Designing Efficient Parallel Software via Compositional Performance Modeling

Alexandru Calotoiu
Department of Computer Science
Technische Universität Darmstadt
64293 Darmstadt, Germany
calotoiu@cs.tu-darmstadt.de

Thomas Höhl
Department of Computer Science
Technische Universität Darmstadt
64293 Darmstadt, Germany
hoehl@cs.tu-darmstadt.de

Heiko Mantel
Department of Computer Science
Technische Universität Darmstadt
64293 Darmstadt, Germany
mantel@cs.tu-darmstadt.de

Toni Nguyen
Department of Computer Science
Technische Universität Darmstadt
64293 Darmstadt, Germany
tutoni.nguyentuan@stud.tu-darmstadt.de

Felix Wolf
Department of Computer Science
Technische Universität Darmstadt
64293 Darmstadt, Germany
wolf@cs.tu-darmstadt.de

***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 alternatives. However, creating a performance model – whether theoretically or empirically – for an entire application that does not exist yet is challenging unless the interactions between all system components are well understood, which is often not the case during design. 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 such as pipeline or task pool. 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.**

***Index Terms*—performance modeling, parallel design patterns, composition operators**

I. 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]–[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. 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 [5]. 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.

In this paper, we show how empirical performance modeling can still support 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—similar to how one creates

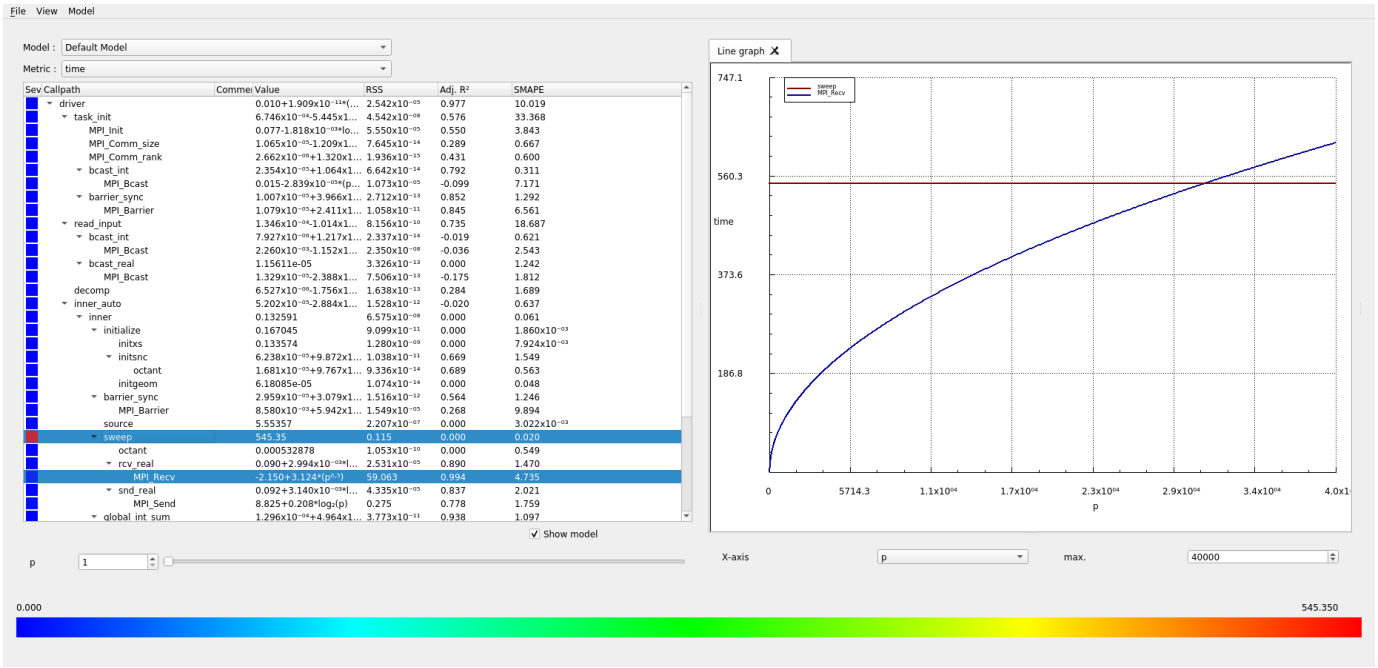


Fig. 1: Interactive exploration of performance models using Extra-P. The screen shot shows performance models generated for call paths in SWEEP3D, a neutron transport simulation

software in a data-flow model. Assuming that performance 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 empirical performance models of software components to be plugged together based on well-defined rules.
- We define composition operators for two design patterns, pipeline and task pool, that can be used in our modular construction approach.
- We provide evidence that our method produces useful performance models by comparing their predictions with actual measurements and their equations with empirical models derived from these measurements.

Before presenting the details of our approach in Section III, we review related work in Section II. Then, we support our claim with performance results in Section IV. Finally, we discuss future work and opportunities in Section V and conclude in Section VI.

II. Related Work

Our approach leverages the concept of design patterns for parallel programs [2]. Design patterns are usually arranged in a pattern language, which is less a formal language with well-defined syntax and semantics than a kind of decision tree that guides the software developer through various design spaces of decreasing levels of abstraction. A well-known 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 to make them run in parallel.

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 [6], TAU [7], Scalasca [8], Score-P [9] and Vampir [10], can be used to gather measurements, usually at the level of individual code regions, that capture the behavior of an application in a given runtime configuration. 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 the large-grain data-flow perspective [11].

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 those 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 map reduce, can be re-interpreted according to this model.

Multiple tools have been developed to help create performance models, some of them based on neural networks [12], [13], others requiring code annotations to supplement measurements in order to derive models [14], [15]. Finally, Siegmund et al. [16] analyze the interaction of different performance relevant parameters to generate models of applications as a whole.

Parallel design patterns have been successfully used to understand Quality of Service attributes of complex software systems at design time using probabilistic analysis [17]. 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 [5], an empirical performance-modeling tool, which we tailor towards modeling the traversal time of data elements through the program.

Extra-P automatically derives human-readable performance models from performance measurements. It is based on the assumption that the behavior of most practical programs can be expressed as n terms involving logarithmic and polynomial expressions of a parameter p, usually representing the number of processors. Extra-P therefore represents models using the *performance model normal form* (PMNF):

$$f(p) = \sum_{k=1}^{n} c_k \cdot p^{i_k} \cdot log_2^{j_k}(p)$$

A search space for potential models defined, by the sets I and J from which i_k and j_k can be chosen, is either generated automatically [18] or can be set by the user, and a combination of regression and cross-validation is then used to find the best coefficients c_k and then select the model with the optimal fit. The resulting human-readable performance models express the execution time, number of floating point operations, bytes send over the network or any other captured metric as a function of the number of processes or other performance-relevant parameters such as the problem size.

Fig. 1 shows how the results of the model generator can be interactively explored. The GUI annotates each call path with a performance model. The formula represents a previously selected metric as a function of the number of processes, and allows other parameters to be represented as well. The user can select one or more call paths and plot their models on the right. In this way, the user can visually compare the scalability of different application kernels.

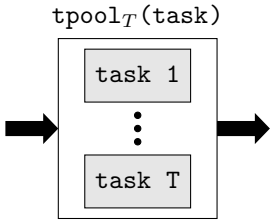
III. Composition Operators for Performance Models

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, which we use to demonstrate our approach. After that, we explain the performance metrics we deem appropriate for our data-flow-centric analysis. In a next step, we clarify the notion of a composition operator for performance models and define operators for the two parallel design patterns we consider in this study. Finally, we introduce an algebra for these composition operators that allows us to reason about the performance impact of applying parallel design patterns.

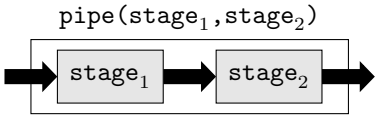
A. Parallel Design Patterns

In this work we focus on two parallel design patterns, namely pipeline and task pool. 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 consequence, the number of threads used may not exceed the available hardware concurrency. Below, we provide definitions of the two patterns pipeline and the task pool.

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. The implementation 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 2a. The task queue stores the work that has to be done and the thread pool is a set of workers that pop



(a) Task-pool pattern



(b) Pipeline pattern

Fig. 2: Parallel design patterns from a data-flow perspective

data elements from the queue and process them. The pattern has the benefit of being able to reuse threads to process data rather than using short-lived threads. This limits the overhead for the creation and destruction of the worker threads. We write $\texttt{tpool}_{\texttt{T}}\texttt{(task)}$ for a task pool that concurrently executes the block `task` for each data element with T threads.

2) Pipeline.: The pipeline pattern can be compared to an assembly line in a factory that creates a product in multiple stages. Once they 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]. 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 a 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 2b. 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 $\texttt{pipe(stage}_1\texttt{,stage}_2\texttt{)}$ for a pipeline that is composed of the stages $\texttt{stage}_1$ and $\texttt{stage}_2$.

B. Performance Metrics

As discussed in Section II, performance models are equations that represent a performance metric as a function of one or more parameters. We write $M(\cdot)$ to denote the performance model of an application. Below, we identify two different performance metrics for applications that process a stream of data elements.

- **Throughput:** The rate at which data elements can be processed.
- **Latency:** The total computational time a single data element requires.

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 both patterns are shown in Figure 3.

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.

C. Composition Operators for Performance Models

Below, we define operators for composing performance models of applications based on parallel design patterns. We interpret the patterns as higher-order functions that take the performance models of the blocks the pattern is composed of, which are themselves functions, as input and returns a function that represents the resulting performance model for the entire pattern implementation. In this sense, the composition operator represents the performance impact of the parallel design pattern itself. It will therefore contain scalar parameters which express the configuration of the parallel design pattern, such as the number of worker threads in a task pool. In the following, we present instances of composition operators for the patterns pipeline and the task pool.

1) Task Pool: Given a task pool $\texttt{tpool}_{\texttt{T}}\texttt{(task)}$, a performance model for the sequential block $M(\texttt{task})$, and a fixed number of threads T, we define the composed performance model of the task-pool pattern as follows:

$$M(\texttt{tpool}_T\texttt{(task)}) := \frac{1}{T} \cdot M(\texttt{task}) \tag{1}$$

Intuitively, a task pool with T threads can process T data elements at once. Therefore, the average runtime of a single data element is only a Tth of the average runtime of the sequential block `task`.

2) Pipeline: Given a pipeline $\texttt{pipe(stage}_1\texttt{,stage}_2\texttt{)}$ and performance models for the states $M(\texttt{stage}_1)$ and $M(\texttt{stage}_2)$, we define the composed performance model of the pipeline pattern as follows:

$$\begin{aligned} &M(\texttt{pipe(stage}_1\texttt{,stage}_2\texttt{)}) \\ &\quad := max(M(\texttt{stage}_1), M(\texttt{stage}_2)), \end{aligned} \tag{2}$$

where max applied to performance models is defined as follows:

$$max(m_1, m_2) = \begin{cases} m_1 & \text{if} \quad \mathcal{O}(m_1) \geq \mathcal{O}(m_2) \\ m_2 & \text{else} \end{cases} \tag{3}$$

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 [19].

D. An Algebra for Composition Operators

The composition operators provide a structured way of combining multiple performance models. This enables the definition of an algebra for compositional reasoning of performance models. In the following, we provide examples of rules that govern our composition operators and the benefits they provide. For all stages $\texttt{stage}_1$, $\texttt{stage}_2$, and $\texttt{stage}_3$, sequential blocks $\texttt{task}_1$ and $\texttt{task}_2$, thread counts T, the following equalities hold:

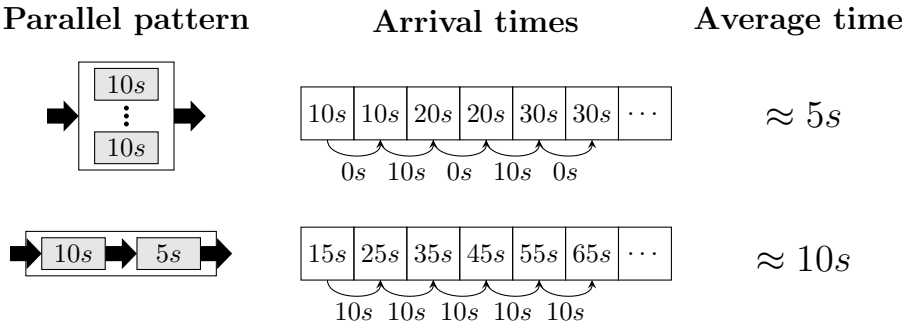


Fig. 3: Examples of average runtimes of individual data elements

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

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

iii) $M(\texttt{pipe(tpool}_\texttt{T}\texttt{(task}_1\texttt{),tpool}_\texttt{T}\texttt{(task}_2\texttt{))})$
= $M(\texttt{tpool}_\texttt{T}\texttt{(pipe(task}_1\texttt{,task}_2\texttt{))})$.

Intuitively, rule (i) states that the performance of a pipeline with more than two stages does not depend on the composition order and rule (ii) states that the performance of a pipeline does not depend of the order of the stages. Rule (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.*

This kind of statements are not possible if the performance of the whole program is modeled as a black box, yet they are evident using our compositional modeling. 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.

IV. Evaluation

In this Section, we support the claims made in Section III. 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.

TABLE I: Models of sequential task blocks.

Task	Model
`nop`	0.00864
`inc`	$0.02599 \cdot n$
`qsort`	$0.03899 \cdot n \log_2 n$

We compare the compositional models with both the performance models generated by Extra-P for the entire systems as well as with the actual performance measurements of the entire systems themselves. We show that the prediction error of the performance model constructed using composition is less than 12% over all our experiments.

All experiment have been carried out on a single node from the Lichtenberg high-performance computer of TU Darmstadt. The node includes 2 Intel XEON E5 2680 v3 processors (12 cores, hyper-threading disabled, 2.5GHz base, 3.3GHz boost), 64GB of DDR4-RAM running under CentOS Linux 7 (kernel 3.10.0-957.21.3.el7.x86_64). The machine is managed by the slurm workload manager in version 17.02.09 and we have used GCC in version 4.8.5 for all experiments.

1) Tasks.: For simplicity, the tasks we use in the following experiments all take an array of integers as input, and return the array after applying their computational task which can potentially alter the array. We have selected the following tasks:

1) `nop`: Performs no operation and return the array unchanged [O(1)].
2) `inc`: Increases the value of each element in the array by 1. [O(n)].
3) `qsort`: Sorts the array using the quicksort algorithm. [O$(n \log_2 n)$].

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 vary 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 repeat each measurement 256 times. The runtime is expressed in microseconds.

The performance models of these tasks express their runtime as a function of the number of elements n in the array. The models are summarized in Table I.

A. Composition Operators

The composition operators we defined are adequate for all possible combinations of tasks. We performed experiments

TABLE II: Comparison between modular and monolithic models of the task-pool and pipeline patterns.

	Configuration	Modular model	Monolithic model	Relative error
Task pool	$\texttt{tpool}_1\texttt{(qsort)}$	$0.03899 \cdot n \log_2 n$	$0.03900 \cdot n \log_2 n$	0.160
	$\texttt{tpool}_2\texttt{(qsort)}$	$0.01946 \cdot n \log_2 n$	$0.01950 \cdot n \log_2 n$	0.180
	$\texttt{tpool}_4\texttt{(qsort)}$	$0.00975 \cdot n \log_2 n$	$0.01015 \cdot n \log_2 n$	4.043
	$\texttt{tpool}_8\texttt{(qsort)}$	$0.00488 \cdot n \log_2 n$	$0.00545 \cdot n \log_2 n$	11.267
Pipeline	pipe(qsort,nop)	$0.03899 \cdot n \log_2 n$	$0.03873 \cdot n \log_2 n$	0.067
	pipe(qsort,inc)	$0.03899 \cdot n \log_2 n$	$0.03891 \cdot n \log_2 n$	0.02
	pipe(inc,qsort)	$0.03899 \cdot n \log_2 n$	$0.03865 \cdot n \log_2 n$	0.089
	pipe(inc,inc)	$0.02599 \cdot n$	$0.02700 \cdot n$	3.802
	pipe(inc,nop)	$0.02599 \cdot n$	$0.02718 \cdot n$	4.437

for all combinations but, for the sake of brevity, focus on analyzing the most relevant four configurations for the pipeline and most relevant two for the task-pool pattern.

1) Task Pool.: The evaluation of this composition operator is straightforward. We initialized the task pools with a different number of threads, ranging from 1 to 8, and measured the execution time for each run. Our composition operator, $M(\texttt{tpool}_T\texttt{(task)}) = \frac{1}{T} \cdot M(\texttt{task})$, 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 II. 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.

2) Pipeline.: We focus on the following configurations of stages and tasks: `pipe(qsort,nop)`, `pipe(inc,inc)`, `pipe(inc,nop)`, and `pipe(inc,qsort)`. Our composition operator, $M(\texttt{pipe(stage}_1\texttt{,stage}_2\texttt{)}) = max(M(\texttt{stage}_1), M(\texttt{stage}_2))$, predicts that the performance model of the pipeline is the performance model of the slower stage. The models summarized in Table II 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.

B. Algebra of Composition Operators

In this subsection, we test the claims from Section III, where we define an algebra of composition operators that apply to the parallel design patterns. The first two rules state that the composition operator for the parallel pipeline is (i) associative and (ii) commutative with respect to the tasks. The third rule (iii) states that the order in which parallel design patterns are layered is performance neutral.

We show a representative subset of pipeline configurations that use the `inc`, `qsort` and `nop` tasks. In Table III, we display 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 II, where the models of `pipe(qsort,inc)` and `pipe(inc,qsort)` are effectively equal.

The third algebra rule we introduce 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,

TABLE III: Performance neutrality of pipeline associativity and commutativity

	Configuration		Monolithic model
Associativity	pipe(pipe(qsort,inc),nop)	**Commutativity**	$0.03873 \cdot n \log_2 n$
	pipe(qsort,pipe(inc,nop))		$0.03900 \cdot n \log_2 n$
	pipe(pipe(qsort,nop),inc)		$0.03873 \cdot n \log_2 n$
	pipe(qsort,pipe(nop,inc))		$0.03888 \cdot n \log_2 n$
	pipe(pipe(inc,qsort),nop)		$0.03877 \cdot n \log_2 n$
	pipe(inc,pipe(qsort,nop))		$0.03894 \cdot n \log_2 n$
	pipe(pipe(inc,nop),qsort)		$0.03860 \cdot n \log_2 n$
	pipe(inc,pipe(nop,qsort))		$0.03894 \cdot n \log_2 n$

TABLE IV: Performance neutrality of different layerings for parallel design patterns.

Parameter	`pipe(tpool`$_T$`(qsort),tpool`$_T$`(inc))`	`tpool`$_T$`(pipe(qsort,inc))`
$T = 1$	$0.03827 \cdot n \log_2 n$	$0.03914 \cdot n \log_2 n$
$T = 2$	$0.01930 \cdot n \log_2 n$	$0.01996 \cdot n \log_2 n$
$T = 4$	$0.01015 \cdot n \log_2 n$	$0.01033 \cdot n \log_2 n$
$T = 8$	$0.00565 \cdot n \log_2 n$	$0.00563 \cdot n \log_2 n$

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 IV and show that all configurations perform similarly— well apart from inherent run-to-run variations. 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.

V. Future Work

So far, we have modeled the task-pool and pipeline parallel design patterns, their interactions, and discussed the types of insights that can be gained from them. We plan to expand our approach to a wider range of patterns, and test our approach on real scientific applications.

By combining DiscoPop [20], an approach aimed at discovering parallel patterns in existing applications with our method we hope to allow the benefits of compositional performance modeling to be attained with a minimum of developer effort. We further wish to investigate the software engineering benefits of our method by supporting the developers of high-performance applications.

In this work, we have performed measurements and applied the tool Extra-P to derive the performance models for the sequential tasks and the entire systems. However, our modular approach is not limited to the models generated by Extra-P and not even to performance models in general. We plan to investigate the insights we can gain by using other dynamic or static analysis tools instead.

VI. Conclusion

We introduce 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. 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.

Acknowledgment

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Research Foundation (DFG) through the Program *Performance Engineering for Scientific Software* and the ExtraPeak project, and by the US Department of Energy under Grant No. DE-SC0015524. The authors gratefully acknowledge to conduct part of this study on the Lichtenberg high-performance computer of TU Darmstadt and thank all anonymous reviewers for the constructive and elaborate feedback.

References

[1] K. Keutzer and 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, and B. Massingill, *Patterns for Parallel Programming*, 1st ed. Addison Wesley, 2004.

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

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

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

[6] L. Adhianto, S. Banerjee, M. W. Fagan, M. W. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, April 2010.

[7] S. S. Shende and A. D. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, 2006.

[8] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca Performance Toolset Architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, April 2010.

[9] 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, and F. Wolf, "Score-P: A Unified Performance Measurement System for Petascale Applications," in *Proceedings 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.

[10] W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[11] J. Šilc, B. Robič, and T. Ungerer, "Progress in Computer Research," F. Columbus, Ed. Commack, NY, USA: Nova Science Publishers, Inc., 2001, ch. Asynchrony in Parallel Computing: From Dataflow to Multithreading, pp. 1–33.

[12] E. Ipek, B. R. de Supinski, M. Schulz, and S. A. McKee, "An Approach to Performance Prediction for Parallel Applications," in *Proceedings of the 11th International Euro-Par Conference*. Springer-Verlag, 2005, pp. 196–205.

[13] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of Inference and Learning for Performance Modeling of Parallel Applications," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. (PPoPP '07). ACM, 2007, pp. 249–258.

[14] K. L. Spafford and J. S. Vetter, "Aspen: A Domain Specific Language for Performance Modeling," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 84:1–84:11.

[15] N. R. Tallent and A. Hoisie, "Palm: Easing the Burden of Analytical Performance Modeling," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14. New York, NY, USA: ACM, 2014, pp. 221–230.

[16] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 284–294. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786845

[17] A. Brogi, M. Danelutto, D. De Sensi, A. Ibrahim, J. Soldani, and M. Torquati, "Analysing Multiple QoS Attributes in Parallel Design Patterns-Based Applications," *International Journal of Parallel Programming*, 11 2016.

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

[19] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "MPI Collective Algorithm Selection and Quadtree Encoding," *Parallel Computing*, vol. 33, no. 9, pp. 613–623, Sep. 2007.

[20] Z. Li, R. Atre, Z. U. Huda, A. Jannesari, and F. Wolf, *DiscoPoP: A Profiling Tool to Identify Parallelization Opportunities*, 08 2015.