mathematics

MDPI

Article

A Theoretical Model for Global Optimization of Parallel Algorithms

Julian Miller [1,*], Lukas Trümper [1,2], Christian Terboven [1] and Matthias S. Müller [1]

[1] Chair for High Performance Computing, IT Center, RWTH Aachen University, 52074 Aachen, Germany; lukas.truemper@rwth-aachen.de (L.T.); terboven@itc.rwth-aachen.de (C.T.); mueller@itc.rwth-aachen.de (M.S.M.)
[2] Huddly AS, Karenslyst Allé 51, 0279 Oslo, Norway
* Correspondence: miller@itc.rwth-aachen.de

Abstract: With the quickly evolving hardware landscape of high-performance computing (HPC) and its increasing specialization, the implementation of efficient software applications becomes more challenging. This is especially prevalent for domain scientists and may hinder the advances in large-scale simulation software. One idea to overcome these challenges is through software abstraction. We present a parallel algorithm model that allows for global optimization of their synchronization and dataflow and optimal mapping to complex and heterogeneous architectures. The presented model strictly separates the structure of an algorithm from its executed functions. It utilizes a hierarchical decomposition of parallel design patterns as well-established building blocks for algorithmic structures and captures them in an *abstract pattern tree (APT)*. A data-centric flow graph is constructed based on the APT, which acts as an intermediate representation for rich and automated structural transformations. We demonstrate the applicability of this model to three representative algorithms and show runtime speedups between 1.83 and 2.45 on a typical heterogeneous CPU/GPU architecture.

Keywords: parallel programming; parallel patterns; program optimization; optimizing framework; dependence analysis

Citation: Miller, J.; Trümper, L.; Terboven, C.; Müller, M.S. A Theoretical Model for Global Optimization of Parallel Algorithms. *Mathematics* **2021**, *9*, 1685. https://doi.org/10.3390/math9141685

Academic Editors: Gabriel Rodríguez and Juan Touriño

Received: 25 June 2021
Accepted: 15 July 2021
Published: 17 July 2021

1. Introduction

Advances in science are intrinsically linked to a steady rise in computing power. Due to the *three walls* [1], this demand is met with increasingly parallel processors and hardware specialization. Modern clusters are thus heterogeneous and the computing nodes are often equipped with different, specialized processors. However, the size and complexity of typical legacy codes challenge the programming productivity of many domain scientists and foster their reliance on (automatic) performance portability between architectures.

The major challenge for maintaining the performance of a parallel algorithm on new architectures is the optimal utilization of parallelism on three levels: (i) *instruction-level*, (ii) *routine (local)*, and (iii) *algorithm (global)*. As such, optimization on all these levels is laborious and requires expert knowledge. Thus, significant efforts target the design of programming abstractions such as parallel programming models and automatic transformation techniques: There are various transformation techniques and capable production compilers for level (i) [2]. Level (ii) is mainly addressed with parallel programming models providing an abstraction over specific processor features such as *OpenMP* [3], *CUDA*, and *MPI* [4]. Furthermore, a sophisticated parallel programming methodology has been developed around the concept of *parallel patterns* [5,6] and the abstraction to local structural parallelism of algorithms. Based on these advances, recent programming models like *RAJA* [7], *Kokkos* [8], and *Stateful Dataflow Multigraphs* [9] enable the implementation and (partially) automatic optimization for different target architectures. These optimiza-

tions include local dataflow and control flow transformations such as data layouts and nested parallelism.

However, there is a lack of effective global transformation techniques. On the one hand, global transformations require large-scale static analyses for identifying concurrency. These analyses are highly combinatorial complex or even infeasible for dynamic programs. On the other hand, global transformations intertwine with the more general decision of mapping the workload to different processors. While lower-level transformations are applied for a single target architecture, global transformations need to tradeoff significant structural changes based on their effect on the concurrency of the algorithm and its utilization of the available hardware. For instance, the fusion of multiple routines might enable the use of a massively parallel accelerator. At the same time, each of these routines might, however, be executed on dedicated processors simultaneously. With increasing heterogeneity, the global transformation of parallel algorithms and their mapping decision cannot be separated.

This paper provides the theoretical model of a framework that enables global optimizations and automatic hardware mapping. It abstracts parallel algorithms in a global, structural representation called *abstract pattern tree (APT)*. This APT captures high-level data dependencies between local parallel structures formalized in a generic parallel pattern definition. Global transformations and automatic mappings are then derived based on optimizing *algorithmic efficiencies*. These efficiencies are necessary performance conditions defined over the global properties of the APT.

In summary, our key contributions are as follows:

- A model of parallel algorithms is introduced, which abstracts the algorithmic structure from the executed functions. This model facilitates the analysis of global algorithmic properties while building on a flexible definition of local parallel structures.
- A new class of global transformation techniques is enabled by introducing necessary performance conditions called algorithmic efficiencies. Three main efficiencies are identified: synchronization, inter-processor dataflow, and intra-processor dataflow efficiency.
- The model's applicability is demonstrated on three typical parallel algorithms showcasing the major transformation capabilities, and their performance improvements are investigated.

The remainder of this paper is structured as follows: Related work is analyzed in Section 2. Section 3 introduces the essential algorithm representation in the form of the APT. Section 4 introduces the idea of algorithmic efficiencies and identifies three main efficiencies. The applicability of the model is investigated with three case studies of typical parallel algorithms in Section 5. Section 6 provides a discussion of the presented techniques towards optimality, applicability, and its integration into compiler frameworks. Section 7 concludes the work and provides an outlook into future work.

2. Related Work

There is extensive literature that addresses related problems or subproblems of this work. In the following, the most relevant literature for this work is discussed in groups based on the specific research question they target.

2.1. Abstractions for Parallel Programming

The used separation of structure and function in this work relates to the original work on design patterns introduced by Christopher Alexander et al. [10] for the architectural domain and later applied to the design of software [11,12]. Based on these works, algorithmic skeletons [13] and parallel design patterns were developed by Mattson et al. [5] and McCool et al. [6]. These building blocks can be found in most parallel programming models and provide interoperability and a common terminology with this work. Cole [13] and Darlington et al. [14] have developed algorithmic skeleton frameworks as the first approach to this end. They were later extended to a wide variety of similar approaches such as GrPPI [15], Fastflow [16], and many others [17]. These frameworks are typically designed as libraries of pre-defined patterns. While they provide high-performance imple-

mentations, they typically lack the global transformations and optimal hardware mapping targeted in this work.

Similarly, OpenMP [3], OpenACC [18], OpenCL [19], and SYCL [20] focuses on local loop-level optimizations, RAJA [7] and Kokkos [8] on task-based parallelism and rule-based performance portability, Halide [21] on image and array processing, and MPI [4] on communication and SPMD optimizations. Julia [22], MATLAB [23], and similar programming languages focus on computer algebra, while Tensorflow [24], MapReduce [25], and similar programming languages mainly target data-centric and machine learning workloads. These approaches allow expressing parallel algorithms adequately but focus on specific domains or local optimization. Instead, our holistic approach aims at global optimizations for the broad domains of scientific software. We use generic parallel patterns as formal elements that provide structural and syntactical information about local concurrency. The proposed model exposes concurrency, guarantees correctness, and globally optimizes the execution for a specific hardware architecture.

2.2. Transformation Techniques

The optimization techniques developed in this work are targeted at global optimizations. For instruction-level optimizations, there is extensive literature on optimizing compilers targeted at HPC, such as Bacon et al. [2]. The framework CHiLL [26] proposes transformations to complex loop nests described by a sequence of high-level transformations. Flattening transformations as seen in NESL [27] and data-parallel Haskell [28], as well as studies by Blelloch et al. [29] and Chakravarty and Keller [30], provide means to compile nested data parallelism to flattened data-parallel code. While nested parallelism allows for high-level abstractions, its transformations can significantly impact the performance of the generated code [31]. Moreover, there exist many rule-based approaches for transforming routines such as Lift [32], Steuwer et al. [33], Rasch et al. [34]. In contrast, this work provides automatic global transformations and hardware mappings based on a static performance model.

2.3. Architectural Mapping and Code Generation

There are multiple approaches for mapping parallel algorithms to specific hardware architecture. The NP-hard MAKESPAN SCHEDULING problem on unrelated machines is a static approach for which different approximation algorithms were proposed [35]. Beaumont et al. [36] have discussed the automatic mapping of parallelism to heterogeneous architectures for local parts of programs instead of the global approach suggested in this work. While this work focuses on static optimizations, there exist many dynamic approaches such as cluster resource management systems [37] and runtime systems that place threads and processes according to their memory affinities and communication patterns [38]. Furthermore, this work integrates the mapping decision into the global optimization to minimize the overall runtime and optimize the utilization of the given hardware architecture.

Intel's Array Building Blocks [39] dynamically generates code from a high-level specification of data-parallel patterns to target heterogeneous architectures. Similarly, Copperhead [40] and Stateful Dataflow Multigraphs [9] optimize and lower data-parallel Python code. Furthermore, modern polyhedral compilers such as Pluto [41], PetaBricks [42], PPCG [43], and Tensor Comprehensions [44,45] can expose parallelism and target multiple hardware architectures. They typically provide advanced optimizations for common problems such as loop-level parallelism.

3. A Theoretical Model of Parallel Algorithms

In the following, the basic theoretical model of the optimization framework is defined, and a representation of parallel algorithms based on parallel building blocks denoted *abstract pattern tree (APT)* is introduced. The APT is a self-contained structure for the global

optimizations to be applied, and the whole framework is therefore widely decoupled from language-specific properties.

3.1. Performance Definition and Model Assumptions

The framework's goals are global optimization and the optimal mapping of parallel algorithms onto a target architecture. Thereby, optimality is defined in terms of minimal overall execution time for given hardware. This can include multiple nodes with heterogeneous architectures. The focus on global properties allows for the following separation: A parallel algorithm consists of *structural* information such as concurrency, synchronization, dependencies, input and output data and *functional* information, which provide the actual computation. While the functional information may guide specific information, e.g., in a static performance model, the global transformations rely on structural information only. Furthermore, the model uses the following assumptions on the parallelism of the algorithm provided by the developer:

- *Local Optimality:* Locally, the parallel hotspots have been identified, and the potential independence of their operations is expressed optimally in the algorithmic structure. This assumption can be assured by methods for identifying concurrency, such as introduced by Mattson et al. [5].
- *Correct:* All dependencies are well-defined, and the algorithm is free of data races, deadlocks, and similar correctness issues.

3.2. Algorithmic Representation

The algorithmic structure is represented in a dataflow-centric fashion with the following structural elements:

- A *data item*, in short data, is produced and consumed as the result of computations during the execution of an algorithm. Thereby, a data item is immutable and does not refer to a memory location.
- An *operation* is a set of instructions producing and consuming data, which resembles the task definition of typical parallel programming models. This set of instructions is interpreted atomically.
- A *place* is a source (does not consume data) or a sink (does not produce data), i.e., external inputs and outputs of the parallel algorithm.
- A *data dependency* between operations occurs when one operation consumes a data item another operation produces. The definition by A. J. Bernstein [46] corresponds to a flow dependency (or true dependency, read-after-write, RAW).

Hence, the model represents a directed data-dependency graph based on flow dependencies. It covers data and control dependencies, while name dependencies like the anti-dependency (write-after-read) and the output dependency (write-after-write) are not interpretable in this model because of the missing relation to memory locations. This also ensures that data-dependency graphs are always acyclic in this model.

3.2.1. Local Structures: Serial and Parallel Patterns

Local structures like loops or functional calls reoccur throughout parallel algorithms and are thus called *patterns*. Formally, each pattern is a data-dependency graph with operations and data items; the places of this graph are the results of preceding local structures. The resulting directed graph is denoted a *pattern diagram (PD)*, $PD = (V_{PD}, E_{PD})$. PDs allow for the efficient analysis of local parallelism, which is defined as follows:

- An operation $o \in V_{PD}$ *depends* on another operation $o' \in V_{PD}$, iff there is a non-empty directed path from o' to o. All other operations are *independent* (cf. *happens-before* relation [47]).
- Let $o \in V_{PD}$ be an operation and let $o_1, \ldots, o_k$ be the depending operations of o, i.e., $(o_i, o) \in E_{PD}, \forall i = 1, \ldots, k$. The *earliest-execution-time* of o is defined as

$t(o) := \max\{t(o_1), \ldots, t(o_k)\} + 1$ (all operands are available) with $t(p) := 0$ for any place p.

- Let $STEP_n \subseteq V_{PD}$ denote the set of all pairwise independent operations with earliest-execution-time of n. The operations of $STEP_n$ are pairwise *parallel* and form a single logical step of the pattern.

A pattern is called a *parallel pattern* if there are at least two parallel operations. Otherwise, it is called a *serial pattern*. For example, Figure 1 shows four operations, each consuming overlapping items from an array of data in a regular access scheme. This is the structural abstraction of the four applications of a 2×2 kernel on a flattened 3×3 matrix. The structure is commonly called a *stencil*, which is a specific instance of the generic parallel pattern definition.

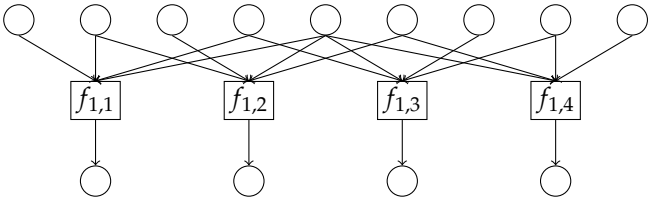

Figure 1. Pattern diagram of a stencil on 3×3 matrix with a 2×2 filter.

3.2.2. Global Structure: Abstract Pattern Tree

The APT is the high-level representation of an algorithmic structure providing a global perspective on a parallel algorithm. Formally, the APT is an undirected graph with the nodes being the patterns occurring in the algorithm and the edges reflecting the execution order specified by the developer. The execution starts with the topmost serial node, which calls its child nodes in sequential order from left to right. Rectangular boxes represent the serial nodes, and undirected edges represent the dependencies between patterns. The parallel nodes are shown through circular boxes and summarize the local pattern diagrams. In a typical application, the pattern nodes are instances of specific parallel patterns such as a *map* or a *reduction*. Such patterns can be understood as higher-order functions with fixed schemes of data dependencies repeated over the input data. In these cases, the well-defined regularity of the pattern allows significant compression of the structural information making a global static analysis between different local patterns in the APT feasible.

Furthermore, the APT is enriched with information regarding hardware mapping. During the optimization step, as described in the following chapter, the execution schedule and target hardware is determined and added as metadata to the nodes of the APT. This includes splitting of parallel patterns into partitions to be executed by processors and the required data transfers. Additionally, the hardware description as used by the optimization can be stored in the APT or through a separate hardware description language. This includes the abstraction of the hardware with the main performance metrics such as computational throughput, sustainable memory bandwidths and latencies, and the memory hierarchy. The final APT, after the optimization steps, then contains all information required to generate machine code.

An example of an APT is provided in Figure 2, which shows the algorithmic structure of a typical image manipulation algorithm that first computes the image gradients and then applies a 1D filter to the image. Its structure consists of a stencil with a PD similar to Figure 1 and a subsequent composition of a map and a reduction corresponding to the structure of matrix-vector multiplication.

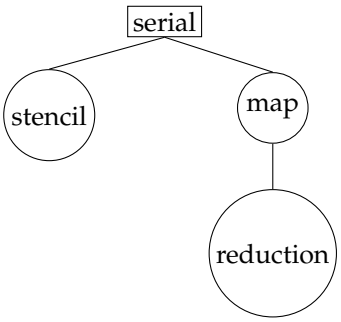

Figure 2. Example of the algorithmic structure of an image manipulation algorithm that first computes the image gradients and then applies a 1D-filter to the image.

3.2.3. Basic Notations

Throughout this paper, a target architecture comprises a set of processors $\mathcal{P}$. Graphically, an operation in a PD, $o \in V_{PD}$, is described by a rectangular box, a place by a circular box, and a data dependency by a directed edge. Parallel operations of a PD form a $STEP$ and are arranged on the same horizontal axis. The APT consists of rectangular boxes representing serial patterns, circular boxes representing parallel patterns, and undirected edges representing the dependencies between patterns. The children of a serial pattern are to be executed from left to right. Additionally, the $STEP$ notation introduced on the level of PDs is also used at the scope of the whole algorithm. These global steps $GSTEP_1, \ldots, GSTEP_N$ are defined analogously and can be constructed directly from the local steps $STEP_n$. The disjoint union of all global steps equals the set of all operations $\mathcal{O}$.

4. Algorithmic Efficiencies

The following chapter introduces a static performance model, enabling the derivation of global mapping decisions and transformations. The proposed performance model makes use of the concept of algorithmic efficiencies as sketched in [48]. Algorithmic efficiencies define necessary optimality conditions of performance over different global properties of algorithms. The separation into different properties allows to optimize the performance separately and reduces the complexity compared to a joint optimization. Runtime estimates in algorithmic efficiencies are thereby parameterized into cost functions, modeled with existing performance models. Furthermore, each efficiency is defined over the properties of the APT. Thus, transformations of the APT are also directly captured by the performance model.

4.1. Algorithmic Steps and Synchronization

The *synchronization efficiency* seeks to maximize the potential parallelism before mapping the operations to an architecture. On the global algorithmic level, this potential is mainly limited by unidentified parallelism such as false *linearization* of independent parallel patterns. Linearization of parallel patterns is thereby defined as the sequential order of two patterns due to data dependencies.

Although asynchronous techniques on the instruction-level may hide such linearization to some extent, linearization on the global level still reduces the potential parallelism during optimization. Therefore, maximizing this potential by resolving false linearization is a prerequisite before deriving the actual mapping and applying transformations. Formally, this problem is described by the global steps, $GSTEP_1, \ldots, GSTEP_N$, defined by the developer and the goal of pulling patterns into earlier steps so that the overall number of steps is reduced and the width of the steps is maximized. Utilizing static data dependency analysis [46,49], false data dependencies might be identified and the number of steps reduced:

Definition 1 (Synchronization Efficiency)**.** *An algorithm is synchronization efficient if it has a minimal number of global steps.*

4.2. Inter-Processor Dataflow

The inter-processor dataflow efficiency guides the mapping of operations to processors. In this context, the abstract term *processor* refers to any homogeneous group of cores sharing the same processor-local cache. Furthermore, the space of mappings is restricted to functions, i.e., each operation must be executed on a single processor.

Without loss of generality, a mapping $M : \mathcal{O} \to \mathcal{P}$ can be decomposed into a sequence of step-wise mappings M_1^T, where each step-mapping $M_t : GSTEP_t \to \mathcal{P}$ is a function on the subset of operations of the global step. To compare mappings, the efficiency assigns any mapping a cost as follows: The *execution costs* $E_t : \mathcal{P} \times 2^{GSTEP_t} \to \mathbb{R}$ define the costs for executing a set operations on a processor. Hence, the execution costs are step-local. Furthermore, the *network costs* $N_t : \mathcal{P} \times 2^{GSTEP_t} \to \mathbb{R}$ account for the costs of communicating the data between two operations. Those costs may depend on previous steps and mappings, which are indicated by the semicolon notation in the function $N_t(\cdot, \cdot; M_1^{t-1})$. The total costs of a mapping is then the sum of the maximal costs of operations assigned to a processor $M_t^{-1}(P)$ over all steps $t = 1, \ldots, T$:

$$\sum_{t=1}^{T} \max_{P} \left[E_t(P, M_t^{-1}(P)) + N_t(P, M_t^{-1}(P); M_1^{t-1}) \right],$$

leading to the following efficiency:

Definition 2 (Inter-Processor Dataflow Efficiency)**.** *A mapping M_1^T is inter-processor dataflow efficient if it has minimal total costs for the execution and network.*

Conceptually, this efficiency implies a multi-step scheduling problem, where processors are distinguishable in the execution of operations and data access times. This distinction of two dueling properties is an essential aspect of effective mapping and transformation decisions: Minimizing the execution costs is typically achieved by spreading the computation across processors. Simultaneously, the network costs and its full context-dependence require minimizing the dataflow between different processors to keep the need for communication minimal.

Cost Modeling

The costs of the above efficiency are introduced in a modular manner. Thus, different performance models can be used to refine the exact costs with respect to the target architectures and the complexity of the resulting optimization problem. For instance, the modeling may be similar to the *roofline model* [50]:

- *Execution costs:* The execution of operations is captured by the number of *floating point operations (FLOPS)* divided by the peak performance π_P (clock frequency times FLOPS per cycle):

$$E_t(P, M_t^{-1}(P)) := \frac{\sum_o FLOPS(o)}{\pi_P}, \quad o \in M_t^{-1}(P).$$

- *Network costs:* The network costs are defined as the slowest data transfer between two processors. A data transfer thereby bundles all bytes to be transferred from one processor to another to satisfy the data dependencies. The bandwidth $\beta_s(P', P)$ is determined by the slowest interconnect between these two processors and a latency penalty $\Gamma_s(P', P)$ is added:

$$N_t(P, M_t^{-1}(P)) := \max_{P'} \left\{ \frac{\sum_{(o',o)} BYTES((o', o))}{\beta_s(P', P)} + \Gamma_s(P', P) \right\}, \quad o \in M_t^{-1}(P), o' \in M_{1\ldots t-1}^{-1}(P').$$

Furthermore, the roofline model assumes the execution and network costs to overlap entirely. In general, the degree of overlap may, however, be controlled by a set of hyperparameters $\kappa_{P,t}$ yielding:

$$\sum_t \max_P \left[(1-\kappa_{P,t})\cdot E_t(P, M_t^{-1}(P)) + N_t(P, M_t^{-1}(P); M_1^{t-1})\right],$$

$$0 \leq \kappa_{P,T} \leq \min\left\{1, \frac{N_t(P,\cdot)}{E_t(P,\cdot)}\right\}.$$

4.3. Intra-Processor Dataflow

Previous efficiencies result in a global mapping of operations to the different processors. To this end, the notion of steps and linearization is a convenient simplification on the global level, where workloads are assumed to be significant. However, the simplification becomes inadequate for analyzing the execution of operations on the cores of a processor. For instance, multiple operations may be executed simultaneously through vectorization. Furthermore, operations of different steps may overlap due to asynchronous techniques in hardware such as prefetching or hyperthreading.

The *intra-processor dataflow efficiency* therefore seeks to optimize the execution of operations on a processor's cores targeting the execution units and core-local caches. In principle, this assignment may be solved as part of the previous efficiency. However, the cores of a processor are assumed to be homogeneous, and the resulting scheduling problem may therefore be adequately solved with simpler heuristics such as static scheduling based on loop indices. Furthermore, the efficiency involves other instruction-level optimizations, such as improving operations overlap through asynchronous techniques. Because of the global scope of this paper, this optimization must be delegated to downward compilers relying on a comprehensive toolset of best practices; see related work for approaches (RAJA, Kokkos).

5. Evaluation

The theoretical model formulates the automatic mapping and global transformation problem as an optimization over costs. The following evaluation's purpose is to assess whether this provides a suitable basis for a class of optimization and mapping algorithms targeting heterogeneous architectures. In detail, it must be shown that performance-critical transformation and mapping decisions, as typically applied by a performance engineer, can be reproduced as the result of cost minimization on practical problems. This work focuses on algorithmic changes, accelerator offloading, and distributed computing. The evaluation is based on benchmarks representing typical parallel algorithms and two representative heterogeneous CPU-GPU nodes found in modern clusters.

5.1. Experimental Setup

Due to the combinatorial complexity of possible mappings, the evaluation focuses on comparing two mapping hypotheses: A baseline version and an optimized version are identified for each benchmark. The baseline version closely follows the numerical definition of the algorithm, whereas the optimized version comprises the typical performance-critical transformation and mapping decisions. Based on these transformations and the cost definition sketched in Section 4.2, it is investigated whether the optimized version is also preferred in the cost-based framework. Furthermore, both hypotheses are manually implemented in C, and both versions' runtimes are compared to their costs. The difference to an algorithmic setup is then the space of mappings, i.e., an algorithm can search through a more expansive space of mappings of finer distinctions. The exemplary cost function may not be detailed enough to reproduce the quality of the optima in these cases, and a cost function would be required, which captures more architecture-specific properties.

The evaluation comprises the following benchmarks:

- *Jacobi:* The problem showcases the capabilities of the model for applying algorithmic changes. Two linear equation systems $Ax = b$ and $Ax' = b'$ are defined who share A and are of size 8192×8192. The equations are solved iteratively with the Jacobi algorithm and a fixed number of iterations $K = 50$ for both systems.
- *k-means:* The problem showcases the automatic accelerator offloading capabilities of the model. The algorithm partitions $n = 10^7$ data points into $k = 128$ clusters. The mean Euclidean distances of the data points to their respective nearest cluster are minimized iteratively in 100 iterations.
- *Monte Carlo Pi:* The problem showcases the distributed computing capabilities of the model. It defines the approximation of π via the Monte Carlo method. The area of a unit circle is accumulated by averaging over 96 independent estimations with 10^9 random draws each.

The investigated architecture consists of two typical CPU-GPU nodes connected via Intel® Omni-Path with a bandwidth of 100 Gb/s. Each node features two Intel® Skylake® Platinum 8160 processors with 24 cores each, a base clock frequency of 2.1 GHz, disabled HyperThreading, and 192 GB of DDR4 RAM. Two NVIDIA® Tesla® V100 GPUs with 16 GB of HBM2 memory are connected via PCI-express. The implementations are done manually in C with OpenMP 4.5 for shared memory problems, OpenMPI 3.1.3 for distributed memory problems, and CUDA 10.2 for GPU offloading problems. The GCC compiler in version 9.3.0 with compiler flags *-fopenmp -std=c99 -O2* is used to generate CPU code and the NVIDIA compiler version 10.2 with GCC version 8.2.0 with compiler flags *-Xcompiler -fopenmp* for the GPU code. The costs of the different transformation and mapping hypotheses are derived manually, the runtime measurements are repeated 30 times, and median values are reported.

5.2. Results

The experiments and results for each benchmark are explained in the following sections. Table 1 provides a brief overview of costs, runtimes, and transformations.

Table 1. The estimated costs and the median measured runtime in seconds for the baseline version (base) and the optimized version (opt.).

Algorithm	Cost [s]		Runtime [s]		Transformations
	Base	Opt.	Base	Opt.	
Jacobi	0.410	0.288	0.986	0.539	Fusion, re-ordering, pipelining
k-means	15.543	5.052	9.594	3.921	GPU offloading
Monte Carlo Pi	12.381	6.190	43.449	22.238	Distributed computing

5.2.1. Jacobi Algorithm

The benchmark consists of two linear equation systems $Ax = b$ and $Ax' = b'$ sharing the same matrix A. Both systems are to be solved iteratively with the Jacobi algorithm. Each iteration is a map over the rows of the matrix, yielding an APT consisting of a sequence of maps. The setup exploits two high-level properties to be considered by an optimization framework:

1. The two Jacobi applications are independent and could be fused into a single sequence of Jacobi iterations.
2. In each iteration, the corresponding rows between both equation systems share the same data from matrix A.

Hypotheses: The baseline hypothesis does not fuse the Jacobi applications and solves the systems one after another. Each Jacobi iteration is split into an upper half and lower half of equations. These halves are always assigned to the two 24-core CPUs of a single node. The optimized version fuses the two Jacobi applications and executes the whole workload on the same two 24-core CPUs of a single node. It thereby assigns the corresponding halves

of both equation systems of a single iteration to the same CPU according to the shared data of matrix A. The resulting costs are reported in Table 1, which shows that the baseline costs are 0.410 s, and the optimized costs are 0.288 s. Furthermore, the implementation of both versions yield median runtimes of 0.986 s and 0.539 s, respectively.

5.2.2. *k*-Means Algorithm

The benchmark consists of a clustering task on a dataset with the *k*-means algorithm. In detail, 10^7 two-dimensional points of a synthetic dataset need to be assigned to $k = 128$ clusters, where the result is obtained after 100 iterations. Each iteration of the *k*-means algorithm comprises two stages, an assignment and an update step. Both steps contain map patterns on large, dense data defining a massive, data-parallel task. Therefore, a typical manual optimization is to offload the whole computation to a suitable accelerator such as the GPU in the considered setup.

Hypotheses: The baseline version executes the whole computation on the CPUs of a single node with 48 cores. The optimized version offloads the computation to the GPU of a single node considering the massive data parallelism of the application. The resulting costs are reported in Table 1, showing the costs of baseline and optimized with 15.543 and 5.052 s. The runtimes are measured at 9.594 and 3.921 s.

5.2.3. Monte Carlo Pi

The benchmark approximates π by accumulating the area of a unit circle obtained from 10^9 random draws. The estimation is repeated 96 times, and the final result is obtained by averaging. The benchmark defines a typical, compute-bound Monte Carlo method, where each estimation is embarrassingly parallel. Therefore, a typical manual mapping decision distributes the independent estimation over multiple nodes as the communication overhead is minimal.

Hypotheses: The baseline version executes the whole computation on the CPUs of a single node with 48 cores. The optimized version, which represents the optimized mapping decision, executes the same computation of the four CPUs of both nodes. The resulting costs and median runtimes are shown in Table 1. The costs are 12.381 and 6.190 s, respectively, while the runtimes are 43.449 and 22.238 s.

6. Discussion

This paper provides a theoretical model for global optimizations of parallel algorithms and their mapping to specific target architectures. The optimization techniques are derived from algorithmic efficiencies, which state generic criteria for performance.

6.1. Analysis of the Results

The model's capabilities to automatically optimize parallel algorithms through algorithmic changes, accelerator offloading, and distributed computing was shown. The three real-world use-cases were optimized successfully, and a significant speedup in the estimated runtime costs between 1.42 and 3.08 was achieved. Similarly, the measured runtime shows speedups between 1.83 and 2.45. Hence, complex transformations utilizing the specific hardware characteristics were automatically applied, significantly increasing the developer's productivity and aiding performance portability.

6.2. Performance Model

The costs for the k-means algorithm are overestimated by a factor of 1.62 for the baseline and 1.29 for the optimized version. The other two benchmarks show underestimations between 0.28 and 0.53, where the Monte Carlo Pi example shows the highest discrepancies. However, the actual mapping and transformation decisions are guided by the relative cost improvement of the optimized version over the baseline version, which is consistent with runtime improvements on all benchmarks.

In general, the costs of the static performance model must be interpreted as a loss/utility rather than a direct runtime estimate. As considered in the evaluation, coarse-grained cost modeling leads to sufficient mapping and transformation decisions if the investigated set of mappings is sparse and each hypothesis differs significantly from others. On the global level, this assumption is valid in cases where the processors of heterogeneous architectures are distinctly different. However, a predictive performance model would improve the resulting transformations' interpretability and allow for finer mapping and transformation decisions. Extensions to the roofline model to include further hardware characteristics, including accelerator-specifics or a more detailed performance model such as the execution-cache-memory performance model [51], could be utilized.

6.3. Optimizations

While the model allows for rich optimizations of parallel algorithms, it is limited in finding completely different algorithms for a specific problem. This could be enabled by adding a library of equivalent structures used as replacements for user-provided parts of the algorithms. Furthermore, the model relies on static information and, thus, dynamic information cannot be included in the algorithm's specification. This limitation could be mitigated by combining the model with a dynamic approach such as autotuning techniques [52–54] to incorporate runtime information into the optimization process. Similarly, monitoring data could be collected and fed back to the optimization step for commonly executed programs.

Moreover, the optimality of the optimization algorithms highly depends on the choice of initial hypothesis and their pruning. Well-tested heuristics will be required to handle the tradeoff between optimality effectively and compile times. Finally, the analysis of the quality of the optimizations is challenging due to the lack of a predictive model, as discussed above. The combination with detailed performance models could enable estimations of lower bounds of necessary synchronization and data movements. Furthermore, additional constraints could be included to cater for energy efficiency and power capping, cost-effective usage of the available resources by minimizing the total cost of ownership of the cluster, and multi-job scheduling problems, such as specific times one needs the respective simulation results. Weighting factors could handle the tradeoff between multiple target metrics.

6.4. Integration into Compiler Frameworks

Three main stages are required to implement the proposed global optimization framework: In the frontend, the hierarchy of patterns needs to be extracted from existing code bases. To this end, pattern recognition tools such as AutoPar [55], Pluto [41,56] or DiscoPoP [57] could be leveraged. In the middleware, the exposed structure is optimized and mapped to target hardware. In the backend, the optimized structure is lowered to a representation that enables loop-level and ILP optimizations. This could be integrated as a pre-processing step into existing compiler frameworks via an intermediate representation (IR). Alternatively, one could implement the framework in a source-to-source fashion where optimized and target-specific code is generated, lowered to an existing programming language. The backend could utilize existing projects such as LLVM [58] and libraries such as Boost [59] to reduce the required implementation efforts. A typical production compiler could then further lower the code to machine code, which can be dynamically optimized with autotuning techniques [52–54].

7. Conclusions

We present a systematic approach to optimize parallel algorithms globally and efficiently map them to heterogeneous architectures. The approach leverages an abstract representation of parallel algorithms via a hierarchical decomposition of parallel patterns. This representation allows for global transformations to optimize their synchronization

and dataflow efficiencies, including pattern matching, siblings fusion, pipelining, cache blocking, and reordering.

We have demonstrated that the proposed model can identify parallelism and restructure real-world parallel algorithms. To this end, dataflow optimizations, including pipelining and cache-blocking, were applied automatically. Furthermore, the model proposed the optimal hardware mapping for these algorithms and provided speedups between 1.83 and 2.45.

The following steps are the design and implementation of the global transformations into a compiler framework. This includes the representation of the APT, the optimization algorithms, and a code generator. Furthermore, a more detailed performance model could improve the cost estimation, and heuristics and libraries could aid in substituting algorithms with more advanced candidates.

Supplementary Materials: The following are available online at https://www.mdpi.com/2227-7390/9/14/1685/s1, Source code, optimization reports, and runtimes S1: Jacobi; source code, optimization reports, and runtimes S2: k-means; source code, optimization reports, and runtimes S3: MonteCarlo-Pi.

Author Contributions: Conceptualization, methodology, J.M.; model, proofs, L.T.; software, validation J.M. and L.T.; writing—original draft preparation, J.M. and L.T.; writing—review and editing, J.M., L.T. and C.T.; funding acquisition, M.S.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The source codes of the case studies and their optimization results are available in the supplemental material.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Asanovic, K.; Bodik, R.; Demmel, J.; Keaveny, T.; Keutzer, K.; Kubiatowicz, J.; Morgan, N.; Patterson, D.; Sen, K.; Wawrzynek, J.; et al. A View of the Parallel Computing Landscape. *Commun. ACM* **2009**, *52*, 56–67.
2. Bacon, D.F.; Graham, S.L.; Sharp, O.J. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.* **1994**, *26*, 345–420.
3. OpenMP API for Parallel Programming, Version 5.0. Available online: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (accessed on 25 June 2021).
4. MPI Forum. MPI: A Message-Passing Interface Standard, Version 3.1. Available online: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (accessed on 25 June 2021).
5. Mattson, T.G.; Sanders, B.; Massingill, B. *Patterns for Parallel Programming*; Pearson Education: Amsterdam, The Netherlands, 2004.
6. McCool, M.D.; Robison, A.D.; Reinders, J. *Structured Parallel Programming—Patterns for Efficient Computation*; Elsevier: Amsterdam, The Netherlands, 2012.
7. RAJA Performance Portability Layer. Available online: https://github.com/LLNL/RAJA (accessed on 25 June 2021).
8. Edwards, H.C.; Trott, C.R.; Sunderland, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distrib. Comput.* **2014**, *74*, 3202–3216.
9. Ben-Nun, T.; de Fine Licht, J.; Ziogas, A.N.; Schneider, T.; Hoefler, T. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC19), Denver, CO, USA, 17–22 November 2019; pp. 1–14.
10. Alexander, C. *A Pattern Language: Towns, Buildings, Construction*; Oxford University Press: Oxford, UK, 1977.
11. Beck, K.; Cunningham, W. Using Pattern Languages for Object Oriented Programs. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Orlando, FL, USA, 4–8 October 1987.
12. Gamma, E. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Reading, MA, USA, 1995; Volume 99.
13. Cole, M. *Algorithmic Skeletons: Structured Management of Parallel Computation*; MIT Press: Cambridge, MA, USA, 1991.
14. Darlington, J.; Guo, Y.K.; To, H.W.; Yang, J. Parallel Skeletons for Structured Composition. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95, Santa Barbara, CA, USA, 19–21 July 1995; ACM: New York, NY, USA, 1995; pp. 19–28.

15. del Rio Astorga, D.; Dolz, M.F.; Fernández, J.; García, J.D. A generic parallel pattern interface for stream and data processing. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e4175.
16. Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. Fastflow: High-level and efficient streaming on multi-core. In *Programming Multi-Core and Many-Core Computing Systems, Parallel and Distributed Computing*; John Wiley & Sons, Inc.: Haboken, NJ, USA, 2017; pp. 261–280.
17. Gonzàlez-Vèlez, H.; Leyton, M. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exp.* **2010**, *40*, 1135–1160.
18. The OpenACC Application Programming Interface. Available online: OpenACC-Standard.org (accessed on 25 June 2021).
19. The Open Standard for Parallel Programming of Heterogeneous Systems. Available online: https://www.khronos.org/opencl/ (accessed on 25 June 2021).
20. C++ Single-source Heterogeneous Programming for OpenCL. Available online: https://www.khronos.org/sycl/ (accessed on 25 June 2021).
21. Ragan-Kelley, J.; Adams, A.; Paris, S.; Levoy, M.; Amarasinghe, S.; Durand, F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* **2012**, *31*, doi:10.1145/2185520.2185528.
22. Bezanson, J.; Karpinski, S.; Shah, V.B.; Edelman, A. Julia: A fast dynamic language for technical computing. *arXiv* **2012**, arXiv:1209.5145.
23. MATLAB (Matrix Laboratory). Available online: https://www.mathworks.com (accessed on 25 June 2021).
24. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; others. Tensorflow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
25. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113.
26. Chen, C.; Chame, J.; Hall, M. *CHiLL: A Framework for Composing High-Level Loop Transformations*; Technical Report; University of Utah School of Computing: Salt Lake City, UT, USA,2008.
27. Blelloch, G.E. *NESL: A Nested Data Parallel Language*; Carnegie Mellon University: Pittsburgh, PA, USA, 1992.
28. Chakravarty, M.M.; Keller, G.; Lechtchinsky, R.; Pfannenstiel, W. Nepal—Nested data parallelism in Haskell. In Proceedings of the European Conference on Parallel Processing, Manchester, UK, 28–31 August 2001; Springer: Berlin/Heidelberg, Germany, 2001; pp. 524–534.
29. Blelloch, G.E.; Sabot, G.W. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.* **1990**, *8*, 119–134.
30. Chakravarty, M.M.; Keller, G. More types for nested data parallel programming. *ACM Sigplan Not.* **2000**, *35*, 94–105.
31. Spoonhower, D.; Blelloch, G.E.; Harper, R.; Gibbons, P.B. Space profiling for parallel functional programs. *J. Funct. Program.* **2010**, *20*, 417–461.
32. Steuwer, M.; Remmelg, T.; Dubach, C. Matrix multiplication beyond auto-tuning: Rewrite-based GPU code generation. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Pittsburgh, PA, USA, 1–7 October 2016; pp. 1–10.
33. Steuwer, M.; Fensch, C.; Lindley, S.; Dubach, C. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. *ACM Sigplan Not.* **2015**, *50*, 205–217.
34. Rasch, A.; Schulze, R.; Gorlatch, S. Generating Portable High-Performance Code via Multi-Dimensional Homomorphisms. In Proceedings of the 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), Seattle, WA, USA, 23–26 September 2019; pp. 354–369.
35. Lenstra, J.; Shmoys, D.; Tardos, É. Approximation Algorithms for Scheduling Unrelated Paralle Machines. In Proceedings of the 28th Annual Symposium on Foundations of Computer Science (SFCS 1987), Los Angeles, CA, USA, 12–14 October 1987 ; Volume 46, pp. 217–224. doi:10.1109/SFCS.1987.8.
36. Beaumont, O.; Becker, B.A.; DeFlumere, A.; Eyraud-Dubois, L.; Lambert, T.; Lastovetsky, A. Recent Advances in Matrix Partitioning for Parallel Computing on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 218–229.
37. Hovestadt, M.; Kao, O.; Keller, A.; Streit, A. Scheduling in HPC resource management systems: Queuing vs. planning. In Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, Seattle, WA, USA, 24 June 2003; Springer: Berlin/Heidelberg, Germany, 2003; pp. 1–20.
38. Broquedis, F.; Clet-Ortega, J.; Moreaud, S.; Furmento, N.; Goglin, B.; Mercier, G.; Thibault, S.; Namyst, R. hwloc: A generic framework for managing hardware affinities in HPC applications. In Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, Pisa, Italy, 17–19 February 2010; pp. 180–186.
39. Newburn, C.J.; So, B.; Liu, Z.; McCool, M.; Ghuloum, A.; Du Toit, S.; Wang, Z.G.; Du, Z.H.; Chen, Y.; Wu, G.; et al. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2011), Chamonix, France, 2–6 April 2011; pp. 224–235.
40. Catanzaro, B.; Garland, M.; Keutzer, K. Copperhead: Compiling an embedded data parallel language. In Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, San Antonio, TX, USA, 12–16 February 2011; pp. 47–56.
41. Bondhugula, U.; Hartono, A.; Ramanujam, J.; Sadayappan, P. A practical automatic polyhedral parallelizer and locality optimizer. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008; pp. 101–113.

42. Ansel, J.; Chan, C.; Wong, Y.L.; Olszewski, M.; Zhao, Q.; Edelman, A.; Amarasinghe, S. PetaBricks: A language and compiler for algorithmic choice. *ACM Sigplan Not.* **2009**, *44*, 38–49.
43. Verdoolaege, S.; Carlos Juega, J.; Cohen, A.; Ignacio Gomez, J.; Tenllado, C.; Catthoor, F. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* **2013**, *9*, doi:10.1145/2400682.2400713.
44. Vasilache, N.; Zinenko, O.; Theodoridis, T.; Goyal, P.; DeVito, Z.; Moses, W.S.; Verdoolaege, S.; Adams, A.; Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv* **2018**, arXiv:1802.04730.
45. Vasilache, N.; Zinenko, O.; Theodoridis, T.; Goyal, P.; Devito, Z.; Moses, W.S.; Verdoolaege, S.; Adams, A.; Cohen, A. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically. *ACM Trans. Archit. Code Optim.* **2019**, *16*, 1–26.
46. Bernstein, A.J. Analysis of Programs for Parallel Processing. *IEEE Trans. Electron. Comput.* **1966**, 757–763, doi:10.1109/PGEC.1966.264565.
47. Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* **1978**, *21*, 558–565.
48. Miller, J.; Trümper, L.; Terboven, C.; Müller, M.S. Poster: Efficiency of Algorithmic Structures. In Proceedings of the IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC19), Denver, CO, USA, 17–22 November 2019.
49. Hennessy, J.L.; Patterson, D.A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed.; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2011.
50. Williams, S.; Waterman, A.; Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* **2009**, *52*, 65–76.
51. Stengel, H.; Treibig, J.; Hager, G.; Wellein, G. Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model. In Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, Frankfurt, Germany, 12–16 June 2015; ACM: New York, NY, USA, 2015; pp. 207–216.
52. Balaprakash, P.; Dongarra, J.; Gamblin, T.; Hall, M.; Hollingsworth, J.K.; Norris, B.; Vuduc, R. Autotuning in High-Performance Computing Applications. *Proc. IEEE* **2018**, *106*, 2068–2083.
53. Frigo, M.; Johnson, S.G. FFTW: An adaptive software architecture for the FFT. In Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181), Seattle, WA, USA, 15 May 1998; Volume 3, pp. 1381–1384.
54. Whaley, R.C.; Dongarra, J.J. Automatically tuned linear algebra software. In Proceedings of the SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, Orlando, FL, USA, 7–13 November 1998; pp. 38–38.
55. Dever, M.; Hamilton, G. AutoPar: Automatic Parallelization of Functional Programs. In Proceedings of the 2014 Fourth International Valentin Turchin Workshop on Metacomputation, META, Pereslavl-Zalessky, Russia, 29 June–3 July 2014; Volume 2014, p. 1125.
56. Bondhugula, U.; Baskaran, M.; Krishnamoorthy, S.; Ramanujam, J.; Rountev, A.; Sadayappan, P. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In Proceedings of the International Conference on Compiler Construction, Budapest, Hungary, 29 March–6 April 2008; Springer: Berlin/Heidelberg, Germany, 2008; pp. 132–146.
57. Li, Z.; Atre, R.; Ul-Huda, Z.; Jannesari, A.; Wolf, F. DiscoPoP: A profiling tool to identify parallelization opportunities. In Tools for High Performance Computing 2014; Springer: Berlin/Heidelberg, Germany, 2015; pp. 37–54.
58. Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, San Jose, CA, USA, 20–24 March 2004; IEEE Computer Society: Washington, DC, USA, 2004; p. 75.
59. Schäling, B. *The Boost C++ Libraries*; Boris Schäling: 2011.