Special Issue Paper

The search-based scheduling algorithm HP* for parallel tasks on heterogeneous platforms

Robert Dietze | Gudula Rünger

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

Correspondence
Robert Dietze, Department of Computer Science, Chemnitz University of Technology, 09111 Chemnitz, Germany. Email: dirob@cs.tu-chemnitz.de

Funding information
Bundesministerium für Bildung und Forschung, Grant/Award Number: 01H16012A/B; Deutsche Forschungsgemeinschaft, Grant/Award Number: EXC 1075

Summary
Scheduling is a widely used method in parallel computing, which assigns tasks to compute resources of a parallel environment. In this article, we consider independent parallel tasks to be scheduled onto a heterogeneous execution platform consisting of a set of multicores of different architecture. Each parallel task has an internal parallelism which allows a parallel execution on any multicore processors. However, the execution time may differ due to a different computation speed of different multicores.

In this article, we propose a new search-based scheduling algorithm HETEROGENEOUS PARALLEL TASK SCHEDULING BASED ON A* (called HP*) to solve the problem of scheduling independent parallel tasks onto heterogeneous multicore platforms. Specifically, we propose a heuristic cost function needed for an informed search. Also, three pruning techniques are proposed, which are shown to significantly reduce the search space of HP*. Performance measurements on a heterogeneous platform are performed and the results of HP* are compared to scheduling results of other popular scheduling methods. The performance results with benchmark tasks from the SPLASH-3 benchmark suite demonstrate the good scheduling results and the improvements achieved by HP*.

Keywords
heterogeneous platforms, parallel tasks, pruning techniques, search-based scheduling

1 Introduction

The execution time of compute-intensive applications depends strongly on the efficient utilization of compute resources. For task-based applications, an efficient utilization of the resources can be achieved by employing a scheduling algorithm, which assigns the tasks to the compute resources with the goal to get a low execution time. The task structure of task-based applications can be of very different nature. Broadly defined, a task can be a sequential task or a parallel task (also known as multiprocessor tasks, rigid tasks, moldable tasks, or malleable tasks) and the set of tasks can consist of independent or dependent tasks. In contrast to a sequential task, which is executed on exactly one processor or core, a parallel task is a parallel piece of code with an internal parallelism which can be executed on a set of processors or cores. Different task structures lead to different scheduling problems to be solved by different scheduling algorithms. In this article, we tackle the problem to schedule independent parallel tasks onto a heterogeneous multiprocessor platform. Independent tasks can be executed concurrently on the execution units which may lead to a significant reduction of the execution time of the application.

Parallel computing environments within or across institutions are often composed of nodes with different capabilities and speeds leading to the so-called heterogeneous architecture platforms. The type of heterogeneous architecture platform considered in this article consists of a set of...
multicores that have different computing speeds possibly leading to a different performance when executing the same parallel task. Achieving a high efficiency on such a heterogeneous system requires a scheduling method, which takes the heterogeneity into account. For the scheduling of parallel tasks, two properties of the compute nodes are particularly important, which are the number of processor cores and the computational speed of each node.

Many proposed task scheduling methods focus on sequential tasks where each task is assigned to exactly one processor of a compute node. Scheduling algorithms for independent parallel tasks usually have to take into account more possibilities of the mapping of tasks, since a task can be executed on an arbitrary number of cores and a concurrent execution of different tasks is possible. Thus, such a scheduling algorithm becomes increasingly complex due to the increasing number of options for placing tasks. For example, for only two tasks, there is the possibility to place them to all available cores and execute them one after another or to execute them concurrently on subsets of the cores. The solution of the scheduling problem is to determine for each task the particular compute node and the number of processor cores to be used on this node.

Since task scheduling is an NP-complete problem, many of the proposed scheduling methods are based on heuristics, see, for example, References 1-5. Heuristic scheduling methods may find solutions of the scheduling problem that are acceptably good for a specific use case but are usually not optimal. To find an optimal solution, a search of the entire solution space may be required in the worst case. Since the computation time required to find an optimal solution can be extremely long, informed search-based algorithms which prune the search space are advantageous.6 Some recent search-based scheduling algorithms are based on the A* (pronounced ‘A-star’) algorithm. The A* algorithm is an informed search algorithm for graph traversal and path search, which strongly relies on an estimation of the cost for the rest of the search. It has been shown that informed search algorithms, such as the A* search algorithm,7 find an optimal solution if a so-called admissible and consistent cost function is used.8 The cost estimation has to underestimate the actual cost in order to guarantee an optimal solution. A scheduling algorithm for sequential dependent tasks to homogeneous platforms based on the A* search algorithm has been proposed in Reference 9 and several pruning techniques to reduce the search space are presented. A scheduling algorithm for the same scheduling problem with an improved cost function has been proposed in Reference 10 with additional pruning techniques in Reference 11. In contrast to these earlier works, we consider the scheduling of independent parallel tasks to heterogeneous platforms with its much higher possibilities of task mappings described above and propose a novel search-based scheduling algorithm HETEROGENEOUS PARALLEL TASK SCHEDULING BASED ON A* (HP*).

The algorithm HP* is also inspired by the A* algorithm. For the search, we define a scheduling decision tree consisting of incomplete schedules which are completed by the informed search. The crucial point was to find an estimation function that underestimates the actual cost. Since our tasks are parallel tasks, these cost are parallel cost which means that the parallel execution time of each task has to be modeled for a varying number of nodes and varying speed of the cores. We have also proposed pruning techniques that are able to reduce the search space of HP*. The HP* has been implemented and the performance has been measured on a heterogeneous multicore using the parallel programs of the SPLASH-3 benchmark suite as parallel tasks. The results are compared with popular scheduling algorithms for parallel task.

The contribution of this article is:

- A new task scheduling method HP* for solving the problem of scheduling independent parallel tasks onto heterogeneous platforms. The algorithm HP* is inspired by the A* search algorithm.
- A modeling of a heuristic cost function for parallel tasks that guides the search of HP*.
- Three pruning techniques that are able to significantly reduce search space and thus increase the performance of HP*.
- Theoretical as well as practical performance evaluations.
- An evaluation of the effects of the pruning techniques on the size of the search space. This is done for each pruning technique in isolation as well as in combination of all techniques together.
- A performance comparison with three popular heuristic task scheduling techniques.
- Experiments with programs from the SPLASH-3 benchmark suite12 used as parallel tasks on a heterogeneous multicore cluster.

The rest of the article is organized as follows: Section 2 defines a scheduling problem for independent parallel tasks and describes the modeling of the task execution times. Section 3 presents the new search-based scheduling algorithms for parallel tasks HP*. Section 4 proposes techniques for pruning the search space of HP*. Section 5 presents experimental results. Section 6 discusses related work and Section 7 concludes the article.

2 SCHEDULING OF PARALLEL TASKS ON HETEROGENEOUS PLATFORMS

In this section, we present the scheduling problem for independent parallel task to be solved in Section 2.1 and the model of the cost function is given in Section 2.2.
2.1 Scheduling problem

The scheduling problem considered in this article comprises \( n_T \) independent parallel tasks \( T_i, i = 1, \ldots, n_T \). A parallel task can be executed on a single compute node utilizing an arbitrary number of processor cores. The number of cores used by each task is fixed during the task execution. This type of tasks is also known as moldable task. The tasks are nonpreemptive, that is, their execution cannot be interrupted. For each task \( T_i \), its parallel execution time using \( p \) cores of compute node \( N_j, j \in \{1, \ldots, n_N\} \) is denoted by \( t_{ij}(p) \).

The heterogeneous platform considered consists of \( n_N \) multicore compute nodes \( N_1, \ldots, N_{n_N} \). The heterogeneity of the platform results from the different architectures of each compute node. Thus, each compute node \( N_j \) might have a different computational speed and a different number of processor cores \( p_j \). It is also stated that each processor core can execute only one task at a time. Thus, each parallel task might be executed on 1 to \( p_j \) cores of a compute node \( N_j \). However, several tasks can be executed on a compute node at the same time depending on the number of cores utilized on a compute node.

A solution for the scheduling problem described above is an assignment of the tasks \( T_i, i = 1, \ldots, n_T \) to the compute nodes \( N_j, j = 1, \ldots, n_N \). For each task \( T_i \), the resulting schedule \( S \) provides the following information:

- the compute node and the number of cores to be utilized,
- the calculated start time \( s_i \), and finish time \( e_i \).

The makespan \( M(S) \) of a schedule \( S \) is the difference between the earliest start time and latest finish time of all tasks. We assume that the execution of the first task starts at time 0. Thus, the makespan is identical to the latest finish time of all tasks. This can be expressed as
\[
M(S) = \max_{i=1}^{n_T} e_i.
\]
The goal is to determine a schedule \( S \) such that the makespan \( M(S) \) is minimized.

2.2 Cost model for parallel tasks

The decisions made by scheduling methods are usually based on predictions of the execution times of single tasks. These predictions can be completely determined by benchmark measurements or can be calculated using a specific cost model. Since the program structures of the parallel tasks are unknown, existing cost models for parallel programming, such as PRAM\(^1\)\(^3\), BSP\(^1\)\(^4\), or LogP\(^1\)\(^5\), cannot be used for the considered scheduling problem. Thus, we use the following runtime formula to model the execution time \( t_{ij} \) of each task \( T_i, i = 1, \ldots, n_T \) on a compute node \( N_j, j = 1, \ldots, n_N \) depending on the number of utilized processor cores \( p \):

\[
t_{ij}(p) = f_j \cdot \left( a_i/p + b_i + c_i \cdot \log p \right).
\]  

The parameter \( f_j \) denotes the performance factor of compute node \( N_j \) that describes its computational speed in relation to a reference compute node \( N_r \). It 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 \( N_r \) is also used to determine the other parameters of the runtime formula, the compute node with the highest number of cores is used as reference node. The remaining part of Equation (1) represents the execution time of task \( T_i \) on the reference node \( N_r \). The structure of this part was chosen to cover the runtime behavior of typical parallel tasks. It 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). To determine the parameters \( a_i, b_i, \) and \( c_i \) of a task \( T_i \), first, the execution times are measured on the reference node with different numbers of cores. Then the concrete values of the parameters are calculated based on a least squares fit of these execution times. In Reference 16, it is shown that this cost model achieves a good prediction of the task execution time. Table 1 summarizes the notations used to describe the scheduling problem.

3 SEARCH-BASED SCHEDULING ALGORITHM

In this section, we propose a new search-based task scheduling method HP\(^*\) for assigning parallel tasks optimally to heterogeneous platforms which is derived from the search algorithm A\(^*\), which is briefly summarized in Section 3.1. The algorithm of the new scheduling method HP\(^*\) is proposed in Section 3.2. The time complexity of HP\(^*\) is analyzed in Section 3.3. Table 2 lists the notation used to describe the A\(^*\) search algorithm and HP\(^*\).
### Table 1: Notation of the scheduling problem

| Notation | Meaning |
|----------|---------|
| \( n_T \) | Number of parallel tasks |
| \( T_1, \ldots, T_{n_T} \) | Independent shared memory tasks |
| \( n_N \) | Number of compute nodes in the heterogeneous cluster |
| \( N_1, \ldots, N_{n_N} \) | Compute nodes of the cluster |
| \( \rho_j \) | Number of processor cores of compute node \( N_j \), \( j = 1, \ldots, n_N \) |
| \( f_j \) | Performance factor of compute node \( N_j \) |
| \( t_j(p) \) | Parallel execution time of task \( T_i \) on \( p \) cores of node \( N_j \) |
| \( s_i \) | Start time of task \( T_i \) |
| \( e_i \) | Finish time of task \( T_i \) |
| \( M(S) \) | Makespan of schedule \( S \) |

### Table 2: Notation of the A* search algorithm and scheduling algorithm HP*

| Notation | Meaning |
|----------|---------|
| \( s \) | Start node |
| \( G \) | Set of goal nodes |
| \( f(n) \) | Cost of a path from \( s \) to a goal node via node \( n \) |
| \( g(n) \) | Actual cost of a path from \( s \) to a node \( n \) |
| \( h(n) \) | Heuristic cost function to estimate the cost of the shortest path from a node \( n \) to a goal node |
| \( h^*(n) \) | Exact cost of a path from a node \( n \) to a goal node |
| \( S_n \) | Schedule represented by node \( n \) |
| \( d(n,u) \) | Edge weight between the nodes \( n \) and \( u \) |
| \( W_s \) | Remaining sequential workload |
| \( K(S_s) \) | Computational capacity available regarding to a schedule \( S_s \) |
| \( L_{open} \) | List of all nodes that have been created but not visited yet |
| \( L_{visited} \) | List of all visited nodes |
| \( s_{init} \) | Initial node in \( HP^* \) which represents an empty schedule |
| \( s_{cur} \) | Node that is currently visited by \( HP^* \) |

### 3.1 The A* search algorithm

The A* search algorithm\(^7\) is commonly used to find the shortest path in a directed graph with positive edge weights. The goal of the algorithm is to find the shortest path in a graph \( G \) from a start node \( s \) to a nonempty set of goal nodes \( T \). For its search, the algorithm uses a function \( f(n) \) representing the cost of a path from \( s \) to a goal node via node \( n \). The function \( f(n) \) consists of two parts: the actual cost \( g(n) \) of the path from \( s \) to \( n \) and the estimated cost \( h(n) \) of the shortest path from \( n \) to a goal node. In Reference 8, it was shown that using an admissible and consistent function \( f(n) \), the A* search algorithm is guaranteed to find an optimal solution.

**Definition 1.** (Admissible cost function). The cost function \( f(n)=g(n)+h(n) \) said to be admissible, if the heuristic function \( h(n) \) underestimates the exact cost \( h^*(n) \) for each node \( n \), that is, if the following holds:

\[
h(n) \leq h^*(n).
\]

**Definition 2.** (Consistent cost function). The cost function \( f(n)=g(n)+h(n) \) said to be consistent, if \( f(n) \) never decreases along any path. This is the case, if for any pair of adjacent nodes \( x \) and \( y \) with edge weight \( d(x,y) \), the following holds:

\[
h(x) \leq d(x, y) + h(y).
\]
Algorithm 1 shows the pseudocode of the A* search algorithm presented in Reference 7. First, the start node \( s \) is marked "open" and the cost function \( f(s) \) is evaluated. All "open" nodes have been discovered but have not been visited yet. Nodes that have been visited are marked as "closed" to avoid revisiting them. Then, the "open" node \( n \) with the smallest cost \( f(n) \) is selected and marked "closed." Each unmarked successor \( u \) of \( n \) is marked "open" and \( f(u) \) is calculated. Nodes \( u \) that are "closed" are marked "open" if the current cost \( f(u) \) is lower than the cost when they were marked "closed." The algorithm continues selecting the next node \( n \) with the smallest cost \( f(n) \) (line 2) until a goal node is reached.

Algorithm 1 A* search algorithm.

1: Mark \( s \) as 'open' and calculate \( f(s) \)
2: Select the open node \( n \) with the smallest value \( f(n) \)
3: if \( (n \in T) \) then
4: Mark \( n \) as 'closed' and terminate the algorithm
5: else
6: Mark \( n \) as 'closed'
7: for (each successor \( u \) of \( n \)) do
8: Calculate \( f(u) \)
9: if (\( u \) is not marked 'closed') then
10: Mark \( u \) as 'open'
11: else if (the current value of \( f(u) \) is lower as when \( u \) was 'closed') then
12: Mark \( u \) as 'open'
13: end if
14: end for
15: Proceed with line 2
16: end if

3.2 Scheduling parallel tasks with HP*

For the scheduling of parallel tasks onto heterogeneous platforms, we propose a new scheduling method HP* (HETEROGENEOUS PARALLEL TASK SCHEDULING BASED ON A*) based on the A* search algorithm. HP* is a search-based scheduling algorithm for independent parallel task and exploits a graph data structure, which guides this search for a schedule. The approach of the scheduling algorithm is derived from the A* search algorithm in the following sense. The scheduling problem described in Section 2.1 is defined as a directed graph in form of a tree, in the following referred to as scheduling decision tree.

Definition 3. (Scheduling decision tree). A scheduling decision tree is a tree where each graph node \( n \) represents a (partial) schedule \( S_n \). The root node represents an empty schedule, that is, where no tasks have been scheduled. The successors of a graph node represent all schedules in which exactly one more task is scheduled. In a scheduling decision tree, the leaf nodes are complete schedules, that is, where all tasks have been scheduled. The weight \( d(n,u) \) of the edge between two nodes \( n \) and \( u \) is the difference between the makespans of the schedules \( S_n \) and \( S_u \), that is, \( d(n,u) = M(S_u) - M(S_n) \).

According to the A* search algorithm, the cost function \( f(n) = g(n) + h(n) \) used by HP* consists of two parts:

- \( g(n) \) that is the makespan \( M(S_n) \) of the schedule \( S_n \) corresponding to node \( n \),
- \( h(n) \) that is a heuristic for the difference between the makespans \( M(S_n) \) and \( M(S_l) \) for a node \( n \) and a leaf node \( l \).

Thus, the graph structure supports a targeted search for a complete schedule and the result of HP* is the leaf node at the end of a shortest path in this graph. The path from an empty schedule (root node) to the solution represents the construction of the schedule found by HP*. For the calculation of the heuristic cost function \( h(n) \), it is assumed that the remaining tasks can be distributed "optimally" to all cores. This means that the execution of a task is not restricted to a single compute node and the number of processor cores used by a task can change during its execution. It is also assumed that each remaining task can be executed fully parallel, which means that the sequential execution time is divided by the number of processor cores utilized. Without loss of generality, it is assumed that in a node \( n \) the tasks \( T_x, \ldots, T_{n_x} \) for \( x \in \{1, \ldots, n_t\} \) have not been scheduled yet. The remaining sequential workload \( W_i \) is then calculated as

\[
W_i = \sum_{i=n}^{n_t} t_i(1),
\]
using Equation (1) considering the reference node $N_s$. The computational capacity available on all cores of the compute nodes regarding to a schedule $S_n$ is defined as

$$K(S_n) = \sum_{j=1}^{n_s} \sum_{k=1}^{n_c} (M(S_n) - \max_{i \in S_n} e_i).$$

(5)

For a compute node $N_j, j = 1, \ldots, n_N$, the set $C_{i,k}$ denotes all tasks that have been assigned to core $k$ of this node. For each node $n$, the heuristic cost function $h(n)$ is defined as follows:

$$h(n) = \begin{cases} \frac{(W_s - K(S_n))}{\sum_{j=1}^{n_s} (p_j \cdot f_j)}, & \text{if } W_s > K(S_n) \\ 0, & \text{otherwise.} \end{cases}$$

If the remaining workload is bigger than the available computational capacity, then $h(n)$ is set to the difference divided by the total compute power, that is, the sum of $p_j \cdot f_j$ over all nodes $N_j, j = 1, \ldots, n_N$. Otherwise, there is enough computational capacity available for the remaining workload, which leads to $h(n) = 0$. Figure 1 shows an illustration of the calculation of the proposed cost function $f(n) = g(n) + h(n)$ with tasks scheduled already (gray) and the remaining workload (blue). In this example, the remaining workload is either lower (left) or greater (right) than the computational capacity.

In Algorithm 2, the pseudocode of the HP* method is shown. HP* maintains two lists, $L_{\text{open}}$ and $L_{\text{closed}}$. $L_{\text{open}}$ contains all nodes that have been created but not visited yet. The list $L_{\text{closed}}$ is used to avoid that nodes are revisited. At the beginning, both lists are empty and the initial node $S_{\text{init}}$ represents an empty schedule. $f(S_{\text{init}})$ is calculated and the node is added to $L_{\text{open}}$. In each step of the main loop (lines 4-22), the node $S_{\text{cur}}$ with the smallest value $f(S_{\text{cur}})$ is selected and removed from $L_{\text{open}}$. If $S_{\text{cur}}$ represents a complete solution, the solution is found and the algorithm terminates. If $S_{\text{cur}}$ is already part of $L_{\text{closed}}$, $S_{\text{cur}}$ is skipped and the algorithm continues with the next node. Otherwise, $S_{\text{cur}}$ is added to $L_{\text{closed}}$ and a task $T$ is selected that has not been scheduled in $S_{\text{cur}}$ yet. For each possible assignment of each unscheduled task $T$, a new node $s$ is created. This is done by an iteration over all compute nodes $N_j, j = 1, \ldots, n_N$, and all numbers of cores $p$ from 1 to $p_i$. In each step of this iteration, all possible assignments of task $T$ to $p$ cores of node $N_j$ are generated. Each assignment is added to the schedule used in $S_{\text{cur}}$ and the resulting schedule is represented by a new node $s$. Then value $f(s)$ of this new node $s$ is calculated and $s$ is added to $L_{\text{open}}$.

**Lemma 1.** HP* uses a heuristic cost function $h(n)$ (Equation (6)) that is admissible and consistent.

**Proof.** Within the scheduling decision tree, the successor $m$ of a node $n$ represents a schedule $S_m$ where exactly one more task is scheduled than in $S_n$. It is assumed that this task is assigned to $p, 1 \leq p \leq p_i$ processor cores of compute node $N_j, j = 1, \ldots, n_N$. $h(n)$ denotes the increase of the makespan $M(S_n)$ of the schedule $S_n$ that is estimated based on the remaining tasks. The exact cost of this increase is denoted by $h'(n)$. It can be shown that for each node $n$, the following holds:

$$h(n) \leq h'(n) \leq \frac{W_s - K(n)}{\sum_{j=1}^{n_s} (p_j \cdot f_j)} \leq \frac{W_s - K(n)}{p \cdot \sum_{j=1}^{n_s} f_j}.$$  

For the calculation of $h(n)$, Equation (6) is used assuming that the sequential workload $W_s$ is bigger than the computational capacity. Since all costs are nonnegative values, $h(n) = 0$ always underestimates $h'(n)$, that is, $0 \leq h(n) \leq h'(n)$ for each node $n$. For the calculation of the exact cost $h'(n)$, the parallel workload $W_s$ is considered instead of the sequential workload. This means that the execution of task $T_i$ requires $p$ cores with a
performance factor $f_j$ for the execution time $t_{ij}(p)$. According to Equation (1), the execution time $t_{ij}(p)$ on compute node $N_j$ can also be expressed as the execution time $t_{ij}(p)$ on the reference node divided by the performance factor $f_j$. The total compute power denoted by $\sum_{j=1}^{nN} (p_j \cdot f_j)$ is always greater or equal to $p \cdot f_j$. For common parallel tasks, it can be assumed that $t_{ij}(1)$ is always less or equal to $t_{ij}(p)$. As a consequence, $h(n) \leq h^*(n)$ holds for each node $n$ and thus, $h(n)$ is an admissible heuristic. According to the definition, both, $g(n)$ and $h(n)$ are nonnegative for each node $n$. Together with the admissibility of $h(n)$, this implies that $f(n)$ is a nondecreasing function along any path. Consequently, $f(n)$ is a consistent function.

**Lemma 2.** Using the heuristic cost function $h(n)$ (Equation (6)), HP* finds an optimal solution.

**Proof.** In Reference 8, it is shown that the $A^*$ search algorithm is guaranteed to find a shortest path if an admissible and consistent function $f(n) = g(n) + h(n)$ is used. Since the cost function used by HP* is admissible and consistent, HP* is guaranteed to find a shortest path in the scheduling decision tree. Per definition of the scheduling decision tree, the leaf node at the end of this path is also a schedule with a minimum makespan.

In each step, HP* selects the node with the smallest cost from all nodes discovered at this point. The first complete node that is visited by HP* is returned as the solution of the search. Since always the node with the smallest cost is visited and $f(n)$ is consistent, no other node in the scheduling decision tree has a lower cost. This implies that the schedule found by HP* is an optimal solution.

**Algorithm 2** Pseudocode of the HP* method.

1. Let $L_{open}$ and $L_{closed}$ be empty lists
2. Let $s_{init}$ be a node with an empty schedule
3. Add $s_{init}$ to $L_{open}$ and calculate $f(s_{init})$
4. while ($L_{open}$ is not empty) do
5. Let $s_{cur}$ be the node in $L_{open}$ with the smallest value $f(s_{cur})$
6. Remove $s_{cur}$ from $L_{open}$
7. if ($s_{cur}$ is a complete schedule) then
8. Terminate the algorithm
9. end if
10. if ($s_{cur} \notin L_{closed}$) then
11. for (each unscheduled task $T$) do
12. Add $s_{cur}$ to $L_{closed}$
13. for ($j = 1, \ldots, n_N$ and $p = 1, \ldots, p_j$) do
14. for (each assignment of task $T$ to $p$ cores of compute node $N_j$) do
15. Create a new node $s$ as a copy of $s_{cur}$
16. Add the assignment to $s$ and calculate $f(s)$
17. Add $s$ to $L_{open}$
18. end for
19. end for
20. end for
21. end if
22. end while

### 3.3 Time complexity of HP*

The time complexity of a scheduling method is important for its practical use. Scheduling is a NP-hard problem even for sequential tasks. Thus, unless $P = NP$, the problem of scheduling parallel tasks onto a heterogeneous platform cannot be solved in polynomial time. The worst time complexity of HP* is exponential in the number of tasks, since it searches for an optimal solution.

**Lemma 3.** The worst case time complexity of HP* is exponential in the number of tasks.

**Proof.** In the worst case scenario, HP* has to explore the whole search space to find an optimal solution. Thus, the worst-case execution time of HP* depends on the size of the search space. As stated in Section 3.2, the search space of HP* has the structure of a tree. The root of this scheduling decision tree is an empty schedule. Starting from the root, in each level of the scheduling decision tree, one more task is assigned to the schedule of the respective predecessor. Thus, the level of the scheduling decision tree indicates the number of tasks that have been scheduled and the depth of the tree is $n_T$. Each inner node of the scheduling decision tree has a successor for the assignment of each remaining task. Each task is assigned to each
combination of 1, …, \( p_j \) cores of each compute node \( N_j, j = 1, \ldots, n_N \). Thus, the branching factor of the scheduling decision tree can be calculated as follows:

\[
\frac{n_T}{n_T} \sum_{j=1}^{n_N} \sum_{p=1}^{p_j} \left( \frac{p_j}{p} \right).
\]

(7)

Since exactly one task is scheduled in each level, the number of remaining tasks \( r \) is decreasing with the depth of the scheduling decision tree. Thus, the maximum number of nodes of the scheduling decision tree can be calculated as follows:

\[
\frac{n_T!}{\left( \sum_{j=1}^{n_N} \sum_{p=1}^{p_j} \left( \frac{p_j}{p} \right) \right)^{n_T}}.
\]

(8)

The heuristic function \( h(n) \) has a big influence on the time complexity, since it guides the search of HP* through the scheduling decision tree. \( h(n) \) uses information about the remaining tasks to prevent HP* from expanding unpromising nodes. Pruning techniques as presented in Section 4 can also reduce the size of the scheduling decision tree drastically. Although the complexity of HP* is exponential in the worst case, the use of a good heuristic function and pruning techniques can reduce its practical running time. With these optimizations, HP* is appropriate for small to medium sized scheduling problems.

4 PRUNING TECHNIQUES AND OPTIMIZATIONS

As described in Section 3.2, the structure that is searched by HP* is the scheduling decision tree. The scheduling decision tree contains all possible schedules, and thus, a lot of unpromising nodes will not lead to an optimal solution. Pruning techniques eliminate such nodes from the search space without losing the ability of finding an optimal solution. The advantages are a reduced memory consumption of HP* and the reduction of the number of nodes that need to be visited. In this section, we present three techniques to prune the search space of HP*. In Section 4.1, a technique to remove all duplicate nodes from the scheduling decision tree is described. A technique for removing all equivalent nodes from the scheduling decision tree is given in Section 4.2. In Section 4.3, a technique that uses a heuristic upper bound to reduce the memory consumption of HP* is presented.

4.1 Removing duplicate nodes

Since HP* considers all possible orders and assignments of the tasks to the processor cores, the scheduling decision tree can contain a lot of duplicate nodes.

Definition 4. (Duplicate nodes). Two nodes in the scheduling decision tree are said to be duplicates if the schedules that are represented by these nodes are equal. Two schedules are equal if each processor core has been assigned to the same tasks in the same order.

Figure 2 shows how multiple paths in the scheduling decision tree can lead to duplicate nodes. In the example, the scheduling problem consists of two tasks (yellow and green), which should be assigned to two compute nodes \( N_1 \) with two processor cores and \( N_2 \) with one processor core. In the first case, one task (yellow) is assigned to the compute node \( N_1 \) and afterward, the second task (green) is assigned to the compute node \( N_2 \). HP* also considers the reverse case, where the green task is assigned to \( N_2 \) before the yellow task is assigned to \( N_1 \). Both cases lead to duplicate nodes (marked blue) that are detected and skipped in HP* by using the list \( L_{\text{closed}} \) (line 10 of Algorithm 2).

4.2 Removing equivalent nodes

For each task, the scheduling decision tree contains assignments to all permutations of the processor cores of a compute node. Since the processor cores of each compute node are homogeneous, the scheduling decision tree can also contain a lot of equivalent nodes. Thus, for each task, its execution time on a compute node is the same for the same number of cores regardless of the assignment to specific cores. This means that every permutation of the cores in a schedule represents an equivalent schedule. A schedule can be normalized by sorting the processor cores of each compute node, respectively, according to the maximum finish time of all tasks assigned to each core.

Definition 5. (Equivalent nodes). Two nodes in the scheduling decision tree are said to be equivalent if they represent the same schedule except for a permutation of the processor cores. To be more precise, two nodes \( n_i \) and \( n_j \) in the scheduling decision tree are said to be equivalent if the following conditions are true:
Figure 2 shows a section of a scheduling decision tree for two tasks and two compute nodes. This scheduling decision tree contains two equivalent nodes in each subtree (marked red). Equivalent nodes are detected and skipped before they are added to the list \( L_{\text{open}} \) (line 17). The newly created schedules are normalized so that equal schedules can be detected. Only unique schedules are added to \( L_{\text{open}} \), all equal schedules are discarded.

### 4.3 Using a heuristic upper bound

A nonoptimal solution for the considered problem of scheduling parallel tasks onto heterogeneous platforms can be found in polynomial time using a heuristic scheduling method. The makespan \( M_{\text{heur}} \) of such a schedule can be used as an upper bound for the cost \( f(n) \) of each node \( n \) in the scheduling decision tree. Since the cost of an optimal solution has to be lower or equal than \( M_{\text{heur}} \), graph nodes with higher costs will never be visited and thus, can be removed from the search space. This optimization will not reduce the number of graph nodes that need to be visited. However, removing such unpromising graph nodes reduces the memory consumption of HP*. The better the solution found by the heuristic method, the more efficient is this optimization strategy.

**Corollary 1.** The presented pruning techniques for removing duplicate and equivalent nodes as well as using a heuristic upper bound have no influence on the optimality of HP*.

**Proof.** When HP* proceeds from each of the duplicate nodes, the same remaining tasks are assigned to all combinations of processor cores. This leads to the same subtrees in the scheduling decision tree, each of which contains the same possible solutions. Graph nodes are only skipped by HP* if a duplicate node has already been visited. Thus, at least one of the subtrees of the duplicate nodes remains in the scheduling decision tree and no possible solution is eliminated. Therefore, removing duplicate nodes reduces the search space without eliminating optimal solutions.

Equivalent nodes are equal except for the naming of the processor cores which has no influence on the makespan. Proceeding from each equivalent node, HP* assigns the same remaining tasks to all combinations of processor cores. This leads to equivalent subtrees in the scheduling decision tree, each of which contains equivalent solutions with the same makespan. Since one of the equivalent subtrees is kept in the scheduling decision tree, no possible solution is eliminated. Thus, equivalent nodes can be removed from the search space without eliminating optimal solutions.

In HP*, a node \( n \) is skipped if its cost \( f(n) = g(n) + h(n) \) are higher than a heuristic upper bound \( b \). The heuristic cost function \( h(n) \) (Equation (6)) underestimates the exact cost \( h'(n) \) for each node \( n \). Thus, for each node \( n \), the costs \( f(n) \) are lower than the exact costs \( f'(n) \). A solution \( n_{\text{opt}} \) is optimal if no other solution has less cost, that is, \( f'(n_{\text{opt}}) \leq f'(n) \) holds for each node \( n \). As a consequence, the heuristic upper bound \( b \) is greater or equal to \( f'(n_{\text{opt}}) \). Thus, the optimal solution is not skipped by HP*.

**Corollary 2.** The worst case time complexity for pruning the search space of HP* does not exceed the computing time of HP*.

**Proof.** Duplicate nodes can be detected by checking whether the node has already been visited. This means that each node has to be compared with each previous node. Thus, a total of \( \sum_{i=1}^{n_T} i = \frac{n_T(n_T+1)}{2} \) comparisons are needed to detect duplicate nodes.
Equivalent nodes can be detected during the expansion of the current node. For each compute node, the generated schedules have to be compared. Thus, in each step of HP:\[ \sum_{j=1}^{nN} (\binom{n}{j}) \] comparisons are needed to detect equivalent nodes.

5 | EXPERIMENTAL RESULTS WITH PARALLEL TASKS ON A HETEROGENEOUS COMPUTE CLUSTER

This section presents experimental results of HP* for the execution of independent parallel tasks on a heterogeneous compute cluster. The experimental setup is described in Section 5.1. In Section 5.2, the makespans of the produced schedules of HP* are measured and compared to the results of three other popular scheduling methods. The influence of the pruning techniques from Section 4 on the performance of HP* is evaluated in Section 5.3. In Section 5.4, HP* is compared with the newly implemented metaheuristic scheduling methods TS* and SA* in terms of the makespans of the produced schedules and the computing times of the scheduling algorithm itself.

5.1 | Experimental setup

The heterogeneous compute cluster used consists of three nodes with a total of 16 processor cores. Table 3 lists the properties of these compute nodes. The compute node sb1 is used as reference node for the determination of the parameters described in Section 2.2. The scheduling method described in Section 3.2 is implemented in C++ using the gcc compiler with optimization level 2. Additionally, we have implemented three existing heuristic scheduling methods that are suitable for the scheduling of parallel tasks on heterogeneous platforms:

- **HCPA**: The HETEROGENEOUS CRITICAL PATH AND ALLOCATION method transforms a heterogeneous compute cluster with individual computational speeds of the processors into a “virtual” homogeneous cluster with equal speed. Then, an existing method for homogeneous compute clusters (ie, CPA) is used for the scheduling. HETEROGENEOUS CRITICAL PATH AND ALLOCATION is a two-phase scheduling algorithm: one phase to determine the number of processors that are allocated to each task, and a second phase to schedule the tasks on the platform using a list scheduling algorithm.

- **Δ-CTS**: The Δ-CRITICAL TASK SET method is an extension of an existing scheduling method for sequential tasks on heterogeneous compute clusters (ie, HEFT) to parallel tasks. Δ-CRITICAL TASK SET is a modification of a list-scheduling approach where in each step, a set of tasks with similar sequential execution time are selected. For each of these particular tasks, the compute node and number of cores are determined separately such that the earliest finish time of the task is minimized. The maximum number of cores to be used by each task depends on the number of selected tasks.

- **WLS**: The WATER-LEVEL-SEARCH method combines list scheduling with a search based approach. The method uses a limit for the predicted makespan that must not be exceeded by the finish time of any task. First, a list scheduling approach is applied repeatedly while the limit is increased until all tasks are scheduled. All computed finish times of all tasks are collected in a set of limits. Then a binary search on this list is performed to find the smallest limit where all tasks can be scheduled.

A separate front-end node of the compute cluster is responsible for the scheduling and for starting the task execution using SSH connections to the compute nodes. This front-end node is equipped with an Intel Xeon E5-2683 cpu and 128 GB of RAM. The tasks are executed according to the determined schedule and the total execution time is measured. A task is started as soon as all of its predecessors in the schedule have finished. The measurements are performed five times and the average result is shown.

As parallel tasks, two application tasks and two kernel tasks from the SPLASH-3 benchmark suite are used. Unless otherwise stated, the default parameters or the provided “parsec-simlarge” parameter sets are used for the different benchmark tasks. The following application and kernel tasks were selected:

- **BARNES (application)**: Barnes-Hut algorithm for a simulation of a particle system of $2^{18}$ particles.
- **FMM (application)**: Fast multipole method for a simulation of a particle system of $2^{19}$ particles.

| Nodes  | Processors | number of cores | total RAM | GHz  |
|--------|------------|-----------------|-----------|------|
| skylake1 | Intel i7-6700 | 4               | 16 GB     | 3.40 |
| hw1     | Intel i7-4770K | 4               | 16 GB     | 3.50 |
| sb1     | Intel Xeon E5-2650 | 8            | 32 GB     | 2.00 |

Table 3: List of nodes of the utilized heterogeneous compute cluster.
• **CHOLESKY (kernel):** Cholesky factorization of a sparse matrix. The matrix "tk29.O" of size 13992×13992 is used.

• **LU (kernel):** LU factorization of a dense matrix. The size of the input matrix is set to 4096×4096.

### 5.2 Performance results with benchmark tasks

In the following, the search-based scheduling method (HP*) proposed in Section 3.2 and the scheduling methods (HCPA, \(\Delta\)-CTS, WLS) described in the previous subsection are investigated in several measurements. These methods are used to determine schedules for the execution of the SPLASH-3 benchmark tasks on a heterogeneous cluster. The heterogeneous cluster used for the following measurements consists of all compute nodes listed in Table 3.

Figure 3 (top) shows the measured total execution times of the BARNES application tasks (left) and FMM application tasks (right) of the SPLASH-3 benchmark depending on the number of tasks. For both types of application tasks, the measured times using the HP* method are lower or equal than the results of the three heuristic scheduling methods (HCPA, \(\Delta\)-CTS, WLS). The results of WLS and HP* show a more steady increase compared with HCPA and \(\Delta\)-CTS. Especially for \(\Delta\)-CTS, a strong increase of the execution times for 7 and 13 tasks can be observed. This behavior might be caused by the heuristics used by \(\Delta\)-CTS. Using HP* leads to slightly lower or equal measured execution times compared to WLS, except for 7 and 8 tasks where HP* achieves up to 23% lower execution times.

Figure 3 (bottom) shows the measured total execution times of CHOLESKY kernel tasks (left) and LU kernel tasks (right) depending on the number of tasks using all compute nodes of Table 3. For the CHOLESKY tasks, the execution times using HCPA are up to 87% higher than the best results. A reason for these significant differences might be that HCPA favors a parallel task execution that uses many cores for each task. However, the execution times of the CHOLESKY tasks are too small to achieve a proper reduction of the parallel execution time for increasing numbers of cores. The other methods achieved very similar results, except for task numbers between 3 and 7 where \(\Delta\)-CTS leads to execution times that are up to 42% higher. For the LU tasks, the differences between the results of the methods used are smaller. The execution times using HCPA and \(\Delta\)-CTS are slightly higher than for WLS and HP* with large increases for 7 and 13 tasks. As for the application tasks, the execution times for 7 and 8 tasks are up to 11% lower using HP* compared with WLS. All in all, HP* leads to lower or equal execution times with a more steady increase compared with the other methods.

![Figure 3](image-url)

**FIGURE 3** Top: Measured total execution times of BARNES application tasks (left) and FMM application tasks (right) depending on the number of tasks using all compute nodes of Table 3. Bottom: Measured total execution times of CHOLESKY kernel tasks (left) and LU kernel tasks (right) depending on the number of tasks using all compute nodes of Table 3
5.3 Evaluation of the pruning techniques

In this section, the benefit of the pruning techniques presented in Section 4 is evaluated. As the scheduling time of HP* depends on the hardware and implementation used, we evaluated the number of nodes that are generated and the number of nodes that are visited by HP* to find a solution. These numbers are hardware- and implementation-independent and approximately proportional to the runtime of HP*. The scheduling problem considered in this evaluation consists of 1 to 16 BARNES tasks that have to be assigned to the heterogeneous compute cluster listed in Table 3.

Table 4 shows the numbers of nodes that are generated and visited by HP* using different pruning techniques depending on the number of tasks. The nodes that are generated and also the nodes that have been visited have to be stored during the execution of HP*. Whenever too many nodes have to be stored, the system runs out of memory. The missing entries in Table 4 are the result of such a situation. Thus, without pruning HP* is applicable only to small problems. The pruning technique of removing duplicate nodes described in Section 4.1 reduces the number of nodes significantly. Without duplicates HP* needs to visit and generate up to 94% less nodes to find a solution. Nonetheless, for more than 6 tasks, the system runs out of memory. The biggest impact of all presented pruning techniques is achieved by removing equal nodes as described in Section 4.2. Without equal nodes HP* can schedule up to 11 tasks before running out of memory. The reason is that the number of nodes that need to be handled by HP* in this case is reduced up to 99% compared with the version without pruning. The WATER-LEVEL-SEARCH method that is presented in Section 5.1 is used to determine the heuristic upper bound used in this evaluation. As characterized in Section 4.3, the heuristic upper bound has no influence on the number of nodes visited by HP*. However, the number of nodes that are generated is reduced up to 99% compared with the version without pruning. Even this significant reduction does not prevent the system from running out of memory for more than 7 tasks.

Each pruning technique itself can reduce the number of nodes significantly. However, even for small problems, the memory consumption is still too high. Since each pruning technique removes a certain set of unpromising nodes, a combination of all pruning techniques might further reduce the memory consumption. Therefore, we integrated all presented pruning techniques in HP* to apply a complete pruning of the search space. The heuristic cost function \( h(n) \) presented in Section 3.2 also influences the number of nodes in the scheduling decision tree. A version of HP* using \( h(n) \) is compared with a version without a heuristic cost function, that is, where \( h(n)=0 \). Table 5 shows the numbers of nodes that are generated and visited by HP* using a combination of all pruning techniques depending on the number of tasks. The results of this complete pruning are presented with and without the heuristic cost function compared to HP* without pruning. The complete pruning removes almost all unpromising nodes from the scheduling decision tree except for the nodes that are most relevant for finding an optimal solution. Compared with HP* without pruning this leads to a huge reduction of visited and generated nodes up to about 99.9%. A comparison with the results presented in Table 4 shows that the combination of all pruning techniques reduces the search space much more than each single pruning technique. This confirms that there is no dominating pruning technique and that each technique prunes a different part of the scheduling decision tree. This makes HP* applicable to small and medium sized scheduling problems. Although the heuristic cost function \( h(n) \) is no pruning technique in general, it is important for the performance of HP*. HP* without a heuristic cost function generates up to 9% and visits up to 6% more nodes.

### Table 4 Number of nodes generated and visited by HP* using isolated pruning techniques depending on the number of tasks

| #Tasks | No pruning | Without duplicates | Without equal nodes | Heuristic upper bound |
|--------|------------|--------------------|---------------------|-----------------------|
|        | Visited    | Generated          | Visited             | Generated             | Visited | Generated | Visited | Generated |
| 1      | 1          | 285                | 1                   | 285                   | 1       | 16        | 1       | 1         |
| 2      | 2          | 570                | 2                   | 570                   | 2       | 32        | 2       | 3         |
| 3      | 17         | 4.845              | 12                  | 3.420                 | 8       | 131       | 17      | 33        |
| 4      | 15.965     | 4.550.025          | 3.385               | 964.725               | 146     | 2.660     | 15.965  | 22.138    |
| 5      | 69.137     | 19.704.045         | 7.333               | 2.089.905             | 322     | 5.921     | 69.137  | 84.746    |
| 6      | 236.157    | 67.304.745         | 12.871              | 3.668.235             | 550     | 10.087    | 236.157 | 305.762   |
| 7      | —          | —                  | —                   | —                     | 23.089  | 489.373   | 81.178.658 | 104.911.723 |
| 8      | —          | —                  | —                   | —                     | 81.229  | 1.589.633 | —       | —         |
| 9      | —          | —                  | —                   | —                     | 90.481  | 1.819.179 | —       | —         |
| 10     | —          | —                  | —                   | —                     | 756.512 | 16.604.150 | —       | —         |
| 11     | —          | —                  | —                   | —                     | 2.560.273 | 60.473.639 | —       | —         |
| 12     | —          | —                  | —                   | —                     | —       | —         | —       | —         |
The search-based scheduling method (HP*) proposed in Section 3.2 is compared with the newly implemented metaheuristic scheduling methods TS* and SA*. TS* and SA* are methods for scheduling parallel tasks onto a heterogeneous platform based on a tabu search and simulated annealing, respectively. These methods are used to determine schedules for the execution of the SPLASH-3 benchmark tasks on a heterogeneous cluster. The heterogeneous cluster used for the following measurements consists of all compute nodes listed in Table 3.

Figure 4 (top) shows the measured total execution times of the BARNES application tasks (left) and FMM application tasks (right) of the SPLASH-3 benchmark depending on the number of tasks. For both types of application tasks, the measured times using the HP* method are lower or equal than the results of the metaheuristic scheduling methods TS* and SA*. Using HP*, the measured times of both application tasks are up to 22% lower than for TS* and SA*. In four cases, for 27 and 28 BARNES tasks and for 22 and 23 FMM tasks, the scheduling of HP* was aborted due to too high memory consumption. One reason might be that the heuristic cost function \( h(n) \) (Equation (6)) underestimated the exact cost of the remaining tasks too much. As a result, HP* has to consider much more possible solutions in its search space. A more precise heuristic cost function may avoid such situations. Except for 12 and 13 BARNES tasks, the measured times using SA* are lower or equal compared with TS*. For the FMM tasks, both metaheuristics show similar results with slightly lower execution times using SA*.

Figure 4 (bottom) shows the measured total execution times of CHOLESKY kernel tasks (left) and LU kernel tasks (right) depending on the number of tasks using all compute nodes of Table 3. For both types of kernel tasks, the execution times for the HP* method are lower or equal compared to the metaheuristic scheduling methods TS* and SA*. For the CHOLESKY tasks, the execution times using HP* are up to 17% lower than for TS* and up to 22% lower than for SA*. The scheduling of HP* for 25-30 CHOLESKY tasks and 24, 25, and 32 LU tasks was aborted due to too high memory consumption. Especially for the short CHOLESKY tasks, the heuristic cost function \( h(n) \) (Equation (6)) underestimates the exact cost of the remaining tasks too much. Thus, HP* needs to search a wider range of the search space. A more precise heuristic cost function may avoid such situations. For 1 to 22 CHOLESKY tasks, SA* achieved lower execution times than using TS*, whereas for 23 to 32 tasks, the measured times for TS* are lower than for SA*. Especially for 29 to 32 CHOLESKY tasks, the execution times are up to 27% higher using SA*. For the LU tasks, there are less differences between the results of all three methods compared with the other tasks. The execution times using HP* are up to 10% lower than for TS* and up to 31% lower than for SA*. For more than 20 LU tasks, the execution times for SA* are up to 8% lower compared with TS*.

Figure 5 shows the measured computing times of the search-based scheduling methods depending on the number of BARNES application tasks using all compute nodes of Table 3. The increase in the computing time of TS* and SA* is steadier than for HP*. A reason for this might be the use of the heuristic cost function \( h(n) \) (Equation (6)) and the pruning methods described in Section 4. The influence of the heuristic

### Table 5: Number of nodes generated and visited by HP* using all pruning techniques depending on the number of tasks

| Tasks | No pruning | Complete pruning | Complete pruning without heuristic cost function |
|-------|------------|------------------|--------------------------------------------------|
|       | Visited  | Generated | Visited | Generated | Visited | Generated |
| 1     | 1       | 285       | 1       | 1         | 1       | 1         |
| 2     | 2       | 570       | 2       | 3         | 2       | 3         |
| 3     | 17      | 4.845     | 6       | 13        | 6       | 13        |
| 4     | 15.965  | 4.550.025 | 46      | 112       | 46      | 112       |
| 5     | 69.137  | 19.704.045| 76      | 186       | 76      | 186       |
| 6     | 236.157 | 67.304.745| 106     | 280       | 106     | 280       |
| 7     | —       | —         | 936     | 2.827     | 936     | 2.827     |
| 8     | —       | —         | 1.824   | 5.724     | 1.824   | 5.724     |
| 9     | —       | —         | 1.900   | 5.734     | 1.929   | 5.789     |
| 10    | —       | —         | 7.272   | 22.771    | 7.479   | 23.244    |
| 11    | —       | —         | 15.416  | 49.352    | 16.335  | 53.922    |
| 12    | —       | —         | 62.370  | 220.145   | 62.999  | 222.054   |
| 13    | —       | —         | 534.397 | 2.195.234 | 534.397 | 2.195.234 |
| 14    | —       | —         | 1.274.832 | 4.518.356 | 1.275.255 | 4.519.432 |
| 15    | —       | —         | 4.957.128 | 20.415.293 | 4.957.318 | 20.415.054 |
| 16    | —       | —         | 12.637.246 | 49.721.482 | 12.638.539 | 49.727.127 |
cost function and the pruning methods differs strongly with the concrete scheduling problem. Influencing factors are the size and structure of the heterogeneous platform as well as the number of tasks and their runtime formula. Except for 16, 22, 30, 31, and 32 tasks, the computing time of HP\(^*\) is lower than for SA\(^*\). For small numbers of tasks, HP\(^*\) needs even less computing time than TS\(^*\). In average, TS\(^*\) is 73 times faster than SA\(^*\).

6 | RELATED WORK

In Reference 22, a classification of parallel tasks into rigid tasks, moldable tasks, or malleable tasks is given. For rigid tasks, the number of processors used is fixed a priori. Malleable tasks can change the number of processors during their execution. Most parallel applications are moldable. This means that the number of processors is fixed during the execution but not before.
Search-based methods are promising approaches to solve scheduling problems. A task scheduling problem can be formulated as the search for an optimal assignment of a set of tasks onto a set of processors, such that the total execution time, also called makespan, is minimized. In the past decade, different types of search-based scheduling algorithms, the so-called metaheuristics, such as genetic algorithms, simulated annealing, or tabu search, have been proposed. A comparison of search-based and heuristic approaches for scheduling independent tasks onto heterogeneous systems can be found in Reference 2. Different approaches for assigning sequential tasks to heterogeneous machines are presented, like genetic algorithms, simulated annealing, tabu search, and A*. All approaches focus on the scheduling of sequential tasks and, thus, cannot be applied to the scheduling problem of parallel tasks.

An experimental comparison of several scheduling algorithms, including A*, genetic algorithms, simulated annealing, tabu search, as well as popular list scheduling heuristics, is given in Reference 23. Since the work considers the problem of mapping sequential tasks with dependencies onto a homogeneous cluster, the algorithms are not suitable for the scheduling of parallel tasks.

Reference 9 proposed a scheduling algorithm for the assignment of sequential tasks with dependencies to homogeneous platforms based on the A* search algorithm. Several pruning techniques to reduce the search space as well as a parallelization of the algorithm are presented. Reference 10 proposed a scheduling algorithm based on the A* search algorithm. The algorithm uses an improved cost function along with several pruning techniques to reduce the search space. In Reference 11, additional pruning techniques for the same scheduling problem are proposed. In contrast to these works, we consider the scheduling of independent parallel tasks to heterogeneous platforms. To the best of our knowledge, the A* search algorithm has not been applied to the scheduling of parallel tasks onto a heterogeneous platform. Although some pruning techniques proposed in References 9 and 11 as the heuristic upper bound can be applied to this problem, specific techniques are needed to reduce the search space.

In Reference 24, a method based on a tabu-search for scheduling independent rigid tasks (i.e., parallel tasks with a fixed number of cores) onto a homogeneous cluster is proposed. Since the numbers of processors are fixed a priori, such a scheduling problem can be seen as a two-dimensional packing problem. The tabu search approach is compared with three greedy strategies for this packing problem. We focus on the scheduling of moldable tasks, which is more complex and allows more choices.

A simulated annealing approach for the scheduling of sequential tasks with dependencies onto heterogeneous multiprocessor systems is proposed in Reference 25. In Reference 26, the best practices for defining the temperature, the acceptance functions, and move heuristics are presented. Although the given information is helpful for scheduling with simulated annealing in general, the scheduling approaches themselves are not suitable for parallel tasks.

Genetic algorithms are inspired by biological evolution, such as reproduction, mutation, recombination, and selection. In Reference 27, a genetic algorithm for the scheduling of independent sequential tasks onto a heterogeneous cluster is proposed. In genetic algorithms, the schedules are represented by chromosomes on which genetic operators such as crossover or mutation are applied. However, the chromosomal representation of schedules for sequential tasks is not suitable for parallel tasks.

7 | CONCLUSION

In this article, we have proposed a task scheduling method HP* for assigning parallel tasks to heterogeneous platforms which is based on the A* search algorithm. In addition, a heuristic cost function has been proposed that is able to reduce the search space of our algorithm. The pruning techniques presented are able to significantly reduce the search space and improve the efficiency of HP*. Theoretical results as well as practical measurements have been presented. Measurements with benchmark tasks from the SPLASH3 benchmark suite have been performed and the scheduling results of HP* have been compared to three heuristic scheduling methods as well as newly implemented scheduling methods based on tabu search and simulated annealing. The performance results demonstrate that the use of HP* leads to a reduction of the total execution times (makespan) of the resulting schedules in comparison with the other scheduling algorithms. We have shown that the computing time of HP* is reasonable for small to medium sized scheduling problems. The evaluation of the pruning techniques has shown that each pruning technique reduces the search space significantly. The highest reduction in the search space is achieved by a combination of all pruning techniques. Also, this complete pruning can prevent the system from running out of memory and allows the computation of larger scheduling problems.

ACKNOWLEDGEMENTS

This work was supported by the German Ministry of Science and Education (BMBF) project “SeASiT e,” Grant No. 01IH16012A/B and the German Research Foundation (DFG), Federal Cluster of Excellence EXC 1075 “MERGE.”

ORCID

Robert Dietze <https://orcid.org/0000-0002-7475-8135>

Gudula Rünger <https://orcid.org/0000-0002-5364-2088>
REFERENCE
1. Arabnejad H, Barbosa JG. List scheduling algorithm for heterogeneous systems by an optimistic cost table. IEEE Trans Parall Distrib Syst. 2014;25(3):682-694.
2. Braun TD, Siegel HJ, Beck N, et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J Paral Distrib Comput. 2001;61(6):810-837.
3. Daoud MI, Kharna N. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. J Paral Distrib Comput. 2008;68(4):399-409.
4. Radulescu A, van Gemund A JC. Low-cost task scheduling for distributed-memory machines. IEEE Trans Parall Distrib Syst. 2002;13(6):648-658.
5. Topcuoglu H, Hariri S. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans Parall Distrib Syst. 2002;13(3):260-274.
6. Russell S, Norvig P. Artificial Intelligence: A Modern Approach. 3rd ed. Upper Saddle River, NJ: Prentice Hall Press; 2016.
7. Hart PE, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern. 1968;4(2):100-107.
8. Dechter R, Pearl J. Generalized best-first search strategies and the optimality of A. J ACM. 1985;32(3):505-536.
9. Kwok Y-K, Ahmad I. On multiprocessor task scheduling using efficient state space search approaches. J Paral Distrib Comput. 2005;65(12):1515-1532.
10. Shahul AZS, Sinnen O. Scheduling task graphs optimally with A. J Supercomput. 2010;51(3):310-332.
11. Sinnen O. Reducing the solution space of optimal task scheduling. Comput Operat Res. 2014;43:201-214.
12. Sakalis C, Leonardsson C, Kaxiras S, Ros A. Splash-3: A Properly Synchronized Benchmark Suite for Contemporary Research. Washington, WA: IEEE; 2016:101-111.
13. Fortune S, Wyllie J. Parallelism in Random Access Machines. New York, NY: ACM; 1978:114-118.
14. Skillicorn DB, Hill J, McColl WF. Questions and answers about BSP. Sci Progr. 1997;6(3):249-274.
15. Culler D, Karp R, Patterson D, et al. LogP: A Realistic Model of Parallel Computation. New York, NY: ACM; 1993:1-12.
16. Dietze R, Hofmann M, Rünger G. Analysis and Modeling of Resource Contention Effects based on Benchmark Applications. Washington, WA: IEEE; 2018:330-337.
17. N’Takpé T, Suter F. Critical path and area based scheduling of parallel task graphs on heterogeneous platforms. Paper presented at: Proceedings of the IEEE Computer Society; ICPADS 06; 2006:3-10; IEEE.
18. Radulescu A, Van Gemund A JC. A Low-Cost Approach Towards Mixed Task and Data Parallel Scheduling. Washington, WA: IEEE; 2001:69-76.
19. Suter F. Scheduling A-Critical Tasks in Mixed-Parallel Applications on a National Grid. Washington, WA: IEEE; 2007:2-9.
20. Topcuoglu H, Hariri S, Wu M-Y. Task Scheduling Algorithms for Heterogeneous Processors. Washington, WA: IEEE; 1999:3-14.
21. Dietze R, Hofmann M, Rünger G. Water-Level scheduling for parallel tasks in compute-intensive application components. The Journal of Supercomputing. 2016;72(11):4047–4068. http://dx.doi.org/10.1007/s11227-016-1711-1.
22. Dutot P-F, Mounié G, Trystram D. Chapter 26: scheduling parallel tasks: approximation algorithms. In: Leung Joseph T, ed. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Boca Raton, FL: CRC Press; 2004:26-1-26-24.
23. Jin S, Schiavone G, Turgut D. A performance study of multiprocessor task scheduling algorithms. J Supercomput. 2008;43(1):77-97.
24. Bożejko W, Nadybski P, Wodecki M. Two Level Algorithm with Tabu Search Optimization for Task Scheduling Problem in Computing Cluster Environment. Washington, WA: IEEE; 2017:238-242.
25. Biswas SK, Jagdev R, Muhuri PK. Energy Efficient Scheduling in Multiprocessor Systems Using Archived Multi-objective Simulated Annealing. Washington, WA: IEEE; 2018:1-8.
26. Orsila H, Salminen E, Hamalainen TD. Best practices for simulated annealing in multiprocessor task distribution problems. In: Tan CM, ed. Simulated Annealing. Rijeka, Croatia: IntechOpen; 2008.
27. Orts F, Ortega G., Puertas A. M., García I., Garzón E. M. On solving the unrelated parallel machine scheduling problem: active microrheology as a case study. The Journal of Supercomputing. 2020. http://dx.doi.org/10.1007/s11227-019-03121-z.

How to cite this article: Dietze R, Rünger G. The search-based scheduling algorithm HP* for parallel tasks on heterogeneous platforms. Concurrency Computat Pract Exper. 2020;32:e5898. https://doi.org/10.1002/cpe.5898