Noname manuscript No.
(will be inserted by the editor)

Water-Level scheduling for parallel tasks in compute-intensive application components

Robert Dietze · Michael Hofmann · Gudula Rünger

Received: date / Accepted: date

Abstract The development of complex simulations with high computational demands often requires an efficient parallel execution of a large number of numerical simulation tasks. Exploiting heterogeneous compute resources for the execution of parallel tasks can be achieved by integrating dedicated scheduling methods into the complex simulation code. In this article, the efforts for developing an application from the area of engineering optimization consisting of various individual components are described. Several scheduling methods for distributing parallel simulation tasks among heterogeneous compute nodes are presented. Performance results and comparisons are shown for two novel scheduling methods and several existing scheduling algorithms for parallel tasks. A heterogeneous compute cluster is used to demonstrate the scheduling and execution of benchmark tasks and FEM simulation tasks.

Keywords component-based development · distributed simulations · parallel tasks · scheduling methods · heterogeneous platforms

1 Introduction

Complex simulations in science and engineering often comprise of large numbers of compute-intensive numerical tasks that dominate significantly the time-to-solution. Increasing the problem sizes for these kinds of applications is usually limited since the number of tasks and their individual execution times should not be too high. Exploiting HPC resources, such as current heterogeneous platforms or future ultrascale computing systems [6], provides an important way to overcome such limitations. However, especially for a variable number of tasks that can be executed in parallel itself (i. e., parallel tasks), the

Robert Dietze · Michael Hofmann · Gudula Rünger
Department of Computer Science, Technische Universität Chemnitz, Chemnitz, Germany
E-mail: {dirob,mhofma,ruenger}@cs.tu-chemnitz.de

efficient utilization of compute resources requires dedicated scheduling methods to be integrated and used within the application program.

The complex simulation which we consider is an application from mechanical engineering for optimizing lightweight structures based on numerical simulations. This class of applications is extensively studied in the research project MERGE[1] in a research group comprising of mathematicians, computer scientists, and especially mechanical engineers who are interested in developing new lightweight parts and constructions. The simulations cover the manufacturing process of short fiber-reinforced plastics and the characterization of their mechanical properties for specific operating load cases [9]. To optimize these parts and constructions, an optimization problem is set up which is then solved by performing the simulations several times with different parameter sets. The goal is to develop an optimal set of parameters. The optimization problem is to be solved using HPC platforms with a variety of compute nodes and depending on the specific optimization problem at hand it is desirable to be able to port the optimization software to very different platforms. This leads to some challenges for the parallel and distributed implementation.

The aim of the software development is twofold. On the one hand, the specific optimization problem developed in the project should be implemented such that the software is able to run on very different hardware and for applications of different sizes. Thus, the software should be sustainable in the sense that it can be easily adapted by the mechanical engineer or mathematician. On the other hand, different simulation and optimization problems might come up and the software development process itself should be sustainable so that new applications can be easily implemented. Our approach is to build a flexible application program whose individual components can be easily replaced or extended. Long-term reusability of the application program (e. g., for increasing problem scales) is achieved by a thorough support for distributed and heterogeneous platforms. Executing the simulations efficiently on a variety of HPC platforms leads to a scheduling problem for parallel tasks on heterogeneous compute nodes.

In this article, we present a component-based approach for the development of a complex simulation application from engineering optimization. Especially, we investigate the use of scheduling methods for assigning simulation tasks to compute resources with the goal to reduce the total parallel runtime of the entire simulations. We employ task and data parallel scheduling and propose two new scheduling methods for parallel tasks. The Water-Level method uses a best-case estimation of the total parallel runtime to determine the compute resources for each parallel task. The Water-Level-Search method repeats the Water-Level method several times to achieve an iterative improvement of the resulting schedule. All presented methods have been implemented and we show performance results with different tasks on a heterogeneous compute cluster.

[1] MERGE Technologies for Multifunctional Lightweight Structures, `http://www.tu-chemnitz.de/merge`

The three benchmarks for testing the scheduling methods developed have been chosen carefully to reflect different potential behavior of the parallel execution time of parallel tasks in a set of tasks to be scheduled. The benchmark for synthetic parallel tasks is described by a runtime formula of the parallel execution time in which the parallelization overhead can be modified through a parameter, thus reflecting very different performance behavior. A benchmark from numerical linear algebra consists of a set of matrix multiplication tasks each of which executes a DGEMM operation from the OpenBLAS library [16] using matrices of size 4000×4000. This benchmark leads to parallel tasks which have the same execution time and scale well up to the full number of cores of a single compute node. The last benchmark comes from the complex application code that we consider, in which a set of simulation tasks has to be executed. The specific simulation tasks needed in the application code lead to parallel tasks that scale moderately well and, hence, the goal is to achieve a performance improvement by our scheduling methods.

The rest of the article is organized as follows: Section 2 describes the application from mechanical engineering. Section 3 presents the component-based application development. Section 4 presents the scheduling methods for executing the simulations on heterogeneous compute resources. Section 5 shows corresponding performance results. Section 6 discusses related work and Sect. 7 concludes the article.

2 Simulation and Optimization of Lightweight Structures

The numerical optimization of lightweight structures consisting of fiber-reinforced plastics to be developed in the project MERGE is performed by a simulation approach described in the following.

2.1 Simulation of fiber-reinforced plastics

The lightweight structures are manufactured by injection molding, which represents one of the most economically important processes for the mass production of plastic parts. The parts are produced by injecting molten plastic into a mold, followed by a cooling process. Fillers, such as glass or carbon fibers, are mixed into the plastic to improve mechanical properties, such as the stiffness or the durability of the parts. Besides the properties of the materials used, the orientation of the fibers and the residual stresses within the parts have a strong influence on the resulting mechanical properties. Thus, determining the mechanical properties of such short fiber-reinforced plastics requires to consider both the manufacturing process and specific operating load cases for the potential use of the plastic parts.

The manufacturing process is simulated with a computational fluid dynamics (CFD) method that simulates the injection of the material until the mold is filled. The input data of the CFD simulation include the geometry of the

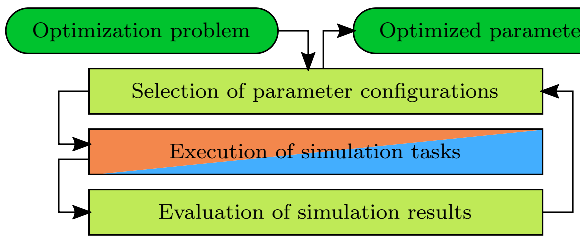
Optimization problem
Optimized parameters
Selection of parameter configurations
Execution of simulation tasks
Evaluation of simulation results

Figure 1 Overview of the coarse structure of the optimization process for lightweight structures.

part, the material properties, such as the viscosity or the percentage of mixed in fibers, and the manufacturing parameters, such as the injection position or pressure. The results of the simulation describe the fiber orientation and the temperature distribution within the part. These results are used for simulating the subsequent cooling process with a finite element method (FEM) that computes the residual stresses within the frozen part.

The simulation of the manufacturing process is followed by an evaluation of the resulting part. Mechanical properties are determined by simulating the behavior of the manufactured part for specific operating load cases of its future use. These simulations are also performed by FEM simulations. Boundary conditions represent the given load cases. The FEM application code employs advanced material laws for short fiber-reinforced plastics and uses the previously determined fiber orientation and residual stresses within the part as input data. The final simulation results describe the behavior of the part, for example, its deformation under an applied surface load.

2.2 Optimization of manufacturing parameters

The goal of the simulation process is not only to simulate one specific manufacturing process of a plastic part but to optimize the properties of the lightweight structures. This is achieved by an optimization process that varies material and manufacturing parameters, such as the fiber percentage or the injection position. Figure 1 shows a coarse overview of the optimization process. The optimization is executed by repeatedly selecting specific values for the parameters to be used and then simulating the manufacturing process and the load cases with the selected parameter configurations as described in the previous subsection. Thus, there is a number of simulation tasks to be executed (i. e., one for each parameter configuration to be simulated) that are independent from each other. The specific number of independent simulation tasks depends strongly on the number of parameters to be varied and on the optimization method to be employed.

Solving optimization problems with objective functions that involve numerical simulations leads to various challenges, such as high computational costs for each evaluation of the objective function, missing derivatives of the objective function, discontinuous and noisy simulation results in many cases, and large numbers of evaluations required for high-dimensional parameter spaces. Therefore in engineering optimization, techniques based on metamodels (or surrogate models) are often used [24]. The goal of these techniques is to create an alternative model of the function that maps input parameters of experiments to experimental results. A small number of experiments, e. g. computationally expensive simulations, is used for the training of a metamodel that is inexpensive to evaluate. The metamodel can then be used for the prediction of experimental results for various input parameters which have not been simulated, for example, to solve an optimization problem.

The optimization of lightweight structures is performed with a Kriging metamodel approach for the global optimization [12] that proceeds as follows. Initial input parameter configurations for creating a metamodel are chosen based on statistical methods for the design of experiments (i. e., either full factorial or Latin Hypercube) [14]. The objective function is evaluated for each initial configuration and the results are used to train the metamodel. The Kriging metamodel provides an interpolation-based method that is especially appropriate for deterministic computer experiments [25]. Furthermore, the Kriging metamodel allows the calculation of the expected improvement for new parameter configurations, thus providing an efficient method for determining new candidates for parameter configurations of the optimization process. The objective function is evaluated for each candidate and the results are used to refine the metamodel. After a specific number of refinement steps, the best remaining candidate is chosen as the optimal solution of the optimization problem.

Evaluating the objective function involves the execution of the simulation tasks and is the most time consuming part of the optimization process. However, since the evaluations are independent from each other, the simulation tasks can be executed at the same time. Thus, the chosen optimization method provides the opportunity to parallelize the evaluations during both the initial training step and in each refinement step. The actual number of evaluations depends, for example, on the complexity of the optimization problem or on the availability of computational resources, but is usually expected to be in the order of tens or hundreds. Performing these computations efficiently on HPC platforms can be supported by an efficient scheduling method that maps the parallel execution of simulation tasks on the various compute nodes.

As an example for the optimization of lightweight structures, we consider the determination of an optimal injection position for the manufacturing of a part made of short fiber-reinforced plastics. Figure 2 (left) shows a plastic part, which is a plate with a hole on one side. The plate is clamped on two sides and a circular surface load is applied leading to the shown deflection in force direction. Figure 2 (right) shows a contour plot of the objective function for the corresponding optimization problem. The optimal injection point shown in the figure leads to a fiber orientation within the plate that minimizes the deflection.

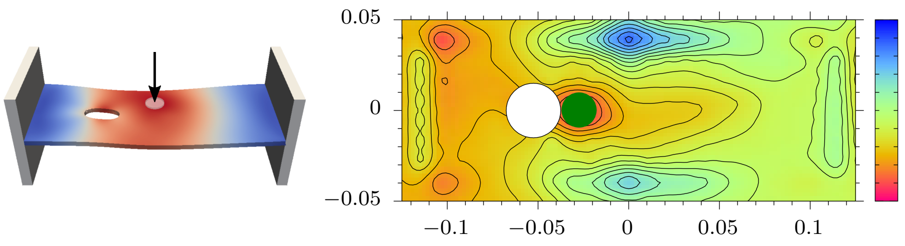

Figure 2 Left: Clamped plate with hole and applied surface load (arrow). Right: Contour plot of the objective function including the obtained minimum (green).

3 Component-based application development

Solving engineering optimization problems as described in Sect. 2 uses a large variety of software programs. In the following, the different software components and their integration into a sustainable application are described.

3.1 Software components for complex optimization applications

For the optimization of lightweight structures, we distinguish the following groups of software programs that are combined into our complex application:

Simulation: The simulation of the manufacturing process and the operating load cases uses computationally intensive numerical applications. The manufacturing by injection molding is simulated with a customized open-source CFD application based on OpenFOAM, which is a C++ library implementing the finite volume method [11] parallelized with MPI. The simulation of the subsequent cooling process and the operating load cases is performed with an in-house adaptive 3D FEM application [5] parallelized with OpenMP.

Optimization: Compared to the objective function of the optimization problem, the optimization methods employed are less computationally intensive. The main optimization component is implemented in Python, since Python provides a fast and flexible way of composing the solution methods for different optimization problems. Existing Python modules, such as NumPy [30], scikit-learn [18], and pyDOE [20], are used to implement the Kriging-based optimization method.

Data management: Scientific simulations and their integration into complex applications require appropriate strategies for the management and exchange of various data. For the optimization application, this includes a repository for optimization problems, dedicated storages for simulation results as well as programs for data conversion and analysis. Depending on the specific optimization problems and application programs involved, local or distributed file-based storages as well as a centralized MySQL database are employed.

User interaction: The specification of optimization problems and their parameters as well as the evaluation of simulation and optimization results are to be performed manually by the user. For the optimization application, a dedicated Python application called Injection Molding Process Optimization Tool (IMPOT) with a graphical user interface has been developed to support the configuration of optimization problems and parameters, the selection of optimization methods, simulation applications, and compute resources as well as the evaluation of the optimization results. Further analysis and visualization of simulation results is performed by scientific applications, such as ParaView [8] or VTK [26].

The large variety of the software components means that several different execution platforms might be involved in the execution. For example, computationally intensive numerical simulations require the use of HPC platforms while user-oriented software, such as IMPOT, will be executed on desktop platforms, such as laptops or PCs. The selection of the execution platforms might also depend on the specific optimization problem to be solved or on the availability of compute resources. For example, during the development process usually only small problem sizes are considered and, thus, all software components can be executed locally on a single platform (e. g., desktop PC). Increasing the problem sizes or switching to production runs will then require to distribute one or more software components among several platforms.

3.2 Component-based development for sustainable applications

Achieving a sustainable solution for the complex application requires the flexibility to add or replace existing software components and to distribute the execution among different platforms without extensive additional development efforts. In [10], we have proposed a method for building complex simulation programs for distributed computing systems to enable such a sustainable development process. Furthermore, we presented the Simulation Component and Data Coupling (SCDC) library specifically designed as programming support for these applications. Our approach leads to an application development that is similar to other component-based architectures [4,17]. However, instead of providing an entire task-based computational framework, the SCDC library is designed as lightweight application-independent programming support that can be easily integrated into existing application codes. The detailed description of the SCDC library and its comparison to other approaches for the development of complex scientific simulations is given in [10].

The SCDC library provides a service-oriented approach for the coupling of independent software components and can be utilized through C or Python programming interfaces. All interactions between software components are organized as data exchanges between client and service components. These data exchanges are performed transparently by the SCDC library utilizing different data exchange methods, such as direct function calls, inter-process communication, or network communication. The library functions provide mechanisms

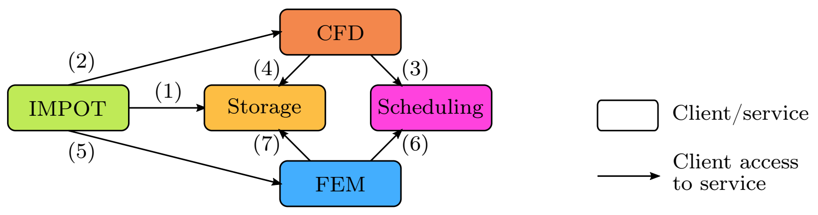

Figure 3 Overview of the components for a service-oriented execution of the optimization application for lightweight structures. Interactions (1)–(7) represent client accesses to services performed with the SCDC library.

for setting up existing application programs as services as well as to access these services from within other application programs as clients.

Figure 3 illustrates the service-oriented implementation of the complex optimization application based on the SCDC library. The IMPOT application with its graphical user interface executes the optimization method, thus representing a client that accesses other components. Executing several simulation tasks (e. g., for different parameter configurations, see Sect. 2.2) is separated in two phases. The first phase performs the CFD simulations as follows:

- The input data of the CFD simulations is first transferred from the IMPOT application to the storage service (1) and then the CFD simulation tasks are submitted to the CFD service (2). Interaction between client and service components is performed by blocking operations, which means that the submission of the CFD simulation tasks to the CFD service (2) is not completed until all current CFD simulation tasks are finished.
- The CFD service uses the information about the CFD simulation tasks and the available compute resources to derive a scheduling problem (see Sect. 4). Solving this problem requires to determine an assignment of the compute resources to the CFD simulation tasks (i. e., a schedule) such that the total time for executing all CFD simulation tasks is minimized. The scheduling algorithms are implemented as a dedicated scheduling service that is accessed by the CFD service (3).
- The CFD simulation tasks are executed by the CFD service, which first retrieves the input data from the storage service (4) and then executes the parallel CFD application on the compute resources given by the schedule. The simulation results are also transferred to the storage service. After the CFD service finished all CFD tasks, the submission of the CFD simulation tasks performed by the IMPOT application (2) is completed.

In the second phase, the same steps are performed for the FEM simulation tasks (i. e., interactions (1), (5)–(7) in Fig. 3). Finally, the IMPOT application retrieves the simulation results from the storage service and decides how to continue the optimization method. For example, if the Kriging-based optimization method requires a refinement of its metamodel, then a new set of simulation tasks is executed by repeating the two phases. In each of these phases, a scheduling problem is solved using the scheduling algorithms presented in the next section.

4 Scheduling parallel tasks on heterogeneous HPC platforms

The scheduling problem emerging when simulating lightweight structures and several scheduling methods for utilizing heterogeneous HPC clusters are presented in the following.

4.1 Scheduling problem for independent parallel tasks

Independent numerical simulations are given as n_T parallel tasks $T_1, \ldots, T_{n_T}$. The parallel runtime $t_i(p)$ of the tasks T_i, $i = 1, \ldots, n_T$ is assumed to be previously determined with benchmark measurements on a specific reference compute node where p denotes the number of cores employed. It is also assumed that it is known whether a task is capable of being executed either on a single node only (SN task, e. g., for OpenMP-based codes) or on a cluster of nodes (CN task, e. g., for MPI-based codes).

The compute resources of the heterogeneous HPC platform are described by a machine model with n_N compute nodes $N_1, \ldots, N_{n_N}$. For each node N_j, $j \in \{1, \ldots, n_N\}$, its number of processor cores p_j and a performance factor f_j is given. The performance factor f_j defines the computational speed of compute node N_j as the ratio of the sequential execution time of a task on the reference compute node and on the compute node N_j. The nodes are grouped into n_C clusters $C_1, \ldots, C_{n_C}$ such that each cluster is a subset of nodes and each node is part of exactly one cluster. Each cluster is able to execute a CN task in parallel (e. g., MPI-based) on all its nodes.

A schedule for the tasks T_i, $i = 1, \ldots, n_T$ to be executed on the compute nodes N_j, $j = 1, \ldots, n_N$ is given by the following information for each task T_i:

- the set of compute nodes and their numbers of cores to be utilized,
- the estimated start time s_i and finish time e_i.

The makespan of a schedule is then defined as the time difference between the earliest start time of all tasks and the latest finish time of all tasks. We assume that the earliest start time is 0 and, thus, the makespan is equal to $\max_{i=1,\ldots,n_T} e_i$. The goal is to determine a schedule such that the makespan is minimized. Furthermore, we will also determine all tasks that are immediate predecessors of a task and executed on the same compute node. With this information, it will be possible to wait for the completion of the predecessor tasks, especially if the runtimes in practice differ from the estimated runtimes.

4.2 Task and data parallel executions

The following task and data parallel schemes [7] are used as reference methods:

Pure Task Parallel (TaskP): The task parallel scheduling scheme for independent parallel tasks is defined to be a scheme in which exactly one core is assigned to each task (i. e., executed sequentially). This leads to an

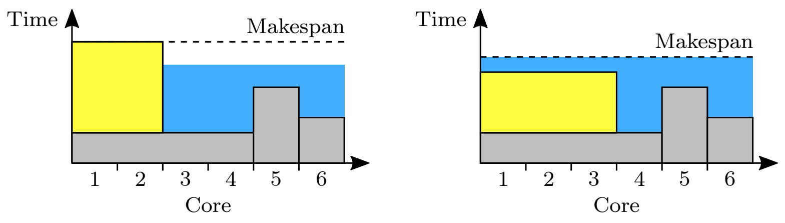

Figure 4 Scheduling of one task (yellow) either on two (left) or three (right) cores with previously scheduled tasks (gray) and optimally executed remaining tasks (blue).

execution of as many tasks as possible at the same time in parallel to each other.

PURE DATA PARALLEL (DATAP): The data parallel scheduling scheme for independent parallel tasks is defined to be a scheme in which as many cores as possible are assigned to each task. Depending on the properties of the tasks (i. e., SN or CN task, see Sect. 4.1), either all cores of a node or all cores of a cluster are used.

The scheduling is performed by iterating over the tasks and selecting the compute resources to be utilized according to the task parallel or data parallel scheme. To favor an early execution of long running tasks, the tasks are first sorted in descending order based on their sequential runtimes. A single task is then assigned to the compute resource that provides the earliest finish. Both schemes are adapted to heterogeneous compute resources by scaling the given runtimes of the tasks with the performance factors of the compute nodes.

4.3 WATER-LEVEL method

We propose a new scheduling strategy which we call WATER-LEVEL method (WATERL). For each task, the WATER-LEVEL method selects the compute nodes and the number of cores for which an estimation of the resulting makespan reaches a minimum. This estimation is determined by assigning a task temporarily to specific compute nodes and cores and assuming all remaining tasks are executed fully parallel without parallelization overhead on the entire set of compute resources. Figure 4 shows an illustration of the WATER-LEVEL strategy in which the current task to be scheduled (yellow) will use either two (left) or three (right) cores. All tasks that are not scheduled yet (blue) are assumed to be executed fully parallel on all cores (i. e., they are distributed like “water” over the “task landscape”). In this example, the current task will be assigned to three cores since the estimation of the resulting makespan (i. e., the “water level”) reaches a minimum.

The pseudocode of the WATER-LEVEL method is given in Fig. 5. To ease the following description, we assume that only SN tasks are given. Supporting also CN tasks can be achieved by iterating over clusters and their numbers of cores instead of compute nodes and their numbers of cores (line 13). The method

```
1  **input** : tasks $T_i$, $i = 1, \ldots, n_T$, with runtimes $t_i(p)$
2  **input** : nodes $N_j$, $j = 1, \ldots, n_N$, with $p_j$ cores and performance factors $f_j$
3  **output:** compute node, number of cores, start time and finish time for each task
4  total compute power $P = \sum_{j=1}^{n_N} p_j \cdot f_j$
5  sequential work $W_S = \sum_{i=1}^{n_T} t_i(1)$
6  free work $W_F = 0$
7  latest finish time $e_{max} = 0$
8  sort $T_i$, $i = 1, \ldots, n_T$ in descending order of $t_i(1)$
9  // *assume* $T_1, \ldots, T_{n_T}$ *are sorted in descending order of their sequential runtimes*
10 **for** $i = 1, \ldots, n_T$ **do**
11     $W_S = W_S - t_i(1)$
12     best estimated makespan $m^* = \infty$
13     **for** $j = 1, \ldots, n_N$ and $p = 1, \ldots, p_j$ **do**
14         assign $T_i$ temporarily to $p$ cores of compute node $N_j$
15         $s$ = start time of task $T_i$
16         $e = s + t_i(p)/f_j$
17         $dW_F = (\max(e, e_{max}) - e_{max}) \cdot P$
18         $dW_F = dW_F - (e - s) \cdot f_j \cdot p$
19         estimated makespan $m = \max(e, e_{max})$
20         **if** $W_S > W_F + dW_F$ **then** $m = m + W_S - (W_F + dW_F)/P$
21         **if** $m < m^*$ **then** { $(m^*, j^*, p^*, s^*, e^*, dW_F^*) = (m, j, p, s, e, dW_F)$ }
22     assign $p^*$ cores of node $N_{j^*}$ to task $T_i$ with start time $s^*$ and finish time $e^*$
23     $W_F = W_F + dW_F^*$
24     $e_{max} = \max(e^*, e_{max})$
```

Figure 5 Pseudocode of the WATER-LEVEL scheduling method.

starts by calculating the total compute power P of all compute nodes with respect to a single core of the reference compute node (line 4). The sequential work W_S is determined as the work for executing all tasks sequentially on the reference compute node (line 5). Additionally, the free work W_F and the latest finish time of all tasks e_{max} are initialized (lines 6 and 7). While e_{max} corresponds to the makespan of the determined schedule, the free work W_F is equal to amount of work that can be executed by all compute resources without increasing this makespan.

The determination of a schedule of the tasks proceeds as follows. The tasks are sorted in descending order based on their sequential runtimes (line 8) and a loop iterates over all tasks in that sorted order (line 10). For the current task $T_i, i \in \{1, \ldots, n_T\}$, the remaining sequential work W_S of all unscheduled tasks is calculated (line 11). The currently best estimated makespan m^* that is achieved for scheduling the task T_i is initialized in such a way (line 12) that it is larger than the following estimation. Then, a loop iterates over the compute nodes and their numbers of cores (line 13) to determine the compute node and the number of cores to be assigned for the task T_i.

The current task T_i is temporarily assigned to the currently considered p cores of compute node N_j (line 14), thus leading to a potential start time s (line 15). The potential finish time e is then calculated from the parallel runtime $t_i(p)$ of the task scaled with the performance factor f_j of the currently considered compute node N_j (line 16). Estimating the makespan for this assignment starts by calculating the potential change dW_F of the free work as follows. The free work is increased by the amount of work that can be executed by all compute nodes if the latest finish time e_{max} is increased to e (line 17). The free work is decreased by the amount of work that is required to execute task T_i with p cores on compute node N_j (line 18). The estimated makespan m is now calculated as the maximum of the current latest finish time e_{max} and the finish time e of the current task (line 19). However, if the remaining sequential work W_S is larger than the free work $W_F + dW_F$, then the estimated makespan m has to be further increased (line 20). This increase corresponds to the time required to execute additional work $W_S - (W_F + dW_F)$ on all compute resources (i. e., with compute power P). If the estimated makespan m is smaller than the best estimated makespan m^*, then the corresponding values are store (line 21).

After the loop over the compute nodes and their numbers of cores is completed, the stored values of the best estimated makespan are used for the task T_i (line 22). Additionally, the free work W_F and the latest finish time e_{max} are updated with the stored values (lines 23 and 24). Estimating the makespan (lines 14–21) requires only constant time. Thus, the overall time of the Water-Level method is linear in the number of tasks, compute nodes, and cores.

4.4 Water-Level-Search method

The Water-Level method underestimates the makespan when the behavior of the parallel tasks differs strongly from the assumptions of that method. This can happen, for example, when tasks have a large parallelization overhead. The Water-Level method then typically assigns too many cores to a parallel task, which would perform better with a smaller number of cores. In this case, the resulting makespan of the schedule is to high. To improve the scheduling method, we propose a search-based approach called Water-Level-Search method (WLSearch) that is able to correct such misestimations.

Instead of minimizing an estimation of the makespan, the Water-Level-Search method uses a makespan limit $\hat{m}$ to determine the compute node and the number of cores for each task in such a way that this makespan limit is not exceeded. The makespan limit is initialized with $\hat{m} = W_S/P$, i. e. the total sequential work W_S of all tasks on the reference compute node divided by the total compute power P of all compute nodes (see lines 4 and 5 in Fig. 5). Whenever a task can not be scheduled without exceeding the makespan limit, the makespan limit is increased and the scheduling of all tasks is restarted. If the makespan limit was increased and all tasks could be scheduled, then the makespan limit might still be to high and a search for a better lower makespan limit is performed.

The Water-Level-Search method consists of three phases. Each phase proceeds similar to the Water-Level method shown in Fig. 5, but without estimating the makespan m (lines 17–20).

Phase 1: The loop over the compute nodes and their numbers of cores (line 13) is stopped if the potential finish time e is less than or equal to the makespan limit $\hat{m}$. In this case, the current compute node and cores are assigned to the current task and the next task is scheduled. If the makespan limit $\hat{m}$ is exceeded by all finish times e, then the makespan limit is set to the smallest finish time and the scheduling of all tasks is potentially restarted. More specifically, the restarts are only performed after $\frac{n_T}{2}$, $\frac{3n_T}{4}$, $\frac{7n_T}{8}$, … tasks to limit the number of restarts to $\mathcal{O}(\log n_T)$.

Phase 2: Since the last makespan limit $\hat{m}$ might still be to large, a binary search for a smaller makespan limit is performed. A list of candidates for the search is created by collecting all finish times e during an execution of the algorithm shown in Fig. 5. The resulting number of candidates is limited to $n_T \sum_{j=1}^{n_N} p_j$.

Phase 3: The binary search is performed as long as there are candidates left for the makespan limit. In each step of the search, the median of the candidates is used as the makespan limit $\hat{m}$ to perform the algorithm shown in Fig. 5 in the same way as in Phase 1. Depending on whether the makespan limit is exceeded or not, the candidates below or above the median are removed. Finally, the last remaining candidate is used as the makespan limit that leads to the schedule determined by the WATER-LEVEL-SEARCH method.

Each phase performs the algorithm shown in Fig. 5 one or several times. The total number of repetitions depends logarithmically on the number of tasks and the total number of cores.

5 Performance results of the scheduling methods

In this section, we present performance results of the scheduling methods described in Sect. 4 on a heterogeneous compute cluster.

5.1 Experimental setup

The heterogeneous compute cluster used consists of $n_N = 8$ compute nodes, each with two multi-core processors. Table 1 lists the nodes and their specific processors. The compute node `cs1` is used as the reference compute node and the performance factors of all other compute nodes are calculated as the ratio of the corresponding sequential execution times of the tasks (see Sect. 4.1). The scheduling methods described in Sect. 4 are implemented in Python. Additionally, we have implemented two methods that represent existing approaches for the scheduling of parallel tasks on heterogeneous compute clusters:

HCPA: The HETEROGENEOUS CRITICAL PATH AND ALLOCATION method [15] transforms individual computational speeds of processors into additional "virtual" processors with equal speed. The scheduling is then performed with an existing method for homogeneous compute clusters (i. e., CPA [22]).

Table 1 List of the compute resources used.

Nodes	Processors	#Nodes × #Processors × #Cores	GHz
`cs1`,`cs2`	Intel Xeon E5345	$2 \times 2 \times 4$	2.33
`sb1`	Intel Xeon E5-2650	$1 \times 2 \times 8$	2.00
`ws1`,...,`ws5`	Intel Xeon X5650	$5 \times 2 \times 6$	2.66

However, the transformation between real processors and virtual processors requires to use the runtime formula of Amdahl's law for the parallel runtimes of the tasks. We have determined such a runtime formula for our benchmark tasks with a least square fit of the parallel execution times measured on the reference compute node.

Δ-CTS: The *Δ*-Critical Task Set method [27] extends an existing scheduling method for sequential tasks on heterogeneous compute clusters (i. e., HEFT [28]) to parallel tasks. The compute node and the number of cores is determined separately for each task such that the earliest finish time of the task (i. e., based on the given runtime formula) is minimized. Additionally, the number of similar tasks (i. e., with similar sequential execution time) executed at the same time is maximized, thus limiting the maximum number of cores to be used by each task.

For the experiments, we have obtained the predicted makespan of each determined schedule and the measured makespan for executing the tasks according to this schedule. Executing the tasks is conducted by a Python script that runs on a separate front-end node of the compute cluster and uses SSH connections to the compute nodes. The measurements are performed 5 times and the average result is shown.

5.2 Results with synthetic tasks

As synthetic benchmark, we employ "sleep" tasks that perform no computations, but only wait for a specific time $t(p) = 10s \cdot \left[x \cdot \frac{1}{p} + (1-x) \cdot (\log p + p)\right]$ to simulate the runtime of typical parallel tasks. The time comprises of a fraction $x \cdot \frac{1}{p}$ which decreases linearly with the number of cores p and a remaining part $(1-x) \cdot (\log p + p)$, which increases logarithmically and linearly. The formula was chosen to model the runtime of a typical parallel task that comprises of parallel computations and of parallelization overhead (e. g., for synchronization or communication) that slows down the parallel execution if the number of cores is to large.

Figure 6 shows the measured makespan of the synthetic tasks with $x = 1.0$, i. e. without parallelization overhead (left) and $x = 0.95$, i. e. with a parallelization overhead (right) depending on the number of tasks, using 4 different scheduling methods. The compute node `cs1` with a total of 8 cores is used as compute resource. The TaskP method achieves the same results for both kinds of tasks. This is expected, since the tasks are always executed sequentially in this scheduling scheme. The makespan shows a step-wise increase after

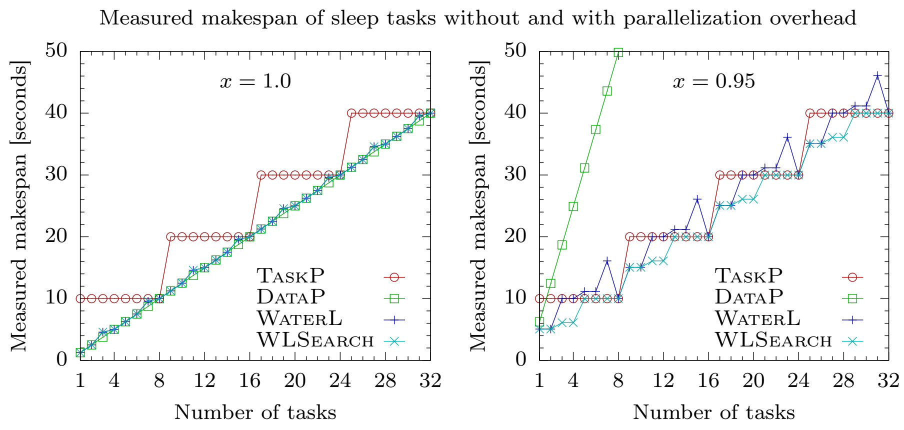

Figure 6 Measured makespan of synthetic tasks with $x = 1.0$, i. e. without parallelization overhead (left) and $x = 0.95$, i. e. with parallelization overhead (right) using different scheduling methods and compute node `cs1`.

every 8 additional tasks. With these numbers of tasks, i. e. 8, 16, 24, . . . , all 8 cores of the compute node are equally utilized and any additional task leads to a higher makespan. The DataP method uses always the maximum number of 8 cores for each task and the makespan shows strong differences between the two kinds of tasks. With $x = 1.0$, the parallel runtime of a task decreases linearly and the minimum runtime is achieved using the maximum number of cores (see Fig. 6 (left)). However, with $x = 0.95$, using the maximum number of cores leads to a strong increase of the makespan due to the increasing parallelization overhead of the parallel tasks (see Fig. 6 (right)).

The WaterL and WLSearch methods achieve a good trade-off between the TaskP and DataP methods. With $x = 1.0$, the minimal makespan as with the DataP method is achieved, because the WaterL method estimates the behavior of tasks without parallelization overhead very well. A further improvement with the WLSearch method is not possible. With $x = 0.95$, the results are close to the results of the TaskP method, but with a more continuous increase instead of the step-wise increase. Especially with less than a multiple of 8 tasks, the WaterL method leads to a higher makespan than the TaskP method. This behavior is caused by the estimation of the makespan by the WaterL method that is based only on the sequential execution time of the tasks. Neglecting the parallelization overhead underestimates the makespan and leads to a choice of a number of cores for tasks that is too high. The WLSearch method solves the drawback if such an underestimation occurs by restarting the scheduling of all tasks with an increased estimation of the makespan. The restarts provide opportunities to use a smaller number of cores for the tasks, thus allowing to reach at least the same result as the TaskP method.

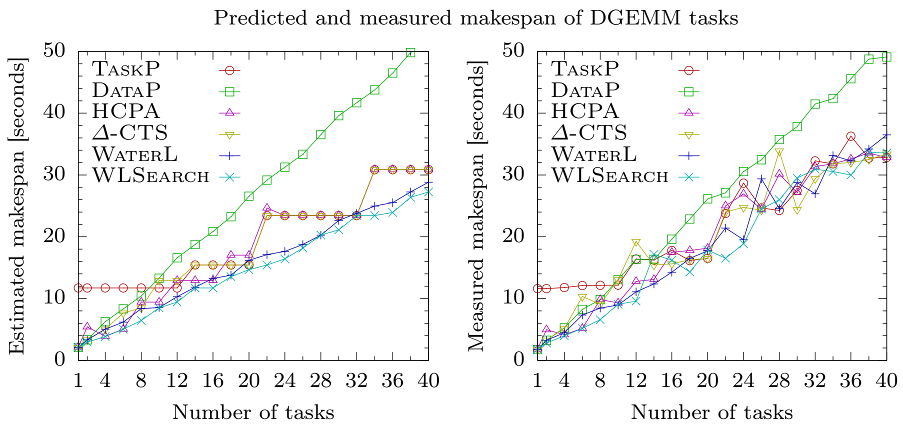

Figure 7 Predicted makespan (left) and measured makespan (right) of DGEMM tasks using different scheduling methods and compute nodes `cs1` and `ws1`.

5.3 Results with DGEMM tasks

The number of cores utilized by the DGEMM operation of the OpenBLAS library is controlled with the environment variable `OPENBLAS_NUM_THREADS`. The parallel runtime of a DGEMM task using p cores is modeled with the runtime formula $t(p) = {}^{a}/_{p^b} + c$. The parameters a, b, and c are determined with a least square fit (by Gnuplot) of the parallel execution times measured on the reference compute node `cs1`. The resulting runtime with matrices of size 4000×4000 comprises of a constant part $c = 2.30$ seconds and a parallel part $a = 13.09$ seconds that decreases proportionally to ${}^{1}/_{p^{1.09}}$. Thus, even though the parallel part scales very well, there is always a significant sequential part.

Figure 7 (left) shows the predicted makespan of DGEMM tasks depending on the number of tasks, using 6 different scheduling methods. The two compute nodes `cs1` and `ws1` with a total of 20 cores are used as compute resources. Similar to Fig. 6 (left), the TaskP method shows a step-wise increase of the execution times due to the sequential execution of the tasks while the DataP method shows a strong increase due to the parallelization overhead caused by always using the maximum number of cores. The existing methods HCPA and Δ-CTS lead to an improved makespan for small numbers of tasks. However, if the number of tasks is larger than the total number of cores, then the same step-wise increase as the TaskP method occurs. In this case, only one core is assigned to each task, because both methods chose the maximum number of cores to be assigned to a task without considering the potential utilization of cores by other tasks.

The WaterL method leads to a continuous increase of the makespan for increasing numbers of tasks. Furthermore, the makespan of the WaterL method is always below or at most equal to the best results of the TaskP method. This is the expected behavior, because the estimation of the makespan that is used to determine the number of cores to be assigned to a task represents a prediction of the remaining utilization of the cores. If there are many tasks left, then this utilization is high and a task parallel execution is preferred. Otherwise, the predicted utilization is low and a data parallel execution of the few remaining tasks is chosen. The results demonstrate this intended trade-off between task and data parallel executions. However, due to the significant sequential part within the parallel DGEMM task, the WaterL method also underestimates the makespan. By repeatedly improving the estimation of the makespan, the WLSearch method further improves the makespan of the WaterL method by about 6 % on average.

Figure 7 (right) shows the measured makespan achieved by executing the DGEMM tasks according to the determined schedule. The results of the DataP method are close to the predicted makespan (left), but still lead to the highest makespan results. All other methods lead to strongly varying results. This behavior is caused by the DGEMM tasks which influence each other when executed on the same compute node. Due to the significant sequential part of the tasks, all methods except the DataP method prefer the execution of multiple tasks at once. However, especially the step-wise increase of the TaskP, HCPA, and Δ-CTS methods is still visible. The majority of the smallest measured makespans is achieved with the WLSearch method, thus confirming the results of the predicted makespan (left).

5.4 Results with FEM simulation tasks

The FEM code [5] is parallelized with OpenMP and simulates the cooling of a lightweight structure as described in Sect. 2. The number of cores utilized by threads is controlled with the OpenMP environment variable `OMP_NUM_THREADS`. Since the different parameter sets to simulate usually do not affect the runtime of the FEM code, all parallel tasks have the same execution time. The parallel runtime is modeled with the formula $t(p) = a/p^b + c$ and the parameters a, b, and c are determined with a least square fit (by Gnuplot) of the parallel execution times measured on the reference compute node `cs1`. The resulting runtime comprises of a constant part $c = 4.47$ seconds and a parallel part $a = 71.07$ seconds that decreases proportionally to $1/p^{0.42}$. Thus, each single FEM task scales only moderately well.

Figure 8 (left) shows the predicted makespan of FEM tasks depending on the number of tasks, using 6 different scheduling methods. All compute nodes listed in Table 1 with a total of 92 cores are used as compute resources. The DataP method leads to a strong increase of the makespan, which is expected since the parallel FEM code scales only moderately well. For the FEM tasks, the DataP method is only advantageous if there are very small numbers of tasks (i. e., less than twice the number of compute nodes). The TaskP method shows a step-wise increase of the makespan, but with different widths and heights of the steps due to the different numbers of cores and computational speeds of the compute nodes used. The makespan of the HCPA method varies

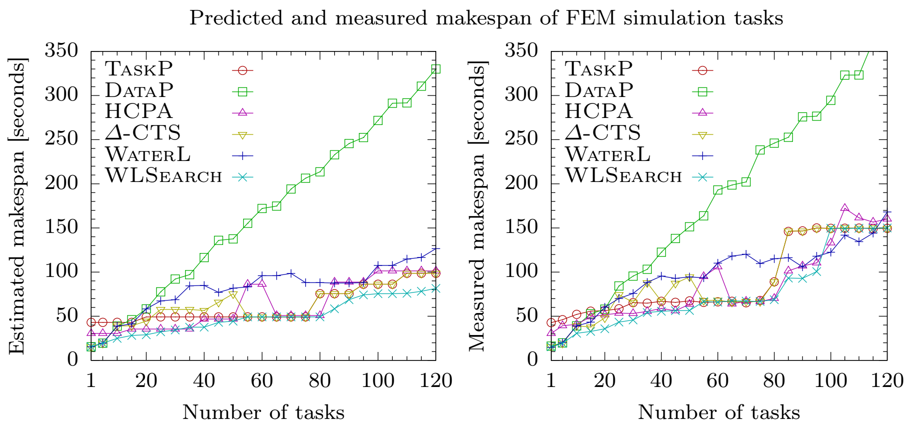

Figure 8 Predicted makespan (left) and measured makespan (right) of FEM tasks using different scheduling methods and all compute nodes listed in Table 1.

around the TaskP method while the makespan of the Δ-CTS method is either higher than or equal to the TaskP method.

The makespan of the WaterL method is up to a factor of two higher than the TaskP method. This behavior is caused by the FEM code that favors an execution with small numbers of cores and the fact that the WaterL method assumes an optimal parallel execution for its estimation of the makespan, which differs strongly from the actual parallel runtimes of the FEM tasks. The WLSearch method solves this problem by repeating the WaterL method several times with improved estimations of the makespan. In general, this improvement is especially important for parallel codes that scale only moderately well. The resulting makespan of the WLSearch method is always the best in comparison to all other methods.

Figure 8 (right) shows the measured makespan achieved by executing the FEM tasks according to the determined schedules. The results confirm the general behavior of the methods, which was previously seen in the predicted makespan results, i. e. the WLSearch method achieves the smallest execution times for a broad range of numbers of tasks. However, the measured makespan results are up to about a factor of 1.4 higher with less than 80 tasks and up to a factor of 2 higher for larger numbers of tasks. This strong increase occurs if a high number of sequential FEM tasks is executed on single nodes at the same time. One reason might be that due to the limited memory bandwidth of the compute nodes `cs1` and `cs2`, the sequential runtimes of FEM tasks almost double if all eight cores execute a separate (sequential) FEM task. However, this affects all scheduling methods that rely on a given runtime formula for the execution times of the tasks.

The parallel speedup $^{T_s}/_{T_p}$ of the FEM tasks shown in Figure 9 demonstrates the performance improvement. The sequential runtime T_s corresponds to a sequential execution of all FEM tasks using only one core of compute node `sb1` (i. e., the compute node with the best sequential performance). The

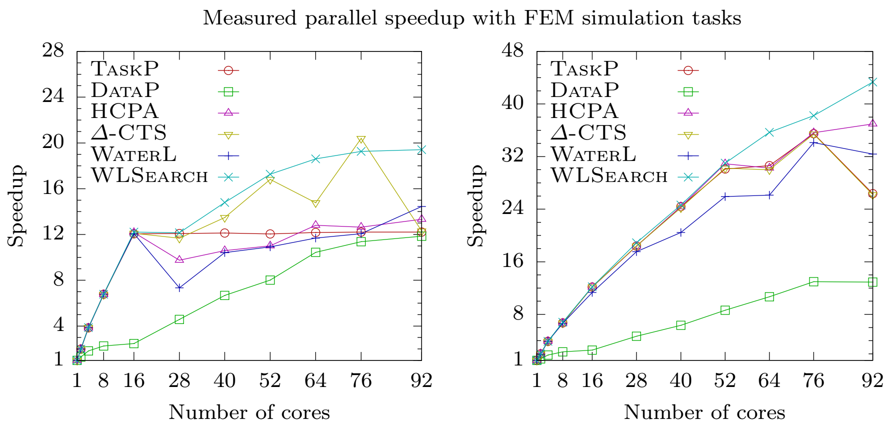

Figure 9 Parallel speedup achieved for the execution of 16 (left) and 92 (right) FEM tasks using different scheduling methods.

parallel runtime T_p corresponds to a parallel execution of all FEM tasks based on a schedule determined for using compute nodes with a total number of p cores. Increasing the number of cores utilizes the compute nodes in order of their performance, i. e. 1–16 cores with compute node `sb1`, 28–76 cores with additional compute nodes `ws1`–`ws5`, and 92 cores with all compute nodes listed in Table 1. Thus, the scheduling methods have to handle the heterogeneity of the different compute nodes and the varying ratios between the number of tasks and the number of cores.

Figure 9 (left) shows the parallel speedup with 16 FEM tasks depending on the number of cores, using 6 different scheduling methods. The DataP method always leads to the lowest speedup results, thus demonstrating the need for an appropriate scheduling of the FEM tasks in favor of the parallel execution of the FEM application itself. Up to 16 cores, all methods except the DataP method, assign the 16 cores of compute node `sb1` equally to the tasks, thus leading to the same results. However, especially with 16 cores, the parallel speedup of about 12 is lower than expected for executing all 16 tasks sequentially at the same time. This is caused by an increase of the sequential execution times if several FEM tasks are executed on a single compute node, as mentioned before.

Using more than 16 cores leads to varying speedups for the different scheduling methods. The speedup of the TaskP method remains constant, because all 16 tasks are executed sequentially on the 16 cores of the compute node `sb1` and additional compute nodes are not used. The speedup of the HCPA method and the WaterL method decreases with 28 cores and increases only slightly when using additional cores. Both methods assign too many cores to single tasks based on an expected parallel behavior (i. e., Amdahl-like or water-like), which differs strongly from their actual parallel execution. The Δ-CTS method does not experience these effects, because it prevents an unbalanced assignment of cores to tasks. However, the balancing ignores the different computational speeds of the processors, thus leading to a strong decrease of the speedup when additionally using the slow compute nodes `cs1` and `cs2`. The WLSearch method achieves an increasing speedup up to 92 cores. The problematic assignment of too many cores to single tasks is resolved by repeating the assignment of cores to tasks with improved estimations of the makespan. Furthermore, using slower computes nodes does not deteriorate the speedup, because the assignment is based on the parallel runtime of the tasks recognizing also the different computational speeds of the processors.

Figure 9 (right) shows the parallel speedup with 92 FEM tasks depending on the number of cores, using 6 different scheduling methods. In comparison to the results with 16 FEM tasks, all scheduling methods, except the DataP method, achieve a significant speedup with more than 16 cores. This is the expected behavior when using less cores than tasks, because in these cases the additional cores are used to execute more tasks sequentially at the same time. Using 92 cores leads to a significant decrease of the speedup for the TaskP method and the Δ-CTS method. In this case, both methods execute all tasks sequentially and the slow compute nodes `cs1` and `cs2` decrease the overall speedup. Especially, the WLSearch method achieves a further increase when using all 92 cores, thus confirming the good results that were already shown with less tasks (left).

In comparison to the DGEMM tasks used in the previous subsection, the parallel behavior of the FEM simulation tasks has shown to be very challenging for the scheduling methods. The usage of slow compute nodes as well as cases with less tasks than cores affect the efficiency of most of the scheduling methods such that their results are often not better than the task parallel scheme. In contrast to that, the WLSearch method has demonstrated consistently good results in all these cases, thus showing that both benchmark tasks and realistic application tasks can be handled appropriately.

6 Related Work

Scheduling is an important problem supporting the efficient processing in different application areas [19]. One prominent area is the scheduling of sequential and/or parallel tasks to be executed on a given set of hardware resources (e. g., cores, processors, or nodes) while additional dependencies between the tasks may restrict their execution order. Determining an optimal schedule (e. g., with minimal makespan) for tasks with dependencies is an NP-hard problem that is usually solved with heuristics or approximation algorithms [13]. Layer-based scheduling algorithms [23] decompose a set of parallel tasks with dependencies into layers of independent tasks. Each layer is scheduled separately with a scheduling algorithm for independent tasks, e. g. list scheduling. Since the simulation tasks of our optimization application are independent, a decomposition into layers can be omitted.

List scheduling algorithms add priorities to the tasks and assign the tasks in descending order of their priority to the processors. Algorithms, such as

Largest Processing Time (LPT) [3] and Longest Task First (LTF) [29], use the given runtime of the tasks as priorities to assign compute intensive tasks first. Algorithms for heterogeneous platforms, such as Heterogeneous Earliest Finish Time (HEFT) [28] and Predict Earliest Finish Time (PEFT) [1], also take the runtime of the tasks on individual processors into account for the priorities. The Water-Level method proposed in this article is also a list scheduling algorithm that prioritizes the tasks according to their runtime. However, the Water-Level method uses only the sequential runtime as priority while the individual processor speeds of a heterogeneous platform are used for the allocation of cores to parallel tasks and for the selection of compute nodes.

Scheduling parallel tasks can also be performed with a two-step approach consisting of an allocation step followed by a scheduling step. The scheduling step assigns the parallel tasks to specific processors and is usually based on a list scheduling algorithm. The allocation step determines the number of processors for each parallel task. This step is usually performed iteratively starting with an initial allocation (e. g., one processor per tasks) and then repeatedly assigning additional processors to tasks (e. g., to shorten the critical path). The resulting number of repetitions depends linearly on the number of tasks. Example algorithms are Critical Path Reduction (CPR) [21], Critical Path and Allocation (CPA) [22], and Modified Critical Path and Area-based (MCPA) [2]. The Water-Level method performs the allocation of cores only once for each task during the list scheduling and, thus, omits repeated assignments of additional processors to tasks. The Water-Level-Search method repeats the Water-Level method, but with a number of repetitions that depends only logarithmically on the number of tasks.

The HCPA method [15] is an extension of the CPA method and requires to calculate the parallel runtime of tasks according to Amdahl's law (see Sect. 5.1). In contrast, both the WaterL method and the WLSearch use the given parallel runtime of the tasks, thus omitting such limitations to a specific runtime formula. The Δ-Critical Task Set method (Δ-CTS) [27] represents an extension of the HEFT method for parallel tasks and, thus, uses the earliest finish time according the given parallel runtime (see Sect. 5.1). This is also done by the Water-Level method. However, the Δ-CTS method considers only subsets of tasks together (i. e., tasks within a specific range of the so-called bottom level) while the Water-Level method always considers all tasks. Furthermore, the Water-Level-Search method iteratively improves the schedule in several steps while the Δ-CTS method performs only one step. Especially for the FEM simulation tasks from our complex application, the experiments in Sect. 5 have demonstrated that these advantages of the WLSearch method lead to a smaller makespan and to better parallel speedup results than the HCPA method and the Δ-CTS method and, thus, the WLSearch method outperforms the other methods.

7 Conclusion

In this article, we have described a component-based development of a complex scientific application from the area of engineering optimization and have shown that a scheduling problem arises, which has to be solved efficiently for an overall efficient execution. Achieving a sustainable application has been accomplished by a flexible development approach that enables both to add or replace individual software components and to distribute their execution among different platforms. To support this approach, we have proposed two scheduling methods for the efficient execution of parallel simulation tasks on heterogeneous HPC platforms. The Water-Level method performs an iterative assignment of the parallel tasks to compute resources and uses a best-case estimation of the makespan to determine the number of cores to be assigned to tasks. The Water-Level-Search method repeats this assignment with improved estimations of the parallel behavior of the tasks. Performance results for benchmark tasks demonstrate that the goal to achieve a good trade-off between task and data parallel execution schemes has been reached. In comparison to other existing scheduling methods for parallel tasks, the Water-Level-Search method leads to consistent good results for both benchmark tasks and realistic application tasks. A significant speedup is also achieved for the execution of the set of specific FEM simulation tasks from the complex simulation application considered.

Acknowledgements This work was performed within the Federal Cluster of Excellence EXC 1075 "MERGE Technologies for Multifunctional Lightweight Structures" and supported by the German Research Foundation (DFG). Financial support is gratefully acknowledged.

References

1. Arabnejad, H., Barbosa, J.: List scheduling algorithm for heterogeneous systems by an optimistic cost table. Transactions on Parallel and Distributed Systems **25**(3), 682–694 (2014)
2. Bansal, S., Kumar, P., Singh, K.: An improved two-step algorithm for task and data parallel scheduling in distributed memory machines. Parallel Computing **32**(10), 759–774 (2006)
3. Belkhale, K., Banerjee, P.: An approximate algorithm for the partitionable independent task scheduling problem. In: Proc. of the 1990 Int. Conf. on Parallel Processing, (ICPP'90), pp. 72–75 (1990)
4. Bernholdt, D., Allan, B., Armstrong, R., Bertrand, F., Chiu, K., Dahlgren, T., Damevski, K., Elwasif, W., Epperly, T., Govindaraju, M., Katz, D., Kohl, J., Krishnan, M., Kumfert, G., Larson, J., Lefantzi, S., Lewis, M., Malony, A., Mclnnes, L., Nieplocha, J., Norris, B., Parker, S., Ray, J., Shende, S., Windus, T., Zhou, S.: A component architecture for high-performance scientific computing. Int. J. High Performance Computing Applications **20**(2), 163–202 (2006)
5. Beuchler, S., Meyer, A., Pester, M.: SPC-PM3AdH v1.0 - Programmer's manual. Preprint SFB/393 01-08, TU-Chemnitz (2001)
6. Bongo, L.A., Ciegis, R., Frasheri, N., Gong, J., Kimovski, D., Kropf, P., Margenov, S., Mihajlovic, M., Neytcheva, M., Rauber, T., Rünger, G., Trobec, R., Wuyts, R., Wyrzykowski, R.: Applications for ultrascale computing. Supercomputing Frontiers and Innovations **2**(1), 19–48 (2015)

7. Dümmler, J., Kunis, R., Rünger, G.: A comparison of scheduling algorithms for multiprocessortasks with precedence constraints. In: Proc. of the High Performance Computing & Simulation Conference (HPCS'07), pp. 663–669. ECMS (2007)
8. Henderson Squillacote, A.: The ParaView guide: A parallel visualization application. Kitware (2008)
9. Hofmann, M., Ospald, F., Schmidt, H., Springer, R.: Programming support for the flexible coupling of distributed software components for scientific simulations. In: Proc. of the 9th Int. Conf. on Software Engineering and Applications (ICSOFT-EA 2014), pp. 506–511. SciTePress (2014)
10. Hofmann, M., Rünger, G.: Sustainability through flexibility: Building complex simulation programs for distributed computing systems. Simulation Modelling Practice and Theory, Special Issue on Techniques And Applications For Sustainable Ultrascale Computing Systems **58**(1), 65–78 (2015)
11. Jasak, H., Jemcov, A., Tukovic, Z.: OpenFOAM: A C++ library for complex physics simulations. In: Proc. of the Int. Workshop on Coupled Methods in Numerical Dynamics (CMND'07), pp. 1–20 (2007)
12. Kleijnen, J., van Beers, W., van Nieuwenhuyse, I.: Expected improvement in efficient global optimization through bootstrapped kriging. J. of Global Optimization **54**(1), 59–73 (2012)
13. Leung, J. (ed.): Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press (2004)
14. Montgomery, D.: Design and analysis of experiments, 5 edn. Wiley (2000)
15. N'Takpé, T., Suter, F.: Critical path and area based scheduling of parallel task graphs on heterogeneous platforms. In: Proc. of the 12th Int. Conf. on Parallel and Distributed Systems (ICPADS'06), pp. 1–8. IEEE (2006)
16. OpenBLAS: An optimized BLAS library. `http://www.openblas.net/`
17. Parker, S.: A component-based architecture for parallel multi-physics PDE simulation. Future Generation Computer Systems **22**(1–2), 204–216 (2006)
18. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. J. of Machine Learning Research **12**, 2825–2830 (2011)
19. Pinedo, M.: Scheduling: Theory, algorithms, and systems. Springer (2012)
20. pyDOE: Design of experiments for Python. `http://pythonhosted.org/pyDOE`
21. Radulescu, A., Nicolescu, C., van Gemund, A., Jonker, P.: CPR: Mixed task and data parallel scheduling for distributed systems. In: Proc. of the 15th Int. Parallel and Distributed Processing Symposium (IPDPS'01), pp. 1–8. IEEE (2001)
22. Radulescu, A., van Gemund, A.: A low-cost approach towards mixed task and data parallel scheduling. In: Proc. of the Int. Conf. on Parallel Processing (ICPP'01), pp. 69–76. IEEE (2001)
23. Rauber, T., Rünger, G.: Compiler support for task scheduling in hierarchical execution models. J. of Systems Architecture **45**(6–7), 483–503 (1999)
24. Roux, W., Stander, N., Haftka, R.: Response surface approximations for structural optimization. Int. J. for Numerical Methods in Engineering **42**(3), 517–534 (1998)
25. Sacks, J., Welch, W., Mitchell, T., Wynn, H.: Design and analysis of computer experiments. Statistical science **4**(4), 409–423 (1989)
26. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit: An Object-oriented Approach to 3D Graphics. Kitware (2006)
27. Suter, F.: Scheduling Δ-critical tasks in mixed-parallel applications on a national grid. In: Proc. of the 8th IEEE/ACM Int. Conf. on Grid Computing, pp. 2–9. IEEE (2007)
28. Topcuoglu, H., Hariri, S., Wu, M.Y.: Task scheduling algorithms for heterogeneous processors. In: Proc. of the 8th Heterogeneous Computing Workshop (HCW'99), pp. 3–14. IEEE (1999)
29. Turek, J., Wolf, J., Yu, P.: Approximate algorithms scheduling parallelizable tasks. In: Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92), pp. 323–332. ACM (1992)
30. van der Walt, S., Colbert, S., Varoquaux, G.: The NumPy array: A structure for efficient numerical computation. Computing in Science Engineering **13**(2), 22–30 (2011)