Design-Time Performance Modeling of Compositional Parallel Programs

Fabian Czappa[a,*], Alexandru Calotoiu[b], Thomas Höhl[a], Heiko Mantel[a], Toni Nguyen[a], Felix Wolf[a]

[a]Department of Computer Science, Technical University of Darmstadt, Hochschulstraße 10, 64289 Darmstadt, Germany
[b]Department of Computer Science, Eidgenössische Technische Hochschule Zürich,Universitätstrasse 6, 8092 Zürich, Switzerland

Abstract

Performance models are powerful instruments for understanding the performance of parallel systems and uncovering their bottlenecks. Already during system design, performance models can help ponder alternative development options. However, creating a performance model—whether theoretically or empirically—for an entire application that does not exist yet is challenging. In this paper, we propose to generate performance models of full programs from performance models of their components using formal composition operators derived from parallel design patterns. As long as the design of the overall system follows such a pattern, its performance model can be predicted with reasonable accuracy without an actual implementation. We demonstrate our approach with design patterns of varying complexity, including pipeline, task pool, and eventually MapReduce, which is representative of a broad class of data-analytics applications.

Keywords: Performance Modeling, Design Pattern, MPI, MapReduce

1. Introduction

The main motivation for letting software exploit parallelism is performance, making it a first-class citizen in the development process. However, permanent pressure to reconcile functional with performance requirements poses serious challenges already during the design phase when an actual implementation is not yet available. To simplify the design of parallel software, several authors proposed design patterns to guide the creation of parallel programs [1, 2, 3]. With its origins in the field of civil engineering, the notion of design patterns has been introduced to document good solutions for recurring problems [4]. In the parallel-computing community, design patterns help identify and express parallelism on different levels, ranging from the decomposition of an abstract computational problem down to the selection of specific parallel-programing constructs.

If performance cannot be measured, it must be predicted. This is why designing efficient software requires performance models, at least as long as one lacks a running prototype that can serve as the basis for performance measurements. Formally, a performance model is an equation that describes a performance metric, usually the execution time, as a function of one or more parameters such as the size of the input data or the number of processing elements.

Deriving performance models analytically from software blueprints, that is searching for equations that accurately reflect the performance of the final product, is, unfortunately, both difficult and time consuming, as can be seen in the analytical performance models for MapReduce, e.g., in [5] and [6], which showcase the necessity for a thorough understanding of the internal scheduling mechanisms of the implementations. This is why it is rarely tried for entire programs but rather for selected kernels such as functions or loops expected to consume the majority of the compute time. More than often, software developers avoid even this and, instead, restrict their analysis to the comparison of pre-existing performance models available in the literature when they select appropriate algorithms.

For runnable code, empirical performance modeling presents an effective but less laborious alternative to analytical modeling. Empirical performance modeling learns performance models from measurements, for example, using regression. One tool that follows this approach is Extra-P [7], which we use extensively in our work to validate our method. While being much faster than analytical modeling, a prerequisite for using this technique is the ability to obtain performance measurements, which is usually not possible during the design phase we want to address here. Even during re-design, the ability to run the product only materializes after all changes have been implemented, which is often too late for major revisions.

We succeed in showing that empirical performance modeling supports the (re-)design of parallel software as long as the construction of the software follows a certain path, closely aligned with the concept of design patterns. In our approach, we exploit the idea that many parallel design patterns can be interpreted as composing an application of (at least initially) serial building blocks that represent the application logic. Very often, these building blocks are already available, for example as components of a serial program to be parallelized. Following the rules of the pattern, they are subsequently connected through communication and synchronization facilities, such as shared queues and MPI [8] intrinsics — similar to how one creates software in a data-flow model. Assuming that perfor-

---

*Corresponding author
Email addresses: `fabian.czappa@tu-darmstadt.de` (Fabian Czappa), `felix.wolf@tu-darmstadt.de` (Felix Wolf)

mance models exist at the level of these elementary building blocks, derived empirically through unit performance tests (i.e., measurements), we define for each design pattern a matching composition operator that allows the performance model of the pattern-based implementation to be constructed from the less complex performance models of the pattern components. Applying this approach recursively, we can quickly model the performance of an entire application without the need to run more than any of its components in isolation. Beyond uncovering performance bottlenecks early, our approach also helps find optimal execution configurations, for example, by suggesting the replication of slower stages in a pipeline. We summarize our contributions as follows:

- We propose a modular approach to the construction of performance models during the design of parallel software that reduces the conceptual complexity of the construction, allowing performance models of software components to be plugged together based on well-defined rules in the form of formal composition operators.

- We define composition operators for the design patterns pipeline, task pool, and MapReduce to be used in our modular construction approach.

- We use Extra-P [7] to show that our approach is working within a reasonable margin of error by comparing models constructed using our operators to empirical models of the whole program.

This work extends a previously published paper [9], expanding the construction of performance models based on building blocks and composition operators beyond simple design patterns such as pipeline and task pool to encompass also a much more complex example, the popular MapReduce [10]. This design pattern was first introduced by Google to enable distributed computations on multiple compute nodes. Unlike the simple patterns we treated before, MapReduce does not act on one item atomically. It rather accepts a list of inputs and processes each element independently from the others (like a task pool). The results are then shuffled and each resulting group is processed again, with the number of groups not having to match the number of parts the input was initially split into. Our previously used framework supported only parallel design patterns that connected computations by simply forwarding the data. This cannot handle MapReduce because of the possibly unrelated cardinalities in the different stages. As a consequence, we had to restructure and extend our framework, in the process remeasuring the old results. To evaluate MapReduce in a relevant environment, we conducted experiments with multiple compute nodes and thus introduced two different levels of parallelism. As a realistic use case to complement our testing framework, we made use of a Hadoop cluster [11] with 24 compute nodes.

Before presenting the details of our approach in Section 3, we review related work in Section 2. Then, we support our claim with performance experiments in Section 4. We give an relational theory for the defined operators in Section 5, comparing the performance of different parallelization variants of the same program. Finally, we summarize our results in Section 6.

2. Related Work

Our approach leverages the concept of design patterns for parallel programs [2]. Design patterns are a form of construction principle, which is less a formal specification than a kind of decision tree that guides the software developer through various design spaces of decreasing levels of abstraction. A well-known design pattern language for parallelism is OPL by Keutzer and Mattson [1], which distinguishes five pattern categories or design spaces: structural, computational, algorithm strategy, implementation strategy, and parallel execution patterns. Another way of looking at parallel design patterns is to think of them as algorithmic building blocks that connect serial computations according to a construction principle that makes them run in parallel. This approach was already introduced in [12], with an extended account given in [13].

In addition to simple design patterns such as pipeline and task pool, in this work we also consider MapReduce [10]. Rather than merely representing a basic construction principle, MapReduce exists in the form of frameworks that allow users to quickly design whole data-analytics applications following this pattern. Existing implementations include, for example, GrPPi [14], Phoenix++ [15], and Hadoop [11]. GrPPi and Phoenix++ only target single compute nodes with multiple threads, while Hadoop has been designed for computations spanning multiple nodes. Hadoop targets commodity clusters and is implemented in Java. It comes with a distributed file system, which takes care of transferring the input files to the compute nodes where they are processed.

The performance analysis of parallel programs has been a primary concern since the very beginning of high-performance computing. A variety of tools, including HPCToolKit [16], TAU [17], Scalasca [18], Score-P [19], and Vampir [20], can be used to gather measurements that capture the behavior of an application in a given runtime configuration, usually at the granularity of individual code regions or call paths. Many of them use profiling to summarize metrics across each code region or thread, keeping the storage requirements at a minimum. Similar to profiles, we summarize performance metrics in our analysis. However, instead of considering the runtime of particular code regions, we rather focus on data elements and the time they require to traverse the application. Basically, we look at applications from a coarse-grained data-flow perspective [21]. Therein, applications are viewed as directed graphs of blocks, where a stream of data elements flows along the arcs of the graph and the blocks implement sequential transformations on the input data elements. The key property of such blocks is that they are independent of the global memory state. Many parallel design patterns, including pipeline, task graph, data flow, fork/join, master/worker, and MapReduce, can be re-interpreted according to this model.

The functional correctness of parallel programs can also be verified in a compositional manner along design patterns. In [22], an approach is proposed for verifying parallel programs for analyzing very large security logs by exploiting the map reduce pattern. While the approach is similar in spirit, the verification of functional correctness is complementary to the com-

positional performance analysis, which we focus on in this article.

Parallel design patterns have been successfully used to understand Quality of Service attributes of complex software systems at design time using probabilistic analysis [23]. We wish to apply a similar concept towards understanding performance. To model the performance of pattern components and validate the composition operators we propose in this paper, we build on Extra-P [7], an empirical performance-modeling tool, which we tailor towards modeling the traversal time of data elements through the program.

To support the creation of performance models, tools with varying degrees of automation have been introduced [24, 25, 26]. Many state-of-the-art tools, including Extra-P, support empirical performance modeling, a method which derives performance models from measurements [27, 28, 29]. To generate performance models from measurements, a range of techniques is applied [30], many of them classifiable as machine learning, including regression, artificial neural networks, and other statistical methods such as Gaussian process regression [31]. Because the approach presented in this paper is orthogonal to the chosen method, we rely on the regression-based modeling tool Extra-P.

Extra-P automatically derives human-readable performance models from performance measurements. It is based on the assumption that the performance (e.g., the runtime) of most practical programs can be expressed as a function involving logarithmic and polynomial expressions of a parameter x, representing, for example, the number of input elements. A search space for potential models is either generated automatically [32] or can be set by the user, and a combination of regression and cross-validation is then used to find the best coefficients and select the model with the optimal fit. The resulting performance models express the execution time, number of floating point operations, bytes sent over the network or any other captured metric, in a human-readable manner, as a function of the number of processes or other performance-relevant parameters such as the problem size.

In contrast to our method, there exists extensive literature that studies analytical performance models of MapReduce. For example, Gandomi et al. [33] use Hadoop internal job metrics on heterogeneous Hadoop clusters. They, as well as Glushkova et al. [34], Yang and Sun [5], and Vianna et al. [5], also incorporate the internal scheduling capabilities and synchronization mechanisms into their performance models. Their approaches showcase the trade-off between their analytical models and our empirical one: On the one hand, they can account for non computationally spent time, on the other hand, their models require a profound knowledge of the used MapReduce implementation. Herodotou and Babu [35] provide one utilization of the performance models by making a cost aware engine for different kind of MapReduce jobs.

3. Composition Operators for Performance Models

Our goal is the modular construction of performance models for parallel applications based on our understanding of parallel design patterns—after their re-interpretation from a data-flow perspective. We first clarify our general assumptions regarding parallel design patterns and then define two specific patterns, namely pipeline and task pool that in our interpretation act on one compute node and the pattern MapReduce, that performs computations across a network of compute nodes. Furthermore, we allow sequential concatenation in our framework, which we do not see as a parallel design pattern but which will help us phrase relational claims in Section 5. We use these patterns and our implementations to demonstrate our approach. In the following section, we give a term language in which the programs we consider can be expressed. After that, we explain the performance metrics we deem appropriate for our data-flow-centric analysis. In the last step, we clarify the notion of a composition operator for performance models and define operators for the three parallel design patterns we consider in this study.

3.1. Parallel design patterns

In this work we focus on three parallel design patterns, namely pipeline, task pool, and MapReduce. In our understanding, a parallel design pattern is a construction principle that helps a developer to parallelize an application by providing hints on how and where to apply concurrent computation. Utilizing this, a developer has to simply choose an appropriate parallel design pattern and can follow a well established way to split a program into blocks in order to exploit parallelism.

We consider the individual blocks combined with the help of a pattern to be sequences of operations of variable computational intensity, without side-effects. We further assume that the implementation of a parallel design pattern has no impact on the performance of the sequential blocks, but can and will affect the performance of the system as a whole. As a simplifying assumption, we do not consider the cases where the introduced parallelism brings other bottlenecks to the surface, e.g. when multiple blocks compete for memory bandwith, mutually lowering their individual performance in the process. As a consequence to this, the number of threads used may not exceed the available hardware concurrency. We also assume that a sequential building block always takes roughly the same time to process an element, e.g., we do not consider worst case scenarios for Quicksort, and so on. Not considering hardware restraints, in general, limits our approach. However, we have decided not to focus on these restraints for now, as they do not matter from the data-flow perspective.

Below, we provide definitions of the three patterns pipeline, task pool, and MapReduce.

3.1.1. Task pool

The task pool pattern utilizes a group of worker threads to execute multiple blocks, henceforth called tasks, in parallel, decreasing the overall time in comparison to serial computation of the tasks. This way, its main feature is the separation of problem decomposition and hardware concurrency. An instance of the pattern in our work consists of two components, a queue for data elements representing the tasks and a thread pool of fixed size as shown in Figure 1 (a). The task queue stores the work
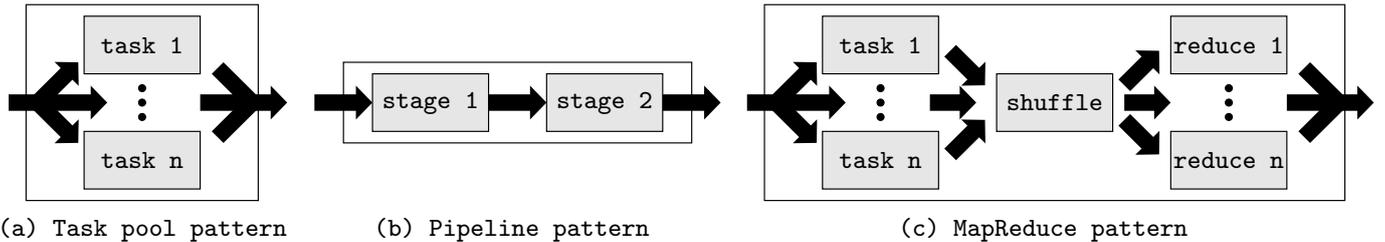
Figure 1: Parallel design patterns from a data-flow perspective.

that has to be done and the thread pool is a set of workers that pop data elements from the queue and process them. This limits the overhead for the creation and destruction of the worker threads. We write $\texttt{tpool}_n(\texttt{task})$ for a task pool that concurrently executes the block $\texttt{task}$ for each data element with n threads.

3.1.2. Pipeline

The pipeline pattern can be compared to an assembly line in a factory that creates a product in multiple stages. Once filled, all stages run in parallel, albeit working on different product instances. A pipeline is useful if the computational tasks can be expressed as a consumer-producer relationship [3]. For example when processing images in real time applying a multitude of different filters and kernels, each application can be realized as one stage. A pipeline consists of a sequence of stages where each stage corresponds to a computational task that can be either a sequential block or an instance of another parallel design pattern. Each stage consumes a data element from the prior stage and produces a data element for the next stage. Conceptually, all stages run in parallel for different data elements, as shown in Figure 1 (b). Without loss of generality, we model a pipeline as having two stages. A pipeline with more stages can be modeled as a composition of pipelines with only two stages. We write $\texttt{pipe}(\texttt{stage}_1,\texttt{stage}_2)$ for a pipeline that is composed of the stages $\texttt{stage}_1$ and $\texttt{stage}_2$.

3.1.3. MapReduce

The MapReduce pattern can express a task pool or a pipeline, but it can also express fare more complicated control flows. It is parameterized by two functions and by a count of workers. A concept of how we see MapReduce is shown in Figure 1 (c). Often, MapReduce is used to process multiple (text) files and combine the results in the reduction step, as was proposed in MapReduce [10] or used in [22].

The input to an instance of the MapReduce pattern is a list of elements, which is uniformly distributed across all workers. The first of the two functions is applied in parallel to all elements of the list. For each element, the function returns a list of key-value pairs. Across all workers, these lists are shuffled such that for every key, all elements are grouped together. Once this is completed, the second function is applied to the associated list of elements for every key, again in parallel. Thus, this pattern consists of three different stages, executed in sequence, whereas the parallelism occurs only within each stage.

In this work, we make the following assumptions regarding the workers: The workers split into a number of computation nodes which we will denote as m, having a fixed number of threads compute at each node, which we will denote as n. We make this distinction for the reason that in the structure of the pattern, the workers have to communicate with one another. This communication might be far easier between threads on a single node than between threads on different compute nodes.

We do not consider the shuffle phase parameterizable in accordance with the state of the literature and common implementations (e.g., [10], [11], [14], [15]). We interpret it as an intrinsic operation of the network due to its nature of being mostly communication and hardly any computation. For reasons we will elaborate in the evaluation, we do not consider the number of keys per instance fixed. We rather assume it to be a function $k : \mathbb{N} \to \mathbb{N}$, mapping the number of input elements to the number of keys. In agreement with this choice, we also assume the number of elements per key to be variable, however, uniform across all keys. Thus, we model elements per key as a function $d : \mathbb{N} \to \mathbb{N}$, mapping the number of input elements to the desired value. This allows us that for a fixed instance of MapReduce with x inputs to have varying a number of keys in the reduction phase ($k(x)$) and a varying number of elements per key ($d(x)$). For example, when counting occurrences of words in text files, more text files might produce more words, while simultaneously emitting more elements per key. In conclusion, we write $\texttt{mapReduce}_{m,n}^{d,k}(\texttt{map},\texttt{reduce})$ for the MapReduce instance that (1) is built from the functions $\texttt{map}$ and $\texttt{reduce}$, (2) runs on m compute nodes, each utilizing n threads, (3) uses $k(x)$ keys, each associated with (4) $d(x)$ elements, for x input elements.

3.2. A term language for structured programs

In this section, we introduce a formally defined language for specifying parallel programs constructed with design patterns. In this language, programs are described by terms. We build on our term language when presenting our approach to constructing performance models in a modular fashion. In our notation, the variables for atomic functions are grouped in the set V and named f. We use m, n, o as variables for natural numbers $\mathbb{N}$, by which we mean all positive integers, not including zero. Lastly, we use d, k as variables of type $\mathbb{N} \to \mathbb{N}$ which will allow us later to get a grip on the cardinality of the lists to reduce within MapReduce and the number of keys. We specify the term language in Backus-Naur form, in which we use T as
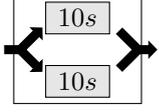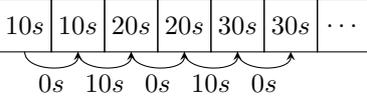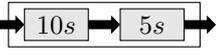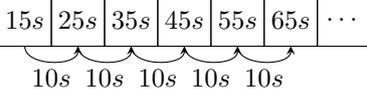
	Schematic depiction	Arrival times of the data packages		Average duration between the arrival of two data packages
Task Pool	$10s$ / $10s$	$10s$ $10s$ $20s$ $20s$ $30s$ $30s$ $\cdots$ $0s$ $10s$ $0s$ $10s$ $0s$		$\approx 5s$
Pipeline	$10s$ → $5s$	$15s$ $25s$ $35s$ $45s$ $55s$ $65s$ $\cdots$ $10s$ $10s$ $10s$ $10s$ $10s$		$\approx 10s$

Figure 2: Examples of average runtimes of individual data elements. Depicted on the top is a task pool with two different workers, each performing a task that takes 10s to complete. In a continuous flow of data packages, starting at time 0s, we expect them to arrive in pairs every 10s, i.e. the average in-between time between the arrival of two successive data packages is 5s. On the bottom a pipeline is depicted, with a task taking 10s in the first stage and a task taking 5s in the second stage, each with one worker. When considering a continuous flow of data packages traversing the pipeline starting at time 0s, we expect the first one to be completed after 15s, and from there on a package will arrive every 10s. In the limit (increasing the number of data packages), the average in-between time between the arrival of two successive data packages is 10s.

a non-terminal symbol, also being the start symbol. We include `shuffle` here to emphasize the importance of having a shuffle phase, but will omit it later on for readability.

$$T ::= \mathtt{f} \mid \mathtt{seq(T,T)} \mid \mathtt{pipe(T,T)} \mid \mathtt{tpool}_n\mathtt{(T)} \mid$$
$$\mathtt{mapReduce}_{m,n}^{d,k}\mathtt{(T, shuffle}_{m,n}^{o}\mathtt{, T)} \tag{1}$$

Intuitively, we use m to denote the number of compute nodes, n for the number of threads per compute node, o as the number of keys used in the shuffle phase, k as a function that maps the number of inputs to the number of keys, and d as a function that maps the number of inputs to the number of elements per key. We later substitute the concrete value of $k(x)$ for the placeholder o. In general, however, o can be a different number, so we do not restrict it here.

We do consider the shuffle phase in MapReduce an intrinsic to the computation cluster and do not want to allow programs that consist of a shuffle phase to be wrapped inside a task pool. Following the above rules, a structured program consists of an atomic function, sequential concatenation, or one of the above's patterns wrapped around a structured program. We summarize all constructed terms in the set $\mathcal{T}$.

3.3. Performance metrics

As discussed in Section 2, performance models are equations that represent a performance metric as a function of one or more parameters. In a first step, we identify two different performance metrics for applications that process a stream of data elements below. We then clarify how we formalize the notion in the later parts of this work.

- **Throughput:** The rate at which data elements can be processed by a program in a given time frame.

- **Latency:** The total execution time a program needs to process a single data element.

In this work, we select throughput, since we focus on applications that process large streams of data elements. However, instead of using throughput directly, we use its inverse,

that is, the average time a data element spends in the application, both considering the computations and the waiting and transport times between these computations until the next data element reaches the same stage in the workflow. We call this the average runtime of a single data element. Examples of this metric for task pool and pipeline are shown in Figure 2, an example for MapReduce is shown in Figure 3. The concrete durations of the tasks are exemplary and should only showcase our intuition for the latency of the respective patterns.

We write $M[\cdot]$ to denote the performance model for the average runtime of a single data element through the application. More formally, we define M to be a function that maps a structured program to a performance model, i.e., $M : \mathcal{T} \to (\mathbb{N} \to \mathbb{R})$. So, for any program, it returns a function of type $\mathbb{N} \to \mathbb{R}$, stating how long the program will execute depending on the number of input elements.

This performance metric enables both assessing the asymptotic complexity of the program as a whole but also predicting the specific runtime for a fixed number of input data elements. Note that our approach of compositional reasoning for performance models works for latency as well—in a similar fashion.

3.4. Composition operators for performance models

Defining a performance model for parallel programs within our pattern language amounts to defining a function $M : \mathcal{T} \to (\mathbb{N} \to \mathbb{R})$ that models the throughput of all such programs. The definition of such a performance model is a tedious process that occurs stepwise.

We propose a modular construction of such performance models. This reduces the effort substantially because one only needs to define performance models for each primitive programs (specified by a term $f \in V$ or $\mathtt{shuffle}_{m,n}^o$), and obtains performance models for more complex programs (specified by terms $t \in (\mathcal{T} \setminus V)$) for free. In the following, we assume that $M : V \to (\mathbb{N} \to \mathbb{R})$ is given, and we show how $M : \mathcal{T} \to (\mathbb{N} \to \mathbb{R})$ can be constructed from it.

Again, we cover sequential concatenation, task pool, pipeline, and MapReduce. For each of these patterns, we provide a composition operator that allows one to assemble a per-

5

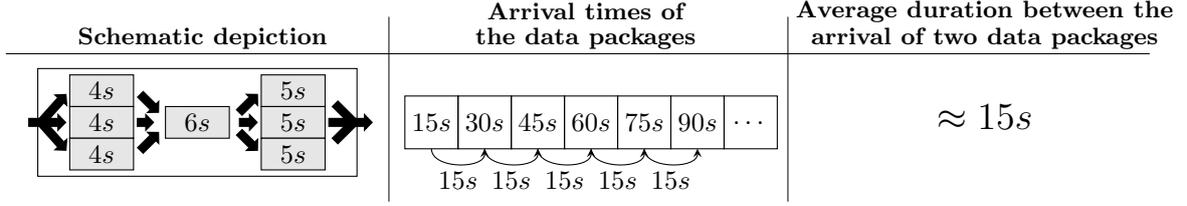| Schematic depiction	Arrival times of the data packages	Average duration between the arrival of two data packages
		$\approx 15s$

Figure 3: Example of average runtimes of individual data elements in the MapReduce pattern. Depicted is a MapReduce instance that utilizes 3 workers, and each data package is split into exactly three map and three reduce tasks. The phases themselves can be run in parallel, however, we do not assume that different phases of jobs can overlap, thus taking a black box approach. Starting at the time 0s, we assume that a data package will arrive every 15s, having to traverse each phase sequentially. The shuffle and the reduction phase can start once the other phases are completed, with no internal overlap.

formance model for a program from performance models of the components. In the construction of a performance model for any program described by a term in $\mathcal{T}$, one firstly decomposes the program recursively until one arrives at primitive programs, and secondly applies the composition operators iteratively from the bottom up.

For each composition operator, we describe the rationale behind our construction. This explanation shall clarify motivation and provide intuition. The explanation is not meant to justify that these are the only possible definitions. We justified experimentally that our choices are sensible, as we will show in Section 4.

3.4.1. General preliminaries

We will define $M : \mathcal{T} \to (\mathbb{N} \to \mathbb{R})$ in a recursive manner, each step unpacking one layer of the constructed term. For a final evaluation, we thus require values for the atomic cases $M[f]$ for $f \in V$ and $M[\text{shuffle}_{m,n}^o]$ for $m, n, o \in \mathbb{N}$ on a given hardware configuration.

3.4.2. Sequential concatenation

Given a sequential concatenation $\text{seq}(f_1, f_2)$ and performance models for the parts $M[f_1]$ and $M[f_2]$, we define the composed performance model as follows:

$$M[\text{seq}(f_1, f_2)] := \lambda x.\ M[f_1](x) + M[f_2](x) \quad (2)$$

Intuitively, executing two functions sequentially takes the time to execute the first function plus the time to execute the second function.

3.4.3. Task pool

Given a task pool $\text{tpool}_n(\text{task})$, a performance model for the sequential block $M[\text{task}]$, and a fixed number of threads n, we define the composed performance model of the task pool pattern as follows:

$$M[\text{tpool}_n(\text{task})] := \lambda x.\ 1/n \cdot M[\text{task}](x) \quad (3)$$

Intuitively, a task pool with n threads can process n data elements at once. Therefore, the average runtime of a single data element is only a nth of the average runtime of the sequential block task. In this sense, the performance model then maps the number of inputs x to the expected running time.

3.4.4. Pipeline

Given a pipeline $\text{pipe}(\text{stage}_1, \text{stage}_2)$ and performance models for the stages $M[\text{stage}_1]$ and $M[\text{stage}_2]$, we define the composed performance model of the pipeline pattern as follows:

$$M[\text{pipe}(\text{stage}_1, \text{stage}_2)] := \\ \lambda x.\ \max(M[\text{stage}_1](x), M[\text{stage}_2](x)), \quad (4)$$

where $\max(f_1, f_2)$ is defined as the function with the worse scaling behavior (in terms of O). Intuitively, the performance model of the pipeline pattern is equal to the performance model of the slower stage for a selected data element size since it will become the bottleneck of the execution. This is why we can use the asymptotic complexity of the stages to make this choice.

In some cases, the performance for a given parameter range can mean that a different selection than the asymptotic choice has to be made. This is not the case for the types of computational tasks considered in this paper, but an expansion of the composition operator to handle such cases would be similar to the way collective operations in MPI optimize runtime by selecting algorithms based on the number of ranks involved [36].

3.4.5. MapReduce

For an instance $\text{mapReduce}_{m,n}^{d,k}(\text{map}, \text{reduce})$ and performance models $M[\text{map}]$, $M[\text{shuffle}]$, and $M[\text{reduce}]$, we define the performance model of the complete pattern as the following higher order function:

$$M[\text{mapReduce}_{m,n}^{d,k}(\text{map}, \text{reduce})] := \lambda x.\frac{x \cdot M[\text{map}](1)}{m \cdot n} \\ + M[\text{shuffle}_{m,n}^{k(x)}](d(x)) + \frac{k(x) \cdot M[\text{reduce}](d(\text{x}))}{m \cdot n} \quad (5)$$

This operator, as the ones for pipeline and task pool before, returns a function that maps the number of inputs (x) to a predicted execution time.

Following the line of thought that the three conceptual phases (map, shuffle, reduce) of MapReduce flow sequentially, the execution time for the complete pattern is the summed execution time of the different stages. In the map phase, the function is applied to x many elements on $m \cdot n$ many threads in parallel. Afterwards, the shuffle phase takes care of distributing $d(x)$ elements across a network of m nodes and n threads, while having to manage $k(x)$ keys. Lastly, the reduction function is applied $k(x)$ times with $m \cdot n$ threads, each function call having to deal with $d(x)$ many elements.

4. Evaluation

Table 1: Models of sequential task blocks, measured in nanoseconds. For `inc`, the performance model for 10 repeated increments is given to counteract noise interference. For the number of input elements x, this gives the respective performance models. 1) The minimum, 0.25 quartile, median, 0.75 quartile, and maximum coefficient of variation. 2) The standard deviation.

Row	Task	Model [ns]	CV. [%] / Std. [%]
1	`nop`	5422.97	$(4.82, 22.75, 28.61, 34.4, 131.05)^{1)}$
2	`inc`	$536.185 \cdot x$	$(0.43, 1.51, 2.14, 3.05, 16.67)^{1)}$
3	`qsort`	$1034.17 \cdot x \log_2 x$	$(0.01, 0.06, 0.29, 0.48, 3.13)^{1)}$
4	$\texttt{map}_{\texttt{own}}^{\texttt{small}}$	197598	$11.76^{2)}$
5	$\texttt{map}_{\texttt{own}}^{\texttt{medium}}$	1274060	$3.03^{2)}$
6	$\texttt{map}_{\texttt{own}}^{\texttt{large}}$	11410300	$2.53^{2)}$
7	$\texttt{reduce}_p$	38316	$8.82^{2)}$
8	$\texttt{reduce}_v$	$0.353395 \cdot x$	$(4.8, 9.41, 22.53, 34.1, 34.66)^{1)}$
9	$\texttt{map}_{\texttt{grppi}}$	$1.241e7$	$6.27^{2)}$
10	$\texttt{reduce}_{\texttt{grppi}}$	$9.449e6$	$6.85^{2)}$
11	$\texttt{map}_{\texttt{hadoop}}$	$3.529e7$	$2.55^{2)}$
12	$\texttt{reduce}_{\texttt{hadoop}}$	$9376 \cdot x$	$(3.13, 3.89, 4.27, 4.78, 5.79)^{1)}$
13	`transfer`	$0.125 \cdot x$	N/A

In this section, we support the definition of the operators from Section 3 experimentally. To create examples of parallel systems, we first introduce a number of computational tasks with different complexities, which we use as the sequential building blocks for our parallel design patterns. We then create performance models using the composition operators we have introduced for these patterns.

We compare the compositional models (which we call modular) with the performance models generated by Extra-P for the entire systems (which we call monolithic), and with the actual performance measurements of the entire systems themselves. We show that the prediction errors of the modular performance models are moderate across all our experiments.

As an error metric, we choose the relative error, calculated by the difference between the predicted value and the measured value divided by the measured value. We calculate the relative error of either the actual execution time, or of the dominating scaling terms of the performance models we constructed. In the later case, we do this in order to compare the actual scaling behavior, as non-dominating terms get absorbed the larger the inputs become. This shows that our approach is flexible enough to support constructing a performance model and evaluating it at a certain point, as well as to support calculation by hand.

All experiment have been carried out on the Lichtenberg high-performance computer of TU Darmstadt, on hardware of its second upgrade phase. Each compute node of our setup includes 2 Intel XEON E5 2680 v3 processors (12 cores, hyper-threading disabled, 2.5GHz base, 3.3GHz boost which is requested to be disabled via Slurm), 64GB of DDR4-RAM running under CentOS Linux 7 (kernel 3.10.0-957.21.3.el7.x86_64). The nodes are connected via 1Gb/s Ethernet. The machine is managed by the workload manager Slurm

in version 17.02.09 and we used GCC in version 9.2.0 and Open MPI 4.0.3 with maximal optimizations for all experiments. The models are generated by Extra-P version 3.0, which is the latest publicly available version at the time of writing. To extend our previous method to also encompass MapReduce, we had to extend our testing environment and add some capabilities, which changed the atomic models for `nop`, `inc`, and `qsort` by adding a small amount (visible in the atomic model of `nop`, which takes approximately 5 microseconds now). With the changes, we now allow every task the be executed in any parallel design pattern (for example, an instance of MapReduce could be executed inside a task pool), thus complicating the way data is moved in and out of the parallel design patterns through virtual functions. We pass the actual data by relying on asynchronous communication mechanisms, adding another slight overhead to every computation. We re-produced the old results [9], which now show a slightly larger deviation between model constructed with our operators and the model generated from performance measurements of the whole program.

The measurements for all models but Hadoop were done with the high resolution clock defined in the C++ standard. While this method is subject to small influences, this is one of the most accurate methods available to us. Furthermore, reducing the clock's granularity only results in rounding errors, which we can circumvent this way. In order to compensate for the cold start up of a program which usually results in a slow first execution, we configured Extra-P to use the median of all values. Lastly, for the atomic models, we timed the functions in the same way they are called by the actual program. In the case of GrPPi, we had to disable inlining because of the way GrPPi internally calls functions.

4.1. Tasks

Here, we introduce the tasks we used to evaluate our composition operators. Because of the fundamentally different nature of task pool and pipeline on the one hand and MapReduce on the other, we chose to use a different set of tasks for evaluation. In general, the order of tasks within MapReduce cannot be reversed, disqualifying them as evaluation tasks for pipeline.

In order to quantify the variability of the measurements, we give the standard deviation or a summary of the coefficient of variation in Table 1. For Rows 1, 2, 3, 8, and 12, which are models constructed from measurements at multiple input sizes, we calculated the coefficient of variation for each such input size. We give the minimum, median, and maximum, as well as the 0.25 and 0.75 quartiles. The comparably high values for `nop` (Row 1) and $\texttt{reduce}_p$ (Row 8) are due to the fast execution overall, and thus the higher vulnerability to noise. For Rows 4, 5, 6, 7, 9, 10, and 11, we give the standard deviation, as these models were constructed for one input size. Row 13 is analytically computed, thus neither the coefficient of variation nor the standard deviation are applicable.

4.1.1. Tasks for task pool and pipeline

For simplicity, the tasks we use in the following experiments all take an array of integers as input, and return the array

Table 2: Comparison between modular and monolithic models of sequential concatenation and the task pool and pipeline patterns. Execution time is measured in nanoseconds. For the number of input elements x, this gives the respective performance models.

	Configuration	Monolithic model [ns]	Modular model [ns]	Relative error [%]
Sequential	`seq(qsort,nop)`	$1037.42 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x + 5422.97$	0.31
	`seq(qsort,inc)`	$1063.25 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x + 536.185 \cdot x$	2.74
	`seq(inc,qsort)`	$1070.71 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x + 536.185 \cdot x$	3.41
	`seq(inc,inc)`	$1030.44 \cdot x$	$1072.37 \cdot x$	4.07
	`seq(inc,nop)`	$540.897 \cdot x$	$536.185 \cdot x + 5422.97$	0.87
Task pool	`tpool`$_1$`(qsort)`	$1072.01 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x$	3.53
	`tpool`$_2$`(qsort)`	$553.72 \cdot x \log_2 x$	$517.09 \cdot x \log_2 x$	6.62
	`tpool`$_4$`(qsort)`	$291.46 \cdot x \log_2 x$	$258.54 \cdot x \log_2 x$	11.29
	`tpool`$_8$`(qsort)`	$151.31 \cdot x \log_2 x$	$129.27 \cdot x \log_2 x$	14.56
	`tpool`$_{12}$`(qsort)`	$108.34 \cdot x \log_2 x$	$86.18 \cdot x \log_2 x$	25.71
	`tpool`$_{24}$`(qsort)`	$70.25 \cdot x \log_2 x$	$43.09 \cdot x \log_2 x$	63.03
Pipeline	`pipe(qsort,nop)`	$1086.11 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x$	4.78
	`pipe(qsort,inc)`	$1105.29 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x$	6.43
	`pipe(inc,qsort)`	$1076.89 \cdot x \log_2 x$	$1034.17 \cdot x \log_2 x$	3.97
	`pipe(inc,inc)`	$677.05 \cdot x$	$536.185 \cdot x$	20.81
	`pipe(inc,nop)`	$670.32 \cdot x$	$536.185 \cdot x$	20.01

Table 3: Comparison between monolithic and modular models of the MapReduce pattern implemented in GrPPi. Execution time is in nanoseconds. For the number of input elements x, this gives the respective performance models.

Threads	Monolithic model [ns]	Modular model [ns]	Relative error [%]
$n = 1$	$2.177e7 \cdot x$	$2.186e7 \cdot x$	0.41
$n = 2$	$1.108e7 \cdot x$	$1.093e7 \cdot x$	1.35
$n = 4$	$5.705e6 \cdot x$	$5.465e6 \cdot x$	4.21
$n = 8$	$3.067e6 \cdot x$	$2.733e6 \cdot x$	10.89
$n = 12$	$2.066e6 \cdot x$	$1.822e6 \cdot x$	11.81
$n = 24$	$1.065e6 \cdot x$	$0.911e6 \cdot x$	14.46

after applying their computational task which can potentially alter the array. We selected the following tasks:

1. `nop`: Performs no operation and return the array unchanged [Average complexity is O(1)].
2. `inc`: Increases the value of each element in the array by 1. [Average complexity is O(x)]. We repeat this operation ten times to reduce the absolute noise and give the total time of the ten runs. This ten-times increment is also used in every configuration of parallel design patterns that we tested.
3. `qsort`: Sorts the array using the quicksort algorithm. [Average complexity is O($x \log_2 x$)].

We look at the average complexity rather than the worst case complexity as we are interested in the behavior across a large number of samples in a realistic execution scenario rather than specific outliers. To gather data, we varied the number of elements of the array from 1.024 to 262.144 in increments of 1.024. To ensure the statistical soundness of our results, we repeated each measurement 256 times. The runtime is expressed in nanoseconds.

The performance models of these tasks express their runtime as a function of the number of elements x in the array. The models are summarized in rows 1, 2, and 3 of Table 1.

4.1.2. Tasks for MapReduce

As the general task for MapReduce, we chose to count the frequencies of pixel-channel-values in pictures. The map task consists of taking an image and emitting a histogram of the values, i.e., a collection of key-value pairs, in which the keys are represented by the pixel-channel-values (red/green/blue in the range 0 - 255) and the values are their frequencies in the picture. The reduce task takes care of adding the values appropriately.

We decided to use only one picture and replicate it as often as needed in order to enhance the comparability of different runs. We furthermore load this image and pass only a pointer to the memory into the functions. This allows us to bypass cache-inference and I/O-blocking as much as possible, which is something we do not want to focus on.

As test image, we used picture number 15 from Weber et al. [37] in the resolutions 3072×2048, 1024×681, and 341×227, downscaled by the algorithm provided in the paper with $\lambda = 0.5$. This enables us to investigate boundaries of our approach, which are discussed below. The atomic models for this map task in our framework with small, medium, and large resolution can be found in Rows 4, 5, and 6 in Table 1.

We generally assume that one has the freedom to chose the keys freely, so we assumed the keys to be continuous in the

integers, which allows the results to be stored in a vector. For us, this means the keys represent the pixel-channel-values and take values in the range 0 - 767.

Thus, the function $k : \mathbb{N} \to \mathbb{N}$, which maps the number of inputs to the number of keys, will always be constant to 768. The function $d : \mathbb{N} \to \mathbb{N}$, which maps the number of inputs to the number of elements per key, in our case the number of occurrences of a certain pixel value in a picture.

This leaves us two different methods of reduction. We can reduce the key-value collections pairwise until there is only one left, in which case each reduction takes a constant amount of time with respect to the number of inputs, but the number of reductions increases linearly. This is the way GrPPi [14] reduces key-value collections. Orthogonally, we can reduce on a per-key basis. In this way, there is a constant amount of reduction tasks with respect to the number of inputs, but each task grows linearly. This way is used by Phoenix [15] and Hadoop [11], although they do not reduce all values for a given key at once, but rather in stages. We chose to investigate both.

The performance models for the reduce functions are given in Table 1 also. reduce_p (Row 7) stands for the performance model for reducing a pair of key-value collections, hence it has a constant performance model (adds two integers a total of 768 times). reduce_v (Row 8) stands for the performance model for reducing a vector of elements on a per-key basis, hence it has a linearly scaling performance model.

To test this method for different implementations, we henceforth used GrPPi and Hadoop. For GrPPi, the map-task performance model (only the largest picture was used) and the reduce-task performance model (the task is repeated 50,000 times to generate a test case in which the reduction function takes a significant portion of the overall running time) are given in Table 1, Rows 9 and 10. The corresponding models for Hadoop are given in Rows 11 and 12 in Table 1.

Lastly, the `transfer` (Row 13 of Table 1) is the performance model for sending bytes across the ethernet connection on the Lichtenberg high-performance computer with *mpi_send* and *mpi_recv*. In this model, x is the number of bytes.

All the models are generated by executing the tasks five times and giving these timings to Extra-P. reduce_v was tested with int vectors of size 128, 256, 512, 1024, 2048, 4096, and 8192, $\text{reduce}_{\text{hadoop}}$ was tested for 1, 2, 4, 8, 12, and 24 nodes, and `transfer` is calculated by 1Gb/s = 0.125 B/ns.

4.2. Composition operators

The composition operators we defined are adequate for all possible combinations of tasks we considered. Although we performed experiments for all combinations, we focus, for the sake of brevity, on analyzing the most relevant four configurations for pipeline and the most relevant two for the task pool.

4.2.1. Sequential concatenation

To evaluate the operator for sequential concatenation, $M[\texttt{seq}(f_1, f_2)] := M[f_1] + M[f_2]$, we chose five different configurations, namely `seq(qsort,nop)`, `seq(qsort,inc)`, `seq(inc,inc)`, `seq(inc,nop)`, and `seq(inc,qsort)`. We chose these five in accordance to the test cases of the pipeline pattern so that we can compare them later on. The cases `seq(qsort,nop)`, `seq(inc,inc)`, and `seq(inc,nop)` are accurately predicted, which is partly due to the small overhead of the implementation of sequential concatenation. In the cases of `seq(qsort,inc)` and `seq(inc,qsort)` we encountered the limitation of our approach when working with performance models constructed with Extra-P. In both cases, Extra-P returns a model with only one term, even when allowed to use multiple. For the calculation of the relative error, we only used the leading term's coefficient. We do this in order to not use approximations that convert a linear term into a fitting term of scaling behavior $x \log_2 x$. We also ignore the constant part in the modular model that is introduced by the nop operation, as we do with all constant offsets. The results are summarized in Table 2.

4.2.2. Task pool

The evaluation of the composition operator for task pool is straightforward. We initialized the task pools with a different number of threads, ranging from 1 to 24, and measured the execution time for each run. Our composition operator, $M[\texttt{tpool}_n(\texttt{task})] = 1/n \cdot M[\texttt{task}]$, predicts that the performance model is the division of the `task` performance model by the number of threads. The results show that each run yields a different execution time and as such a different performance model, as can be seen in Table 2. And indeed, the speedup achieved by the number of threads according to the model is effectively the division of the base model by the number of threads used.

Table 4: Performance neutrality of pipeline associativity and commutativity. Execution time is measured in nanoseconds. For the number of input elements x, this gives the respective performance models.

	Configuration		Monolithic model [ns]
Associativity	`pipe(pipe(qsort,inc),nop)`	Commutativity	$1057.54 \cdot x \log_2 x$
	`pipe(qsort,pipe(inc,nop))`		$1092.77 \cdot x \log_2 x$
	`pipe(pipe(qsort,nop),inc)`		$1068.45 \cdot x \log_2 x$
	`pipe(qsort,pipe(nop,inc))`		$1098.45 \cdot x \log_2 x$
	`pipe(pipe(inc,qsort),nop)`		$1060.94 \cdot x \log_2 x$
	`pipe(inc,pipe(qsort,nop))`		$1078.37 \cdot x \log_2 x$
	`pipe(pipe(inc,nop),qsort)`		$1068.17 \cdot x \log_2 x$
	`pipe(inc,pipe(nop,qsort))`		$1129.66 \cdot x \log_2 x$

4.2.3. Pipeline

For the evaluation of the composition operator for pipeline, we picked five different configurations. The composition operator, $M[\texttt{pipe}(\texttt{stage}_1,\texttt{stage}_2)] = \max(M[\texttt{stage}_1], M[\texttt{stage}_2])$, predicts that the performance model of the pipeline is the performance model of the slower stage. The models summarized in Table 2 show that the measured data support using maximum as a composition operator: the runtime and models of the pipeline where two `inc` tasks are performed are effectively the same as those of the pipeline where one `inc` task and one `nop` task is performed. Similarly, the models and measurements for the pipeline composed of a `qsort` task and a `nop` task and that of the `qsort` task and an `inc` task are the same. Therefore, the average runtime of a data element in a parallel pipeline depends only on the runtime of the stage with the highest complexity.

4.2.4. MapReduce, Hadoop and GrPPi

For the evaluation of MapReduce's composition operator, we used the following application: The input consists of bitmap pictures and the output are histograms of these pictures. The map task counts the different RGB values and their intensity from a picture. The reduce task adds all numbers for a given RGB and intensity value together.

When using MapReduce as implemented by GrPPi [14], the performance model for $M[\texttt{mapReduce}_{m,n}^{d,k}(\texttt{map}_{\texttt{grppi}},\texttt{reduce}_{\texttt{grppi}})]$ can be simplified by using the following considerations:

- GrPPi is designed to run on a single node, so $m = 1$, so no shuffling is necessary.

- GrPPi reduces the key-value-collections pairwise, starting with a neutral element. This means that the number of elements per key is fixed ($d(x) = 768$) and the number of reduce tasks is $k(x) = x$.

Considering these simplifications,

$$
\begin{aligned}
& M[\texttt{mapReduce}_{m,n}^{d,k}(\texttt{map}_{\texttt{grppi}},\texttt{reduce}_{\texttt{grppi}})](x) \\
& = \frac{x \cdot M[\texttt{map}](1) + k(x) \cdot M[\texttt{reduce}](d(\texttt{x}))}{n} \\
& = \frac{x \cdot (M[\texttt{map}](1) + M[\texttt{reduce}](768))}{n} = \frac{x \cdot 2.186e7}{n}
\end{aligned}
\tag{6}
$$

This performance model is evaluated for the thread counts $T = 1, 2, 4, 8, 12, 24$ and the results are shown in Table 3. The relative error increases with increasing thread count which does not surprise. GrPPi uses multithreading opportunities internally, which play a bigger role the more threads are participating. Overall, the relative error is below 17%, which shows that our composition operator does faithfully reflect the running time of the single node use case of GrPPi.

In the case of Hadoop [11], we do not have low-level control like with GrPPi and our own implementations. One example are data types that are sent to/ received from compute nodes, which are not Java's value types but rather custom classes. From a developer's point of view, one does not know how large these classes are and how their internal workings are in detail, so in the later calculations we assumed the custom classes to be as large as their value type counter parts. The map task does the same as before in principle: It receives a byte array and constructs a pixel-value histogram based on the array. This is done 24000 times per map task to simulate a larger input, helping to reduce the noise the distributed file system inherent to Hadoop introduces. Each map task reduces its histograms locally and has a single histogram left. The reduce task is on a per-key basis and adds all occurrence counters.

One histogram is of type *OutputCollector<LongWritable, LongWritable>*, having $768 (= 3 \cdot 256)$ entries. As noted above, we take the stance of a common developer and assume the collection has a size of $768 \cdot (8 + 8) = 12288$ bytes. The consideration in the previous paragraph lead to $k(x) = 768$ keys for Hadoop and $d(x) = x$ entries per key. To Hadoop, it does not make a difference whether the worker threads are on the same or a different compute node, so, assuming that the keys are distributed equally, the cluster has to transfer $(m \cdot n - 1)/(m \cdot n)$ of all data in the shuffle phase. This formula does ignore optimization of communication, it only calculates the bare minimum of bytes that have to be sent. This amount is a lower bound on the number of bytes that have to be transferred without compression. Indeed, if a library were to compress the bytes that have to be sent, we would not catch that. However, as we assume no particular value-distribution, we cannot make other assumption on the rate a library might compress the data by. Thus, for Hadoop, we can conclude:

$$
\begin{aligned}
& M[\texttt{shuffle}_{m,n}^{k(x)}](d(x)) \\
& = M[\texttt{shuffle}_{m,n}^{768}](x) = \frac{m \cdot n - 1}{m \cdot n} \cdot 12288 \cdot x
\end{aligned}
\tag{7}
$$

Figure 4 shows the relative error of the measured execution time and the execution time predicted by our composition operator.

We tested Hadoop with 1, 2, 4, 8, 12, and 24 compute nodes, each with 1, 2, 4, 8, 12, and 24 threads. For each of these configurations, we had a separate node take care of the synchronization and distribution of work, similar to a general use case. Here, in order to keep the amount of overhead low, we tested with the number of input images equal to the number of compute nodes, making sure one picture is processed by one compute node. On each node, the program replicates the picture once per thread, and each thread processes the picture 24000 times, so we can deal with weak scaling in a pure form.

Our method is able to predict the running time of the different job configurations with an relative error in all but one case below 20%. In general, the prediction is worse the more threads on a node are computing simultaneously, however, the number of nodes does not influence the prediction's quality. We attribute this error to factors we did not include in our model, e.g., hardware contention in the form of a limited memory bandwidth, other threads interrupting the computing ones, and time for creation and deletion of the threads themselves.

Overall, this shows that our composition operator is also suitable to estimate the performance of a multi node cluster set up with Hadoop, as long as the computational tasks are far

10

Figure 4: This figure showcases the relative error between the measured execution time and the one predicted by our composition operator.
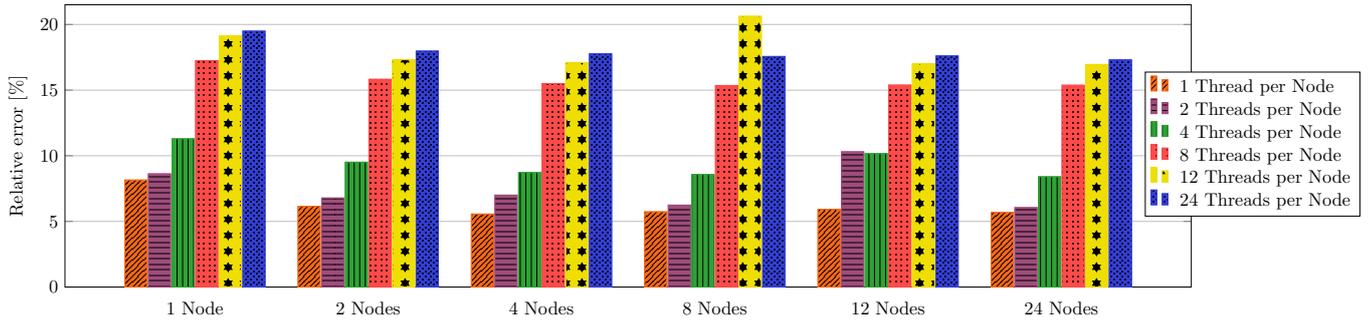
more time consuming than the noise, for example, introduced by loading relatively much data from the disk.

4.2.5. MapReduce, own framework

To get a better understanding of the dependencies between the emitted keys and values in the map phase and the other phases, we expanded the testing framework we used to evaluate task pool and pipeline with by three different implementations for MapReduce. They mirror the way GrPPi and Hadoop realize the pattern, but we refrained from implementing communication and overhead that is not necessary for the actual computation. We used the following ways to reduce the results of the map phase:

- Reducing the key-value collections on each compute node until there is just one left. Sending those ones left to a single node, which reduces the remaining key-value collections to a single one and outputs the result. This leaves $768 \cdot (8 + 8) \cdot (m - 1)/m$ bytes to send for m compute nodes.

- Reducing the key-value collections on a per-key-basis on each node until one key-value collection is left. Sending those ones left to a single node, which reduces them again on a per-key-basis to a single collection and outputs the result. This, too, leaves $768 \cdot (8 + 8) \cdot (m - 1)/m$ bytes to send for m compute nodes.

- Each compute node is assigned a subset of the keys. All compute nodes send all values associated with a key to the respective node, which then reduces all values on a per-key-basis. The results are then send to one node, which outputs the result. This leaves $768 \cdot 8 \cdot (m - 1)/m$ bytes to send for the keys in the reduce phase, together with $i/m \cdot (m - 1)/m \cdot 8$ bytes for the values in the reduce phase, and $768 \cdot (8 + 8) \cdot (m - 1)/m$ bytes to send in the end for m compute nodes and i inputs.

We did this for 1, 2, 4, 8, 12, and 24 threads per compute node, for 1, 2, and 4 compute nodes and for the small, medium, and large pictures, giving a total of 162 performance models.

To give an overview of the models we summarize them in Table 5. Each configuration has four degrees of freedom, the number of nodes, the number of threads, the reduction method, and the size of the pictures. In Table 5, we fixed each degree

once and aggregated all models with the respective feature. We give the minimum, maximum, median, mean, and standard deviation of the relative errors for each such column.

The best results throughout the complete evaluation were achieved with the large input pictures, indicating the comparatively high impact of noise for the small tasks. Furthermore, with growing number of threads and compute nodes, the relative error grows. This artifact arises due to our choice of composition operator. We have chosen to disregard modeling inter-thread synchronization, for example required for using thread-safe data structures, which shows better the more threads or compute nodes are involved.

Table 5: The relative error of the modular models for our own implementation in percent, grouped by the size of the input pictures, the type of reduction phase, the number of threads per compute node n, and the number of compute nodes m.

Group	Min	Max	Median	Mean	Std.
all	0.08	93.68	31.01	37.24	21.24
small	27.05	93.68	57.68	59.68	16.85
medium	6.21	62.75	31.23	32.34	12.88
large	0.08	35.88	24.41	20.03	9.06
reduce_p	0.15	72.62	28.26	32.02	16.92
reduce_v local	0.48	91.52	31.44	38.93	22.37
reduce_v global	0.08	93.68	34.93	40.75	22.89
$n = 1$	0.08	48.69	27.54	25.47	13.96
$n = 2$	2.11	57.87	28.60	29.35	15.31
$n = 4$	5.09	68.02	31.01	35.58	18.36
$n = 8$	10.99	77.54	32.38	40.53	19.89
$n = 12$	12.95	76.81	33.55	40.42	20.36
$n = 24$	14.64	93.68	44.67	51.98	26.28
$m = 1$	0.08	88.57	19.89	28.22	23.62
$m = 2$	23.48	91.91	33.02	41.59	18.31
$m = 4$	23.22	93.68	36.16	42.09	18.26

Overall, the evaluation shows that the usage of our composition operators yield faithful performance models, when considering use cases that do not push the boundaries of the work to noise ratio too much. Altogether, Extra-P reported 31 out of the 162 performance models to be non-linear. These models were $O(x^{0.5} \cdot \log(x))$ (2 times), $O(x^{0.66667} \cdot \log(x))$ (3 times), $O(x^{0.75})$ (1 time), $O(x^{0.5} \cdot \log(x))$ (7 times) and $O(x^1 \cdot \log(x))$ (18 times).

We judged them close enough to the linear models and forced Extra-P to generate a linear model, as these non-linear ones are likely due to statistical uncertainty.

5. Relational theory for composition operators

In this section, we want to explore some consequences of the operators from Section 3, multiple patterns interact. Generally, composition operators provide a structured way of combining multiple performance models. They enable compositional reasoning in form of an relational theory on the higher order function for the performance models.

These kinds of statements relating multiple parallel design patterns to one another are not possible if the performance of the whole program is modeled as a black box, yet they are evident using our compositional modeling. However, statements like these are not possible unless the affected parts of a program can be completely disassembled. MapReduce as a parallel design pattern does not support this kind of disassembly because the map and reduce phase rely crucially on one another. This surfaces in the model of `shuffle`, which's model relies on the number of participating threads and processes. Thus, disassembling the map phase into an additional task pool hides this kind of information by our black box approach. The map phase directly influences the reduce phase by the number of keys it generates, so they cannot be pulled apart. Furthermore, in implementations such as Hadoop or GrPPi, it is assumed that the map and reduce functions are serial and that all parallelization is achieved by replicating the map and reduce phases.

5.1. Relational claims for composition operators

In the following, we provide examples of rules that govern our composition operators and the benefits they provide. For all stages $stage_1$, $stage_2$, and $stage_3$, sequential blocks $task_1$ and $task_2$, thread counts n, the following relations hold:

(i) $M[\texttt{pipe}(\texttt{stage}_1,\texttt{pipe}(\texttt{stage}_2,\texttt{stage}_3))]$
$= M[\texttt{pipe}(\texttt{pipe}(\texttt{stage}_1,\texttt{stage}_2),\texttt{stage}_3)]$,

(ii) $M[\texttt{pipe}(\texttt{stage}_1,\texttt{stage}_2)]$
$= M[\texttt{pipe}(\texttt{stage}_2,\texttt{stage}_1)]$,

(iii) $M[\texttt{pipe}(\texttt{tpool}_n(\texttt{task}_1),\texttt{tpool}_n(\texttt{task}_2))]$
$= M[\texttt{tpool}_n(\texttt{pipe}(\texttt{task}_1,\texttt{task}_2))]$, and

(iv) $M[\texttt{pipe}(\texttt{task}_1,\texttt{task}_2)]$
$\geq M[\texttt{tpool}_2(\texttt{seq}(\texttt{task}_1,\texttt{task}_2))]$.

Intuitively, Equation (i) states that the performance of a pipeline with more than two stages does not depend on the composition order and Equation (ii) states that the performance of a pipeline does not depend on the order of the stages. Equation (iii) states that a parallel pipeline where each stage is a task pool executing some work has the same performance as a task pool where the tasks are parallel pipelines executing the same work. This is equivalent to stating that when layering parallel design patterns correctly, the performance of the resulting system will not change, regardless of the ordering of these patterns.

On a practical level, this means that once the models of the individual stages are known, the model for the performance of the entire system can be derived, and no new measurements have to be performed if the ordering is changed. As long as the performance model of each stage is known, stages can be arbitrarily added or removed from the system and the performance can still be derived without any new measurements.

Furthermore, Relation (iv) states that in a pipeline, there is usually wasted potential. Intuitively, the faster stage always has to wait for the slower stage, which wastes time while the faster stage is stalling. This can be circumvented when sequentially concatenating both tasks and using the two threads from the former pipeline inside a task pool. This way both threads work as long as there are items to process and do not have to wait for one another, resulting in a potentially faster program.

We refrain from giving a relational claim that encompasses MapReduce in addition to task pool or pipeline. On the one hand the Equations (i) and (ii) are mere sanity conditions on our understanding of the pipeline pattern. On the other hand MapReduce exhibits an intrinsic internal flow of elements, so it cannot be split as in Equation (iii) or translated into a simple task pool that executed sequential operations as in Relation (iv).

5.2. Validation of relational claims

In this subsection, we test the claims from Subsection 5.1, where we defined an relational theory for composition operators that apply to the parallel design patterns task pool and pipeline, as well as sequential concatenation. The first two Equations state that the composition operator for the parallel pipeline is (i) associative and (ii) commutative with respect to the tasks. The third Equation (iii) states that a parallel pipeline where each stage is a task pool executing some work has the same performance as a task pool where the tasks are parallel pipelines executing the same work. The fourth Relation (iv) asserts that when using a pipeline without further load-balancing capabilities, there might be a better distribution of threads.

We show a representative subset of pipeline configurations that use the `inc`, `qsort`, and `nop` tasks. Table 4 shows the measured performance models for these configurations. Not only are the models in the same complexity class but even the coefficients show less then 10% variation across all configurations. The models show that the parallel pipeline design pattern is associative and commutative. The commutative property can also be seen in Table 2, where the models of `pipe(qsort,inc)` and `pipe(inc,qsort)` are effectively equal.

The third Equation states that layering parallel design patterns correctly should have no impact on performance. In order to compare the performance models, we have created software systems following both design patterns, but applied in a different order. The first configuration is a pipeline that uses stages that rely on task pools to process the tasks, while the second type is a task pool that uses a pipeline containing stages that are responsible for processing the data. The resulting models are summarized in Table 6 and show that all configurations perform similarly—well apart from inherent run-to-run variations. We did not evaluate this claim, in contrast to the other claims, with more than 8 threads per task pool. This is due to the way

Table 6: Performance neutrality of different layerings for parallel design patterns. Execution time is in nanoseconds. For the number of input elements x, this gives the respective performance models.

Parameter	`pipe(tpool`$_n$`(qsort),tpool`$_n$`(inc))` [ns]	`tpool`$_n$`(pipe(qsort,inc))` [ns]
$n = 1$	$1101.71 \cdot x \log_2 x$	$1072.10 \cdot x \log_2 x$
$n = 2$	$578.47 \cdot x \log_2 x$	$602.41 \cdot x \log_2 x$
$n = 4$	$298.89 \cdot x \log_2 x$	$315.89 \cdot x \log_2 x$
$n = 8$	$165.51 \cdot x \log_2 x$	$181.19 \cdot x \log_2 x$

the implementations of task pool and pipeline use threads internally. With this we can confidentially say that layering parallel patterns and the ordering in which patterns are layered, if done correctly, does not affect performance.

For the fourth relation we evaluated the different pipeline configurations we used in Table 2 but substituted the pipeline for a sequential concatenation inside a task pool with two threads. This way, the total number of threads in the programs stays the same, but they are redistributed in order to reduce the stalling time of the threads. As shown in Table 7, the claim holds true for our test cases. However, we would expect an equality in Column 4, when performing two `inc`, one after the other. We attribute this to the communication between the two threads when passing data from one stage of the pipeline to the other one.

6. Conclusion

We introduced a modular approach to the construction of performance models that can be used not only to optimize existing software or choose between implementation alternatives, but even (and arguably especially) during the design of parallel software. Leveraging the properties of parallel design patterns, we can now construct accurate performance models in a brick-by-brick fashion from performance models of software components, as long as the combination of the components follows the design pattern.

In this extended version of our previously published paper, we also show how to extend the previously published method to incorporate multiple compute nodes, as well as the ability to deal with complex operations that change the number of processed items. We demonstrate this by the example of MapReduce, and depending on its implementation our modular performance models had an relative error smaller than 17 % (GrPPi), 21 % (Hadoop in realistic use case), and 36 % (own implemen-

tation with large pictures). Even in the other test cases, our method still produced a lower bound and the modular model was in the same order of magnitude as the monolithic model.

The resulting models can be created not just without needing to repeat the entire measurement process whenever a component of the system is changed, but rather allow detailed performance prediction before an implementation of the system as a whole even exists. Our modular approach provides significant support to developers trying to design, maintain, or optimize parallel programs. It also reduces the time to obtain a practical performance model significantly by canceling the need to have a working prototype for an application completely. Thus, if a developer wishes to estimate the run time of a new application built from the considered design patterns, they can simply measure the execution time of the serial parts (even with dummy input), and extrapolate a precise performance model for the whole application.

We plan to further investigate this method and study the current limitations, which include synchronization overhead, hardware contention, and work sharing, to name the three most prominent ones.

Table 7: Exchange of pipeline with task pool and sequential concatenation. Execution time is measured in nanoseconds. For the number of input elements x, this gives the respective performance models.

Configuration	Monolithic model [ns]	Monolithic model of pipeline [ns]
`tpool`$_2$`(seq(qsort,nop))`	$546.40 \cdot x \log_2 x$	$1086.11 \cdot x \log_2 x$
`tpool`$_2$`(seq(qsort,inc))`	$561.84 \cdot x \log_2 x$	$1105.29 \cdot x \log_2 x$
`tpool`$_2$`(seq(inc,qsort))`	$564.59 \cdot x \log_2 x$	$1076.89 \cdot x \log_2 x$
`tpool`$_2$`(seq(inc,inc))`	$607.06 \cdot x$	$677.05 \cdot x$
`tpool`$_2$`(seq(inc,nop))`	$355.67 \cdot x$	$670.32 \cdot x$

References

[1] K. Keutzer, T. Mattson, Our Pattern Language – A Design Pattern Language for Engineering (Parallel) Software, https://patterns.eecs.berkeley.edu/ (Online: February 19, 2019).

[2] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, 1st Edition, Addison Wesley, 2004.

[3] M. McCool, J. Reinders, A. D. Robison, Structured Parallel Programming: Patterns for Efficient Computation, Morgan Kaufmann, 2012.

[4] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977.

[5] X. Yang, J. Sun, An analytical performance model of mapreduce, in: 2011 IEEE International Conference on Cloud Computing and Intelligence Systems, 2011, pp. 306–310. doi:10.1109/CCIS.2011.6045080.

[6] E. Vianna, G. Comarela, T. Pontes, J. Almeida, V. Almeida, K. Wilkinson, H. Kuno, U. Dayal, Analytical performance models for mapreduce workloads, International Journal of Parallel Programming 41 (4) (2013) 495–525.

[7] A. Calotoiu, T. Hoefler, M. Poke, F. Wolf, Using automated performance modeling to find scalability bugs in complex codes, in: Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 45:1–45:12.

[8] M. P. Forum, Mpi: A message-passing interface standard, Tech. rep., USA (1994).

[9] A. Calotoiu, T. Höhl, H. Mantel, T. Nguyen, F. Wolf, Designing efficient parallel software via compositional performance modeling, in: Proc. of the Workshop on Programming and Performance Visualization Tools (ProTools), held in conjunction with the Supercomputing Conference (SC19), Denver, CO, USA, 2019, pp. 17–24. doi:10.1109/ProTools49597.2019.00008.

[10] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.

[11] Apache Software Foundation, Hadoop.
URL https://hadoop.apache.org

[12] S. Gorlatch, M. Cole, Parallel Skeletons, Springer US, Boston, MA, 2011, pp. 1417–1422. doi:10.1007/978-0-387-09766-4_24.

[13] S. Gorlatch, Toward formally-based design of message passing programs, IEEE Transactions on Software Engineering 26 (3) (2000) 276–288. doi:10.1109/32.842952.

[14] D. del Rio Astorga, M. F. Dolz, J. Fernández, J. D. García, A generic parallel pattern interface for stream and data processing, Concurrency and Computation: Practice and Experience 29 (24) (2017) e4175, e4175 cpe.4175. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4175, doi:10.1002/cpe.4175.

[15] J. Talbot, R. M. Yoo, C. Kozyrakis, Phoenix++: Modular mapreduce for shared-memory systems, in: Proc. of the Second International Workshop on MapReduce and Its Applications, MapReduce '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 9–16. doi:10.1145/1996092.1996095.

[16] L. Adhianto, S. Banerjee, M. W. Fagan, M. W. Krentel, G. Marin, J. Mellor-Crummey, N. R. Tallent, HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs, Concurrency and Computation: Practice and Experience 22 (6) (2010) 685–701.

[17] S. S. Shende, A. D. Malony, The TAU Parallel Performance System, International Journal of High Performance Computing Applications 20 (2) (2006) 287–331.

[18] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, B. Mohr, The Scalasca Performance Toolset Architecture, Concurrency and Computation: Practice and Experience 22 (6) (2010) 702–719.

[19] D. an Mey, S. Biersdorff, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. D. Malony, W. E. Nagel, Y. Oleynik, C. Rössel, P. Saviankou, D. Schmidl, S. S. Shende, M. Wagner, B. Wesarg, F. Wolf, Score-P: A Unified Performance Measurement System for Petascale Applications, in: Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010, Gauß-Allianz, Springer, 2012, pp. 85–97.

[20] W. Nagel, M. Weber, H.-C. Hoppe, K. Solchenbach, VAMPIR: Visualization and Analysis of MPI Resources, Supercomputer 12 (1) (1996) 69–80.

[21] J. Šilc, B. Robič, T. Ungerer, Progress in Computer Research, Nova Science Publishers, Inc., Commack, NY, USA, 2001, Ch. Asynchrony in Parallel Computing: From Dataflow to Multithreading, pp. 1–33.

[22] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, H. Mantel, Scalable offline monitoring, in: Proc. of the 14th International Conference on Runtime Verification (RV), 2014, pp. 31–47.

[23] A. Brogi, M. Danelutto, D. De Sensi, A. Ibrahim, J. Soldani, M. Torquati, Analysing Multiple QoS Attributes in Parallel Design Patterns-Based Applications, International Journal of Parallel Programming (11 2016). doi:10.1007/s10766-016-0476-8.

[24] N. R. Tallent, A. Hoisie, Palm: Easing the burden of analytical performance modeling, in: Proc. of the 28th ACM International Conference on Supercomputing, ICS '14, ACM, New York, NY, USA, 2014, pp. 221–230. doi:10.1145/2597652.2597683.
URL http://doi.acm.org/10.1145/2597652.2597683

[25] S. Lee, J. S. Meredith, J. S. Vetter, Compass: A framework for automated performance modeling and prediction, in: Proc. of the International Conference on Supercomputing (ICS), Newport Beach, CA, USA, ACM, 2015. doi:10.1145/2751205.2751220.

[26] J. Hammer, G. Hager, J. Eitzinger, G. Wellein, Automatic loop kernel analysis and performance modeling with kerncraft, in: Proc. of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems, PMBS '15, ACM, New York, NY, USA, 2015, pp. 4:1–4:11. doi:10.1145/2832087.2832092.
URL http://doi.acm.org/10.1145/2832087.2832092

[27] S. F. Goldsmith, A. S. Aiken, D. S. Wilkerson, Measuring Empirical Computational Complexity, in: Proc. of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, ACM, New York, NY, USA, 2007, pp. 395–404. doi:10.1145/1287624.1287681.
URL http://doi.acm.org/10.1145/1287624.1287681

[28] L. Carrington, A. Snavely, N. Wolter, A performance prediction framework for scientific applications, Future Generation Computer Systems 22 (3) (2006) 336–346.

[29] A. Grebhahn, C. Rodrigo, N. Siegmund, F. J. Gaspar, S. Apel, Performance-influence models of multigrid methods: A case study on triangular grids, Concurrency and Computation: Practice and Experience 29 (17) (2017) e4057.

[30] B. C. Lee, D. M. Brooks, B. R. D. Supinski, M. H. Schulz, K. Singh, S. McKee, Methods of inference and learning for performance modeling of parallel applications, in: Proc. of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP), San Jose, CA, USA, 2007, pp. 249–258.

[31] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, R. Ricci, Taming performance variability, in: Proc. of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, 2018, pp. 409–425.

[32] P. Reisert, A. Calotoiu, S. Shudler, F. Wolf, Following the Blind Seer – Creating Better Performance Models Using Less Information, in: Proc. of the 23rd Euro-Par Conference, Santiago de Compostela, Spain, Lecture Notes in Computer Science, Springer, 2017, pp. 106–118.

[33] A. Gandomi, A. Movaghar, M. Reshadi, A. Khademzadeh, Designing a mapreduce performance model in distributed heterogeneous platforms based on benchmarking approach, The Journal of Supercomputing (2020) 1–27.

[34] D. Glushkova, P. Jovanovic, A. Abelló, Mapreduce performance model for hadoop 2.x, Information Systems 79 (2019) 32–43, special issue on DOLAP 2017: Design, Optimization, Languages and Analytical Processing of Big Data. doi:https://doi.org/10.1016/j.is.2017.11.006.

[35] H. Herodotou, S. Babu, Profiling, what-if analysis, and cost-based optimization of mapreduce programs, Proc. VLDB Endow. 4 (11) (2011) 1111–1122. doi:10.14778/3402707.3402746.

[36] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, J. J. Dongarra, MPI Collective Algorithm Selection and Quadtree Encoding, Parallel Computing 33 (9) (2007) 613–623.

[37] N. Weber, M. Waechter, S. C. Amend, S. Guthe, M. Goesele, Rapid, detail-preserving image downscaling, ACM Trans. Graph. 35 (6) (Nov. 2016). doi:10.1145/2980179.2980239.