$ is the cost of externally merging a byte from pre-sorted inputs just read from the network. This establishes the grouping of the complete input of one `Reducer` task into

Prepared using cpeauth.cls
DOI: 10.1002/cpe

different groups, which will then be processed by the `Reducer` function, one group at a time. Additionally, it establishes the ordering in which the groups are handed over to the `Reducer` function, and can also be used to establish some order inside each group.

$cost_{\texttt{Reducer}}$ is the cost of executing the `Reducer` function on a byte of input. As for the `Mapper` and `Combiner` functions, this cost depends on the application code.

$cost_{\text{writeDFS}}$ (1h) is the cost of writing a byte to the distributed file system. This is a local write plus, typically, two additional redundant copies on other nodes in the same cluster.

3.3.3. Relation to the Functional Model Our cost model is based on our functional model. Yet, there are some differences.

First, there are some functions in the functional model that are not explicit in the cost model. This includes the user-defined comparison functions `partF`, `sortF`, and `groupF`. These functions are comparison operators on user-defined data. As such, they should take constant time per comparison, and their total cost should be subsumed by the costs of sorting and merging, which are already part of our model. Recall that the choice of function parameter `partF` influences the value of $numReducers_{\text{Eff}}$ heavily, so its effect is already incorporated indirectly in our cost model. In the same vein, we have also mentioned the Hadoop `Splitter` function in the cost model without attributing a separate cost to it. This is legitimate because the `Splitter` function induces negligible overhead when it computes the split points for each input file; later on, this guarantees that each task has at most $chunkSize$ input data to process, so the data will be more evenly distributed between tasks. Furthermore, we also do not attribute costs to `dmrConcat` because, in practice, it does not constitute an extra step of computation, but rather happens on the fly without incurring a discernible cost.

Second, there are additional cost terms that do not refer explicitly to elements of the functional model. This includes the startup costs of jobs and tasks (1a–b), which are considerable for some MapReduce jobs, although they do not appear in a simplified data-flow view. More importantly, our cost model contains individual I/O costs for input, intermediate, and output data, which takes the amount of data transferred at the different stages of a MapReduce computation into account. We believe that an explicit inclusion of I/O in the functional data-flow model would complicate this model unnecessarily, whereas, in the cost model, I/O is too important to omit.

4. OPTIMIZATION OF MAPREDUCE PROGRAMS

To demonstrate the feasibility of our functional model and our cost model, we use them as a basis for formulating optimization rules for MapReduce programs. The transformations that we discuss in this section aim mainly at performance optimization. Nevertheless, the transformations are undirected: they can be applied forward, in the direction of target code, to optimize performance or, alternatively, backward, in the direction of source code, to refactor MapReduce programs into a more modular and less tangled form. We formulate two pairs of

DOI: 10.1002/cpe

transformation rules for MapReduce programs—one pair for each of the algorithm classes considered, as we explain next.

Classes of Algorithms Considered We concentrate on two important classes of homomorphisms that are compatible with MapReduce, namely, homomorphisms on multisets (Section 4.3) and on lists (Section 4.4), both with one restriction concerning resource consumption: they may only produce a single result of constant size. To produce a global result, as homomorphisms do, the MapReduce programs considered here must have a single `Reducer` task in the final program step.[§] This is in line with the common view that, in the MapReduce world, only linear-time, constant-space algorithms are considered to be really scalable. More precisely, the size of the data processed by each parallel task created should be bounded by a constant. As a consequence, all programs that do not reduce the volume of data massively—using heavy filtering (in the `Mapper` function) or heavy reduction (in the `Combiner` function)—need multiple `Reducer` tasks in the first MapReduce program step. This also amounts to producing only a single result of constant size in the single `Reducer` task in the last program step, for example, a single maximum value or a single count of entities of some kind. Results that are more complicated are also allowed, for example, a histogram of values (which consists of a fixed number of counts of entities of some kind). Results whose size depends on the size of the input are not considered. This excludes many simple, embarrassingly parallel `map`-only problems, which is okay because we do not consider them to be very interesting. It also excludes interesting complex data transformations with big distributed results, which form only a small percentage of MapReduce programs executed, although the MapReduce programming model really excels at them. In contrast, we include most reporting and summarization problems, which makes our optimizations very relevant in practice.

4.1. Implementing Homomorphisms Using MapReduce

First, we show how to implement basic versions of both classes of homomorphisms in MapReduce. We start with multiset homomorphisms and then extend the approach to include preservation of order, thus handling list homomorphisms as well.

From Multiset Homomorphisms to MapReduce Programs MapReduce does not directly offer the combinators `map` and `reduce`, which are normally used to implement list homomorphisms (see Section 2.4). Let us demonstrate how to replace these combinators with appropriate MapReduce counterparts. Because we need a single global result, computed by taking all input values into account, we can only use one segment of the segmented reduction, which must contain all data. Then, although all `Mapper` tasks can work in parallel in a distributed fashion, the real work is done sequentially by a single `Reducer` task. So, the simplest MapReduce implementation of a homomorphism is effectively sequential. Furthermore, because we want only a single group in the only segment, we need to regard all keys as equal during grouping. If the keys are not used in computing the result, then the

[§]There is no such restriction on the other program steps in these MapReduce programs.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

simplest way to guarantee this is to project all input keys to the same constant intermediate key in the `Mapper` function.[¶] Then, the function subject to a `map` in a homomorphism has to be applied to the current value by the `Mapper`. Similarly, the function subject to a `reduce` in a homomorphism has to be applied to any two values in the iterator by the `Reducer`—so, the programmer needs to implement also the `reduce` combinator in the `Reducer`.

From List Homomorphisms to MapReduce Programs Next, we show how to extend our approach of creating MapReduce programs from multiset homomorphisms to list homomorphisms. To implement list homomorphisms in MapReduce, we have to take special care to preserve the order of input elements in all steps. To begin, we need some representation of the input list order. We use positions (or *indices*), for simplicity. In Hadoop MapReduce, there is no notion of a global index for a datum in the input; to remedy this limitation, we assume that the input has been preprocessed and each element is carrying a globally unique index, which is exactly 1 greater than the index of its predecessor (we will call such indices *contiguous*). An even better (but more costly, and thus not pursued) solution would be to use ranges of begin and end indices, because they allow to represent exactly the segments on which list homomorphisms work.

4.2. Optimization Rules

We can now proceed to the formalism that we use to describe optimization rules for MapReduce programs.

Rule Notation Formally, we denote a program transformation rule as follows. A tuple of original program steps is transformed (denoted by a horizontal line) into a tuple of new program steps. Often, these tuples only have a single element, in which case we omit the enclosing parentheses. In all rules, we denote variables in *italics*, and constants and predicates in sans-serif font. As a simple but not very useful example, Rule EXAMPLE, which divides, under a specific condition, a MapReduce program s into two MapReduce program steps i and t, would look as follows:

$$\frac{\mathsf{isCompositionOf}(s, (i, t))}{s \;\rightarrow\; (i, t)}\,(\textsc{Example})$$

In the condition (premise) of a rule, above the horizontal bar, we use binary relations (for example, isCompositionOf), whose names hint at their meaning when they are read as infix relations ("s is composition of (i, t)"). Multiple conditions are combined with conjunction.

Number of `Reducer` Tasks as Parameter In addition to the parameters introduced in the functional model (Section 3.1), the transformation rules depend on the number of `Reducer` tasks specified in the MapReduce program. We specify the number of `Reducer` tasks as an additional parameter that can be either zero (0), one (1), or more than one (N). These three

[¶]Of course, alternatively, one could employ a user-defined `GroupingComparator` instead.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

cases correspond to three semantically different kinds of behavior: do not execute `Reducer` tasks, always execute a single (sequential) task, and execute *at least two* tasks (in parallel, if the cluster has more than one CPU core). The latter case corresponds to the optimal number of parallel, distributed `Reducer` tasks given the specification of the problem and the cluster at hand.

4.3. Optimization of Multiset Homomorphisms

Let us describe first the two optimization rules on multiset homomorphisms (see Section 2.4). The basic idea of the first rule is to parallelize a sequential MapReduce program, whereas the second rule optimizes this or another parallel program further by reducing communication overhead and discarding intermediate results. Before we come to the rules, we provide some technical context that is common to both rules.

Context Although there is no order in multisets, programs on multisets make special use of the three order-related parameters (`Partitioner`, `SortComparator`, and `GroupingComparator`) of our functional model (see Section 3.1).

- For grouping, all keys are considered equal (allEqualCmp). This gives us more freedom in defining keys.
- For sorting, we do the same to avoid some of the overhead associated with sorting.
- We do not change the default (hash) `Partitioner` function, because it is well suited for distributing most kinds of data to different partitions.

These order-related functions are the same for all programs treated by the multiset rules. For brevity, we omit the parameters for `SortComparator` and `GroupingComparator` from all rules. We keep the parameter for the `Partitioner` function to be able to show that, in cases that have only a single `Reducer` task, the choice of the `Partitioner` function does not matter (which we denote by an asterisk, $*$). Other than the order-related parameters, we have four parameters for each MapReduce program step: the user-defined `Mapper`, `Combiner`, and `Reducer` function, and the number of `Reducer` tasks. Because `Combiner` functions are optional in MapReduce, we model the absence of a `Combiner` function with the special value ϵ. Functionally, if we do not consider differences in performance, this amounts to specifying the identity function as `Combiner`.

4.3.1. Multiset Parallelization Rule The first rule (Rule M-Par, short for multiset parallelization) describes the transformation of a sequential MapReduce program to a potentially faster two-step MapReduce program with parallelism.[‖] The two parameters m and r of the original program describe fully the multiset homomorphism that we want to optimize.

[‖]Note that, for performance, we desire distributed execution mostly for the parallelism gained; fault tolerance and other aspects are not in the center of our discussion.

DOI: 10.1002/cpe

$$\frac{\mathsf{isCompositionOf}(m, (m', \mathsf{oneKeyMapper})) \quad \mathsf{isMapperFor}(m', (\mathsf{stdPartitioner}, \mathsf{N})) \quad \mathsf{isCombinerFor}(c, r)}{(m, \epsilon, r, \mathsf{1}, *) \;\rightarrow\; ((m', \epsilon, c, \mathsf{N}, \mathsf{stdPartitioner}), (\mathsf{oneKeyMapper}, \epsilon, r, \mathsf{1}, *))}\,(\text{M-PAR})$$

Parallelism In the original program, only the `Mapper` function is executed in parallel. We consider the original MapReduce program to be sequential, for its sequential execution of the single `Reducer` task. This judgment stems from the view that it is the `Reducer` function parameter that is responsible for the expressiveness of homomorphisms, and of much of MapReduce as well. In the resulting two-step MapReduce program, the first step is parallel (N means many `Reducer` tasks; the number is approximately one `Reducer` task per processor core), while the subsequent reduction in the second step is executed sequentially by a single `Reducer` task (1). Of course, the only step of the sequential program has a single `Reducer` task as well.

Computation on Values, only Grouping on Keys For simplicity, we have chosen to use the keys of the key–value pairs as meta-data only. So, the original program needs to store all user data in the *value* part of the key–value pairs on which it works, and to use the keys only to make the MapReduce framework use the default partitioning and grouping (and sorting) on them. This is asserted by the condition isCompositionOf, which constrains the original program's `Mapper` function m. To this end, it makes use of one special function, named oneKeyMapper, which has to be predefined to be used in MapReduce programs for homomorphisms. It maps each input key–value pair to a pair consisting of a single constant key and the value of the input. This way, we can guarantee that the output produced by this `Mapper` function only contains a single key, which will then lead to a single group to be processed by a single `Reducer` task. Likewise, the condition isMapperFor asserts that the `Mapper` function m' in the transformed program produces sufficiently many different keys to make up N different partitions of roughly equal size when using the default partitioning function (stdPartitioner). Note that, here, we benefit from the fact that all keys are considered equal for grouping. This way, we can use a key type with many different values, allowing for a balanced partitioning of keys, while still grouping and reducing all values in a partition into a single value.

Decomposition of Multiset Homomorphisms in MapReduce Rule M-PAR is a variant of a classical *bookkeeping rule* [16], for multiset homomorphisms in a MapReduce framework. The idea is to decompose the `Reducer` function r into a `Combiner` function that is executed in a parallel, distributed fashion, and a `Reducer` function for the final computation on the intermediate results. So, in the target program, we make use of segmented reduction implemented by MapReduce. This program produces multiple partial results: one local result for each of the many `Reducer` tasks. We regard these partial results as intermediate results in

Prepared using cpeauth.cls
DOI: 10.1002/cpe

a compound computation (a multi-step program or *workflow*[**]) consisting of two MapReduce programs. The parallelism in the target program has the structure of a two-level tree with the global result as the root, the intermediate results as its children, and all partitions of the initial input in the first MapReduce job as the grandchildren.

Correctness Condition for Decomposition For correctness, function c, used as the `Reducer` function in the first step of the transformed MapReduce program, must also be a `Combiner` function compatible with r as a `Reducer` function: isCombinerFor asserts that both functions implement the same reduction, and composing them in the order stated does not change the result. This is needed to guarantee that the composition of a segmented reduction and a global reduction (the two MapReduce programs in the transformed program) is semantically equivalent to a single global reduction (the original program). We need to impose this constraint on function c, not allowing arbitrary `Reducer` functions here, because a `Reducer` function in MapReduce is more general: after executing a reduction, a `Reducer` function may also execute some other function (for example, an additional `map` or `filter` function). In the same vein, note that a function parameter m for specifying a `Mapper` function is not even needed to implement global reduction; allowing only an identity function here would suffice to implement multiset homomorphisms. However, parameter m gives the user the flexibility to specify an additional preprocessing function to be applied to each input value, which is executed in the same MapReduce program step, without creating the need to add an extra program step with its associated (communication) overhead.

Cost If we compare the original program (left of the arrow) of Rule M-Par and the transformed program (right of the arrow) using our cost model, there are several things to note. First, there are two MapReduce program steps in the transformed program, so it may take up to twice the resources of the original program, including twice the time. Actually, this is the worst case, which happens only for small input sizes, in which the startup overheads outweigh any other costs. Second, through parallelization on $numReducers$ nodes, the cost associated with the execution of the `Reducer` function (see 1g in the cost model of Figure 2) is decreased by a factor of $numReducers$. In the best case, this cost dominates all other costs. For large input sizes, this is also more or less the expected typical case. So, we can expect to attain a cost reduction by a factor of between $\frac{numReducers}{2}$ and $numReducers$ when applying Rule M-Par to real programs. The resulting program can be optimized further using the following transformation rule.

4.3.2. Multiset Combiner Rule The second rule (Rule M-Comb, multiset `Combiner`) describes the transformation of a two-step MapReduce program without `Combiner` function (for example, a program produced by applying Rule M-Par above) to a (somewhat faster) single-step program that uses a standard Hadoop `Combiner` function.

[**]Although not directly modeled in the basic MapReduce papers of Google [2, 8, 9] and by Lämmel [12], multi-step MapReduce programs are very common in practice. A drawback of using them is that the programmer support provided by MapReduce implementations (libraries) is limited even for single-step MapReduce programs, and quasi non-existent for workflows. Thus, the use of multiple, chained MapReduce programs places a significant additional burden on the programmer.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

$$\frac{\begin{array}{c}\mathsf{isCompositionOf}(m, (m', \mathsf{oneKeyMapper})) \\ \mathsf{isMapperFor}(m', (\mathsf{stdPartitioner}, \mathsf{N})) \qquad \mathsf{isCombinerFor}(c, r)\end{array}}{((m', \epsilon, c, \mathsf{N}, \mathsf{stdPartitioner}), (\mathsf{oneKeyMapper}, \epsilon, r, \mathsf{1}, *)) \;\rightarrow\; (m, c, r, \mathsf{1}, *)} (\text{M-Comb})$$

Parallelism Observe that it is not obvious whether the transformed program is sequential (like the input program to Rule M-Par) or parallel (which it is, in fact). The only syntactic difference between these two programs is the presence of a `Combiner` function (c) in the transformed program. Yet, a syntactic difference may not be a semantic difference and, more importantly, a semantic difference may not be visible in the syntax.[††] The important difference here is semantic: the `Combiner` function c is reducing the size of the intermediate data by a factor large enough that the subsequent single `Reducer` task can run in very short time, shorter than that of an average `Mapper` task. If this were not the case, the transformed program would also be sequential, like the input program to Rule M-Par. Of course, this line of argument also applies to the two-step program produced by Rule M-Par, but it is easier to show using the example just discussed.

Overview Both programs, the original one on the left and the transformed one on the right side of Rule M-Comb, implement a global reduction in MapReduce. In contrast, the first step of the original program alone implements a *segmented reduction*, a local reduction in each partition/segment of input data. The idea is best explained by applying three smaller transformations in a row: first move the reduction in the first step of the original MapReduce program to the empty `Combiner` position, then move the single-`Reducer` reduction of the second step in the original MapReduce program to the now empty position of the `Reducer`, and finally omit the now empty second MapReduce program step. The first part of this transformation is only possible because the original program does *not* make use of the repartitioning and grouping facilities of MapReduce. Actually, this is an instance of a bigger pattern:

> *If, in a MapReduce program with multiple* `Reducer` *tasks, you need only the (Map and) Reduce part(s) but not the implicit Group-By or Sort-By, try to make use of a* `Combiner` *function to speed up processing.*

Details In the original program, parameters m', c, and r describe fully the multiset homomorphism that we want to optimize. A necessary correctness condition is here that the function parameter c is a `Combiner` function that is compatible with the `Reducer` function r (isCombinerFor; see above). As a slight generalization beyond multiset homomorphisms, in both the original and the transformed program, function parameter r may, again, apply also some other function (for example, an additional `map` or `filter` function) after this reduction. Much like in Rule M-Par, we also need isCompositionOf to constrain the `Mapper` function m' in the original program with respect to `Mapper` function m in the transformed program.

[††]Without changing the semantics, we could add an identity `Combiner` function in the other program, thus, removing any difference in syntax between the two programs with `Combiner` function.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Effects on Parallelism and I/O Rule M-Comb does *not* change the degree of parallelism: both the original and the optimized program are equally parallel. Only the second MapReduce program step in the original program and the `Reducer` task execution in the optimized program are sequential—and necessarily so, as described earlier. The costs for I/O operations are reduced only slightly (because we do not create redundant copies of intermediate data). The performance optimization intended here is to reduce the delay and the overhead incurred by a second MapReduce program step.

Effects on Programming Comfort Having only a single program left also avoids the hassle of programming multiple consistent MapReduce programs. As described in Section 2, `Combiner` functions have been introduced in MapReduce for a closely related kind of optimization [MR]: they shall reduce the size of intermediate data that has to be communicated from all the `Mapper` nodes over the cluster-network to all the `Reducer` nodes. Because the original MapReduce model [MR] assumes a single MapReduce program, this is the only application that could have been thought of. Yet, in our case of a chain (workflow) of two consecutive, closely related MapReduce programs, the benefit of using `Combiner` functions is even greater than in the original use case.

Cost In the case of a small input, the cost decreases by, at least, a factor of 2, because the overhead of creating an extra MapReduce job with all its tasks disappears. This is the best case for Rule M-Comb. There cannot be a cost *increase* due to additional `Combiner` runs compared to a single `Reducer` run in the original program, because the result of each application of the `Combiner` function c to intermediate data of any size has constant size. As a consequence, in the worst case, the transformed program incurs the same cost as the original program. Furthermore, because the first program step in the original program incurs much cost for the data transfer between `Mapper` and `Reducer` tasks to achieve a deterministic grouping (which is not actually needed), the transformed program will be faster for large input sizes. In summary, we expect a speedup of slightly more than a factor of 2 for most practical problem sizes.

4.4. Optimization of List Homomorphisms

Now that we have seen two optimization rules for the comparatively simple case of multiset homomorphisms, we strive to port these optimization rules to the more difficult-to-parallelize case of list homomorphisms. Recall that the basic idea of the first rule is to parallelize the execution of `Reducer` functions, whereas the second rule fuses a two-step MapReduce program, possibly created by the first rule, into a—likely faster—single-step program. In the end, we will see these rules ported from multiset to list homomorphisms but, first, we have to work out what even a basic sequential list homomorphism looks like when cast as a MapReduce program. As mentioned earlier, in MapReduce, the order of input data is not preserved by default. Yet, to implement list homomorphisms, we need exactly this feature. We will have to take extra care to preserve order in all MapReduce programs of this section, including the sequential program that is the starting point of the transformations.

DOI: 10.1002/cpe

Preserving Order in MapReduce Programs (Sequential) One might assume that the original, sequential program need not take the list order into account, because computation can proceed along the list. But, because MapReduce is inherently parallel, this is not true—at least not after the shuffle has happened in the execution of a MapReduce program: Even though the single `Reducer` function works only on a single partition, we need non-standard `SortComparator` and `GroupingComparator` functions to preserve order and create a single group. Sorting is achieved using the natural ordering on the indices (naturalCmp; this is different from the multiset case!) and, for grouping, all indices are considered equal (allEqualCmp). Of course, these two functions and the `Partitioner` function must match the types of the data to be processed; this needs to be coded manually, by overloading these functions on the data types used. Fortunately, we can even re-use the `SortComparator` and `GroupingComparator` functions from the original program in both optimized programs. For simplicity, we will also omit them from the rules for list homomorphisms. So, compared to the rules for multiset homomorphisms, we do not need to add additional parameters to each program step. As for multiset homomorphisms, once again we use keys only as meta-data.

4.4.1. List Parallelization Rule The third rule (Rule L-PAR, list parallelization) describes the transformation of a sequential MapReduce program on lists to a two-step program. It is a variant of the *bookkeeping rule* for list homomorphisms, applied to MapReduce.

$$\frac{\begin{array}{c}\mathsf{isListMapperFor}(m, (\mathsf{semiContiguousPartitioner}, \mathsf{N})) \\ \mathsf{isCombinerFor}(c, r)\end{array}}{(m, \epsilon, r, 1, *) \;\rightarrow\; ((m, \epsilon, c, \mathsf{N}, \mathsf{semiContiguousPartitioner}), (\mathsf{idMapper}, \epsilon, r', 1, *))}\,(\text{L-PAR})$$

Preserving Order (Parallel) The transformed program works in parallel on individual list segments, which form individual partitions of intermediate data. We need to take special care of the contiguity of index values in both steps of this MapReduce program.

In the first program step, we depend on the previously stated assumption of contiguous indices in the input, and use a custom partitioner function (semiContiguousPartitioner) that preserves this contiguity (as much as possible, i.e., for input sizes of at most 2^{31} records). The `Partitioner` function in Hadoop MapReduce projects the indices to the small data range of partition numbers (a Java `int` value). Furthermore, the `Mapper` function m needs to produce roughly the same number of keys in each of the different partitions for the parallelization to be effective (compare for the discussion of the parameter $numReducers_{\text{Eff}}$ in the cost model, Section 3.3.1). We combine all these constraints into the following condition: isListMapperFor(m, (semiContiguousPartitioner, N)).

In the second program step, the input data have no longer contiguous indices, because all values of one partition have been combined into a single value, with a single index as the key, leaving a gap between subsequent indices. As mentioned previously, we could resort to transforming indices to ranges, either directly in the first `Mapper` function (m), incurring an additional overhead of around 20% for storing and transferring six instead of five numeric values per record, or in the first `Reducer` function (c), which would then no longer be a `Combiner` function. We have opted for another solution: We continue working with indices,

Prepared using cpeauth.cls
DOI: 10.1002/cpe

and accept that we will not be able to detect errors (that is, intermediate data in an incorrect order) in a fail-fast manner, but only at the end of the computation. This means that, in the second `Reducer` function (r'), we reduce any values with strictly increasing indices, and we can only detect (programming) errors when there are duplicate or decreasing indices. Put simply, we perform a *relaxed* check on the indices.

Input and Output Programs The input program (left of the arrow) of Rule L-PAR consists of one program step. It has variable `Mapper` (m) and `Reducer` functions (r), no `Combiner` function, uses a single `Reducer` task (1) and any partitioner function. The output program is a two-step MapReduce program, with a parallel first step (N) and a sequential second step (1). Like the input program, it does not use `Combiner` functions. It is parameterized with the `Mapper` function (m) of the original program, a `Combiner` (c) function used as a `Reducer` function, and a second `Reducer` function (r'), as described above. In the second step, it uses an identity function as the `Mapper` function (idMapper). We could also use the oneKeyMapper function here, that we introduced in Rule M-PAR, with the advantages of not having to store a key for each record, and of more flexibility in the `Mapper` function of the original program, but we prefer to pursue an alternative approach that spares us one rule condition. Once again, because the second step is sequential, any `Partitioner` function can be used ($*$).

Additional Rule Conditions In addition to constraint isListMapperFor described above, this rule is only applicable if the `Reducer` functions c and r are in relation isCombinerFor, that is, if the first `Reducer` function c is a `Combiner` function compatible with the second `Reducer` function r.

Cost The cost estimates are very much the same here, for ordered (list) data, as those for unordered (multiset) data, except for two small differences. One difference is the extra sorting step necessary to guarantee preservation of order. Although the data are almost sorted (as they consist of a comparably small number of sorted chunks), they need to undergo the complete process of external sorting with multiple reads and writes to disk. There seems to be some opportunity for optimization in the Hadoop framework here. Yet, this work is also needed to achieve grouping alone, and so there is no difference to the case for multisets. The second difference concerns the type of data processed. In the multiset case, we do not need to store anything in the keys of intermediate data whereas, in the list case, we have chosen to store unique (and contiguous) list indices. Consequently, we incur more overhead during I/O and comparison. But the additional overhead is present in both list programs, before and after application of Rule L-PAR, so there is no difference compared to Rule M-PAR for multisets: We expect the same best and worse cases and, for real programs, we can expect to attain a cost reduction by a factor of between $\frac{numReducers}{2}$ and $numReducers$ via Rule L-PAR.

4.4.2. List Combiner Rule The fourth rule (Rule L-COMB, list `Combiner`) describes the transformation of a two-step MapReduce program on lists to a single-step program that uses a custom `Combiner` function that is run exactly once. In this rule, the two steps of the original

Prepared using cpeauth.cls
DOI: 10.1002/cpe

program together implement a global, ordered reduction in MapReduce (whereas the first step alone implements a segmented, ordered reduction).

$$\frac{\mathsf{isListMapperFor}(m, (\mathsf{semiContiguousPartitioner}, \mathsf{N})) \quad \mathsf{isMapperWithCombinerOnceFor}(mc1, (m, c")) \quad \mathsf{isCombinerFor}(c, r)}{((m, \epsilon, c, \mathsf{N}, \mathsf{semiContiguousPartitioner}), (\mathsf{idMapper}, \epsilon, r', 1, *)) \;\rightarrow\; (mc1, \epsilon, r', 1, *)} (\text{L-COMB})$$

Hadoop `Combiner` Function The idea of Rule L-COMB is, once again, to speed up processing using a `Combiner` function. Unfortunately, we cannot use Hadoop `Combiner` functions on list data for three reasons that are caused by reasonable design choices in distributed programming. First, to be able to reduce the volume of intermediate data as much as possible, Hadoop applies `Combiner` functions arbitrarily often; thus, they must be idempotent. This is problematic because our `Combiner` function does not preserve the contiguity it requires, and an alternative `Combiner` function that preserves contiguity by transforming indices to ranges is not idempotent either. Second, to keep the implementation simple and fast, Hadoop `Combiner` functions are applied to arbitrary subsets of intermediate data; thus, they must be associative-commutative. This means that non-contiguous values will occur frequently, rendering the `Combiner` function almost useless. Third, Hadoop prioritizes sorting over combining, so `Combiner` functions are only applied to groups of values that are equal according to `GroupingComparator` *and also* `SortComparator`—and, for sorting, we need groups of single values, which cannot be combined, rendering the `Combiner` function a no-op.

Preserving Order (Optimized Parallel) As a resort, we use a custom-built function (*mc1*). It is a `Mapper` function that also executes the logic of a `Combiner` function, but only once, on the complete list of intermediate values produced by the `Mapper`, and in the order in which they have been produced. This requirement is expressed by relation isMapperWithCombinerOnceFor, which is parameterized with the new `Mapper` function *mc1* and the defining `Mapper` and `Combiner` functions (here, m and $c"$). For our experiments (Section 5), we have implemented this by refactoring the `Combiner` code, thereby adding a new function that can perform the reduction in a streaming fashion, and we call this function from the `Mapper` code for each new intermediate value produced. This eliminates problems with memory management (the worry of whether the intermediate values fit into the memory assigned to the `Mapper` task), and it does not require a reimplementation of the rather complicated interface between `Mapper` and `Combiner` functions (a read-once iterator reusing mutable singleton containers for possibly serialized keys and values, grouped by some function).

Variables Rule L-COMB is parameterized with two different `Mapper` functions (m, *mc1*), and different variants of `Combiner` (c, $c"$) and `Reducer` functions (r, r'). The difference between the `Reducer` functions is that function r' only performs a relaxed check, as described above; the `Combiner` function $c"$ does not perform any check at all.

Input and Output Programs The input program is a two-step MapReduce program. It uses the functions idMapper and semiContiguousPartitioner described above. More precisely,

DOI: 10.1002/cpe

it is of the form produced by Rule L-Par. The resulting program is single-step. It uses the custom-built function *mc1* as its `Mapper` function, no Hadoop `Combiner` function (as explained above), the relaxed `Reducer` function *r'*, any `Partitioner` function, and it has only a single `Reducer` (`1`; no parallelism here). All parallelism is associated with function *mc1*.

Additional Rule Conditions isListMapperFor and isCombinerFor are the same as in Rule L-Par, with the same parameters as mentioned there; concerning the relation isMapperWithCombinerOnceFor, see the explication on preserving order given above.

Cost All arguments that we have given for Rule M-Comb also hold for Rule L-Comb. So, we expect a speedup of slightly more than a factor of 2 for most practical problem sizes.

5. EXPERIMENTS

So far, we have developed much theory: We have started with a functional model of MapReduce, continued with a cost model, and finished with four optimization rules—two rules for each of two classes of programs, resembling multiset and list homomorphisms.

We do not prove the correctness of these rules formally, an approach others have pursued [17]. Instead, we take an experimental approach to illustrate their feasibility. While a rigorous empirical study would be certainly worthwhile, applying our models and optimizations to a wide array of application is well beyond the scope of the article. Instead, we limit our attention to two instances of the two problem classes, which serve as canonical representatives for whole classes of practical applications: the Maximum problem, as an instance of a multiset homomorphism, and the Maximum Segment Sum problem, as an instance of a list homomorphism.

5.1. Research Question

We consider Hadoop Java programs and perform tests to measure performance and speedup. Our research question is: *"Do the optimization rules achieve the indicated performance gain?"* So, we report on an exploratory evaluation of the optimization rules in this section. Our conclusion is that the results justify the formal model, on which the rules are based.

5.2. Experimental Setup

In the following, we describe the code of the Hadoop MapReduce programs that we have implemented. First, we describe briefly the Java interface to Hadoop that we developed to resemble our functional model. Then, we proceed with the example programs for two problems to which we apply the optimization rules.

5.2.1. Java MapReduce Skeleton Hadoop MapReduce programs are meta-programs that make heavy use of Java reflection. In particular, the semantics and the type of result data depend on a numeric parameter (called *numReducers* in the cost model). When porting the formal

DOI: 10.1002/cpe

model of MapReduce (in Section 3.1) from Haskell to Java with Generics, we needed to create two Java instances to enable type checking: one with a Hadoop `Combiner` function and one without.[‡‡] In Java, we do represent the key–value pairs that we have abstracted away in the functional model. They are modelled as separate keys and values, forgoing the notion of being part of a pair. So, the `Mapper` function parameter in our formal model (Section 3.1) with Haskell type `m -> [r]` becomes an object parameter of our Java skeleton, which takes the form of a Java function; the object needs to be an instance of the generic Hadoop class `Mapper<K1,V1, K2,V2>`, where `K1` and `V1` are the Java types of the keys and values in the input type `m`, and `K2` and `V2` are the Java types of the keys and values in the result type `r` (the list in the result type `[r]` is not represented explicitly). When passed to Hadoop, the `Mapper` function parameter will have the even less expressive reflective type of `Class<? extends Mapper>`. In both instances of the Java skeleton, parameter *numReducers* is also passed on to Hadoop. Furthermore, we pass on some more type parameters needed by Hadoop for (de)serialization. Lastly, we use Hadoop's `Partitioner` function on keys and values instead of a third comparator. Apart from this, the two resulting Java skeletons are straightforward adaptations of our Haskell skeleton. More information on the Java skeletons are available in Appendix A as well as in Doerre et al. [18].

5.2.2. Subject Programs For each of the two pairs of optimization rules, we created an example program from the class of programs to which the first rule of the pair applies, optimized it according to this rule, and optimized it further using the second rule of the pair. This enables us to assess the performance of the original and the transformed programs. Thus, for both multiset and list homomorphisms, we have three program variants of an example program each.

Sequential: the original, unoptimized program, using one Hadoop program step; the baseline variant

TwoStep: the program after application of the corresponding parallelization rule (Rule M-Par or L-Par), using two Hadoop program steps; no `Combiner` function

Optimized: the final program after also applying the corresponding `Combiner` rule (Rule M-Comb or L-Comb), using one Hadoop program step; a suitable `Combiner` function

Multiset Homomorphism: Maximum We will first apply the optimization rules on multisets to the Maximum (Max) problem. The problem consists of finding the maximum value in a list of signed 32-bit Java `int` values. This problem represents a whole set of database aggregation operators, for example, count, sum, and average (i.e., it can be easily modified to implement these operators). Because the binary operator `max` is associative and commutative, Max is a multiset homomorphism.

Given the Java MapReduce skeleton for Hadoop, it is now straightforward to implement the Sequential Maximum problem in Hadoop, and to derive the TwoStep and the Optimized

[‡‡]Two additional instances will be needed to support MapReduce programs without a `Reducer` function

Prepared using cpeauth.cls
DOI: 10.1002/cpe

variant using the optimization rules stated in Section 4.3. Each of them makes use of the same user-defined `Reducer` function that implements a `max` computation on a list (or, more exactly, on an iterator) of intermediate values. Note that, for Maximum, the optimized parallel MapReduce program is known and easy to write without this method. This example is mainly used as an intermediate step to the more complex Maximum Segment Sum problem that follows.

List Homomorphism: Maximum Segment Sum Our next example is the Maximum Segment Sum (MSS) problem, which is defined as follows: For an input list of integers, look at each segment (a sublist containing only consecutive list elements), and compute its sum; return the maximum of these sums. We have selected MSS for several reasons: It works on lists, it is non-trivial to parallelize, using a complex operator on intermediate data and post-processing, and it has been studied extensively in parallel functional programming [3]. Nevertheless, MSS is grounded in a practical use case: Its original two-dimensional formulation was intended to be used as a simplified maximum likelyhood estimator on digital images [6]. A naïve implementation of this algorithm is of cubic time complexity. Optimal sequential implementations have linear time complexity and run in a streaming fashion (as a linear scan), requiring only constant space at any time. The algorithm described next will also have these properties when run sequentially. Nevertheless, we are concerned with parallel implementations here. MSS is both an instance of the divide-and-conquer program skeleton and a list homomorphism [3].

As with the Maximum problem, we have created three different MapReduce implementations of MSS. More information on the subject programs are available in Appendix B.

5.2.3. Test Input Data Now that we have described the test programs, let us describe the test input data which they process.

Format The test input data are made up of random 32-bit signed integer values. For consistency, we use the same input data for both problems in the evaluation. This means that, although this is only needed for list homomorphisms, all input records also contain the current (32-bit signed integer) index in the key part of a key–value pair. For ease of access, each input record is stored as the textual representation of the key and the textual representation of the value, separated by a tab and terminated by a newline.

Scaling Because Hadoop is optimized for larger files, we start testing with a size of 16 million (2^{24}) input records (see parameter *chunkSize* of our cost model, described in Section 3.3.1). We double the input size between consecutive tests, which gives us an evenly-spaced doubly-logarithmic scale. Technically, we re-use the input file(s) of the smaller size and add the same number of input files of size 2^{24} records each to reach the next (binary) order of magnitude. We stop at 2^{31} input records for two reasons. First, disk space and run time do not allow for much bigger inputs, given the cluster on which we run our experiments, as described in Section 5.3.1. Second, and more importantly, the return type of the Java `Object`'s `hashCode` method is a 32-bit signed integer (a Java `int`). It is used pervasively by the Hadoop MapReduce framework for

DOI: 10.1002/cpe

partitioning and cannot be changed easily to a bigger type. We depend on this functionality when preserving order for list homomorphisms and, for consistency, we use the same input sizes also for multiset homomorphisms.

5.3. Experiments

Next, we describe the experiments performed, to quantify the performance gains obtainable using the optimization rules.

5.3.1. Measurement Setup The measurement environment consists of hardware, software, and measurement tools.

Hardware All experiments were run on the same 16-node cluster (which has one additional master node with a similar configuration); the input was distributed across the cluster (in the Hadoop distributed file system); each node has

- $2 * 4$ CPU cores (Intel(R) Xeon(R) CPU E5405 @ 2.00GHz)
- 16 GB of RAM (plus 8 GB of swap space on hard disk)
- 1 hard disk volume (striped on $2 * 72$ GB physical disks; one node has $2 * 600$ GB)
- 1 GBit Ethernet, connected through a switch

In total, there are $16 * 2 * 4 = 128$ CPU cores in the cluster. In the optimal case, CPU-bound programs that can run fully parallel will be able to make use of this degree of parallelism. Any additional speedup is likely due to cache effects.

Third-Party Software The operating system was a 64-bit openSUSE 10.3 Linux (kernel 2.6.22.17-0.1-default). The Java Virtual Machine (from Oracle) was: Java(TM) SE Runtime Environment (build 1.6.0_15-b03) with Java HotSpot(TM) 64-Bit Server VM (build 14.1-b02, mixed mode).

Hadoop Configuration With the given data encoding, *inputSize* is 310 MB for each file of 2^{24} records. Given the *chunkSize* of 64 MB, this amounts to 6 input splits per input file. We have verified that the *outputSize* for all programs is a very small constant (some dozen bytes). We used a replication factor of 2 in the Hadoop DFS. In Hadoop MapReduce, we used at maximum 8 `Mapper` tasks and 8 `Reducer` tasks per 8-core cluster node, allowing for potentially full utilization in all phases, even if only `Mapper` or only `Reducer` tasks are at work. Following standard guidelines [*numReducers* in Hadoop Wiki], in all parallel program steps, we used 111 `Reducer` tasks (which is slightly less than the number of CPU cores).

Measuring Runtime We used the tool `GNU time 1.7` to measure the wall-clock run time, including startup overheads of the user program. The server processes on each cluster node ran for weeks, and the Java just-in-time compiler and the data caches have been warmed up with the real input data for both programs before the measurements start for each input size. We only measured each point once, because the small measurement differences that occurr have little influence on the trends that we expected to observe and compare.

DOI: 10.1002/cpe

5.3.2. Performance Results In the Figures 3 and 4, we show, for both the Max and the MSS problem, the performance of the original program and the program after applying only the first and both optimization rules, respectively.

For both problems, we used the same setup. The results are plotted on a doubly-logarithmic scale. On the horizontal axis, we have the binary (or dyadic) logarithm of the number of elements in the input, as described in Section 5.2.3; it starts at 2^{24}, because this is the minimum number of input records. The vertical axis records the binary logarithm of the program run time in seconds, as just described. The raw results are also shown in Table I.

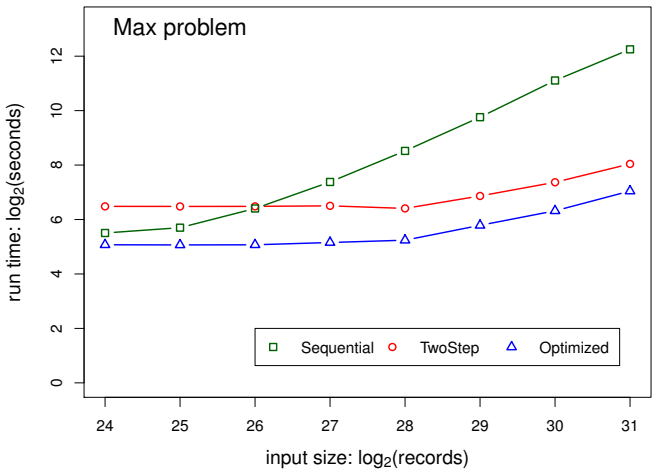

Figure 3. Wall-clock run time of the Sequential, TwoStep, Optimized programs for the Max problem on input sizes between 2^{24} and 2^{31} records.

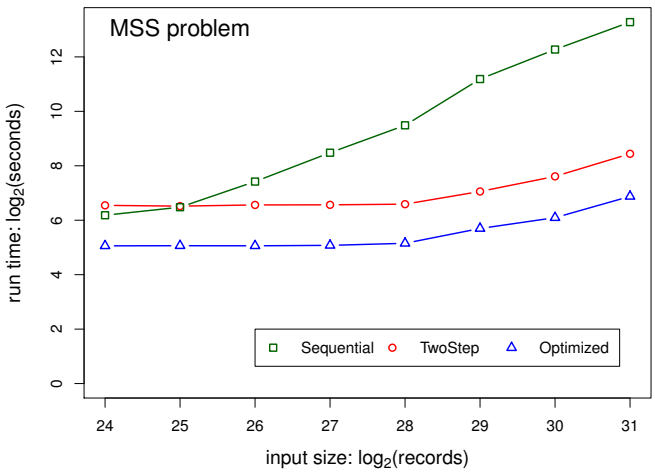

Figure 4. Wall-clock run time of the Sequential, TwoStep, Optimized programs for the MSS problem on input sizes between 2^{24} and 2^{31} records.

Max Problem For the Max problem, the TwoStep program is much faster than the Sequential program for large inputs (by a factor of $2^{4.2}$), and the Optimized program is even faster for all inputs (up to a factor of $2^{5.2}$ compared to the Sequential program).

DOI: 10.1002/cpe

MSS Problem For the MSS problem, the TwoStep program is orders of magnitude faster than the Sequential program for medium and large inputs (up to a factor of $2^{4.8}$), and the Optimized program is even faster for all inputs (up to a factor of $2^{6.4}$).

Raw data Table I shows the raw data for Figure 3 and Figure 4, again as doubly-logarithmic values. The run time for input sizes between 2^{24} and 2^{31} records is shown in log_2 seconds.

5.4. Discussion

Next, we discuss the performance gains obtained by applying the optimization rules, and the consequences for both the functional and the cost model.

Performance Overall, the results for the optimization of the Max and MSS problems are very similar. The biggest difference is that Max is up to twice as fast as MSS, especially when comparing the Sequential variants for the two problems. So, let us discuss them together. Looking at the diagrams, the Sequential variants take very long for larger input sizes (more than 2^{28} records); the TwoStep variants are much faster and the Optimized variants are the fastest by some factor of 2. For smaller input sizes (2^{24} to 2^{28} records), the execution times of the non-sequential variants do not depend on the input size, but are constant; this can be explained by the dominance of the startup overhead. For even smaller input sizes, this is likely to apply also to the Sequential variants. The TwoStep variants take around three times as long as the Optimized variants—except for the Max program with large inputs, in which case the slowdown factor is 2, not 3.

Speedup Compared to the Sequential variant, the speedup of the TwoStep program for the Max problem grows from 2^{-1} to $2^{4.2}$ with an increase of a factor of 2^7 in input size (from 2^{24} to 2^{31} records), and the speedup of the Optimized program grows from $2^{0.4}$ to $2^{5.2}$ in the same frame. Likewise, for the MSS problem, the TwoStep program shows a speedup of between $2^{-0.4}$ and $2^{4.8}$ (with a slightly different slope), and the Optimized program exhibits a speedup of between $2^{1.1}$ and $2^{6.4}$ (which is a speedup of around 85).

Table I. Raw data (log_2 of wall-clock run time in seconds) of the Sequential, TwoStep, Optimized programs for both the Max and the MSS problem on input sizes between 2^{24} and 2^{31} records.

	Max			**MSS**		
log_2 **records**	**Optimized**	**TwoStep**	**Sequential**	**Optimized**	**TwoStep**	**Sequential**
24	5.07	6.48	5.50	5.06	6.55	6.18
25	5.07	6.48	5.70	5.06	6.52	6.48
26	5.07	6.48	6.40	5.06	6.56	7.42
27	5.16	6.50	7.38	5.08	6.57	8.48
28	5.24	6.41	8.52	5.15	6.59	9.49
29	5.79	6.86	9.76	5.70	7.06	11.19
30	6.32	7.37	11.11	6.09	7.61	12.27
31	7.05	8.04	12.25	6.88	8.44	13.28

DOI: 10.1002/cpe

Interpretation The observed speedups are not unexpected. In Section 4, we describe the cost reduction associated with each optimization rule. These costs ranged from a slowdown by a factor of 2 (as exhibited by the TwoStep program variant of the Max problem on the smallest input size) to a speedup of $2 * numReducers \approx 2 * 128 = 256 = 2^8$ (theoretically, if the best cases of both rules apply to the same input size; we observed $2^{6.4}$ for the Optimized program variant of the MSS problem on the biggest input size). The data points between these extreme values also align well with our expectations.

Justification We have chosen a formal and algebraic approach to explore the possibilities of MapReduce program optimizations. Our approach has the advantage of being both better founded and able to give more practical advice than the current best practice for MapReduce program development, namely, rules of thumb as, for example, the following one, taken from the introduction of the Hadoop documentation: *If your program is slow, try to use a* `Combiner` *function.* We had to apply several simplifications in the functional and cost model and the optimization rules compared to the practical evaluation using Hadoop. We have already discussed the performance-related points in Section 3.2, Section 3.3.3 and Section 4.2. Nevertheless, the optimization rules worked. This is because, at least, for the problem classes described, the models seem to match the Hadoop programs well.

Recommendation of Use For all but trivially small input sizes, the parallelization sule M-Par and L-Par should be applied to MapReduce programs. Concerning the combiner rules M-Comb and L-Comb, the situation is less clear: The performance gain alone (a factor of 2 or 3) may not pay off, given the development effort needed to implement it. But, the target program has the extra benefit of expressing a one-step computation (in the logical/homomorphism view) by a one-step program (instead of two separate MapReduce steps). This is more intuitive, and it renders operations more manageable.

Further Applications We have demonstrated that our functional model and our cost model are useful and principled tools for formulating and reasoning about performance optimization of MapReduce programs. Beyond formulating further optimizations, the models can serve as a basis for reasoning about properties other than performance. First, the functional model can be a foundation for modeling and reasoning about functional correctness of MapReduce programs and their optimizations. Second, the cost model can be extended to cover also other non-functional properties, such as reliability and energy consumption. Overall, a formal approach to this domain—as taken by us—will help to classify, compare, inspire, and guide further work on MapReduce programming and similar models for distributed programming.

6. RELATED WORK

Since its invention, a huge amount of research has been conducted on Hadoop MapReduce optimization [10]. Yang et al. [19] state the main principles and requirements of MapReduce implementations.

DOI: 10.1002/cpe

The authors of MapReduce themselves recommend that one should take advantage of natural indices whenever possible [9]. This supports our interest in list homomorphisms, in which order plays an important role. Map-Join-Reduce extends MapReduce by user-specified join functions that allow to control the order or items during joining [20].

HaLoop optimizes the execution of MapReduce programs by caching intermediate results between MapReduce jobs [21]. This is an alternative to our approach of merging jobs structurally based on our model. Similarly to HaLoop, iMapReduce also optimizes the execution of MapReduce programs, not by caching, but by pooling and reusing suspended MapReduce jobs [22].

The MRShare system transforms a batch of queries into a new batch by merging jobs into groups and an evaluating each group as a singlen query [23]. The transformation is based on a cost model. Again, this is not really a structural program rewrite, as we do it.

Babu [24] introduces a profiler that monitors, based on instrumentation, the execution of a Hadoop MapReduce program, and a cost-based optimizer that tunes the underlying parameters of the Hadoop framework. This is parameter tuning, which is complementary to our approach of a structural program optimization. Similarly, Herodotou et al. [25, 26, 27] perform parameter tuning, in this case, based on clustering.

The Manimal system performs a static analysis to determine relational optimizations and to generate proper indexes for the raw data [28, 29], no architectural optimization, as we do it.

Several approached concentrate on increasing fault tolerance, mostly by monitoring and job re-scheduling: ParaTimer [30, 31], LATE [32], RAFT [33, 34] and Hadoop++ [35]. Several systems perform optimization on logical query plans, much like in ordinary databases: FlumeJava [36], Pig Latin [37] and Tenzing [38].

Finally, some tools provide declarative interfaces to Hadoop, but these are rather SQL-like data processing interfaces, and not architectural models like ours: Sawzall [39], Pig Latin [37], Tenzing [38], Hive [40] and SQL/MapReduce [41].

7. CONCLUSIONS AND FUTURE WORK

Google's MapReduce programming model has been one of the contributions with highest practical impact in the field of distributed computing in the recent year. It is closely related to functional programming, especially to the algebraic theory of list homomorphisms. List homomorphisms facilitate program composition, optimization of intermediate data, and parallelization. To put these theoretical benefits to practical use, we strive for a combination of the formal basis of list homomorphisms with the scalability and industrial-strength distributed implementation of MapReduce.

In particular, we have developed a formal model of MapReduce programs suitable for optimization (comprising a functional model and a cost model), an approach to model MapReduce programs that operate on lists instead of on multisets only, a total of four optimization rules for MapReduce programs formulated on top of our formal model, and a series of experiments to validate the model and the optimization rules.

DOI: 10.1002/cpe

We believe that the development of MapReduce programs, for example using Hadoop, benefits from the use of a formal functional model and cost model. Our work is a first step on the way to better understand and ease MapReduce programming, and further step shall follow:

- The power of our transformational approach rests on the fusion of MapReduce phases, which avoid barrier synchronization and the storage of intermediate data. That this fusion principle is very powerful has been demonstrated previously in the literature [3, 42]. We offer a few transformations here to prove the point that MapReduce is a suitable target for this approach. Further such transformations shall be pursued.
- Although it proved promising in our first experiments, our cost model has, so far, received limited study. For real-world applications, one will likely want to refine it—which does not necessarily mean complicating it! In the end, it were nice if MapReduce programs could be optimized automatically based on the cost model. We believe that our formal approach to modeling and reasoning about MapReduce programs would simplify this task greatly.
- Also, it is certainly useful to look beyond Hadoop and try to apply our approach to other implementations of MapReduce.

8. ACKNOWLEDGEMENTS

This work received financial support under DFG project *MapReduceFoundation*, grant no. Le 912/13-1.

References

1. Sterling TL, Savarese D, Becker DJ, Dorband JE, Ranawake UA, Packer CV. BEOWULF: A parallel workstation for scientific computation. *Proc. Int. Conf. Parallel Processing (ICPP)*, vol. 1, CRC Press, 1995; 11–14. 1
2. Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI)*, USENIX Association, 2004; 137–150. 1, 4, 6, 22
3. Gorlatch S. Toward formally-based design of message passing programs. *IEEE Trans. Software Engineering (TSE)* 2000; **26**(3):276–288. 1, 3, 30, 36
4. Gorlatch S. Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming (SCP)* 1999; **33**(1):1–27. 1
5. White T. *Hadoop: The Definitive Guide*. 3rd edn., O'Reilly, 2012. 3, 5
6. Bentley J. Programming pearls: Algorithm design techniques. *Comm. ACM* 1984; **27**(9):865–873. 3, 30
7. Bird R. Algebraic identities for program calculation. *Computer J.* 1989; **32**(2):122–126. 3
8. Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. *Comm. ACM* 2008; **51**(1):107–113. 4, 8, 13, 22
9. Dean J. MapReduce: A flexible data processing tool. *Comm. ACM* 2010; **53**(1):72–77. 4, 22, 35
10. Sakr S, Liu A, Fayoumi AG. The family of MapReduce and large-scale data processing systems. *ACM Comp. Surveys* 2013; **46**(1):44 pp. Article 11. 4, 34
11. Bird R. Lectures on constructive functional programming. *Technical Report PRG-69*, Programming Research Group, Oxford University Computing Laboratory Sep 1988. 8
12. Lämmel R. Google's MapReduce programming model – Revisited. *Science of Computer Programming (SCP)* 2008; **70**(1):1–30. 8, 22

DOI: 10.1002/cpe

13. Forum MPI. *MPI: A Message Passing Interface Standard, Version 2.1.* High-Performance Computing Center Stuttgart, 2008. 8, 9
14. Gorlatch S. Systematic efficient parallelization of scan and other list homomorphisms. *Proc. European Conf. Parallel Processing (Euro-Par), Vol. II*, LNCS 1124, Springer, 1996; 401–408. 8
15. Marlow S ((ed.)). *Haskell 2010 Language Report.* haskell.org, 2010. URL `http://www.haskell.org/onlinereport/haskell2010/`. 9
16. Bird R. *Introduction to Functional Programming using Haskell.* 2nd edn., Prentice Hall Series in Computer Science, Prentice Hall Europe, 1998. 21
17. Ono K, Hirai Y, Tanabe Y, Noda N, Hagiya M. Using Coq in specification and program extraction of Hadoop MapReduce applications. *Proc. Int. Conf. Software Engineering and Formal Methods (SEFM)*, LNCS 7041, Springer, 2011; 350–365. 28
18. Dörre J, Apel S, Lengauer C. Static type checking of Hadoop MapReduce programs. *Proc. Int. Workshop on MapReduce and its Applications (MapReduce)*, ACM, 2011; 17–24. 29, 39
19. Yang HC, Dasdan A, Hsiao RL, Parker DS. Map-Reduce-Merge: Simplified relational data processing on large clusters. *Proc. Int. Conf. Management of Data (SIGMOD)*, ACM, 2007; 1029–1040. 34
20. Jiang D, Tung AKH, Chen G. Map-Join-Reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. Knowledge and Data Engineering (TKDE)* 2011; **23**(9):1299–1311. 35
21. Bu Y, Howe B, Balazinska M, Ernst MD. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endowment* 2010; **3**(1–2):285–296. 35
22. Zhang Y, Gao Q, Gao L, Wang C. iMapReduce: A distributed computing framework for iterative computation. *J. Grid Computing* 2012; **10**(1):47–68. 35
23. Nykiel T, Potamias M, Mishra C, Kollios G, Koudas N. MRShare: Sharing across multiple queries in MapReduce. *Proc. VLDB Endowment* 2010; **3**(1–2):494–505. 35
24. Babu S. Towards automatic optimization of MapReduce programs. *Proc. Int. Symp. Cloud Computing (SoCC)*, ACM, 2010; 137–142. 35
25. Herodotou H, Babu S. Profiling, what-if analsyis, and cost-based optimization of MapReduce programs. *Proc. VLDB Endowment* 2011; **4**(11):1111–1122. 35
26. Herodotou H, Dong F, Babu S. MapReduce programming and cost-based optimization? Crossing this chasm with starfish. *Proc. VLDB Endowment* 2011; **4**(12):1446–1449. 35
27. Herodotou H, Lim H, Luo G, Borisov N, Dong L, Cetin FB, Babu S. Starfish: A self-tuning system for big data analytics. *Proc. Bienn. Conf. Innovative Data Systems Research (CIDR)*, Online proceedings, 2011; 261–272. 35
28. Cafarella MJ, Ré C. Manimal: Relational optimization for data-intensive programs. *Proc. Int. Workshop on the Web and Databases (WebDB)*, ACM, 2010; 10:1–10:6. 35
29. Jahani E, Cafarella MJ, Ré C. Automatic optimization for MapReduce programs. *Proc. VLDB Endowment* 2011; **4**(6):385–396. 35
30. Morton K, Balazinska M, Grossman D. ParaTimer: A progress indicator for MapReduce DAGs. *Proc. Int. Conf. Management of Data (SIGMOD)*, ACM, 2010; 507–518. 35
31. Morton K, Friesen A, Balazinska M, Grossman D. Estimating the progress of MapReduce pipelines. *Proc. Int. Conf. Data Engineering (ICDE)*, IEEE Computer Society, 2010; 681–684. 35
32. Zaharia M, Konwinski A, Joseph AD, Katz R, Stoica I. Improving MapReduce performance in heterogeneous environments. *Proc. USENIX Symp. Operating Systems Design and Implementation (OSDI)*, USENIX Association, 2008; 29–42. 35
33. Quiané-Ruiz JA, Pinkel C, Schad J, Dittrich J. RAFTing MapReduce: Fast recovery on the RAFT. *Proc. Int. Conf. Data Engineering (ICDE)*, IEEE Computer Society, 2011; 589–600. 35
34. Quiané-Ruiz JA, Pinkel C, Schad J, Dittrich J. RAFT at work: Speeding-up MapReduce applications under task and node failures. *Proc. Int. Conf. Management of Data (SIGMOD)*, ACM, 2011; 1225–1228. 35
35. Dittrich J, Quiané-Ruiz JA, Jindal A, Kargin Y, Setty V, Schad J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endowment* 2010; **3**(1–2):515–529. 35
36. Chambers C, Raniwala A, Perry F, Adams S, Henry RR, Bradshaw R, Weizenbaum N. FlumeJava: Easy, efficient data-parallel pipelines. *Proc. Int. Conf. Programming Language Design and Implementation (PLDI)*, ACM, 2010; 363–375. 35
37. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: A not-so-foreign language for data processing. *Proc. Int. Conf. Management of data (SIGMOD)*, ACM, 2008; 1099–1110. 35
38. Chattopadhyay B, Lin L, Liu W, Mittal S, Aragonda P, Lychagina V, Kwon Y, Wong M. Tenzing A SQL implementation on the MapReduce framework. *Proc. VLDB Endowment* Aug 2011; **4**(12):1318–1327. 35

DOI: 10.1002/cpe

39. Pike R, Dorward S, Griesemer R, Quinlan S. Interpreting the data: Parallel analysis with Sawzall. *Science of Computer Programming (SCP)* 2005; **13**(4):277–298. 35
40. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R. Hive – A warehousing solution over a Map-Reduce framework. *Proc. VLDB Endowment* 2009; **2**(2):1626–1629. 35
41. Friedman E, Pawlowski PM, Cieslewicz J. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endowment* 2009; **2**(2):1402–1413. 35
42. Wedler C, Lengauer C. On linear list recursion in parallel. *Acta Informatica* 1998; **35**(10):875–909. 36

DOI: 10.1002/cpe

A. JAVA SKELETON FOR HADOOP MAPREDUCE

In this section, we provide more detail of the Java incarnation of our functional model. We pay special attention to the typing issues that arise because of the use of generic types in Java, in continuation of our work on improved static typing for MapReduce programs [18].

On the implementation side, we use Java and Hadoop for our example MapReduce programs. In our functional model (Section 3.1), we have extracted the most important parameters from the many parameters that Hadoop and other MapReduce frameworks accept. If we port the functional model to Java, the result is a Java program skeleton that is very similar to our Haskell program skeleton of MapReduce. The main difference is that, on the Java side, because of the pervasive use of reflection in Hadoop, we really have a meta-program with more flexibility than we could attain in plain Haskell. This flexibility allows us to introduce an additional parameter (a natural number) that specifies exactly the number of `Reducer` tasks to run. (We have already used this parameter in the cost model and in the optimization rules.) There are three possible kinds of behavior and associated (Java) types of result data depending on the value of this numeric parameter.

- With zero `Reducer` tasks, only the `Mapper` function is run. The user-defined `Partitioner`, `SortComparator`, `GroupingComparator`, `Combiner`, and `Reducer` functions are not used. Consequently, the user-defined type of the final output data is given by the return type of the `Mapper` function, and not by the return type of the `Reducer` function, as would normally be the case.
- With one `Reducer` task, any `Partitioner` specified produces only a single partition. Thus, the single `Reducer` task processes all intermediate data, and it has a global view of this data.
- With any larger number of `Reducer` tasks, everything happens as normal (as described in Section 2.1).

The types in last two cases can be unified, regarding a single result as a singleton list, but the MapReduce program skeleton still has two different possible result types.

No Compile-Time Checks Of course, the flexibility of selecting different behaviors, via a numeric value at run time, comes at a price. For example, unlike in Haskell (the programming language of our functional model), two MapReduce programs using the Java skeleton cannot be composed in a type-safe manner, because they are not fully type-checked until run time. One would need to take the number of `Reducer` tasks specified into account to be able to decide between the two possible result types of the first MapReduce program, and to verify type correctness. Of course, this would be possible using an external type checker; yet, in our current implementation, this is impossible to achieve at compile time, because there is only a thin wrapper library around the Hadoop Java API that uses the standard Java type-checker and its support for generics. The best we could do is to provide a separate skeleton for the case of zero `Reducer` tasks, and to throw a run-time error if the user specifies a zero value in the normal skeleton which would, thus, be restricted to accept only a number of `Reducer` tasks larger than zero as parameter. Using this approach, we could then extract the skeleton

Prepared using cpeauth.cls
DOI: 10.1002/cpe

parameters into different variants of data objects, and provide a type-safe composition function with run-time checks for them.

Types To represent the parametric types of our functional model, we have to resort to Java Generics. The drawbacks of using Java Generics are the following: they have no representation of type parameters at run time, not even in reflective values, and their interaction with subclassing is difficult. So, our Java skeleton is a Java method parameterized with six type parameters for the types of keys and values of input, intermediate and output data (Figure 5). Let us proceed to explaining the parameters of this method. Its first parameter (of type `Job`) will be explained further down. We represent the most important parameters using normal Java values to circumvent some of the problems with Java reflection mentioned. These are the `Mapper` and `Reducer` function (and the `Combiner` function in the variant of the skeleton with Hadoop `Combiner` function). We check that the `Mapper` and `Reducer` function work on the same type of intermediate data (key–value pairs with key of type `K2` and values of type `V2`). The next parameter for the (natural) number of `Reducer` tasks is represented using a Java `int`. All other parameters are reflective Java values (`Class<Type>`). For them, we also document expected nested type parameters in Figure 5 in comments (`/* ... */`). The first of these parameters is the `Partitioner` function, defined on keys and values of intermediate data. The comparator functions for sorting and grouping are specified next; they work on serialized data. Compared to the functional model, there are four additional parameters: the types of keys and values of intermediate data and of final output data, respectively. They are needed for serialization.

```
public static  <K1,V1, K2,V2, K3,V3>
  void mrSkeletonNoCombiner(Job job,
    Mapper <K1,V1, K2,V2> mapF,
    Reducer       <K2,V2, K3,V3> redF,
    int numReduceTasks,
    Class<? extends Partitioner/*<? super K2, ? super V2>*/> partF,
    Class<? extends RawComparator/*<K2>*/> sortF,
    Class<? extends RawComparator/*<K2>*/> groupF,
    Class<K2> mapKeyClass, Class<V2> mapValueClass,
    Class<K3> keyClass,    Class<V3> valueClass)
  throws IllegalStateException { /* ... */ }
```

Figure 5. The signature of a variant (without Hadoop Combiners) of our skeleton in Java, interfacing Hadoop.

Implementation The return type of our skeleton is `void`, because its implementation is based on a side-effect. It uses Hadoop's Java API to set, in a Hadoop `Job` object (the first parameter of our skeleton), the corresponding reflective values for all skeleton parameters, whether they be already reflective values or given as Java objects. This `Job` object can then be used to start a distributed Hadoop job. (As an aside: a job already running cannot be configured any further, which leads to the `IllegalStateException` mentioned in the type.)

DOI: 10.1002/cpe

In Figure 6, we show a call to this Java skeleton, which is suitable for the Sequential variant of the MSS example program. As described before, the first parameter is the Hadoop `Job` object to be configured.

```
public class MssMainIndexedSequential /* ... */ {

  Mapper<LongWritable, Text, LongWritable, LongWQuadruple>
      map = new IndexedTextToLongWQuadrupleValueMapper();
  Reducer<LongWritable,LongWQuadruple, LongWritable,LongWQuadruple>
      reduce  = new IndexedMssReducerRelaxed();

  mrSkeletonNoCombiner(job,
         map,
         reduce,
         1,
         HashPartitioner/*<LongWritable,Object>*/
             /*<Object,Object>*/.class,
         NaturalLongWritableComparator.class,
         AllEqualComparator/*<LongWritable>*/ /*<Object>*/.class,
         LongWritable.class, LongWQuadruple.class,
         LongWritable.class, LongWQuadruple.class);
  /* ... */
}
```

Figure 6. A call (from the Sequential program for the MSS problem) to the Java skeleton shown in Figure 5.

We store the `Mapper` (`map`) and `Reducer` (`reduce`) objects in separate variables make the types that they use explicit. The input (when read by the standard Hadoop `TextInputFormat`) consists of the (binary) position offset in the current input file as the key of type `LongWritable`, and the textual values of index and data item as the value of type `Text`. These are the first two type parameters of the `Mapper` object used. The text is then parsed, the index is stored in the intermediate key of type `LongWritable`, and the data item is stored as a quadruple with four identical entries in the value of type `LongWQuadruple`. This is needed to apply a homomorphism to the data: we need to store this additional information in the intermediate data to be able to re-combine partial results (for contiguous data) in multiple passes, and even to forgo the need to store all input data. The `Reducer` function implements a true reduction on this intermediate data; in consequence, it produces output of the same type. The `Mapper` and `Reducer` objects are the second and third parameters of the MapReduce skeleton (in the variant without Hadoop `Combiner` function), which guarantees that the types of intermediate data used by `Mapper` and `Reducer` function coincide. This can be checked by looking at the declarations of `map` and `reduce`. They also need to coincide with the types represented by the Java class literals that are the four last parameters of the skeleton (two for intermediate keys/values, two for output keys/values). The remaining parameters are the number of `Reducer` tasks in a sequential program (1), and the class literals of the functions for partitioning, sorting, and grouping the type of intermediate data used.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

B. DESCRIPTIONS OF SUBJECT PROGRAMS

In this section, we describe how the optimization rules (Section 4) must be instantiated for the subject programs used in the experiments (Section 5.2.2).

B.1. Multiset Homomorphism: Max

The Max problem can be defined using an associative, commutative, binary reduction operator `max` (which is the parameter of `reduce`; see Section 2.4). Commutativity entails that the order of elements in the input list (and also in possible intermediate result lists) does not matter, so the input can be regarded as a multiset. Furthermore, every reduction can be regarded as a list homomorphism with the identity function (`id`), both as the parameter to the `map` combinator and as the post-processing function. Thus, a Haskell implementation of the Max problem might look as follows:

```
maximum = (id) .  reduce(max) .  map(id)
```

Finally, every list homomorphism with a commutative reduction operator is a multiset homomorphism. So, this Haskell implementation of Max is a multiset homomorphism, too.

Max MapReduce Programs Let us now describe three different implementations of Max in MapReduce, and how to get from one to the next, using the optimization rules stated in Section 4.3.

Sequential There are two variables on the left-hand side of rule M-Par (Section 4.3.1), which have to be instantiated for the Max problem: `Mapper` function m and `Reducer` function r. The `Mapper` function m is called on each a line of the input text file. It splits the line into key and value, discards the unnecessary index in the key, parses the value into a Java `long` integer, converts it to a Hadoop `LongWritable`, and returns it together with a constant key. The `Reducer` function r iterates over the `LongWritable` values that are passed to it, converts each value into a Java `long` integer, reduces all these values using the `max` function, and finally converts the single result to a Hadoop `LongWritable` and returns it together with a constant key.

TwoStep We use rule M-Par to create a two-step parallel MapReduce program for the Max problem. On the right-hand side of rule M-Par, `Reducer` function r is already bound to the value used on the left-hand side of this rule. The additional variables that need to be instantiated are `Mapper` function *m'* and `Combiner` function c. They are constrained by the relations in the condition of the rule. The choice of `Combiner` function c is given by `Reducer` function r. Concerning `Mapper` function *m'*, there is some freedom in the set to which the input keys are mapped: we need to map the set of all input keys to a (not too) small set of different intermediate keys. This is because we need a certain number (some hundreds or thousands) of segments, which will be processed in parallel. So, we need sufficiently many keys for parallelism in the first MapReduce program step and, at the same time, not too many

Prepared using cpeauth.cls
DOI: 10.1002/cpe

groups of keys in this step, to reduce the input size in the first program step only, such that the second step does not dominate execution time. Having the `GroupingComparator` defined to yield always a single group simplifies things: we need not bother about the number of groups, and we can just re-use the input key set (`long` integer).

Optimized Now we can use rule M-Comb to derive an even more optimized, single-step parallel MapReduce program for the Max problem. `Combiner` function c and `Reducer` function r on the right-hand side of rule M-Comb are already bound by the left-hand side. As to `Mapper` function m, we can just re-use the `Mapper` function from the Sequential Max program (also named m there).

B.2. List Homomorphism: MSS

To parallelize MSS, we formulate it as a list homomorphism, a function that operates on lists while respecting their structure. To encode MSS as a list homomorphism, we need, as a parameter to the `map` function, a simple function `f` that maps a value to a quadruple (encoded as a pair of pairs), which is necessary to store additional intermediate values. These are needed if we do not process all input sequentially. Parameter `g` of the reduce function has to operate on two of these quadruples, producing a third, and will consequently be more complex than `f`. It can be expressed as a combination of one addition operator and two maximum operators. The post-processing function will then extract the first component from the single final nested pair.

MSS MapReduce Programs As with the Max programs, we now describe three different MapReduce implementations of MSS, and how to get from one to the next.

Sequential In the first variant of the MSS Hadoop program, we also have a single group and a single partition of intermediate values, as for the Max program. Additionally, the elements of the only group have to be in input order for correctness. We state this in the program by specifying an own `SortComparator` that compares elements based on their indices.

TwoStep We use rule L-Par to create a two-step parallel MapReduce program for the MSS problem. Because we have multiple partitions, it is now also important in which way the data are partitioned. By default, Hadoop uses hashing and modulo calculations here. To preserve correctness, we must only process contiguous list segments at a time (instead of lists of interleaved list elements). So, we specify an own `Partitioner` function that uses long integer division internally.

Optimized We can now use rule L-Comb to derive an even more optimized, single-step parallel MapReduce program for the MSS problem. In other words, we would like to replace the `Reducer` tasks in the first MapReduce job with a `Combiner` function, to be able to fuse the two jobs later on. Unfortunately, in Hadoop, the framework may choose to run a `Combiner` function *more* than once. This has two consequences for a MapReduce program using standard Hadoop `Combiner` functions.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

First, correctness is lost if a `Combiner` function assumes to be fed original input while it cannot be applied in a semantically correct way to intermediate results. This happens, for example, if a `Combiner` function for MSS assumes contiguous indices in its input, yet it produces non-contiguous ones, for it can only keep *one* index for all the values which it consumes.

Second, efficiency may degrade even more seriously if a `Combiner` function does not reduce the data volume much. Then, the `Combiner` function runs multiple times over largely the same data, mostly incurring costs without achieving much.

As a consequence, we cannot use the way in which Hadoop runs `Combiner` functions. Instead, we call the `Combiner` function directly from our `Mapper` function. This is a manual replacement for the former Hadoop behavior, and it assures that the `Combiner` function is only called once, thus, preserving correctness and efficiency.

DOI: 10.1002/cpe