NESUS cost IC1305
Network for Sustainable Ultrascale Computing

Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015) Krakow, Poland

Jesus Carretero, Javier Garcia Blas
Roman Wyrzykowski, Emmanuel Jeannot.
(Editors)

September 10-11, 2015

Exploiting Heterogeneous Compute Resources for Optimizing Lightweight Structures

ROBERT DIETZE, MICHAEL HOFMANN, GUDULA RÜNGER

Department of Computer Science, Chemnitz University of Technology, Chemnitz, Germany

{dirob,mhofma,ruenger}@cs.tu-chemnitz.de

Abstract

Optimizing lightweight structures with numerical simulations leads to the development of complex simulation codes with high computational demands. The optimization approach for lightweight structures consisting of fiber-reinforced plastics is considered. During the simulated optimization, independent simulation tasks have to be executed efficiently on the heterogeneous computing resources. In this article, several scheduling methods for distributing parallel simulation tasks among compute nodes are presented. Performance results are shown for the scheduling and execution of synthetic benchmark tasks, matrix multiplication tasks, as well as FEM simulation tasks on a heterogeneous compute cluster.

Keywords Numerical simulations, scheduling, heterogeneous clusters

I. INTRODUCTION

The development of complex simulations in science and engineering leads to various challenges for application programmers, especially when targeting at future ultrascale computing systems [4]. The sustainability and portability of application codes represent important non-functional requirements, which can be provided, for example, with an appropriate methodology for the development process as well as technical support in the form of dedicated programming libraries [7]. By encapsulating the data exchange operations of coupled simulations, it is possible to achieve a flexibly distributed execution of the simulation components on distributed systems. However, for compute-intensive simulations also the efficient utilization of various computing resources, such as HPC servers or clusters, is important.

As an application example for complex simulations, we consider the optimization of lightweight structures based on numerical simulations which are studied in the research project MERGE[1]. The simulations cover the manufacturing process of short fiber-reinforced plastics and the characterization of their mechanical properties for specific operating load cases [6]. For solving the optimization problem, the simulations are performed several times with different parameter sets in order to develop an optimal set of parameters. The efficient execution of the simulations on HPC platforms leads to a task scheduling problem with the following properties:

- The tasks are independent from each other and the number of tasks is usually in the order of tens or hundreds.
- Since each task represents an execution of the same parallel simulation application, all tasks behave almost the same. This means that the expected parallel runtime is the same for all tasks and can be determined previously, for example, with separate benchmark measurements.
- The compute resources to be utilized are hierarchically organized and can comprise several compute clusters. Each cluster consists of several compute nodes and each node contains several compute cores.
- The compute resources are heterogeneous in the sense that each compute node has an individual performance.
- A parallel task can be either a shared-memory application that can be executed only on a single node including several cores or a distributed-memory application that can be executed on a cluster including several nodes.

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

In this article, we investigate the use of scheduling algorithms for assigning simulation tasks to compute resources with the goal to reduce the total parallel runtime of the entire set of simulations. We employ task and data parallel scheduling methods and propose a new scheduling algorithm called WATER-LEVEL method. The WATER-LEVEL method is designed as a trade-off between task and data parallel executions and is based on a best-case estimation of the total parallel runtime of all tasks. All presented methods were implemented to solve to scheduling problem described above. We show performance results with different simulation tasks on a heterogeneous compute cluster.

The rest of this article is organized as follows: Section II presents the application example from mechanical engineering. Section III describes the approaches for scheduling the execution of the simulations on heterogeneous compute resources. Section IV shows corresponding performance results. Section V discusses related work and Sect. VI concludes the article.

II. SIMULATION AND OPTIMIZATION OF LIGHTWEIGHT STRUCTURES

The numerical optimization of lightweight structures consisting of fiber-reinforced plastics can be performed by a simulation approach which is described in the following.

II.1 Simulation of fiber-reinforced plastics

The lightweight structures can be 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 the 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 can be simulated with computational fluid dynamics (CFD) that simulates the injection of the material until the mold is filled. The input data of the CFD simulation include the geometry of the 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 simulation results describe the fiber orientation and the temperature distribution within the part. These result data are used for simulating the subsequent cooling process with an approach based on the finite element method (FEM) that computes the residual stresses within the freezed 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 using boundary conditions that correspond to 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.

II.2 Optimizing 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 done by an optimization process that varies selected material and manufacturing parameters, such as the fiber percentage or the injection position. The optimization is executed by repeatedly selecting specific values for the variable parameters and then simulating the manufacturing process and the load cases for the selected parameter configurations as described in the previous subsection. Thus, there are 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 strongly depends on the number of variable parameters or on the optimization method employed and is usually expected to be in the order of tens or hundreds.

Figure 1 (left) shows an example of a plastic part, which is a plate made of short fiber-reinforced plastics 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. The shown optimal injection point leads to a fiber orientation within the plate that minimizes the deflec-

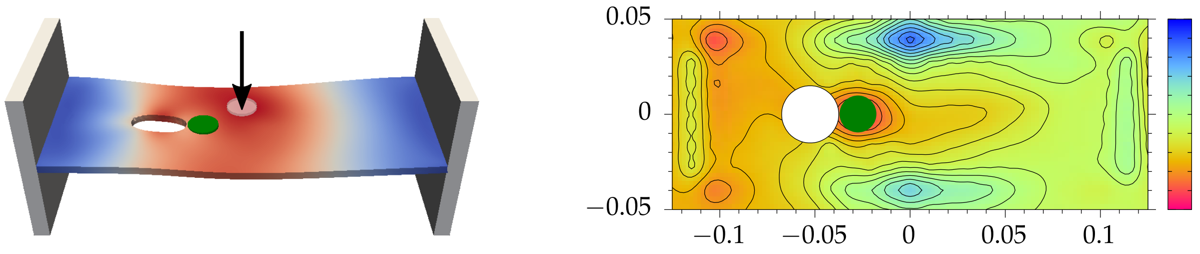

Figure 1: Left: Clamped plate with hole, applied surface load (arrow), and optimal injection point (green). Right: Contour plot of the objective function including the obtained minimum (green).

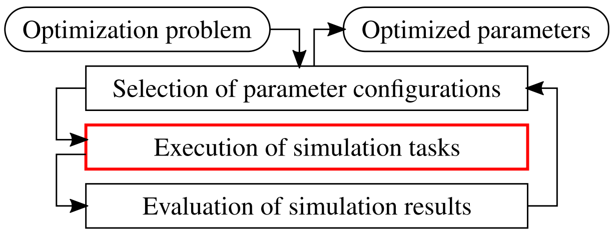
Optimization problem
Optimized parameters
Selection of parameter configurations
Execution of simulation tasks
Evaluation of simulation results

Figure 2: Overview of the optimization process.

tion. Figure 1 (right) shows a contour plot of the objective function for the corresponding optimization problem. The function values were determined during a Kriging-based optimization method [12]. This method creates an arbitrary number of candidate points for the optimal solution which are then recursively improved. The candidate points can be computed independently from each other, thus leading to a number of simulation tasks that can be executed at the same time. Figure 2 gives an overview of the optimization process. The repeated execution of the simulation tasks is the most time consuming part of the optimization process. Thus, performing these computations efficiently on HPC platforms is required and can be supported by an appropriate scheduling method for the parallel execution of simulation tasks on various compute nodes.

III. Distributing Simulations on Heterogeneous HPC Platforms

The optimization process described in Sect. II leads to independent numerical simulations that need be executed efficiently on HPC platforms. In the following, we describe a corresponding scheduling problem and present several scheduling algorithms for utilizing heterogeneous HPC clusters.

III.1 Scheduling problem

Task model: The independent numerical simulations are given as n_T parallel tasks $T_1, \ldots, T_{n_T}$. We assume that the parallel runtime of the simulations was previously determined with benchmark measurements on a specific reference compute node. Thus, for each task T_i, $i \in \{1, \ldots, n_T\}$, the given function $t_i(p)$ specifies the parallel runtime of the simulation when using p processor cores. Furthermore, it is known whether the tasks are capable of being executed either on a single node only (e. g., for OpenMP-based codes) or on a cluster of nodes (e. g., for MPI-based codes).

Machine model: The compute resources of the HPC platform to be used consist of 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 (with respect to the reference compute node) is given. 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 has to be able to execute an appropriate parallel task (e. g., MPI-based) on all its nodes.

Schedule: The goal is to determine an assignment of the given tasks to the compute resources of the HPC platform such that the total runtime for executing all tasks (i. e., the makespan) is minimized. For each task, the resulting schedule contains the compute resources to be used (i. e., nodes and utilized numbers of cores) and the estimated start time. Furthermore, for each task, the list of tasks that utilize the same compute resources immediately before is given. With this information, it will be possible to wait for their completion, especially if the runtimes in practice differ from the estimated runtimes.

III.2 Task and data parallel executions

Scheduling parallel tasks requires to determine the number of parallel cores to be used by each task. The following task and data parallel schemes will be used as reference methods:

Pure task parallel: This scheduling strategy uses only one core for each task and, thus, allows the execution of as many tasks as possible at the same time. The scheduling is performed by creating a list of all cores, using the core from the front of the list for the task to be scheduled next and then moving this core to the back of the list.

Pure data parallel: This scheduling strategy uses as many cores as possible for each task. Depending on the properties of the tasks (see Sect. III.1), either all cores of a node or all cores of a cluster are used as compute resources. The scheduling is performed by creating a list of all compute resources (i. e., either nodes or clusters), using the compute resource from the front of the list for the task to be scheduled next and then moving this compute resource to the back of the list.

Both methods schedule the tasks in their given order and use the compute resources in a round-robin scheme independently from their performance or utilization. We study the following adaptations to create further variants of the task and data parallel scheduling methods: The tasks are sorted in descending order based on their sequential runtimes to favor an early execution of long running tasks. Furthermore, scheduling a task is now performed by selecting the compute resource that provides the earliest finish (EF). This strategy replaces the round-robing scheme and is especially important for heterogeneous compute resources. Overall, we consider four task and data parallel scheduling variants, i. e., the original methods (TASKP and DATAP) and the variants with the earliest finish (TASKP-EF and DATAP-EF).

III.3 WATER-LEVEL method

In addition to the task and data parallel execution schemes described in the previous subsection, we present a further strategy for assigning tasks to compute resources which we call WATER-LEVEL method (WATERL). The method uses the given runtime functions of the tasks and the performance factors of the compute nodes to determine the compute resources for a each task. For realistic tasks, we assume the following

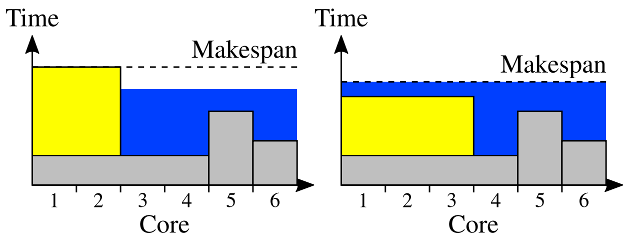

Figure 3: Scheduling of one task (yellow) either on two (left) or three (right) cores, previously scheduled tasks (gray), and optimally executed remaining tasks (blue).

behavior: The parallel runtime of a task decreases for increasing numbers of parallel cores until an optimal number of cores is reached and, thus, a higher number of parallel cores (up until the optimal number) should be preferred. However, the decreasing of the parallel runtime is usually restricted by parallelization overheads (e. g., due to communication or synchronization) and, thus, a lower number of parallel cores should be preferred. Therefore, the WATER-LEVEL method increases the number of cores for a task only up until an estimation of the resulting makespan reaches a minimum. This estimation is determined by assigning a task temporarily to a specific number of cores and assuming all remaining tasks can be executed optimally in parallel on the compute resources. Support for heterogeneous compute resources is achieved by taking the performance factors $f_j, j = 1, \ldots, n_N$, of the compute nodes into account for the estimation.

Figure 3 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 remaining tasks (blue) are assumed to be executed optimally in parallel on all cores (i. e., distributed like "water" over the "task landscape"). In this case, the current task would 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 for tasks that can be executed only on single compute nodes is shown in Figure 4. The method starts by determining the total work W required for executing all tasks sequentially (line 4). Scheduling the tasks proceeds similar to the task and data parallel methods of the previous subsection: The tasks are sorted in descending order based on their sequential runtimes to favor an early execution of long running tasks (line 5) and a loop iterates over all tasks in the sorted order (line 7).

```
1  input  : tasks $T_i$, $i = 1,\dots,n_T$, with runtimes $t_i(p)$
2  input  : nodes $N_j$, $j = 1,\dots,n_N$, with $p_j$ cores
3  output : compute resource and start time for each task
4  seq. work $W = \sum_{i=1}^{n_T} t_i(1)$
5  sort $T_i$, $i = 1,\dots,n_T$ in descending order of $t_i(1)$
6  // assume $T_1,\dots,T_{n_T}$ are sorted
7  for $i = 1,\dots,n_T$ do
8      $W = W - t_i(1)$
9      minimal makespan $m^* = \infty$
10     for $j = 1,\dots,n_N$ do
11         $p = 0$
12         repeat
13             $p = p + 1$
14             select $p$ cores of $N_j$ as resource $R$ with start time $s$
15             estimate makespan $m$ with optimally parallelized
               seq. work $W$ and task $T_i$ assigned to resource $R$
16             if $m < m^*$ then
17                 $m^* = m$ ; $R^* = R$ ; $s^* = s$
18         until $p \geq p_j$ or $t_i(p)$ is minimal
19     use resource $R^*$ and start time $s^*$ for task $T_i$
```

Figure 4: Pseudocode of the WATER-LEVEL *method for tasks that can be executed only on single compute nodes.*

In each iteration for the current task $T_i, i \in \{1,\dots,n_T\}$, the sequential work W of all remaining tasks is calculated (line 8). Then, two loops iterate over the compute nodes (line 10) and their number of cores (line 12) to determine the compute resources for the task T_i that lead to a minimal makespan. The inner loop stops earlier if the parallel runtime $t_i(p)$ reaches a minimum for the current number of cores p. It depends on the given runtime function whether and how this minimum can be determined.

For tasks that can be executed only on a single compute node, a compute resource R consists of a specific node and the number of cores to be used on that node. The final schedule also requires the corresponding start time s on the compute resource (line 14). The selected compute resource R is temporarily used for task T_i and the resulting makespan m with optimally parallelized work W of the remaining tasks is estimated (line 15). If this estimated makespan m is smaller than the current minimal makespan m^*, then the corresponding compute resource R and start time s are stored (line 17) such that they can be later used for the task T_i (line 19).

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

Table 1: List of the compute resources used.

For tasks that can be executed on a cluster of nodes, the pseudocode shown in Fig. 4 has to be modified. The clusters $C_1,\dots,C_{n_C}$ have to be provided as input and loops over the clusters and their cores replace the lines 10 and 12. Additionally, a compute resource R for the current task T_i, $i \in \{1,\dots,n_T\}$ will then contain a subset of the nodes of the current cluster and the numbers of cores to be used on each of these nodes. In general, the distinction between the two kinds of tasks could also be performed on a per task basis in an implementation of the WATER-LEVEL method.

IV. PERFORMANCE RESULTS

The task and data parallel methods as well as the WATER-LEVEL method have been used for the scheduling of different simulation tasks. In the following, we present performance results on a heterogeneous compute cluster.

IV.1 Experimental setup

The heterogeneous compute cluster used consists of 8 compute nodes, each with two multi-core processors. Table 1 lists the nodes and their specific processors. The parallel runtime of the tasks required for the scheduling is determined with separate benchmark measurements on the reference compute node `cs1`. The performance factors of the other compute nodes are derived from the sequential runtimes of a task on those compute nodes. In the following subsections, we show total parallel runtimes for executing a number of tasks according to the determined schedules. Starting the tasks is performed by a Python script running on a separate front-end node of the cluster using SSH connections to the compute nodes to be utilized. Each schedule is executed 5 times and the average result is shown.

IV.2 Benchmark tasks

As synthetic benchmark, we employ “sleep” tasks that perform no computations, but only wait for a specific time

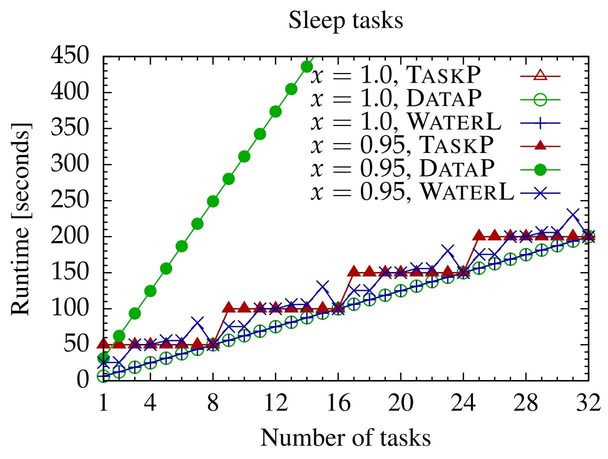

Figure 5: Parallel runtime of different scheduling methods using sleep tasks without ($x = 1.0$) and with parallelization overhead ($x = 0.95$) executed on node `cs1`.

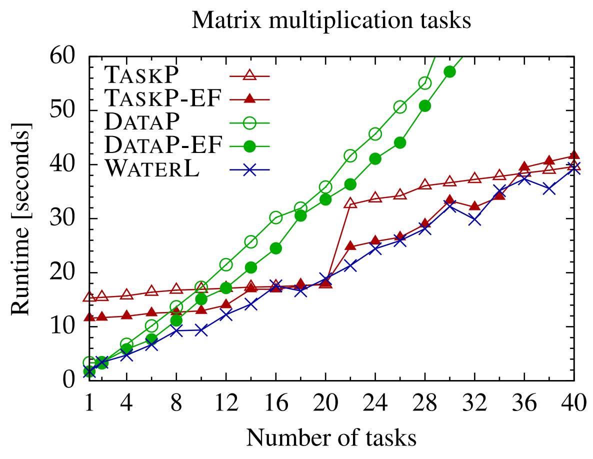

Figure 6: Parallel runtime of different scheduling methods using matrix multiplication tasks executed on nodes `cs1` and `ws1` with a total of 20 cores.

$t(p) = 50s \cdot \left[x \cdot \frac{1}{p} + (1-x) \cdot (\log p + p)\right]$. The runtime comprises of a fraction x which decreases linearly with the number of cores p (e. g., computation) and a remaining part, which increases logarithmically and linearly (e. g., parallelization overhead).

Figure 5 shows parallel runtimes of sleep tasks with $x = 1.0$ (i. e., without parallelization overhead) and $x = 0.95$ (i. e., with parallelization overhead) executed on node `cs1` depending on the number of parallel tasks using different scheduling algorithms. The TaskP method achieves the same results for both kinds of tasks, because the tasks are always executed sequentially. The runtime shows a step-wise increase after every 8 additional tasks, since with these task numbers all 8 cores of the compute node are equally utilized. The DataP method uses always the maximum number of 8 cores for each task and 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. However, with $x = 0.95$, using the maximum number of cores leads to a strong increase of the runtime due to the increasing parallelization overhead. The Water-Level method achieves a trade-off between the TaskP and DataP method. With $x = 1.0$, the optimal result of the DataP method is achieved. With $x = 0.95$, the runtime results are close to the runtime results of the TaskP method, but show a different shape with a more continuous increase instead of the step-wise increase.

The matrix multiplication operation (DGEMM) from the OpenBLAS library is used as parallel benchmark tasks that can be executed on one compute node only. The number of cores utilized is controlled with the environment variable `OPENBLAS_NUM_THREADS`. The matrix size is set to 4000×4000. Figure 6 shows parallel runtimes depending on the number of parallel tasks using different scheduling algorithms and the compute nodes `cs1` and `ws1`. The TaskP methods shows a step-wise increase similar to Fig. 5. Additionally, there is a slight increase between each step since the matrix multiplication tasks on the same compute node influence each other. Selecting the compute nodes according to the earliest finish (TaskP-EF) causes an earlier utilization of the faster node `ws1` and thus, decreases the runtime for specific task numbers. The DataP method shows a strong increase of the runtime due to the parallelization overhead caused by always using the maximum number of cores. The DataP-EF method (i. e., with earliest finish) selects the faster compute node `ws1` more often, thus leading to a smaller runtime. The results of the Water-Level method demonstrate the trade-off between the task and data parallel schemes. With small numbers of tasks, the runtime of the Water-Level method is equal or below the best data parallel scheme and for higher numbers of tasks, the runtime of the Water-Level method is similar to the best task parallel scheme.

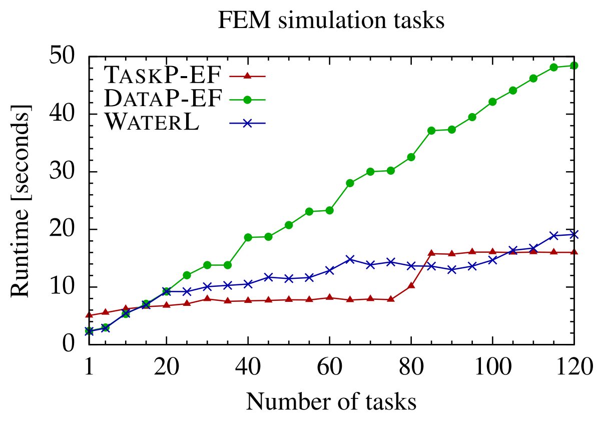

Figure 7: Parallel runtime of different scheduling methods using FEM simulation tasks executed on all nodes listed in Table 1 with a total of 92 cores.

IV.3 FEM simulation tasks

An OpenMP parallel FEM code [3] is used as simulation tasks in the optimization process for lightweight structures as described in Sect. II. Figure 7 shows parallel runtimes depending on the number of parallel tasks using different scheduling algorithms and all compute nodes listed in Table 1. The data parallel scheme with the DATAP-EF method leads to strongly increasing runtimes, which is caused by the low speedup achieved with the parallel FEM code. Thus, using a data parallel execution with high numbers of cores is only advantageous if there are few FEM simulation tasks (i. e., about twice the number of compute nodes). The task parallel scheme with the TASKP-EF method often leads to the smallest runtimes, but shows a steep increase if the number of tasks approaches the total number of cores. The WATER-LEVEL methods leads to the same results as the DATAP-EF method for small numbers of tasks. However, for higher numbers of tasks, the WATER-LEVEL methods is up to a factor of two slower than the TASKP-EF methods. This behavior is caused by the less efficient parallel execution of the FEM code that favors an execution with small numbers of cores. In contrast to that, the WATER-LEVEL method assumes an optimal parallel execution of the unscheduled tasks for the estimation of the makespan. Since this estimation differs strongly from the actual parallel runtime of the tasks, the schedule of the WATER-LEVEL method differs significantly from the faster task parallel schedule.

V. RELATED WORK

Scheduling is a popular problem in computer science involving different application areas and approaches [9]. One of those areas is the scheduling of sequential or parallel tasks to be executed on a given set of hardware resources (e. g., processors) 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 [8]. The layer-based scheduling algorithm from [5] decomposes a set of tasks with dependencies into layers of independent tasks and schedules each layer separately with a so-called list scheduling algorithm. Since the simulation tasks in our optimization process are independent, we can omit a decomposition into layers.

List scheduling algorithms add priorities to the single tasks and assign the tasks in descending order of their priority to the processors. Algorithms, such as *Largest Processing Time* (LPT) [2] and *Longest Task First* (LTF) [14], use the given runtime of the tasks as priorities, thus scheduling compute intensive tasks first. Algorithms for heterogeneous architectures, such as *Heterogeneous Earliest Finish Time* (HEFT) [13] and *Predict Earliest Finish Time* (PEFT) [1], also take the runtime of the tasks on individual processors into account for the priorities. The proposed WATER-LEVEL method 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 and uses the individual processor speeds of a heterogeneous architecture for the allocation of cores by parallel tasks and for the selection of compute nodes.

Scheduling parallel tasks with dependencies can also be performed with a two-step approach consisting of an allocation step and 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). Examples for such algorithms are *Critical Path Reduction* (CPR) [10] and *Critical Path and Allocation* (CPA) [11]. The WATER-LEVEL method performs the allocation of cores only once for each task during the list scheduling and, thus, omits repeated allocation and scheduling steps.

VI. Conclusion

In this article, we have proposed the Water-Level scheduling method for parallel tasks without dependencies on heterogeneous HPC platforms. The method performs an iterative assignment of tasks to compute resources and uses a best-case estimation of the makespan to determine the number of cores to be used for each task. Performance results for benchmark tasks demonstrate a good trade-off between task and data parallel execution schemes. However, for simulation tasks with low parallel efficiency, the best-case estimation may differ strongly from the actual runtimes achieved. This disadvantage might be solved by using the given parallel runtimes of the tasks for a better estimation of the makespan.

Acknowledgment

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] H. Arabnejad and J.G. Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *Transactions on Parallel and Distributed Systems*, 25(3):682–694, 2014.

[2] K.P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *Proc. of the 1990 Int. Conf. on Parallel Processing, (ICPP'90)*, pages 72–75, 1990.

[3] S. Beuchler, A. Meyer, and M. Pester. SPC-PM3AdH v1.0 - Programmer's manual. *Preprint SFB/393 01-08, TU-Chemnitz*, 2001.

[4] L. A. Bongo, R. Ciegis, N. Frasheri, J. Gong, D. Kimovski, P. Kropf, S. Margenov, M. Mihajlovic, M. Neytcheva, T. Rauber, G. Rünger, R. Trobec, R. Wuyts, and R. Wyrzykowski. Applications for ultrascale computing. *Supercomputing Frontiers and Innovations*, 2(1):19–48, 2015.

[5] J. Dümmler, T. Rauber, and G. Rünger. Programming support and scheduling for communicating parallel tasks. *J. of Parallel and Distributed Computing*, 73(2):220–234, 2013.

[6] M. Hofmann, F. Ospald, H. Schmidt, and R. Springer. 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)*, pages 506–511. SciTePress, 2014.

[7] M. Hofmann and G. Rünger. 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*, 2015. (to appear).

[8] J.T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[9] M.L. Pinedo. *Scheduling: Theory, algorithms, and systems*. Springer.

[10] A. Radulescu, C. Nicolescu, A.J.C. van Gemund, and P.P. Jonker. CPR: Mixed task and data parallel scheduling for distributed systems. In *Proc. of the 15th Int. Parallel and Distributed Processing Symposium (IPDPS'01)*, pages 1–8. IEEE, 2001.

[11] A. Radulescu and A.J.C. van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *Proc. of the Int. Conf. on Parallel Processing (ICPP'01)*, pages 69–76. IEEE, 2001.

[12] M. Strano. A technique for FEM optimization under reliability constraint of process variables in sheet metal forming. *Int. J. of Material Forming*, 1(1):13–20, 2008.

[13] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Proc. of the 8th Heterogeneous Computing Workshop (HCW'99)*, pages 3–14. IEEE, 1999.

[14] J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332. ACM, 1992.