Original published: R. Dietze, M. Hofmann, and G. Rünger. Resource contention aware execution of multiprocessor tasks on heterogeneous platforms. In *Proceedings of the 15th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (Heteropar'2017)*, pages 390–402. Springer, August 2017. Online available at `http://dx.doi.org/10.1007/978-3-319-75178-8_32`.

Resource Contention Aware Execution of Multiprocessor Tasks on Heterogeneous Platforms

Robert Dietze✉, Michael Hofmann, and Gudula Rünger

Department of Computer Science, Chemnitz University of Technology, Germany,
{dirob,mhofma,ruenger}@cs.tu-chemnitz.de

Abstract. In high performance computing (HPC), the tasks of complex applications have to be assigned to the compute nodes of heterogeneous HPC platforms in such a way that the total execution time is minimized. Common approaches, such as task scheduling methods, usually base their decisions on task runtimes that are predicted by cost models. A high accuracy and reliability of these models is crucial for achieving low execution times for all tasks. The individual runtimes of concurrently executed tasks are often affected by contention for hardware resources, such as communication networks, the main memory, or hard disks. However, existing cost models usually ignore the effects of resource contention, thus leading to large deviations between predicted and measured runtimes. In this article, we present a resource contention aware cost model for the execution of multiprocessor tasks on heterogeneous platforms. The integration of the proposed model into two task scheduling methods is described. The cost model is validated in isolation as well as within the utilized scheduling methods. Performance results with different benchmark tasks and with tasks of a complex simulation application are shown to demonstrate the performance improvements achieved by taking the effects of resource contention into account.

Keywords: Resource contention, Multiprocessor tasks, Heterogeneous platforms, Scheduling methods, Distributed simulations

1 Introduction

Reducing the overall execution time of compute-intensive applications is a major concern in high performance computing (HPC). The efficient utilization of the available HPC resources represents a key aspect for achieving such reductions of execution times. Complex applications in the area of scientific and engineering simulations usually consist of separated tasks that can be distributed among the compute nodes of a HPC platform. Thus, the goal is to find a distribution that minimizes the execution time of the whole application. This problem is usually solved by applying task scheduling methods. Sequential tasks are assigned to exactly one processor of a compute node. Multiprocessor tasks can be executed in parallel itself to reduce their individual execution times by using more than one processor. Thus, for distributing multiprocessor tasks on HPC platforms, not

only the particular compute node but also the number of processors to be used on this node has to be determined for each task. The resulting distribution problem becomes increasingly complex, thus requiring dedicated scheduling methods.

Task scheduling methods usually base their decisions for distributing the tasks on predictions of the execution times of the tasks. These predictions can be determined with cost models that model the specific execution times of the tasks on the hardware platform to be utilized. For multiprocessor tasks, the cost model also has to include the number of processors employed. Thus, cost models from parallel computing, such as PRAM [9], BSP [11], or LogP [4], might be used. However, since these models abstract from many details of the compute systems, there can be large differences between modeled and measured execution times. These differences lead to improper decisions for scheduling the single tasks and, thus, might deteriorate the overall execution time of all tasks. Keeping these difference as small as possible is therefore an important goal for achieving an efficient execution of multiprocessor tasks on HPC platforms.

Heterogeneous platforms consist of a variety of compute nodes with different computational properties. Existing cost models for heterogeneous platforms take these properties into account, for example, by including different computational speeds of compute nodes. However, the influence of tasks on each other when being executed concurrently on the same node is currently not included in these models. For example, tasks that are executed concurrently on different processors of a compute node can utilize the same hardware resources (e. g., communication network, main memory, or hard disk). The access to these hardware resources has to be shared and might increase execution times due to *resource contention*.

In this article, we present a resource contention aware cost model for the execution of multiprocessor tasks on heterogeneous platforms. The proposed model considers the effects of resource contention, especially due to hard disk and main memory accesses. The integration of the cost model into two scheduling methods for multiprocessor tasks is described. Experiments with different types of tasks on a heterogeneous compute cluster are performed. This includes benchmark tasks with intensive hard disk and main memory accesses. Simulation tasks of a complex application for optimizing lightweight structures are used to represent tasks with accesses to various hardware resources.

The rest of the article is organized as follows: Section 2 discusses related work. Section 3 defines a scheduling problem for multiprocessor tasks and describes the modeling of the task execution times. Section 4 presents a resource contention aware cost model for multiprocessor tasks on heterogeneous platforms. Section 5 describes the integration of the cost model into different task scheduling methods. Section 6 presents experimental results and Section 7 concludes the article.

2 Related Work

Resource contention is mainly considered in the area of thread scheduling for operating systems [14]. Contention for accessing the main memory is integrated into the scheduling, for example, based on memory request rates [12] or cache

miss rates [8]. The measured rates are used to prioritize or group applications to achieve a balanced utilization of memory resources. The measurement approach might also be used to estimate the effects of resource contention for tasks. However, the scheduling approach is not suitable if all tasks exhibit the same memory behavior, such as simulation tasks that execute the same application program.

For task scheduling, contention for communication resources is usually considered. For example, in [10], a model for communication contention is proposed that improves the accuracy of predicted execution times. The integration into scheduling methods is based on task duplication to avoid interprocessor communication and, thus, cannot directly be applied to contention of other hardware resources. Only few works consider contention for other resources. In [2], a contention aware scheduling algorithm for heterogeneous platforms is proposed, but in the context of achieving fault-tolerance by replicating tasks. In [13], the system resources required by tasks are modeled in order to constrain the number of tasks running concurrently. A reduction of the execution time was achieved for tasks that perform memory or file accesses. However, the approach requires that the system resources required by a task are specified manually with user annotations within the program code.

3 Multiprocessor Tasks and Heterogeneous Platforms

The efficient execution of multiprocessor tasks on heterogeneous platforms can be described as a scheduling problem. In the following, the scheduling of multiprocessor tasks and the modeling of the task runtimes is described.

3.1 Scheduling of Multiprocessor Tasks

The considered problem comprises n_T multiprocessor tasks $T_1,\ldots,T_{n_T}$. The term *multiprocessor task* describes a task that can be executed on an arbitrary number of processor cores. It is assumed that all tasks are independent from each other and that the number of utilized cores is fixed during the task execution. The execution of each multiprocessor task is non-preemptive, i. e. it can not be interrupted. For each task T_i, $i = 1,\ldots,n_T$, $t_{i,j}(p)$ denotes its parallel execution time on p cores of a compute node N_j, $j \in \{1,\ldots,n_N\}$. The modeling of the parallel execution time $t_{i,j}(p)$ is described in the following subsection.

The considered heterogeneous HPC platform consists of n_N compute nodes $N_1,\ldots,N_{n_N}$, each having a different computational speed. For each node N_j, $j \in \{1,\ldots,n_N\}$, its number of processor cores p_j and a performance factor f_j are given. The performance factor f_j describes the computational speed of the compute node N_j and is defined as the ratio between the sequential execution time of a task on a reference node N_r and the compute node N_j. Since the reference node is also used for the runtime modeling of the multiprocessor tasks, the compute node with the highest number of cores is used as reference node. It is assumed that each multiprocessor task can only be executed on a single node (e. g., OpenMP-based codes) and that each core can execute only one task

at a time. Thus, each multiprocessor task might be executed on 1 to p_j cores of a node N_j, $j \in \{1, \ldots, n_N\}$. Depending on the number of utilized cores of a compute node, several tasks can be executed on a node at the same time.

The result of the scheduling is a schedule, which defines an assignment of the tasks T_i, $i = 1, \ldots, n_T$, to the compute nodes N_j, $j = 1, \ldots, n_N$. A schedule S includes for each task T_i, $i \in \{1, \ldots, n_T\}$, the information about the compute node and the number of cores to be utilized as well as the estimated start time s_i and finish time e_i. The total execution time $T_{max}(S)$ of a schedule S is defined as the difference between the earliest start time and latest finish time of all tasks. By assuming that the task execution starts at time 0, the total execution time corresponds to the latest finish time of all tasks, i. e. $T_{max}(S) = \max\limits_{i=1,\ldots,n_T} e_i$. The goal is to determine a schedule S such that $T_{max}(S)$ is as small as possible.

3.2 Runtime Modeling of Multiprocessor Tasks

Scheduling methods usually base their decisions on predictions of the execution times of the single task. A high accuracy and reliability of these predictions is required for achieving schedules with a lower total execution time. These predictions can, for example, be calculated regarding to a specific cost model or determined by benchmark measurements. Existing cost models for parallel programming, such as PRAM [9], BSP [11], or LogP [4], are not suitable for the considered scheduling of multiprocessor tasks. The PRAM model, for example, assumes a single shared memory with uniform access by each processor and, thus, heterogeneous platforms with distributed memory are not covered. Furthermore, all of the models calculate the cost of a parallel program based on its program structure and, thus, can not be used if this structure is unknown. In [1], a cost model is presented that uses the amount of work of each task in combination with the relative speed of each compute node. Since this model is designed for the execution of sequential tasks on multiprocessor architectures supported by accelerators, it is not suitable for the scheduling problem described above.

Since the program structures of the considered multiprocessor tasks are unknown, we use the following general runtime formula to model the execution time $t_{i,j}$ of each task T_i, $i \in \{1, \ldots, n_T\}$, on a compute node N_j, $j \in \{1, \ldots, n_N\}$ depending on the employed number of processor cores p:

$$t_{i,j}(p) = f_j \cdot (a_i/p + b_i + c_i \cdot \log p) \tag{1}$$

The parameter f_j denotes the performance factor of node N_j to account for the different computational speeds. The remaining part of Eq. (1) models the execution time of task T_i on the reference node. This part consists of a parallel computation time a_i that decreases linearly with the number of cores p, a constant sequential computation time b_i, and a parallelization overhead c_i that increases logarithmically with the number of cores p (e. g., for synchronization or communication). These components were chosen such that the runtime behavior of common parallel algorithms is covered. The parameters a_i, b_i, and c_i of a task T_i are determined through a least squares fit of the execution times

measured on the reference node with different numbers of cores. In practice, these measurements have to be performed only for tasks with differing execution times.

4 A Resource Contention Aware Cost Model

Shared access to hardware resources can lead to increased execution times of tasks executed concurrently. In the following, a new resource contention aware cost model for predicting the execution time of such tasks is developed.

4.1 Measuring the Effects of Resource Contention

Since resource contention results from shared access to hardware resources, the specific effects on the execution time may depend on the type and number of tasks executed as well as on the hardware utilized. To investigate these effects, we consider three types of tasks. The specific data sizes of the tasks where chosen, such that effects due to data caching are avoided.

I/O bound: The *hdWrite* tasks are used to investigate resource contention due to concurrent hard disk accesses. Each task consists of writing data of size 300 MB to a file on the local hard disk. The parallel implementation as a multiprocessor task is based on MPI where each MPI process writes an equally sized part of the entire file using the function `MPI_File_write`.

Memory bound: The *memWrite* tasks are used to investigate resource contention for the memory bandwidth due to concurrent main memory accesses. Each task consists of writing random integers of size 12 GB to the main memory. The parallel implementation as a multiprocessor task is based on MPI, where each MPI process writes an equally sized share of the entire data.

Compute bound: Numerical simulations based on a *Finite Element Method (FEM)* code [3] are used as compute-intensive tasks. During the numerical optimization of lightweight structures, a large number of structure simulations for varied sets of manufacturing parameters have to be performed [5]. Each simulation applies a preconditioned conjugate gradient method on very large but sparse matrices. The FEM code is parallelized with OpenMP, thus leading to multiprocessor tasks that can be executed in parallel on a single compute node.

Figure 1 shows the sequential execution times for the different types of tasks depending on the number concurrently executed tasks on the same compute node. Each measurement is performed 5 times using the compute node `ws1` with a total number of 12 cores (see Sect. 6.1). The results show that for each type of tasks, the execution times increase almost linearly with an increasing number of concurrently executed tasks. However, the slopes of the curves are different for each type of task. Further measurements have shown that the slope also differs for the same type of task between different compute nodes. These observations imply that the effects of resource contention depend on the type of the tasks, the number of concurrently executed tasks, and the compute node.

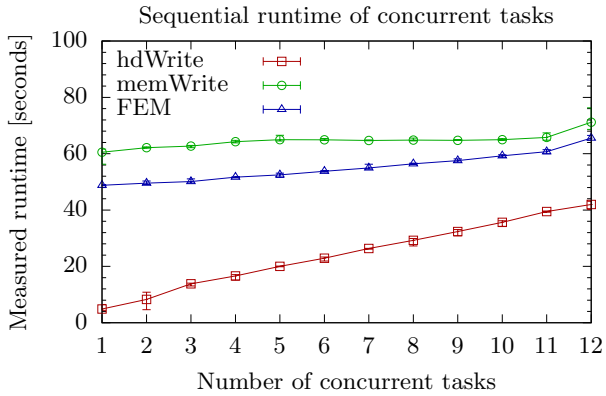

Fig. 1. Measured sequential runtime of different types of tasks on compute node `ws1`.

4.2 Runtime Modeling with Resource Contention

The effects of resource contention on the execution time of tasks is modeled separately for each type of task. For a fixed type of task, we introduce a *contention factor* c_j for each compute node N_j, $j \in \{1, \ldots, n_N\}$. This factor represents the linear slope of the sequential execution times that occurs for executing the tasks concurrently on the compute node N_j. The contention factors are determined by benchmark measurements as described in the previous subsection. Thus, the contention factor captures the entire effects of resource contention due to various hardware resources that may be utilized by a specific type of tasks.

To predict the impact of resource contention on the runtime of a task, the number of concurrently executed tasks on the same compute node has to be known. A task T_k, $k \in \{1, \ldots, n_T\}$ is executed concurrently to a task T_i, $i \in \{1, \ldots, n_T\}$, $i \neq k$, if the start time s_k of task T_k is smaller than the finish time e_i of task T_i and the finish time e_k of task T_k is larger than the start time s_i of task T_i. The period of time during which the two tasks T_k and T_i are executed concurrently lasts from their latest start time to their earliest finish time, i. e. $\min(e_k, e_i) - \max(s_k, s_i)$. During this time, the two tasks content for resources.

Let $K_{i,j}$ denote the set of tasks that are executed concurrently to a task T_i, $i \in \{1, \ldots, n_T\}$, on the compute node N_j, $j \in \{1, \ldots, n_N\}$. To include the effects of resource contention into the prediction of the execution time of the task T_i, its predicted runtime $t_{i,j}(p)$ (see Sect. 3.2) is increased by the additional time during which the task T_i is executed concurrently with the tasks $T_k \in K_{i,j}$. The specific time increase is calculated with the contention factor c_j for the type of tasks on compute node N_j. Thus, the contention aware execution time $\hat{t}_{i,j}(p)$ of task T_i executed on compute node N_j with p cores is modeled as follows:

$$\hat{t}_{i,j}(p) = t_{i,j}(p) + c_j \cdot \sum_{T_k \in K_{i,j}} (\min(e_k, e_i) - \max(s_k, s_i)) \tag{2}$$

Table 1. Difference between measured and predicted execution times without and with resource contention depending on the type and number of tasks on compute node `ws1`.

Type of tasks		hdWrite			memWrite			FEM		
Number of tasks		10	50	100	10	50	100	10	50	100
Difference [percent]	without contention	44.25	25.1	32.01	4.32	2.51	3.02	2.5	2.95	4.35
	with contention	4.92	5.8	2.6	3.2	1.67	2.2	0.07	0.86	2.13

4.3 Validation of the Runtime Modeling

In order to validate the accuracy of the proposed runtime modeling, several benchmark measurements have been performed for each of the three considered types of tasks, i. e. hdWrite, memWrite, and FEM. For each measurement a specific number of tasks (i. e., 10, 50, or 100) is executed on the compute node `ws1`. The number of cores p utilized by each multiprocessor task is chosen between 1 and 6 and each task is started as soon as the chosen number of cores was available. The total execution time of all tasks is measured and the difference to the prediction without resource contention according to Eq. (1) and with resource contention according to Eq. (2) is determined.

Table 1 shows the differences between measured and predicted execution times depending on the type and the number of tasks. For all types and numbers of tasks, the contention aware cost model leads to smaller differences in comparison to the cost model that neglects the effects resource contention. More exactly, the difference between measured and predicted execution times is always smaller than 6% with the contention aware cost model. The biggest improvement is achieved for the hdWrite tasks, where the difference without resource contention is up to about 45%. This corresponds to the previous results shown in Fig. 1, where a significant increase of the runtime was observed. However, even for the memWrite and FEM tasks, the contention aware cost model leads to better predictions of the execution times.

5 Resource Contention Aware Scheduling Methods

The contention aware cost model presented in the previous section has been integrated into two task scheduling methods. In the following, the two task scheduling methods and the necessary adaptions for the integration are described.

5.1 Task parallel execution

The task parallel scheduling scheme (TaskP) presented in [7] is a list scheduling algorithm that assigns each task to exactly one core (i. e., executed sequentially). All tasks are sorted in descending order based on their sequential runtimes. The algorithm iterates over the ordered tasks and selects one core to be utilized. The current task is then assigned to the core that provides the earliest finish.

```
1  total execution time limit m̂ = Σ_{i=1}^{n_T} t_{i,r}(1) / Σ_{j=1}^{n_N} p_j f_j for reference node r
2  repeat
3      potential limits L = ∅
4      clear all assignments of tasks to nodes
5      for task T_i, i = 1, ..., n_T, in descending order of t_{i,r}(1) do
6          for node N_j, j ∈ {1, ..., n_N} and cores p = 1, ..., p_j do
7              select start time s_i such that p cores of node N_j are free
8              calculate finish time e_i = s_i + t_{i,j}(p)
9              add finish time e_i to the set of potential limits L
10             if e_i ≤ m̂ then assign T_i to p cores of N_j and quit the for-loop
11         if task T_i was not assigned then set m̂ to the smallest finish time
             calculated for task T_i and quit the for-loop if restart(i) returns true
12 until all task are assigned
13 repeat
14     total execution time limit m̂ = median of all potential limits L
15     clear all assignments of tasks to nodes
16     for task T_i, i = 1, ..., n_T, in descending order of t_{i,r}(1) do
17         for node N_j, j ∈ {1, ..., n_N} and cores p = 1, ..., p_j do
18             select start time s_i such that p cores of node N_j are free
19             calculate finish time e_i = s_i + t_{i,j}(p)
20             if e_i ≤ m̂ then assign T_i to p cores of N_j and quit the for-loop
21     if all task are assigned then remove all values greater than m̂ from L
22     else remove all values less than m̂ from L
23 until |L| ≤ 1
```

Fig. 2. Pseudocode of the Water-Level-Search method.

5.2 Water-Level-Search method

In [6], a heuristic method for scheduling parallel tasks onto heterogeneous compute resources called Water-Level-Search (WLS) is proposed. Figure 2 shows the pseudocode of this method. The method uses a limit $\hat{m}$ for the predicted total execution time that must not be exceeded by the finish time of any task. An initial guess for this limit is based on the sequential runtimes of the tasks and the total compute capacity of all nodes (line 1). Afterwards, the WLS method performs a search for a better smaller limit that still allows to finish all tasks.

The search for a better limit consists of two phases (lines 2–12 and lines 13–23). In each phase, the current limit $\hat{m}$ is used to determine an assignment of tasks to nodes and cores (lines 5–11 and lines 16–20). The assignment is determined by iterating over the tasks in descending order based on their sequential runtimes and for each tasks T_i, $i \in \{1, \ldots, n_T\}$, all compute nodes N_j, $j = 1, \ldots, n_N$, and their numbers of cores p_j are tested. This test consists of selecting a possible start time s_i and calculating the corresponding finish time e_i with the runtime formula $t_{i,j}(p)$ (lines 7–8 and lines 18–19). If the finish time e_i is valid for the current limit $\hat{m}$, then the task is assigned to the selected node

and cores and then the next task is tested. Otherwise, the limit $\hat{m}$ is adapted and the assignment of tasks is restarted for all tasks. In the first phase, the limit is only increased and a set of potential limits L is created. In the second phase, a binary search among the potential limits in L is performed by repeatedly using the median of L as the current limit $\hat{m}$ (line 14) and adapting L accordingly (lines 21–22). The last value of L is the smallest limit $\hat{m}$ that was found and the corresponding assignment of tasks to nodes and cores is the determined schedule.

Determining an assignment in each phase depends linearly on the number of tasks n_T, the number of nodes n_N, and the highest number of cores of a node p_r. Restarting the assignment in the first phase is limited to at most $\log n_T$ times with a the *restart* function (line 11). The size of L depends linear on the number of tasks n_T, such that the binary search in the second phase requires $\mathcal{O}(\log n_T)$ steps. Thus, the overall complexity of the method is $\mathcal{O}(\log n_T \cdot n_T \cdot n_N \cdot p_r)$.

5.3 Integration of the Contention Aware Cost Model

Both methods use the runtime formula of $t_{i,j}(p)$ in Eq. (1) to predict the execution time of a task T_i, $i \in \{1, \ldots, n_T\}$, executed on compute node N_j, $j \in \{1, \ldots, n_N\}$ with p cores. To integrate the contention aware cost model described in Sect. 4, each occurrence of this usage is replaced by the new formula of $\hat{t}_{i,j}(p)$ in Eq. (2). Additionally, both methods use a list scheduling approach where tasks are assigned gradually to the compute resources. Thus, the number of tasks that are executed concurrently to a specific task that was already scheduled can increase during the scheduling. Since the contention aware cost model depends on this number, the start and finish time of an already scheduled task is recalculated whenever another task is assigned to the same compute node with an overlapping period of time.

For the WLS method, changing the finish times of tasks afterwards may lead to problems. For example, if the finish time of a task increases due to resource contention, then it might exceed the limit $\hat{m}$ that was used when the assignment of this task was determined. However, such a behavior conflicts with the assumption that a valid limit $\hat{m}$ allows to finish the execution of all tasks. To avoid such situations, the prediction of the finish time of a specific task uses always the maximum number of tasks that might be executed concurrently based on the number of currently available cores of the compute node.

6 Experimental Results

The proposed resource contention aware cost model has been integrated into the scheduling methods described in Sect. 5. The following experimental results compare the methods without and with the resource contention aware cost model.

6.1 Experimental Setup

The compute nodes of the heterogeneous compute cluster used for the measurements are listed in Table 2. The scheduling methods described in Sect. 5 have

Table 2. List of nodes of the utilized heterogeneous compute cluster.

Nodes	Processors	#Nodes×#processors×#cores	total RAM	GHz
`sb1`	Intel Xeon E5-2650	$1 \times 2 \times 8$	60 GB	2.00
`ws1`,...,`ws5`	Intel Xeon X5650	$5 \times 2 \times 6$	32 GB	2.66
`cs1`,`cs2`	Intel Xeon E5345	$2 \times 2 \times 4$	16 GB	2.33

been implemented in Python. A Python script running on a separate node performs the execution of the tasks on the compute nodes via SSH connections. Each measurement is performed 5 times and the average values are shown.

6.2 Performance Results with Benchmark and Simulation Tasks

The task parallel scheduling (TaskP) and the Water-Level-Search method (WLS) without resource contention and with resource contention (i. e., TaskP-RC and WLS-RC) have been used to schedule the execution of different types of tasks. Figure 3 (top) shows the measured total runtimes for executing the hdWrite tasks (left) and the memWrite tasks (right) according to the determined schedules depending on the number of tasks. Up to 24 tasks of the corresponding type are executed on the compute nodes `ws1` and `ws2` with a total number of 24 cores. Using the contention aware scheduling methods leads to a significant reduction of the total runtimes with the hdWrite tasks. The biggest differences up to about 60% of the total runtime are achieved for the task parallel scheduling method (i. e., TaskP and TaskP-RC). With the contention aware cost model, both scheduling methods (i. e., TaskP-RC and WLS-RC) lead to about the same results. This behavior can mainly be attributed to the hardware resources utilized by the hdWrite tasks. The hard disk accesses are usually limited by the corresponding hard disk devices, thus leading to high contention for concurrent accesses and low benefits from parallelization. In comparison, the benefits of the contention aware cost model for the memWrite tasks are smaller. This behavior corresponds to the results shown in Fig. 1, where the effect of resource contention was also smaller for the memWrite tasks. However, there are still improvements of the total runtime for both contention aware scheduling methods.

Figure 3 (bottom left) shows measured total runtimes for executing the FEM simulation tasks according to the determined schedules depending on the number of tasks using all compute nodes of Table 2. The results confirm the improvements achieved with the contention aware cost model. Especially, if the number of tasks approaches the number of utilized cores (i. e., 92), both scheduling methods (i. e., TaskP-RC and WLS-RC) lead to a significant reduction of the total runtimes. In general, the results with the resource contention aware cost model show a more steady and less abrupt increase when the number of tasks is increased. Figure 3 (bottom right) shows the parallel speedups for executing 100 FEM simulation tasks according to the determined schedules depending on the number of utilized cores. Up to about 52 cores, there are only small differences

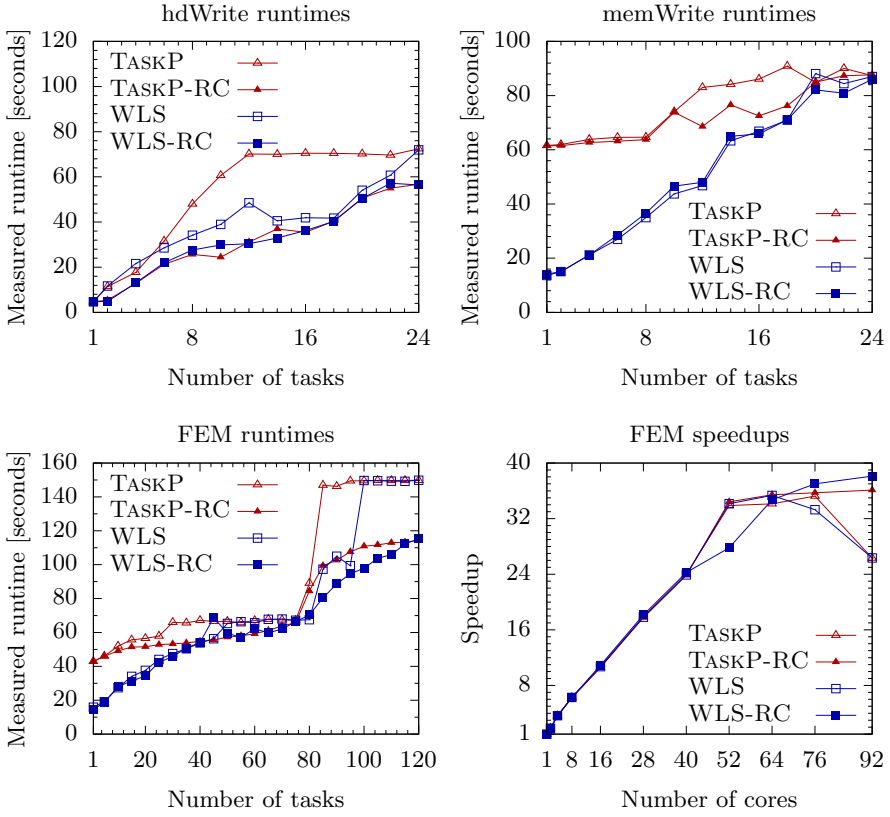

Fig. 3. Top: Measured total runtimes of hdWrite tasks (left) and memWrite tasks (right) depending on the number of tasks using all compute nodes `ws1` and `ws2`. Bottom: Measured total runtimes of FEM simulation tasks depending on the number of tasks using all compute nodes of Table 2 (left) and parallel speedups for the execution of 100 FEM simulation tasks depending on the number of cores (right).

between all scheduling methods. However, when all compute nodes are used, the resource contention aware cost model prevents a decrease of the speedup.

7 Conclusions

In this article, we investigated the effects of resource contention for the execution of multiprocessor tasks on heterogeneous platforms. The development of a contention aware cost model based on a task- and hardware-depending contention factor was described. The proposed cost model was used for the prediction of executions times of multiprocessor tasks and the integration into two existing scheduling methods was described. Measurements with benchmark tasks with hard disk and main memory accesses demonstrated that for both scheduling methods, a reduction of the total task runtimes could be achieved. Further

results with FEM simulation tasks confirmed the performance improvements, especially due to a better utilization of hardware with high contention effects.

Acknowledgments

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).

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exper. 23(2), 187–198 (2011)
2. Benoit, A., Hakem, M., Robert, Y.: Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. Parallel Computing 35(2), 83–108 (2009)
3. Beuchler, S., Meyer, A., Pester, M.: SPC-PM3AdH v1.0 - Programmer's manual. Preprint SFB/393 01-08, TU-Chemnitz (2001)
4. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'93). pp. 1–12. ACM (1993)
5. Dietze, R., Hofmann, M., Rünger, G.: Exploiting Heterogeneous Compute Resources for Optimizing Lightweight Structures. In: Proc. of the 2nd Int. Workshop on Sustainable Ultrascale Computing Systems (NESUS'15). pp. 127–134 (2015)
6. Dietze, R., Hofmann, M., Rünger, G.: Water-level scheduling for parallel tasks in compute-intensive application components. J. of Supercomputing pp. 1–22 (2016)
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. Feliu, J., Petit, S., Sahuquillo, J., Duato, J.: Cache-hierarchy contention-aware scheduling in CMPS. IEEE Trans. Parallel Distrib. Syst. 25(3), 581–590 (2014)
9. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: Proc. of the 10th Annual ACM Symp. on Theory of Computing. pp. 114–118. ACM (1978)
10. Sinnen, O., To, A., Kaur, M.: Contention-aware scheduling with task duplication. J. of Parallel and Distributed Computing 71(1), 77–86 (2011)
11. Skillicorn, D.B., Hill, J., McColl, W.: Questions and answers about bsp. Scientific Programming 6(3), 249–274 (1997)
12. Subramanian, L., Seshadri, V., Kim, Y., Jaiyen, B., Mutlu, O.: MISE: Providing performance predictability and improving fairness in shared main memory systems. In: Proc. of the 19th Int. Symp. on High Performance Computer Architecture (HPCA'13). pp. 639–650. IEEE (2013)
13. Tillenius, M., Larsson, E., Badia, R.M., Martorell, X.: Resource-aware task scheduling. ACM Trans. Embed. Comput. Syst. 14(1), 5:1–5:25 (2015)
14. Zhuravlev, S., Saez, J.C., Blagodurov, S., Fedorova, A., Prieto, M.: Survey of scheduling techniques for addressing shared resources in multicore processors. ACM Comput. Surv. 45(1), 4:1–4:28 (2012)