Karlsruher Institut für Technologie

Master's Thesis

Malleable Distributed Hierarchical Planning

Niko Wilhelm

Date of submission: November 1, 2022

Betreuer: Prof. Dr. Peter Sanders
M.Sc. Dominik Schreiber

Institut für Theoretische Informatik, Algorithmik
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 01. November 2022

Zusammenfassung

Totally Ordered Hierarchical Task Network (TOHTN) planning ist ein Teilgebiet der künstlichen Intelligenz und gehört zum Bereich des domänenunabhängigen Planens. Beim TOHTN planning werden offene Tasks rekursiv aufgelöst und verfeinert bis wir einen ausführbaren Plan aus primitiven Aktionen erhalten. TOHTN planning ist von hoher Komplexität gekennzeichnet. Es gehört zur Komplexitätsklasse D-EXPTIME und ist EXPSPACE-hart.
Malleability bezeichnet die Fähigkeit eines Programms, mit einer sich verändernden Menge zugewiesener Ressourcen zurecht zu kommen. Sie erlaubt den effizienten Einsatz von Ressourcen, da sie erlaubt Ressourcen zu jedem Zeitpunkt vollständig zu nutzen anstatt einen Kompromiss aus Durchsatz und Latenz anstreben zu müssen. Programme zu schreiben die dies erfüllen ist jedoch eine Herausforderung, weswegen es nur wenige Beispiele gibt.
Um die Möglichkeiten des parallelen Planens auszuloten und zudem die Möglichkeiten des Load Balancers und Schedulers Mallob aufzuzeigen haben wir unseren parallelen Planner CrowdHTN in Mallob integriert. Zudem haben wir die Performance von CrowdHTN mithilfe eines verteilten Schemas zur Schleifenerkennung basierend auf Bloom Filtern verbessert und Neustarts genutzt, um die Vollständigkeit unseres Algorithmus' zu garantieren.
In unserer Evaluation zeigen wir, dass sowohl die verteilte Schleifenerkennung als auch die Neustarts die Performanz und die Menge gelöster Instanzen erhöhen. Zudem zeigen wir, dass wir auch in einem veränderlichen Umfeld die Fähigkeit beibehalten, Instanzen zu lösen.

Abstract

Totally Ordered Hierarchical Task Network (TOHTN) planning is a sub-field of artificial intelligence and belongs to the area of domain-independent planning. In TOHTN planning we recursively resolve open tasks, refining them, until we obtain an executable plan consisting of primitive actions. TOHTN planning is computationally intensive and it belongs to the class of D-EXPTIME while being EXPSPACE-hard.
Malleability is the ability of a program to handle a changing amount of assigned resources at run time. It allows for a more efficient use of resources as it frees schedulers to utilize all resources at all times instead of striking a compromise between throughput and latency. At the same time, it is a challenge to implementers and few malleable programs exist.
To further explore parallel TOHTN planning and to demonstrate the capabilities of malleable scheduler and load balancer Mallob we perform an integration of our parallel planner CrowdHTN with the aforementioned Mallob. Additionally, to increase the performance of CrowdHTN, we introduce a new distributed loop detection scheme based on bloom filters and utilizing restarts for correctness.
In our evaluation we find that both distributed loop detection and restarts positively impact performance and coverage of our planner while showing that we retain the capability to solve problems in a malleable environment.

Acknowledgments

First and more than anyone else, I want to thank my supervisor Dominik Schreiber. His advice and patient guidance helped me get through this thesis and I am continuously amazed at how great his program Mallob truly is.
Second, I want to thank my flatmates and friends who took care of me during the stressful parts of this work, be it with regular meals or long walks in the park to help me relax and refresh my mind. Last, I want to thank Colin Bretl who was always ready to have another discussion on the finer points of TOHTN planning.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Planning via Hierarchical Task Networks (HTN) is a popular approach to automated AI planning. HTN planning works by repeatedly decomposing a set of initial tasks until they have been decomposed to the level of simple actions [26, 9]. These actions form a plan which can be executed to achieve the goal set out by the initial tasks. Totally Ordered (TO) HTN planning is an important sub-problem of HTN planning where all tasks are constrained by a total order. Hierarchical planners are easy to use as the hierarchy allows the user to insert a structure into the problem description and to provide the planner with advice to guide the planning procedure. As a result, HTN planning has been used in a number of fields. [46] have used HTN planning for AI in real-time strategy games. Similarly, [49] have improved their minimax game tree search in real-time strategy games via HTN, which allowed them to reduce the branching factor of their problem. This approach was further extended by [41] to also take the opponent's strategy into account. Further applications of HTN planning include automated web service composition [59] as well as the composition of cloud applications [27], socially assistive robotics [28], storyline visualizations [50] and automated machine learning [45].
While popular with users, HTN planning does present challenges for implementers as instances can be very CPU and memory intensive to solve. It was shown that HTN planning itself is only semi-decidable and that TOHTN planning is still in D-EXPTIME while being EXPSPACE-hard [19, 20]. Regarding the expressive power of TOHTN problems they correspond to the class of context-free languages [32]. Part of the complexity in HTN planning stems from its recursive nature. As a result of this, the detection of duplicate states plays an important role in the performance of planners [31].

Malleability is the ability of a parallel job to efficiently integrate new processing elements (PEs) into a parallel job at run time as well as handle a reduction of the available PEs [23]. Malleable programs are well-liked by administrators of supercomputers, as they allow the utilization of all compute resources, maintaining both high throughput and keeping latencies for new jobs low [23, 37]. The malleable model does pose additional challenges for application programmers, though, and malleable jobs are only easy to implement if we restrict our problems to those which split into small, independent work packages [23, 61]. As a result, few malleable applications exist.
However, in recent years a number of malleable SAT solvers have emerged. Among those Mallob [54] and Paracooba [29]. This is of special importance as SAT solvers serve as a building block in many other applications such as hierarchical planning. Having malleable SAT solvers available may allow for other applications to profit from this paradigm while being presented a simple to use interface.

In this thesis, we present three main advances in parallel and malleable TOHTN planning.
Firstly, we present a number of improvements for the parallel search-based planner CrowdHTN. Both HyperTensioN [43] and PANDA [31] have shown the importance of detecting duplicate search nodes to improve the performance of hierarchical planners. PANDA specifically uses an approach based on hashing which may fall back to full node comparisons to avoid false positives. Parallel planner CrowdHTN already includes a loop detection mechanism based on the ideas of PANDA [14]. We take the idea of PANDA to only compare hashes and not full nodes and generalize it to bloom filters which may use any number k of hash functions to reduce false positive rates. Bloom filters then allow us to present a design for a distributed loop detection mechanism. Additionally, PANDA has shown that heuristics can greatly increase the performance of search-based planner [35]. We try to adapt heuristic search for CrowdHTN while

under the added constraints of malleability. Doing so, we implement BFS, heuristic DFS and A* in CrowdHTN. Last, we take inspiration from SAT where restarts have been used since the 90's [18] to design a restarting scheme for CrowdHTN which allows it to be complete even as bloom filter based detection of duplicate nodes may lead to false positives.
Second, we provide a general overview of the completeness of different TOHTN planning approaches. We show that both search-based planners using BFS, A* and current SAT-based planners are complete. In addition we argue that heuristic best-first search may always be incomplete and that, while current loop detection mechanisms are helpful for planner performance, there are cases of recursion in hierarchical planning problems which they are unable to detect. Lastly, we show that our restart mechanism brings random DFS into the list of algorithms which are complete on all problems.
Third, we present our design of a malleable TOHTN planner. In this we integrate our planner CrowdHTN with the malleable job scheduler Mallob. We offer an overview of our design and show how work stealing in general can be adapted to a malleable framework while preserving completeness of the search. Doing so we also show Mallob's capabilities as a general purpose job scheduler and load balancer.
In our evaluation we find that our implementation suffers from some overhead due to the integration into Mallob. However, we also see that bloom filters in general outperform hash sets when it comes to detecting duplicate states in TOHTN planning and that this performance gain extends to our distributed loop detection scheme. We further find that restarts not only serve to ensure the completeness of our planner while using bloom filters but have an additional positive impact on overall performance. Regarding malleability, we see that our proposed malleable work stealing scheme can suffer from some loss of performance when a large number of PEs is frequently reshuffled but that this can be mitigated with frequent restarts.

The rest of this work follows the following structure: in section 2 we introduce a TOHTN planning formalism as well as planning techniques, followed by an intro to malleability and an overview of parallel and distributed computing techniques that inform the design of CrowdHTN. It ends with a short introduction to CrowdHTN and Mallob. Section 3 presents us two potential theoretical improvements to CrowdHTN and the design of a distributed loop detection scheme. It concludes with us showing how current loop detection schemes are unequipped to ensure completeness of hierarchical planners and argue for the use of restarts as an alternative technique. Section 4 presents our design of a malleable parallel TOHTN planner based on work stealing. This is followed by section 5 which contains implementation details of both our improvements and the malleable design. Finally, section 6 evaluates and compares our planner to it's old standalone version, presents the performance impact of our various improvements and provides an overview of the behavior of CrowdHTN as the number of PEs scales as well as under malleable conditions. Section 7 concludes this work.

2 Preliminaries

In this section we introduce the TOHTN formalism and discuss its complexity and differences to classical planning in 2.1. This is followed by an overview and comparison of TOHTN planning techniques in 2.2, and a classification of parallel programs and discussion of malleability in 2.3. In 2.4 we introduce the parallel techniques underlying our planner. We conclude by introducing our parallel planner CrowdHTN in 2.5 and the malleable load balancer and scheduler Mallob in 2.6.

2.1 (TO)HTN Formalism

In this section we first define what HTN and TOHTN problems are from a formal perspective 2.1.1. Afterwards we take a short look at the algorithmic worst case complexity of HTN and TOHTN planning 2.1.2. We conclude by taking a short look at how hierarchical and classical planning compare in 2.1.3 and present the way in which we visualize hierarchical problems in this work in 2.1.4.

2.1.1 Defining (TO)HTN Planning Problems

Both HTN and TOHTN planning are based on decomposing a list of initial tasks down into smaller subtasks until those subtasks can be achieved by simple actions. A number of formalisms for HTN plannings exist [26, 6, 56]. These formalisms are similar but differ slightly to suit specific planning approaches. In this work, we will reuse the formalism we introduced in [14] which is built on the definition by [26].

Definition 1. *A **predicate** consists of two parts. Firstly a predicate symbol $p \in \mathcal{P}$ where $\mathcal{P}$ is the finite set of predicate symbols. Secondly of a list of terms $\tau_1, \ldots, \tau_k$ where each term τ_i is either a constant symbol $c \in \mathcal{C}$, with $\mathcal{C}$ being the finite set of constant symbols, or a variable symbol $v \in \mathcal{V}$, where $\mathcal{V}$ is the infinite set of variable symbols.*
The set of all predicates is called $\mathcal{Q}$.

With the definition of a predicate in place, we can then define a grounding as well as our world state.

Definition 2. *A **ground predicate** is a predicate where the terms contain no variable symbols or, in other words, a predicate that contains only constant symbols.*

Definition 3. *A **state** $s \in 2^{\mathcal{Q}}$ is a set of ground predicates for which we make the closed-world-assumption. Under the closed-world-assumption, only positive predicates are explicitly represented in s. All predicates not in s are implicitly negative.*

Definition 4. *With T_p the set of primitive task symbols, a **primitive task** t_p is defined as a triple $t_p(\tilde{t}_p(a_1, \ldots, a_k), pre(t_p), eff(t_p))$. $\tilde{p} \in T_p$ is the task symbol, $a_1, \ldots, a_k \in \mathcal{C} \cup \mathcal{V}$ are the task arguments, $pre(t_p) \in 2^{\mathcal{P}}$ the preconditions and $eff(t_p) \in 2^{\mathcal{P}}$ the effects of the primitive task t_p. We further define the positive and negative preconditions of t_p as $pre^+(t_p) := \{p \in pre(t_p) : p \text{ is positive}\}$ and $pre^-(t_p) := \{p \in pre(t_p) : p \text{ is negative}\}$. We define $eff^+(t_p)$ and $eff^-(t_p)$ analogously.*
*We call a fully ground primitive task an **action**.*

As preconditions and effects may not be concerned with the whole world state the closed-world assumption does not apply to them. To any HTN instance we could create an equivalent one where each precondition and effect cares about the whole world state. This would be achieved by instantiating all the "don't care" terms in preconditions and effects with all possible combinations of predicates. Doing this would, however, come at the price of a huge blowup of our planning problem.

Definition 5. *An action t_p is **applicable** in state s if $pre^+(t_p) \subseteq s$ and $pre^-(t_p) \cap s = \emptyset$. The **application** of t_p in state s results in the new state $s' = (s \setminus eff^-(t_p)) \cup eff^+(t_p)$.*

Definition 6. *We define a **compound task** as $t_c = \tilde{t_c}(a_1, \ldots, a_k)$, where $\tilde{t_c} \in T_c$ is the task symbol from the finite set of compound task symbols T_c and $a_1, \ldots, a_k$ are the task arguments.*

Primitive and compound tasks together form task networks. In places where both can be used, we will refer to them simply as tasks $t \in T$.

Definition 7. *Let $T = T_p \bigcup T_c$ be a set of primitive and compound tasks. A task network is a tuple $\tau = (T, \psi)$ consisting of tasks T and constraints ψ between those tasks.*

Definition 8. *Let M be a finite set of method symbols and $T = T_p \bigcup T_c$ a set of primitive and compound tasks. A **method** $m = (\tilde{m}(a_1, \ldots, a_k), t_c, pre(m), subtasks(m), constraints(m))$ is a tuple consisting of the method symbol $\tilde{m}$, the method arguments $a_1, \ldots, a_k$, the associated compound task $t_c \in T_c$ the method refers to, a set of preconditions $pre(m) \in 2^{\mathcal{P}}$, a set of tasks $subtasks(m) = \{t_1, \ldots, t_l\}, t_i \in T$ and a set of ordering constraints $c_1, \ldots, c_m$ defining relationships between the subtasks. Any arguments appearing in $t_c, pre(m), subtasks(m)$ must also appear in $a_1, \ldots, a_k$.*
In TOHTN planning, $constraints(m)$ is implicitly set s.t. the subtasks $t_1, \ldots, t_l$ are totally ordered.
*We call a fully ground method a **reduction**.*

Each method m has exactly one associated compound task t_c. However, multiple methods $m_1, \ldots, m_k$ may be associated with a single compound task t_c. Additionally, while any arguments of t_c must be present in m, the contrary is not true and m may have arguments not present in t_c, i.e., m is not fully determined by t_c. As a result, methods represent choice points both in the choice of method itself as well as through the argument instantiation.

Definition 9. *Let $\tau = (T, \psi)$ be a task network, s a state, $m = (\tilde{(m)}(a_1, \ldots, a_k), t_c, pre(m), subtasks(m), constraints(m))$ be a method. m **resolves** τ iff $t_c \in T$, the constraints in ψ allow for t_c to be resolved, $pre^+(m) \in s$ and $pre^-(m) \cap s = \emptyset$.*
Resolving a compound task $t \in T$ results in a new task network $\tau' = ((T \setminus t) \cup \{t : t \in subtasks(m)\}, \psi \cup constraints(m))$ and unchanged state s.
Applying a primitive task results in a new task network $\tau' = (T \setminus t, \psi)$ in state s' where the effects of t have been applied to s.

Definition 10. *An **HTN domain** is a tuple $D = (V, C, P, T, M)$ consisting of finite sets variables V, constants C, predicates P, tasks T and methods M. An **HTN problem** $\Pi = (D, s_0, \tau_0)$ consists of a domain D, an initial state s_0 and an initial task network τ_0.*
*If $subtasks(m)$ has a total order for all $m \in M$ and the tasks in τ_0 are totally ordered, we speak of a **TOHTN domain** and **TOHTN problem**.*

It is possible to simplify the model s.t. τ_0 always consists of only a single task with no constraints. We do this by inserting a new initial task t_0 and method m_0 with no arguments s.t. resolving t_0 via m_0 results in τ_0.
Another way of viewing HTN problems is as AND/OR trees [34] where tasks form OR-nodes where one of may methods is chosen and methods form AND-nodes, as all subtasks need to be resolved.

2.1.2 Complexity of (TO)HTN planning

The complexity of HTN and TOHTN planning has been studied in many papers. Here the problem PLANEXIST describes, whether for any given (TO)HTN instance a plan exists at all. It is not concerned with optimality.
Early on it was shown by [19] and [20] that the complexity of hierarchical planning formalisms depends on things such as the existence and ordering of non-primitive tasks, whether a total order between tasks is imposed and whether variables are allowed. The combination of arbitrary non-primitive tasks, no total order imposed and allowing variables is what we talk about with HTN planning, the same combination but with a total order is what we mean with TOHTN planning. They showed that HTN planning is semi-decidable whereas TOHTN planning is decidable in D-EXPTIME while being EXPSPACE-hard.
We can see what D-EXPTIME means in practise when we consider the maximum size of a task network we need to consider. From [5] we know that if a solution to an HTN instance exists, it can be found within a maximum depth of $|T_c| \cdot (2^{|\mathcal{Q}|})^2$. Similarly, we see that a task network can have exponential width in it's depth. Consider for this an instance constructed such that each compound task has exactly two children and where primitive tasks are only occuring at the bottom most layer. Now each layer will be twice as wide as the one before, giving us exponential width.
Regarding the general relationship of hierarchical planning to complexity theory, [19] and [20] showed early on that HTN instances can be used to simulate context-free languages. This was extended by [32] who showed that TOHTN instances correspond exactly to context-free grammars.
In addition to planning itself, the problem of plan verification was studied. Here, [4] showed that plan verification is NP-complete, even under the assumption that not only the plan but also the decompositions leading to it are provided.

2.1.3 Differences from other Kinds of Planning

[48] creates a classification of planners into domain-specific, domain-independent and domain-configurable planners. They argue that HTN planning falls under domain-configurable with the decompositions providing advice to the planner to gain efficiency. [35] argue that HTN-planning is not simply a domain-configurable version of classical planning on the basis that [19, 20] showed that HTN-planning is strictly more powerful compared to classical planning which is PSPACE-complete.
While we agree with [35], one can still use HTN planning without using the full complexity of the model, using it instead to provide more efficient and guided versions of classical planning problems.

2.1.4 Graphically Representing TOHTN Problems

In the rest of this work we will sometimes represent the structure of TOHTN domains graphically. We will always use the same scheme which we explain here. In our visualization, we

Figure 1: An example TOHTN domain

represent the structure of a domain as a series of methods. To the left we show the compound task, followed by the method itself on its right and followed again by the method subtasks in their fixed order. Compound tasks are always represented by green rounded squares and their name, methods by an arrow with the method name above and actions by blue rounded squares and their name.
A short example is shown in figure 1. It consists of one method, m_1 which resolves task t_1. The subtasks of m_1 are t_2, a_1 and a_2. t_1 and t_2 are compound tasks, a_1 and a_2 are actions.

2.2 Techniques to solve HTN planning problems

In this section, we will give an overview over the different techniques with which HTN problems can be solved. The HTN planners produced by researchers can be classified along two main axes:

- the planning algorithm
- lifted vs grounded approaches

For the algorithms, the two main variations are translation-based algorithms that take an HTN instance and translate it into a problem in a simpler complexity class such as classical planning or propositional logic (SAT) and search-based algorithms that utilize techniques such as plan-space search and progression search. We will focus on progression search here, as it is the technique employed in our own planner, CrowdHTN.
After that we will have a short discussion on lifted versus grounded approaches which is largely independent of the search algorithm.

2.2.1 Translation-based

One of the main techniques employed in HTN planning is to find an efficient encoding into a simpler problem. Two such problems are classical planning [1] and SAT. While translation to SAT was already proposed in 1998 [44], the first complete encoding without assumptions about the instance was publicized in 2018 [5]. In recent years, SAT seems to be the most popular problem to translate an HTN instance into, utilized by planners such as totSAT [5, 6], Tree-REX [57] and Lilotane [56].
As we have seen in the previous section 2.1.2 on (TO)HTN complexity, (TO)HTN problems are in D-EXPTIME and undecidable respectively. Both classical planning and SAT are less powerful. As a result, HTN problems cannot be encoded and even for TOHTN problems we would suffer a blowup in the size of the instance. Instead, as noted in [57], SAT-based planners tend to explore subproblems by encoding the set of potential hierarchies layer by layer, increasing the encoding size as they go. As a result, those SAT-based planners tend to have a BFS-like characteristic to their search.

2.2.2 Search-based

The second main category of techniques to solve HTN planning problems are search-based algorithms, such as plan space search and progression search.
Plan space search searches the space of partial plans, where search nodes represent partial plans and edges represent plan refinements [62].
As progression search is both more prominent in current planners and our own planner, CrowdHTN, also utilizes it we will now focus on this paradigm according to [35]. Progression search generates plans in a forward way. Search nodes are represented as tuples (tn, s) of open tasks tn and world state s. It always chooses an open task that is currently unconstrained, i.e. has no unresolved predecessors under the ordering constraints, and resolves it. This allows the planner to update the world state as it goes along, as the sequence of actions from the start to the current point is known at each step of the search. In case of TOHTN planning, the choice of the next unconstrained task becomes trivial, as there is always exactly one such task. Knowing the full world state gives progression search two main advantages over plan space search. First, it allows the planner to prune parts of the search space by immediately validating action and reduction preconditions against this world state. Second, it gives us maximum information to be used in heuristics that guide our search.
The progression search algorithm is given in pseudocode in algorithm 1. As mentioned, line 6 becomes trivial for TOHTN planning and the loop from lines 7 to 16 is no loop, as there is always exactly one unconstrained task. Additionally, the location of our goal test can be moved around, depending on need. Performing the goal test upon popping a node is useful if we want to find optimal plans and our fringe data structure - and thus popping order - have a notion of node cost. Performing the goal test upon node creation allows us to terminate earlier.
Notable search-based planners are SHOP ([47]), HyperTensioN ([43]), PANDA ([35]) and our own planner CrowdHTN which will be presented in a later section 2.5.

Algorithm 1: Classical Progression Search for HTN as introduced in [35]

```
1  fringe ← {(s_0, tn_I, ϵ)}
2  while fringe ≠ ∅ do
3      n ← fringe.pop()
4      if n.isgoal then
5          return n
6      U ← n.unconstrainedNodes
7      for t ∈ U do
8          if isPrimitive(t) then
9              if isApplicable(t) then
10                 n' ← n.apply(t)
11                 fringe.add(n')
12         else
13             for m ∈ t.methods do
14                 n' ← n.decompose(t, m)
15                 fringe.add(n')
```

2.2.3 Lifted and Ground HTN Planning

As mentioned in section 2.1.1, HTN instances are normally given in a lifted representation and can be ground, i.e. all variables are filled with all possible parameter combinations. Specifying the instance in a lifted fashion is done for ease of use, as it is a more compact representation and allows domains to be reused for different problems [7].
The efficient grounding and pruning of HTN instances is an active field of research [51, 7]. While it is an easier problem than HTN planning itself, it can take exponential time and the ground instance may be exponential in size compared to the lifted instance [7].
Planners may choose to operate on either lifted or ground instances. A discussion on the trade-offs involved is found in [56]. We will reiterate the main advantage of each approach here. Grounded representations have more information available for pruning. As an example, while some parameter combinations in reductions may lead to a contradiction later on and can thus be pruned, not all such combinations may be invalid and thus the corresponding lifted method may not be prunable. Lifted representations on the other hand may be a lot more compact in practice. For example, our TOHTN instance may want us to choose any of N trucks to transport a package from A to B where in practice the choice might not matter. Whereas a grounded representation will have to instantiate all operators concerning a truck N times, a lifted operation will avoid this and only choose a truck ad-hoc.
The choice of grounded vs lifted representation is independent of the choice of planning algorithm. We have examples of grounded translation-based planners (totSat [5], Tree-REX [57]), lifted translation-based planners (Lilotane [56]) and also search-based planners that work on both lifted (HyperTensioN [43]) and ground representations (PANDA [35]).
Our own planner, CrowdHTN, walks a middle ground. It performs its search on a ground representation to allow detailed run time pruning according to the world state. However, it does not front-load the cost of a grounding procedure and instead grounds tasks and methods as needed.

2.2.4 Comparing the Techniques

Current SAT-based planners tend to explore the space of potential task networks in a layer-by-layer fashion, lending a BFS-like characteristic to their search. Progression search on the other hand is often implemented as DFS which may be further guided by heuristics. In the International Planning Competition (IPC) 2020, we saw a demonstration of these different characteristics [3]. HyperTensioN, the overall winner, is a search based planner. Its performance is hit-or-miss, i.e., plans are either found extremely quick or not at all. On four out of 24 domains HyperTensioN failed to find any plan at all. Lilotane, the runner up, is a SAT-based planner. While reaching a lower overall rating, it managed to find plans on a wider selection of domains. Within the IPC, planners were rated with the so-called agile metric. If a plan is found in less than one second, a score of 1 is awarded. If no plan is found within the time limit T, then a score of 0 is awarded. For run times $0 < t < T$, the score is set as $1 - \frac{\log t}{\log T}$. The agile metric focuses on fast run times over the overall number of problems solved. A different metric which can be used is the coverage. Here the run times are ignored and only the overall number of solved problems is measured. According to [56], Lilotane outperforms HyperTensioN in coverage. In addition, it excels at finding short plans.
In our previous research on parallel hierarchical planning we have shown that a portfolio of search-based and SAT-based planners can lead to improved run times and coverage overall, combining the strengths of both approaches [14].

2.3 Malleability

In this work, we follow the classification of [23] regarding parallel jobs. A job is **rigid** if it has a fixed number of required PEs which is hard-coded in the application. This number stays the same between runs. We call a job **moldable** if the number of PEs is variable and can be set at application start but remains fixed within any one run. An **evolving** job is one where the required number of PEs changes during execution and where these changes are initiated by the user. If the number of PEs changes during execution with the changes initiated externally, we call a job **malleable**. Malleability can be defined more generally as the ability to deal with changing resources, not only PEs [60]. In practise, we see that most jobs running on supercomputers follow the moldable model [17]. The moldable model is also supported by programming environments such as MPI [37].
Systems that utilize malleable jobs have been shown to be highly efficient [23]. They achieve this in multiple ways. First, they allow for efficient scheduling, as the scheduler can reevaluate and change previously made decisions [60]. This allows to resolve the conflict in scheduling between throughput and response latency, where low latencies come with the need to keep spare resources on hand instead of fully utilizing them [23], [37]. Second, they allow applications to utilize additional resources as they become available, leading to improved performance [37]. Lastly, [16] make the case that malleable applications are more fault-tolerant which is of increasing importance as applications become more parallel.
While malleable jobs are desirable from a scheduling and administration perspective, they are not popular with the user side, as they impose additional complications [23]. The effort required to make any one application malleable varies depending on the problem. In case the problem at hand is easily split into independent small subtasks, we can use a central work queue from which other PEs can receive new tasks as needed [23], [61]. This approach allows us to redistribute PEs to other jobs in between tasks. It is however limited by the central work queue which tends to be a bottleneck and makes strong assumptions about the structure of our problem. Alternatively, in data driven applications, we may have distributed data structures that are redistributed as the number of available PEs changes [23]. This is more complicated, though, and is an expensive operation which should not be performed too often. Lastly, [58] showed for SAT that a portfolio approach is easy to adapt to a malleable environment. Loosing single workers may slow down progress but completeness is preserved. Additionally, a periodic exchange of knowledge can benefit the remaining workers even as some solvers terminate. To sum it up, making an application malleable is highly dependent on the specific problem and only easy in cases that are trivial to parallelize.
Due to the inherent complexities, in practice there are only few malleable applications. This may change with the introduction of malleable SAT solvers such as Mallob [58] and Paracooba [29]. SAT forms an important building block in many applications. By presenting an easy to use interface while using malleability internally, a SAT solver may unlock the benefits of malleability for at least part of an application's work.

2.4 Parallel and Distributed Computing Techniques

Parallel and distributed computing techniques have been investigated for many years. While there has been little work on parallel hierarchical planning, fields adjacent to it have long been studied. In this section we will revisit our discussion on parallel graph search from [14] before extending it with a short summary of our previous work in parallel hierarchical planning.

2.4.1 Parallel Graph Search

Many hierarchical planners such as HyperTensioN [43], PANDA [35] and our own planner CrowdHTN [14] are based on some form of DFS. Parallel DFS has been a target for researchers for over 35 years [52, 39]. In hierarchical planning specifically, our graph is often so large as to be only implicitly defined. Load balancing under these conditions has been investigated by [53]. Load balancing is important, as parallelizing search may introduce new overheads. These overheads can be classified according to [25].

- Search overhead
- Synchronization overhead
- Communication overhead

Search overhead happens if a parallel search algorithm has to explore more nodes than it's sequential counterpart to find a goal. Synchronization overhead is what occurs when processors are idling, waiting for others to catch up and reach a synchronization point. Communication overhead is characterized by the time spent on communication. There are two main approaches to load balancing in parallel search. These are work sharing and work stealing. During work sharing, a PE with work actively distributes it to other PEs. A popular implementation of this is the hash-distributed A* algorithm [38]. When using work stealing, the responsibility instead lies with those PEs which do not have any work available. They subsequently search out PEs with work available and "steal" some of it. It was shown that work stealing has less communication overhead than work sharing [13].
In graph search a work package can be identified as a graph node as well as the attached subgraph. In hierarchical planning, a search node is identified by the set of open tasks and the world state. While the world state is bound in size, the set of open tasks may be arbitrarily large. As a result, avoiding communication overhead is a priority and CrowdHTN utilizes a work stealing approach.
In addition to this, [25] have shown that work stealing performance may degrade when duplicate search nodes may be encountered. This is important, as [31] have shown that duplicates play an important role in search-based hierarchical planning.

2.4.2 Parallel Hierarchical Planning

We already investigated parallel, moldable hierarchical planning before [14]. In our previous work, we created and evaluated three different planners:

- **S**election of **Hi**erarchical **P**lanners (SHiP), a portfolio planner
- Mallotane, an integration of Lilotane and Mallob
- CrowdHTN, a search-based planner

Out of these three, SHiP performed overall best. As a portfolio planner, it is however limited by the available number of sequential planners with sufficiently different performance characteristics. It's performance only scales meaningfully for up to four cores. Second, we have Mallotane. Here we took the sequential SAT-based planner Lilotane and integrated it with Mallob as a SAT backend. This allowed us to parallelize the SAT solving part of Lilotane's planning. Mallotane, too, has limited scalability as instantiating the task network and pruning unreachable tasks are still sequential. Third, we created the new progression search planner CrowdHTN. It uses randomized work stealing for load balancing and performs random DFS to find a plan. While it performed overall worse than SHiP and Mallotane it has the highest theoretical potential for scalability, as it is fully parallel. A more detailed overview of CrowdHTN will be given later in 2.5.

In our work on parallel TOHTN planners, we found that the typical characteristics of sequential planners extended to the parallel case. That is, CrowdHTN retained the hit-or-miss characteristic of search based planners where Mallotane proved to have more consistency in between runs. SHiP, which successfully emulates a virtual best solver of state of the art hierarchical planners may be considered state of the art in parallel hierarchical planning.
We are not aware of any work on malleable TOHTN planning. However, with the presence of malleable SAT solvers such as Mallob [54] and Paracooba [29] one could argue that Mallotane and any other SAT-based hierarchical planner can be made malleable. This is however limited by the fact that SAT-solving is only part of the work those planners perform which would limit scalability.

2.5 The CrowdHTN Planner

The CrowdHTN (**C**ooperative **r**andomized **w**ork stealing for **D**istributed **HTN**) planner was introduced in our previous work on parallel and distributed TOHTN planning [14]. It is implemented as a parallel state machine that uses work stealing for load balancing. According to the definition of [23] we introduced in section 2.3, CrowdHTN is a moldable program, as the number of workers is arbitrary but fixed during execution.
Each local worker of CrowdHTN owns it's own queue of search nodes and performs progression search on these nodes as explained in 2.2.2. Search nodes are enhanced with information about the reductions that were applied to reach them, allowing for plan reconstruction once a goal has been found. The basic CrowdHTN parallel search algorithm is shown in algorithm 2. The work step in line 2 corresponds to the local progression search.
In the initial state, only the root worker has any search nodes. All other workers start empty. To perform load balancing, randomized work stealing is used. The work package exchange is implemented as a three step protocol

(i) *work request*

(ii) *response*

(iii) *ack* (if response was positive)

Upon sending a positive *response*, a worker increments it's local tracker of outgoing work packages. When receiving an *ack*, the local tracker of outgoing work packages is decremented. This ensures that there is always at least one node that acknowledges the existence of any one search node. This enables CrowdHTN to determine a global UNPLAN. To do this each worker reports whether it has any work left. A worker reports true if it has a non-empty fringe or at least one outgoing work package.
This capability is especially helpful for small instances where it is plausible to explore the whole search space. As we saw in the earlier section on complexity (2.1.2), TOHTN planning is in D-EXPTIME making it infeasible to explore the whole search space on bigger instances.
To extract a work package, a CrowdHTN worker always takes the search node at the back end of the queue while the local search is performed at the front end. This serves as a heuristic to send off a work package that is as large as possible. Nodes at the back end will be higher up in the hierarchy with more left to explore. Additionally, this reduces overall communication volume, as nodes close to the initial search node will have fewer open tasks and a shorter sequence of preceding reductions, reducing their memory footprint.

Algorithm 2: The parallel CrowdHTN algorithm

```
1  while true do
2      work_step()
3      if fringe.empty and not has_active_work_request then
4          r ← random worker id
5          send work request(r)
6          has_active_work_request ← true
7      for (message, source) ∈ incoming messages do
8          if message is work request then
9              if fringe.has_work() then
10                 send positive work response(fringe.get_work(), source)
11                 outgoing work messages += 1
12             else
13                 send negative work response(source)
14         if message is work response then
15             if response is positive then
16                 fringe.add(work response)
17                 send work ack(source)
18             has_active_work_request ← false
19         if message is work ack then
20             outgoing work messages -= 1
```

2.6 The Mallob Load Balancer and SAT Solver

Mallob stands for both **Ma**lleable **Lo**ad **B**alancer and **M**ulti-tasking **A**gi**l**e **Lo**gic **B**lackbox [58]. It provides both a malleable scheduler which focuses on hard jobs with unknown processing times, where the jobs themselves can be small while still being hard [54] and a parallel SAT solving engine which is based on the massively parallel SAT solver HordeSat [2]. Mallob is able to solve jobs using a high degree of parallelism and also allows for the processing of many jobs in parallel. In 2020, the international SAT competition featured a cloud track for the first time, which Mallob has dominated in both 2020 and 2021 [24, 30].

What sets Mallob apart from other malleable schedulers is its flexibility and decentralized nature. Many other malleable schedulers rely on being able to predict run times in general and dependent on the number of assigned PEs specifically [11, 55], whereas Mallob does not need such information [54]. The decentralized nature of Mallob further avoids bottlenecks.

While Mallob is an excellent malleable SAT solver, it is not limited to this problem. Mallob also forms a general job scheduler and load balancer, having a simple programming interface which allows for the integration of other problems [1]. The following overview of how Mallob functions as a job scheduler is taken from [58].

As a scheduler Mallob is able to solve multiple jobs in parallel and adjusts the resources available per job on a dynamic basis. New jobs j can be introduced to Mallob at any time and are described by a number of attributes. Among those, each job has a fixed *priority* $p_j \in (0, 1)$. Additionally, each job has a variable resource *demand* $d_j \in \mathbb{N}$, describing the maximum number of PEs that job j is able to utilize efficiently. In the trivial case, assuming jobs only happen

[1] https://github.com/domschrei/mallob/

one after the other, each job can simply set d_j to the total number of available PEs. When it comes to the total number of active jobs at any one time, Mallob assumes that their number is lower than the total number of PEs. This allows Mallob to assign each PE to only a single job at a time while still making progress on all active jobs and also guarantees that each job will have at least one PE assigned at all times. The total number of PEs assigned to a job is also called the job's *volume* v_j. The volume of each job is set proportional to $d_j p_j / \sum_{j'} d_{j'} p_{j'}$, i.e., proportional to the product of a job's demand and priority.
Mallob follows the message passing paradigm which is realized through the MPI programming interface.

3 Theoretical Improvements of the CrowdHTN Planner

In this section we will introduce two improvements to CrowdHTN. First we changed the underlying implementation of our planner, which allows us to easily switch out the search algorithm we use. We introduce four search algorithms in 3.1 and discuss their expected performance characteristics. Afterwards we start a discussion about loop detection in TOHTN planning, the data structures which may help us and how we design our new distributed loop detection scheme in 3.2. We conclude with an extended discussion on planner completeness in 3.3, taking search algorithms, loop detection and restarts into account.

3.1 Search Algorithms Used in CrowdHTN

As we have seen in our overview of different TOHTN planning techniques, the choice of algorithm and their behavior can have a big impact on performance. We see this in the varying behavior of planners relying on SAT-solving, DFS and heuristic search respectively. For this reason we wanted to explore the behavior of different search algorithms when applied to CrowdHTN. As part of the re-engineering of CrowdHTN, we changed the implementation of the search algorithms to be based on a fringe. As mentioned in [35] we can simply switch out the underlying fringe data structure to emulate different search algorithms without making any changes to our core planner. Enabled by this change, we have implemented four search algorithms and will discuss them in the following section:

- Random DFS
- Random BFS
- Heuristic DFS
- A*-like search

3.1.1 Random Depth-First Search

Random DFS is the only search algorithm that was already present in the previous implementation of CrowdHTN. It is implemented using a Last-In-First-Out queue as our fringe. Resolving an abstract task may create multiple new search nodes. If multiple nodes are created, we randomize their order before insertion into the fringe. This is done to avoid any pathological cases a fixed order may induce. We do not expect any differences in behavior or performance compared to the previous implementation.

3.1.2 Random Breadth-First Search

Random BFS is the first new search algorithm that we implemented. It is done by using a First-In-First-Out queue, allowing us to explore all the potential task hierarchies layer by layer. The insertion order of new nodes is randomized as in the DFS. We do this as the number of search nodes per layer can be exponential in the depth. As such, the last layer may dominate the overall work and the order in which we explore it can have a large impact on performance. In general, we expect a higher memory footprint compared to DFS and assume that the planner will struggle with domains where plans are only found in deep layers or where the branching factor is very high as both will lead to a blowup in the size of our fringe and in the number of nodes we need to explore to find a plan. At the same time, we expect the performance of BFS to be more consistent than DFS, as the layer at which we find a plan stays fixed for any single instance.

Overall, we do not expect high performance of our BFS. It may however prove useful on some domains and help us understand and validate assumptions about the behavior of TOHTN problems.

3.1.3 Heuristic Search

Both random DFS and BFS are unguided and do not adapt the order of search node exploration to information contained in those nodes. Other planners, such as PANDA, use heuristics to guide their search. We will describe search heuristics and their use in PANDA according to [35] and then go over how we try and adapt the use of heuristics with the added constraints of malleability.

Heuristics in hierarchical planning in general and PANDA specifically The general idea of heuristic search is to explore our search space more intelligently. Heuristics achieve this by guiding the search to the most promising search nodes first. In TOHTN planning, our choices during search are restricted by both the hierarchy of tasks as well as the world state. As a result, the best heuristics should make use of both pieces of information for the best results.
One avenue to deriving heuristics for HTN planning is to adapt classical planning heuristics. This proves difficult as these heuristics do not know about the hierarchy and may assume a state-based goal which HTN planning often does not have. To avoid these issues, PANDA instead adapts the hierarchical problem to match the heuristics. PANDA computes a classical planning problem which is a relaxation of the HTN problem at hand. Then a solution to this relaxed model is approximated with the help of classical planning heuristics and the result is used to guide the initial HTN planning procedure. The computation of the classical model is possible in polynomial time and only done fully in the beginning, afterwards the model is only updated for the current state of planning. As a result, the heuristic takes into account both hierarchy and world state with little overhead during search.

Problems with the PANDA heuristic for malleable HTN planning While PANDA has managed to make great use of heuristics in HTN planning, we cannot simply adopt the same heuristics for malleable CrowdHTN. The reason for this lies in the assumption of PANDA that a ground problem instance is already available. Grounding is an expensive operation, though, as discussed in [7]. A full grounding may be exponential in size compared to the input and run times of grounding procedures are accordingly high.
While a grounding is already available in PANDA, CrowdHTN does not perform explicit grounding before planning. In a malleable environment without shared memory we can expect this grounding to take place every time a PE is added to a job, adding a high startup cost. A short-lived worker may be interrupted while still grounding, never starting the actual search. This would interfere with the efficient usage of available resources. For this reason we have decided against using the PANDA heuristics in CrowdHTN and instead tried to design a simpler heuristic to be used in malleable TOHTN planning.

A heuristic for malleable HTN planning To counter the startup cost of the PANDA heuristic, we have devised a simpler heuristic which is cheap to precompute and can easy to use in malleable planning. The goals are to have little startup overhead and to retain the efficient evaluation at each search step. As discussed in the previous paragraphs, this limits any precomputation to the lifted instance. We hope to still find performance gains on at least some problem instances.

As heuristic value, we use a lower bound on the number of reductions we still need to perform to fully resolve our list of open tasks. When computing this lower bound we ignore preconditions and effects, searching the shortest possible way through the hierarchy. We precompute this value for each task as described in algorithm 3. The heuristic value for each action is initialized to zero. The heuristic value for each abstract task is initially unknown. In each step we loop over all tasks t. For each method m of task t we check the heuristic value of all subtasks. The heuristic value of a method is set to the sum of the heuristic over all subtasks plus one. For each task we choose the minimum value over all corresponding methods.
Once there are no more changes in the mapping of tasks to heuristic values, we stop. Any task which at this point does not have an assigned value is not resolvable at all and can be pruned. To visualize the computation of our heuristic, we provide an example TOHTN domain in figure 2 and table 1 shows how the heuristic is computed on this domain.
Computing the final heuristic will take at most as many iterations as there are compound tasks. To show this we look at our hierarchical planning problem as graph. The tasks and methods form the vertices. We get edges from abstract tasks to all applicable methods and from methods to all their subtasks. Actions do not have any outgoing edges. During computation, the heuristic values are initialized at the actions and propagated and update throughout the graph. As tasks and methods alternate, any cycle contains at least one method and as such propagating the heuristic through a cycle would strictly increase it.
To evaluate our heuristic while planning, we now need to look at the whole sequence of open tasks and calculate the sum of our heuristic over those tasks. While the naive approach gives us linear run time in the number of open tasks, we can stretch the computation over task instantiation and reuse parts of it to perform heuristic evaluation in

$$\mathcal{O}\left(\max\left\{\#\text{subtasks of } m | m \in methods\right\}\right)$$

at run time. Details can be found in 5.5 on efficient hashing of the open tasks, the technique also applies to the heuristic computation. Additionally, while we only use this heuristic to guide TOHTN planning, it ignores any orderings between open tasks. As such, it can be applied to HTN planning as well.

Algorithm 3: GBFS heuristic calculation

```
1 task depths ← {(t_c, 0)|t_c ∈ concrete tasks}
2 while task depths changed do
3     for t_c ∈ compound tasks do
4         reduction depths = ∅
5         for r ∈ reductions for t_c do
6             if depths of all subtasks are known then
7                 reduction depths = reduction depths ∪1 + ∑{d|d is depth of a subtask of r}
8         if reduction depths ≠ ∅ then
9             task depths = task depths ∪{(t_c, min(reduction depths))}
```

Using the new heuristic in TOHTN planning With the new heuristic presented in the previous paragraph, we implemented two new search algorithms for CrowdHTN, those being a heuristic DFS and an A*-like search.
The implementation of DFS used so far performs the search in a uniformly random order.

Figure 2: Example TOHTN domain to demonstrate our heuristic

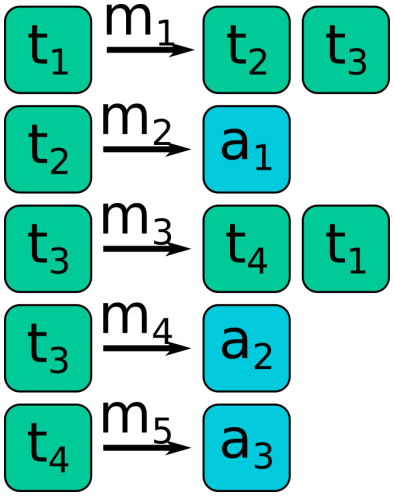


Table 1: Example computation of our TOHTN heuristic for the domain in figure 2. Changing values are bold.

	Iteration			
task	0	1	2	3
a_1	**0**	0	0	0
a_2	**0**	0	0	0
a_3	**0**	0	0	0
t_1			**3**	3
t_2		**1**	1	1
t_3		**1**	1	1
t_4		**1**	1	1

Without knowing anything about the domain, this is a reasonable choice. While it is far from optimal search, it also avoids any pathological cases that may arise from a fixed order of exploration. As an alternative to this random order, we used the heuristic to guide our DFS. We have to note that this is not necessarily fully deterministic. Randomness comes into play both when two reductions lead to search nodes with the same heuristic score - which happens e.g. for different instantiations of the same method - and when performing work stealing in a parallel setting.
In addition to the guided DFS, we also implemented an A*-like search where a node's value is the sum of the heuristic value and the number reductions applied to reach the node. We differ from A* in that we terminate the search as soon as a plan has been found instead of continuing on the search for an optimal plan. Our goal is for the heuristic to guide us towards a plan while giving weight to the number of applied reductions forces us to turn back and explore other parts of the search space without getting lost in endless loops due to pathological cases in our heuristic.

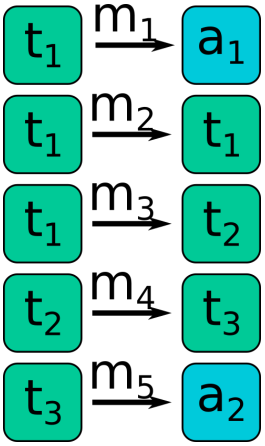
3.1.4 Completeness of different Search Algorithms

In the previous paragraphs we discussed a number of different search algorithms that we implemented for CrowdHTN and how we expect them to affect the performance characteristics of our planner. The search algorithm has a more fundamental impact than that, however, and may affect the completeness as well. In this section we want to give a short overview over the completeness of each of the algorithms. A summary is found in table 2. Note that this discussion is only about the algorithms without any modifications. In section 3.2 we discuss both loop detection and a restart mechanism and in section 3.3 we have a more detailed discussion about the completeness of different planners as well as the completeness of progression search with these additions.
DFS is not complete as it may enter an endless loop and, even if it still explores side-tracks from this loop, will never be able to backtrack out of the loop, cutting of parts of the search space. There is however always a chance to find a plan if it exists. I.e., there is a non-zero chance that the random choices all happen to be done correctly, the loop is never entered and a plan is found.
BFS on the other hand is trivially complete. While the exploration order within each layer is random we can provide an upper bound on the number of search steps required to explore a given search node n on layer i. One such bound is the sum of all layer sizes from 0 up to and including i.
Next in our list is heuristic DFS. Similar to random DFS it may run into an endless loop. Compared to DFS, however, heuristic DFS may do so in a deterministic fashion if the domain triggers a pathological case in the heuristic. One example domain which would trigger such a case in our heuristic is visualized in figure 3. In this instance our heuristic assigns the value 1 to m_1, 2 to m_2 and 3 to m_3. If the preconditions of m_1 are not fulfilled, heuristic DFS will first try to resolve t_1 via m_1, fail, then use m_2 with the goal to try m_1 again afterwards. This will fail again and the loop is repeated indefinitely. While m_3, m_4, m_5 may provide a path out, they will never be used.
Lastly, we implemented A*-like search. While it does reuse the same heuristic, this algorithm achieves completeness by also valuing the number of applied methods so far. Let n be a node with heuristic value h and r previously applied reductions to reach n. Then n is guaranteed to be explored once all nodes with at most $h+r$ previously applied reductions have been explored.
All of this discussion so far has assumed sequential planners. The implications do hold for parallel search, too. For a planner using at most n PEs we may always provide an instance with

Table 2: Completeness of the different search algorithms in CrowdHTN

Algorithm	Completeness
Random DFS	Not complete. Positive probability to find a plan if it exists
Random BFS	Complete
Heuristic DFS	Incomplete
A*-like	Complete

Figure 3: A pathological case in our new HTN heuristic



n ways to enter an infinite recursion.

3.2 Loop Detection

In recent years it has become clear that the recursive nature of HTN instances poses its own set of challenges to planners. As a result, mechanisms to perform loop detection have become an active area of research with both HyperTensioN [43] and PANDA [31] exploring it. In this section we will discuss loop detection specifically in the context of parallel and distributed hierarchical planning. We want to address this, as research indicates that the performance of randomized work stealing may suffer on problems where the detection of duplicate states is important. [25] and [31] have shown that duplicate search nodes play an important role in hierarchical planning.
We start out with a discussion of loop detection techniques in other planners such as PANDA and HyperTensioN in section 3.2.1. This is followed by a short overview of how CrowdHTN specifically differs and how this changes our base assumptions in section 3.2.2. Afterwards we first explore loop detection based on hash sets in section 3.2.3. We conclude by exploring how approximate-membership-query (AMQ) data structures can be used in loop detection, how this affects completeness of the progression search algorithm and present our design for a distributed and global loop detection mechanism in section 3.2.4.

3.2.1 Loop Detection in Other HTN Planners

Loop detection in HTN planning is a recent phenomenon and was introduced in 2020 by the HyperTensioN planner with the so-called 'Dejavu' technique [43]. Dejavu works by extending the planning problem, introducing primitive tasks and predicates that track and identify when a particular recursive compound task is decomposed. These new primitive tasks are invisible to the user. Information about recursive tasks is stored externally to the search as to not loose it

during backtracking. Dejavu comes with performance advantages and protects against infinite loops. However, as Dejavu only concerns itself with information about the task network but ignores the world state it may have false positives. This was also noted by [31] and means that HyperTensioN is not complete. [31] further nodes that the loop detection is limited in that it only finds loops in a single search path but cannot detect if multiple paths lead to equivalent states.
In response to HyperTensioN, the PANDA planner introduced its own loop detection in [31]. Similar to HyperTensioN, PANDA keeps the loop detection information in a separate list of visited states, $\mathcal{V}$. Search nodes (s, tn) of world state s and task network tn, are only added to the fringe if they are not contained in $\mathcal{V}$. To reduce the number of comparisons required to determine whether $(tn, s) \in \mathcal{V}$, $\mathcal{V}$ is separated into buckets according to a hash of s. In the sub-case of TOHTN planning, both an exact comparison of the sequence of open tasks as well as an order-independent hash of the open tasks called *taskhash* are used. Similar to HyperTensioN, using a hash to identify equal task networks can lead to false positives and an incomplete planner. The loop detection in PANDA improves upon the one in HyperTensioN insofar as it is not just able to detect loops but also recognizes when equivalent search nodes are reached on independent paths.

3.2.2 Assumptions in Loop Detection for CrowdHTN

To design the loop detection in CrowdHTN, we have both simplifying and complicating assumptions that we will discuss here.
While both PANDA and HyperTensioN are HTN planners, CrowdHTN concerns itself only with TOHTN planning. As a result, the remaining task network can be represented as a sequence of open tasks with the ordering constraints implicit in how the sequence is stored.
As tasks of equivalent task networks are always in the same order, we can incorporate that order into our hash of tn to reduce the number of collisions compared to PANDA's *taskhash*. This will increase performance where we fall back to comparisons in case of collisions and reduce our false positive rate in case we forgo comparisons for performance reasons.
Both PANDA and HyperTensioN are sequential planners whereas CrowdHTN is highly parallel. This adds an additional design constraint to our loop detection. If we want to efficiently share information about visited states, directly sharing search nodes would be infeasible due to their size. If we perform loop detection only locally, we expect to suffer from decreased performance as the degree of parallelism increases. I.e., if a search node exists multiple times we may encounter it on different PEs, not realizing that it is a duplicate.

3.2.3 Hash Set Based Loop Detection

One simple way to perform loop detection which is also used in PANDA ([31]) is to use a hash set of visited states. The implementation in Crowd is slightly different from PANDA in that we use one combined hash for both world state and open tasks. Other than PANDA, CrowdHTN does incorporate the order of tasks into the hash, which should reduce collisions and makes the two levels of hashing less needed.
Using hashes combined with a full comparison provides us a perfect loop detection, i.e., neither false positives nor false negatives exist. This makes it a useful technique to benchmark other loop detection methods. However, both in the sequential and in the distributed case this technique suffers from performance problems.
In case of hash collisions, we have to fall back to a full comparison of world state s and open tasks tn. While s is bound in size by the total number of predicates, the size of tn is effectively

unbound, making this an expensive operation. Additionally, we have to keep both s and tn around for all nodes ever encountered, increasing the memory footprint of our planner.
The hash function we do use is a combination of the hashes for the task network and the world state. The sequence of open tasks can be hashed as-is in a deterministic fashion by iterating over it from beginning to end. For the world state as a set of predicates we do not have a fixed order. We solve this by combining hash values of predicates by adding their squares, a commutative operation.

3.2.4 Approximate and Distributed Loop Detection

In the preceding sections we have always made the assumption that our loop detection mechanism needs to be perfect, i.e., it needs to avoid both false positives and false negatives. Some hierarchical planners do not share this assumption and both HyperTensioN and PANDA have configurations that allow for false positives. In the following paragraphs, we will also permit false positives to occur and explore the implications.
We start out by introducing the concept of AMQ data structures with a specific focus on bloom filters and how to use the scalable bloom filter in loop detection. Once this is done, we turn to the problem of false positives and introduce a restart mechanism that guarantees that, given enough time, we are able to reach any search node. The section is concluded by us using the special properties of bloom filters to design a distributed loop detection mechanism which allows for efficient information sharing between PEs.

Approximate membership queries AMQ data structures are used as a memory efficient representation of sets that allow for a false positive rate during membership queries to be able to gain memory efficiency [8]. They were introduced with the bloom filter in 1970 [12]. Since then both variations of bloom filters, such as counting bloom filters [22], and other AMQ data structures have been introduced, among them quotient filters [8] and cuckoo filters [21].
As the guarantees of the bloom filter are sufficient for us, we will now focus on this specific data structure using the definition of [12]. A bloom filter is defined by three numbers, the number of bits in the filter m, initially all set to zero, the number of hash functions k, each producing hashes in the range $0, \ldots, m-1$, and the number of elements already present in the filter n. To insert a new element, we use the hash functions to compute k hashes and use them as indices into our bit vector, setting the corresponding bits to 1. Similarly, to query for membership we check whether the corresponding k locations all contain a 1. This leads to highly efficient insertion and membership queries in time $\mathcal{O}(k \cdot h)$ where h is the time required to hash an element.
One limitation of bloom filters is the fact that they do not support element deletion. We cannot simply set a bit to zero, as there may be more than one element requiring it to be 1. To deal with this limitation the concept of a counting bloom filter was proposed [22]. Instead of a single bit per element, multiple bits are used per index to store a counter tracking the number of elements belonging to the index.
Given m, n and k we can compute the probability of encountering a false positive. A detailed discussion of this can be found in [15]. The main result is that the probability for any bit to contain a 1 is

$$p' = 1 - \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}}$$

This gives us an overall probability of false positives of

$$p = p'^k = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

We see that the probability of encountering false positives steadily rises as the number of elements in the filter grows, reaching 1 as all bits are set to 1. As a result, bloom filters in their original form are best suited for static sets of known size and may necessitate setting m conservatively high to guarantee a low rate of false positives. To fix this and guarantee an upper bound on p even in dynamic sets, the concept of scalable bloom filters was introduced [63]. A scalable bloom filter builds a hierarchy of bloom filters, each new bloom filter being twice the size of the previously largest one. To insert an element into a scalable bloom filter, we first check the false positive probability of its largest sub-filter. If it is still under our decided bound, we simply insert the element into the largest filter. If inserting the new element would raise the false positive rate beyond the limit, we add a new filter, twice as large as the previously largest, and insert the element there. Membership queries now have to check all levels of the filter. While there is a performance overhead, only a linear number levels is required to store an exponential number of elements.
Given bloom filters and their variations we have decided on the use of a scalable bloom filter for loop detection in CrowdHTN. While our set of search nodes is theoretically limited in size, in practice the search space is prohibitively large and unlikely to be fully explored. Using a single bloom filter of fixed size would lead to a high size requirement to ensure low false positive rates. At the same time, once inserted we want to forever keep an element in our set of known nodes and do not require deletion, allowing us to forgo mechanisms like the counting bloom filter. To get k hash functions, we reuse the hash function we already use in the hash set based loop detection, varying the seed to generate different hashes.

Completeness in the face of false positives While using approximate membership queries as described in the previous paragraph gives us a number of advantages, it also introduces a new set of challenges. Among these is are false positives. As a result, we can expect to loose parts of our search space. More specifically, if we perform progression search using bloom filters and $n = n_1, \ldots, n_l$ is a path of length l from our initial search node to a goal search node, then for any n_i in n, the search nodes $n_1, \ldots, n_{i-1}$ may collectively set the k hashes associated with n_i, filtering it out. This may end up turning a TOHTN problem unsolvable for us even though a plan exists. As a result our planner, if otherwise unchanged, will no longer be complete.
We will now take a closer look at the probabilities involved. Assuming we use a scalable bloom filter with maximum false positive rate $0 \leq p < 1$ and hash functions which map search nodes to uniformly independent values and n is a shortest path. Then

$$q = 1 - (1 - p)^l < 1$$

is an upper bound on the probability that we are unable to solve the problem even though a solution exists.
It follows that

$$\lim_{u \to \infty} q^u \to 0$$

That is, if we keep re-running our search with new, independent hash functions we regain completeness. It is critical to use independent hash functions between runs to ensure that false positives in different runs are not correlated with each other. In the next step we need to determine when to re-run our planner. There are three main constraints.

(i) As the TOHTN instance may be recursive, the search space may be infinite

(ii) The number of restarts needs to be unbounded as run time goes to infinity

(iii) As a plan may be arbitrarily long, the number of runs with run time at least u needs to be infinite

Constraint one implies that we cannot simply wait until we explore the whole search space before restarting. Instead we will base our restarts on total run time so far. To fulfill constraints two and three, we perform a check every second where after t seconds we perform a restart with probability $\frac{1}{t}$.
For the expected number of restarts we get $\sum_{t=1}^{\infty} \frac{1}{t}$. This is the harmonic series and diverges, giving us the required unbounded number of restarts. As t grows, the probability of restarting decreases, allowing for increasingly long runs, fulfilling the third constraint.
This mechanism allows us to restore completeness to our planner while utilizing AMQs. Restarts may prove to have additional benefits to planner performance, as DFS-based planners tend to be hit-or-miss and restarts increase the number of opportunities for a hit. We do note that approximate loop detection does come at the cost of no longer being able to detect UNPLAN, as we can never guarantee that we explored the full search space. In practice, we do not expect this to matter as the search space of TOHTN problems tends to be too big to feasibly fully explore.

Global Loop Detection In the section 3.2.2 we already mentioned that loop detection in distributed hierarchical planning comes with unique problems. Specifically, current loop detection techniques do not include ways to efficiently share the visited states between PEs. As a result, a search node is only fully filtered out once each PE has encountered it at least once. This leads to a degradation in performance as the degree of parallelism increases. To address this issue, we will start with a short discussion on how previous loop detection mechanisms are hard to adapt for the distributed case and then show how we implement distributed loop detection on the basis of bloom filters.
As mentioned in section 3.2.3 on hash set based loop detection, it suffers from a high memory footprint as we keep whole search nodes around. This problem extends to the distributed case, as we would now have to communicate whole search nodes leading to a large overhead for encoding, sending and decoding. Even if we assume the communication overhead to be low enough, we run into additional problems. Inserting n elements into a hashset takes $\mathcal{O}(n)$ time even without hash collisions. The higher the number of PEs, the more time would be spent on inserting search nodes received from other PEs which would either block us from performing search for large amounts of time or introduce synchronization problems. As a result, we have decided that it is infeasible to extend hash set based loop detection to the distributed case.
Compared to hash sets, bloom filters offer a number of advantages for distributed loop detection. First, bloom filters offer a more compact representation of the already encountered nodes which leads to a lower communication overhead. Second, as the filter itself is stored as a simple bit vector, we have negligible overhead regarding encoding and decoding for communication. Thirdly, we can efficiently merge two bloom filters by performing a simple bitwise or operation of the bit vectors. In a combined filter, we get a conservative upper bound for the total number of contained elements by summing the number of elements of both filters. This guarantees that our maximum rate of false positives is not exceeded.
To integrate bloom filters into our distributed loop detection, we also need to address the question of what to do in case the global filter gets filled up, i.e. it's false positive rate reaches our set limit. For local loop detection we introduced scalable bloom filters. This is a problem as we now communicate whole sets of search nodes whereas locally we introduce new search nodes into the filter one by one, increasing the size at exactly the right moment. As a result, we face the choice of increasing the size early, throwing away some information or loosing our guarantees regarding false positives. Similarly, we face the problem where different PEs may disagree about the current maximum size of the scalable bloom filter, putting more information in a smaller filter that will be thrown away by other PEs.

To deal with this problem, we induce a restart in our search once the global filter is full. As the restart mechanism is already present due to the need to deal with false positives this is an easy adaption. To limit the number of needed restarts and once again allow arbitrarily long runs, we double the size of our global filter with each restart. Additionally, we limit the amount of information present in our filter to further reduce the number of restarts we need. We do this by only putting 'important' search nodes into our filter. Our heuristic to determine search node importance is to put a node into our global filter if it is present in our local filter and encountered again. Other heuristics are possible but beyond the scope of this thesis.

3.3 Discussion of Planner Completeness

In section 3.1 we already did a short discussion on the impact of different search algorithms on the overall completeness of progression search. The current section will start with a short recap of our findings, expand them to other planners and will then do an expanded discussion that takes factors like loop detection and restarts into account.
Before we dive into the more detailed discussion we want to note that we have seen in section 2.1.2 that there is an upper bound to task network depth where, if a plan exists at all, it can be found before that depth. By limiting our planning to task network expansions with lower depth, we can trivially achieve completeness. This is however of little practical use as this depth bound is exponential in the problem size. As a result, we can expect to run out of memory before hitting this bound. For this reason we do not make use of this bound and as far as we know no other planner does. We will now resume a more practical discussion of planner completeness.
As previously noted, we can split our search algorithms into three main groups:

- Algorithms that are complete (BFS, A*-like search)
- Algorithms with a chance but no guarantee to find a plan (DFS)
- Algorithms which for some domains will never find a plan (heuristic DFS with pathological cases)

Completeness in other planners So far we have only classified the different search algorithms present in CrowdHTN. For now we will take a look at other planners, starting with translation-based planners totSAT ([5]), Tree-REX ([57]) and Lilotane ([56]). As we have noted in the discussion on planning algorithms in section 2.2.1, all three of these planners are based on SAT. Additionally, they all explore the set of potential expansions of the task hierarchy in a layer-by-layer fashion, leading to a BFS-like characteristic in their behavior. As a result, these planners are complete.
In contrast to this, we have the space of search-based planners, starting with HyperTensioN [43]. For HyperTensioN, the authors themselves note that their inbuilt loop detection mechanism suffers from false positives with no mechanism to mitigate them [43]. It follows that their planner is not complete. If we disabled the loop detection in HyperTensioN we would be left with a planner performing DFS, which would put it in the category of planners which are not complete but still have a chance to solve any instance.
PANDA on the other hand is a planner based on heuristic progression search that offers a number of configuration options for both search and loop detection. Regarding search, PANDA offers both a pure heuristic DFS and a weighted A* search taking into account the previous path [35]. For loop detection PANDA offers hashing based mechanisms both with and without a fallback to full search node comparison [31]. Completeness of the planner varies depending on the chosen configuration. If loop detection is configured to allow for false positives, we expect

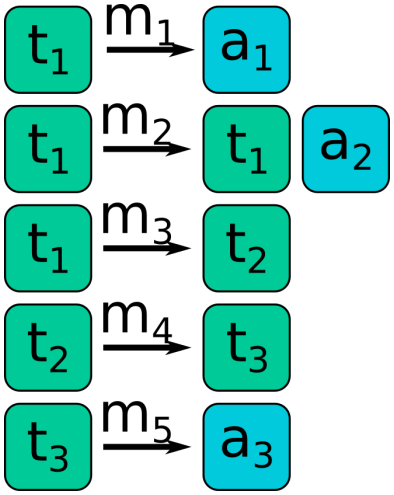
PANDA to not be complete regardless of search algorithm, as there is no mechanism in place to mitigate their effect. If a loop detection mechanism is chosen which does not have false positives, we expect PANDA to be complete if weighted A* with a weight $w > 0$ is used, as this introduces a BFS-like behavior into the search. This leaves pure heuristic DFS combined with loop detection without false positives. Due to the complex nature of the PANDA heuristic we were unable to construct any pathological case which leads PANDA into an infinite recursion. As heuristics are inherently limited we do assume that such cases exist. We will explore this case and the similar case in CrowdHTN in the following paragraph.

Loop detection and completeness As we have seen, heuristic search on its own may increase planner performance but comes at the cost of completeness. Random DFS is able to find any plan but may still get stuck in endless loops. We will now explore the implications of combining heuristic search with loop detection but without restarts to see how this changes the overall situation. In this paragraph we are only interested in loop detection mechanisms that do not suffer from false positives as, without restarts, this automatically disqualifies a planner from being complete.
In general, loops are only a problem in hierarchical planning if there exists at least one recursive task. If no such task exists, there exist only a finite and usually small number of possible task network expansions such that we can easily search the full search space. If we do have a recursive task, we can further classify our instances according to how hard it is to deal with. We identify three categories:

- Tasks which recurse into themselves with no change in the world state
- Tasks which recurse into themselves with changes in the world state
- Tasks which recurse into themselves while adding more tasks afterwards

For general HTN planning, a task recursing into itself also implies that the ordering constraints of the new open tasks are a superset of the old ordering constraints.
The first case is the easiest to detect and fix. If a task recurses into only itself we do not get any changes to the open tasks. As a result, search nodes before and after this recursion are equivalent. They will be detected by loop detection as it is used in PANDA and CrowdHTN and the search will be guided into another direction.
In the second case we have to perform additional work before a loop can be detected. As the set of open tasks stays the same and the world state changes, we do not immediately get equivalent search nodes. However, in an HTN instance with predicates $\mathcal{Q}$, there are only $2^{|\mathcal{Q}|}$ possible world states. We can easily see that we will recurse at most $2^{|\mathcal{Q}|}$ times before our loop detection activates and we backtrack. In practice we can often obtain a smaller upper bound on the possible number of recursions by looking only at the predicates which occur as effects in the resolution of any tasks present in the recursion. We see that, while less efficient, our known loop detection mechanisms correctly deal with this case.
This leaves us with the third case, where a task does not directly recurse into itself but where the resolution of task t gives rise to a new instance of t as well as additional tasks $t_1, \ldots, t_k$ which are restricted to be resolved after t. As a result, once we re-encounter t our open task set has changed. While we were able to limit the number of possible world states in previous case, this is not possible here, as the number of open tasks is unbounded. More specifically, loop detection as used in PANDA and CrowdHTN is unable to handle this case.
Figure 4 provides an example of an instance which would guide our proposed heuristic into a recursion while not creating loops. Our heuristic would assign the values 1 to m_1, 2 to m_2 and 3 to m_3. If the preconditions for m_1 are not fulfilled we would then apply m_2, creating a unique set of open tasks and then repeat application of m_2 indefinitely.

Figure 4: Pathological instance for our proposed heuristic that is not caught by loop detection



Restarts and completeness Together with AMQ based loop detection, we introduced a restart mechanism in 3.3. We will now go over the use of restarts to achieve completeness for random DFS.
To show that restarts help us to achieve completeness for random DFS we can use a similar argument as we did for the loop detection. For any path in our search graph, random DFS gives us a probability $p > 0$ to take this path. As the number of restarts we perform goes to infinity, the probability to take any fixed path at least once goes to 1. We are under the same constraints as previously, needing both an unbounded number of restarts and an unbounded number of runs of at least length u for any u. The second constraint is needed so that we do have the time to fully explore a path once we take it. Our restart mechanism fulfills both constraints, restarting with probability $\frac{1}{t}$ at second t. It follows that random DFS with restarts is complete.

Conclusion In this section we have taken a look at completeness in hierarchical planners and how loop detection and restarts can help us to achieve it. We have shown that completeness is highly dependent on the specific search behavior with BFS-like behavior being trivially complete. In addition we show that loop detection, while helpful, is not able to solve the problem for some instances. Introducing our restart mechanism, we show that it can turn random DFS into a complete algorithm. This does not extend to heuristic DFS if the heuristic exhibits pathological behavior as the heuristic will always guide the search back into the same recursion.

4 A Malleable TOHTN Planner

The goal of this section is to describe how we adapt it to be a malleable TOHTN planner by integrating it with Mallob, preserving both the completeness and scalability of CrowdHTN in the process. Before we get into the details, let's recall that CrowdHTN is already a moldable program according to the definition introduced in section 2.3, i.e., it may utilize any number of PEs as long as that number stays fixed during the run. We will now introduce a design that extends the parallel capabilities to achieve malleability. For this we need to address three main concerns.

- Distributing the job information
- Integrating new PEs into a running job
- Dealing with PEs leaving the job while it runs

In the following sections we will address these problems in this order. Both distributing the job information and integrating new workers do not pose significant problems. Most time will be spent on the handling of disappearing PEs.
Due to the fact that we specifically integrate CrowdHTN with Mallob, we will in some parts refer to implementation details of Mallob. Namely that the v_j workers currently assigned to a job are internally organized as a binary tree, s.t. all levels except the last one of the tree are always full and the last level is filled from left to right. If a PE is taken away from a job, the associated data is not immediately deleted. Instead, a small and constant number of previous jobs is kept around. When the volume v_j grows again, PEs containing a suspended worker of the same job are preferred to increase efficiency.

4.1 Distributing Jobs

When a PE is assigned to a job, it needs to obtain a description of this job. In case of TOHTN planning, the choice is mostly between a lifted or ground TOHTN instance. As we have seen, a ground instance may be up to exponential in size [7]. Encoding and communicating such a ground instance would take up much time, which is why we decided to communicate our problem as a lifted instance.
With the lifted instance, we choose to simply take the textual hddl input [33] and send it as-is. While this does incur the overhead of locally parsing the instance on each PE, communicating the parsed instance would involve re-encoding and effectively re-parsing it locally, too.
In malleable TOHTN planning there is a more general trade-off involved when it comes to precomputation. While parsing the instance is unavoidable, we can choose whether we want to spend time grounding and pruning our instance. It has been shown that grounding and pruning improve the planning performance [7] and allow for the computation of complex and good heuristics [35], but grounding, pruning and other precomputations are expensive operations themselves. As a result, a PE which is only assigned to our job for a short time may never perform any actual planning work before it is reassigned to the next job. For this reason, CrowdHTN takes an alternative path. The TOHTN instance is kept in lifted form. Instantiation is only performed as needed to explore the current search node. This allows CrowdHTN to start working immediately to utilize even short-lived PEs in a highly malleable environment.

4.2 Integrating New PEs Into Malleable CrowdHTN

To integrate a new PE into a running TOHTN job, it needs both the general job description and part of the actual work to handle. In the previous section we explained how the job description

is obtained, now we will focus on the work itself.
The efficient integration of new PEs into a running job is where work stealing shows it's strength. For work stealing, there is no functional difference between a PE which has locally run out of work and a new PE which has the job description but no work yet. Both will message other PEs at random to receive a new work package with no special handling required. As a result, a new PE can perform at full efficiency almost immediately, allowing our job to utilize resources as soon as they become available.

4.3 Handling PEs Leaving at Run Time

The last challenge in designing a malleable CrowdHTN is the fact that PEs may disappear at any time. This represents a potential loss of information. The information loss presents itself in two ways. First, the loss of the local search fringe, if we do not communicate it to another PE, and second messages which may be lost in transit as their receiver no longer belongs to the same job. To deal with this, Mallob does allow us to detect locally when a PE is taken away from a job and additionally provides a message return mechanism. We will present our solutions to both cases with a focus on preserving the completeness property of CrowdHTN.

4.3.1 Handling the Local Fringe

When a local PE is unassigned from a job, we will loose the local search fringe. As Mallob signals a PE when it is unassigned from a job, we are however free to encode parts or all of the fringe and communicate them to another PE. This leaves us with a number of choices where we may trade-off data loss versus efficiency and communication. On this axis we discuss three choices

- Encode and redistribute the whole local fringe
- Communicate the root of the local search space
- Communicate nothing, loose the local fringe

Encoding and redistributing the whole fringe Encoding and sending off the local fringe to another PE is, in a way, the easiest operation. No information is lost, preserving completeness in our planner. It does, however, come with two disadvantages. First, the local fringe may be arbitrarily large, especially considering that TOHTN planning is EXPSPACE-hard as seen in section 2.1.2. Encoding and communicating a large fringe is a very expensive operation which would increase the time from Mallob telling a PE to suspend itself until the PE actually is free for the next job. Second, receiving a large fringe would strain the memory of the receiving PE which may lead to dropping parts of it anyways to avoid crashes.

Communicating the root of the local search space Instead of communicating the whole local fringe, we can simply encode the root node the local fringe emerged from. In a way, this search node represents a very efficient encoding of the local search space. As we would only communicate a single search node, this would be more efficient and could reuse the facilities we already have in place for work stealing. Similar to encoding the whole fringe, communicating only the root search node would lead to no information loss, preserving completeness.
While this approach is very efficient and avoids loss of information, it does suffer from duplicate work. As we loose the local fringe, we will have to re-explore it again. Additionally, other nodes may have received parts of the local search space via work stealing. These nodes will be re-encountered leading to further duplication. In this way, we would trade-off local performance

for encoding and communication against global performance through duplicating parts of our search.
Implementing global loop detection as we propose in section 3.2 further complicates matters. Upon suspension of a PE, we can leave the global loop detection unaffected. This might lead to some losses in our search space as nodes will not be re-explored but may also help the search on our other workers which can still profit from being aware of common loops and prominent duplicate search nodes. Alternatively, we could remove the global loop detection data of our suspended PE from all other PEs. As more than one PE may have committed the same search node to global loop detection and due to the way bloom filters work, this brings its own problems. Namely, it would degrade global loop detection performance as we may delete more search nodes from our filter than strictly necessary.

Communicate nothing Our third option in dealing with disappearing workers is to accept the loss of information and communicate nothing to the remaining PEs. This comes at the cost of loosing information while being easy to implement and allowing for immediate reassignment of PEs and avoiding any duplication of work.
This leaves us with the problem of information loss. To deal with this, we can revisit the restart mechanism we introduced to deal with a similar problem in probabilistic loop detection. There we argued that correctly designed restarts would allow our planner to be complete even when using a loop detection mechanism suffering from false positives. For this we made no assumptions besides the false positive rate being less than 1. As Mallob guarantees that we will always have at least one PE, never loosing all information, we can model the loss of PEs and their local fringes as an extremely high false positive rate. From this it follows that restarts allow us to loose this local information while maintaining overall completeness.

Conclusion As we have seen, there are multiple approaches on how to handle a reduction in the number of available PEs while maintaining planner completeness. In our design, we decided to communicate the root node of our local search space to a random other PE, inserting it at the back end of the fringe. The other PE is chosen at random to avoid turning any single PE into a bottleneck. We choose this design, as it allows us to avoid the large overhead of communicating the whole fringe while not being overly reliant on restarts to achieve completeness. While restarts do offer us completeness from a theoretical perspective, times between restarts increase rapidly as run time increases. As such we are unwilling to loose large parts of our search space and instead allow for the risk of performing duplicate work.
In addition to this, we keep the global loop detection information unchanged. As we have seen in 3.3, loops in progression search may get very long. By keeping this information around we hope to save other PEs from re-exploring the same loop such that they can still profit from the work even after a PE has been taken away.

4.3.2 Handling Lost Messages

In the moldable version of CrowdHTN, we could make a fundamental assumption about all messages, namely that they were guaranteed to be delivered. In malleable CrowdHTN this is no longer possible. If PEs p_1, p_2 are both assigned to the same job, then p_1 may send a message to p_2 with p_2 being reassigned to a different job before the message arrives. Mallob deals with this by recognizing the message can no longer be handled and returning it to the sender. We go over the way we handle such return messages in our implementation chapter.
However, the changing assignments of PEs to jobs imply an additional problem. The return message may be lost as well, if the original sender gets assigned to a different job before the

return message can be received. This may happen if two PEs try to communicate during a large reassignment operation. Mallob provides no further handling mechanism for this case. One way to solve this problem would be to extend Mallob to forward such a message to the job's root PE. In our design we instead chose to not handle this case for two reasons.
First, by not handling this any further we simplify our design and implementation as we avoid special-casing the root PE. Second, as we choose to handle unassigned PEs by preserving the root of their local search space, information is preserved even if any of it's transitive children is lost in the moment. Due to this, the lost message does not represent a lost part of our search space.

5 Implementation

In this section we will give an overview of the implementation work we performed. We start out with an overview of the integration of CrowdHTN into Mallob in 5.1 where we present the Mallob interface, explain how we implemented it while adhering to Mallob's performance guarantees, mention how we improved the reliability of CrowdHTN in low memory conditions and end with a detailed overview of how we handle messages addressed to PEs that no longer belong to the same job. This is followed by our mechanism for efficiently handling restarts in 5.2 and an explanation of how we perform an all-reduction in a malleable environment to allow for global loop detection in 5.3. We conclude with CrowdHTN's improved algorithm for the expansion of search nodes in 5.4 and our presentation of an efficient way to hash search nodes in 5.5.

5.1 Mallob Integration

In this section we will give an overview over how we integrated CrowdHTN with Mallob. More information about how to do this for general problems can be found in the Mallob GitHub repository [2]. There are three steps we need to perform:

- Implement a way to read and encode a TOHTN instance
- Implement the Mallob job interface seen at 4
- Implement a way to encode a result for writing to file

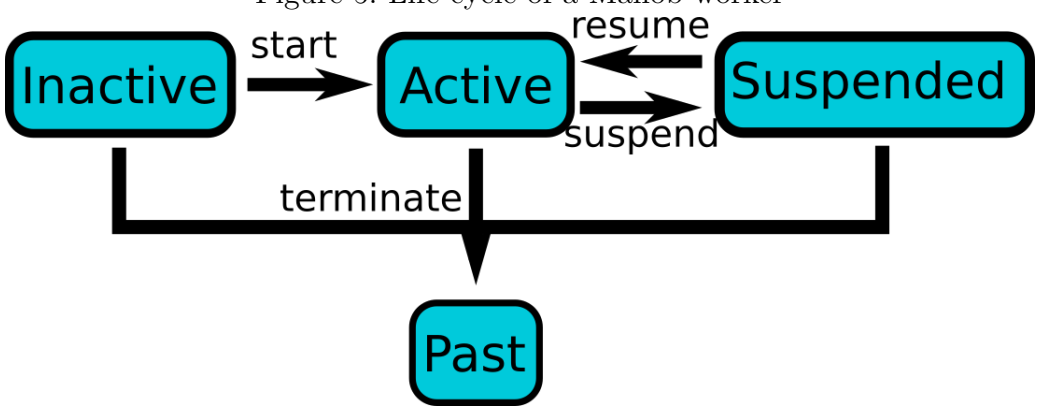
As we discussed in section 4 on how we designed malleable CrowdHTN, we choose to communicate an instance by simply transferring the string contents of the instance file, making the first step easy. The third step, encoding a result for writing, is similarly easy as CrowdHTN already contained a mechanism to write a plan to the terminal. Most of the work was done in the second step which we will now discuss in more detail. As a general principle, CrowdHTN was kept as a separate library which is linked into Mallob. The implementation of the TOHTN job within Mallob is a wrapper around this library. This allowed for a clearer separation of concerns where our job implementation does not need to know anything about the specifics of TOHTN planning while CrowdHTN is agnostic of implementation details of its environment, e.g. how messages are transmitted.
Both CrowdHTN and Mallob are implemented using the C++ programming language.

Algorithm 4: The Mallob job interface

```
1 void appl_start()
2 void appl_suspend()
3 void appl_resume()
4 void appl_terminate()
5 void appl_solved()
6 JobResult appl_getResult()
7 void appl_communicate()
8 void appl_communicate(source, mpi_tag, message)
9 void appl_memoryPanic()
```

[2] https://github.com/domschrei/mallob

Figure 5: Life cycle of a Mallob worker



Implementing the Job Interface In this paragraph we explain how we implemented the Mallob job interface for CrowdHTN while upholding the guarantees demanded by Mallob. For ease of reading we will leave out the *appl* prefix shared by all functions.
The Mallob job interface can be split into four parts. First, a worker in Mallob is implemented as a state machine with *start*, *suspend*, *resume* and *terminate* responsible for the transitions. The corresponding state diagram can be seen in figure 5. Second, the two *communicate* calls allow for communication. For general communication we note that Mallob requires all communication calls to take place in the main thread, i.e. we may not send any messages in any threads we started to perform internal work. For ease of separation we further restrict ourselves to only send messages in the *communicate* calls. Third, we have the functions *solved* and *getResult* for general bookkeeping regarding solutions. Last, we have *memoryPanic* which signals the job that memory usage is critically high. We discuss its use in the next paragraph.
As [58] writes, Mallob aims to achieve millisecond latencies. To enable this goal, we may not block the main thread calling the job interface functions any longer than a few milliseconds at most and must keep work performed directly in any of the job interface functions to a minimum. We achieve this by delegating all planning and handling of the CrowdHTN library to a separate work thread which is initialized in the *start* function. This work thread is the only thread ever directly interacting with CrowdHTN. Due to this, we avoid locking on CrowdHTN which also allows us to keep working at all times. State transitions are communicated to the work thread via a number of atomic variables, *suspend* and *resume* additionally use a condition variable to suspend and wake up the work thread.
To be able to keep communication to the *communicate* functions, the job and the work thread exchange messages via separate buffers. While these buffers do necessitate locking, we restrict the critical section to be at most a copy of a few bytes.
The last problem is the *start* call during which we need to parse our TOHTN instance, setup the CrowdHTN data structures and start our worker thread. Here both the parsing of the instance and, within CrowdHTN, computing the heuristic values may take longer than Mallob allows. For this reason, we have decided to place the initialization itself into a separate thread and return immediately.

Increasing the reliability of CrowdHTN As a scheduler, within a single execution Mallob may work on any number of jobs making it necessary that jobs do not crash. This imposes additional challenges for TOHTN planning, as we have seen in section 2.1.2 that TOHTN planning is EXPSPACE-hard, meaning we may often run out of memory and will be subsequently shut down by the operating system. Luckily, the Mallob job interface we see in algorithm 4 does provide a function for this case. Mallob does periodically check available memory and if it

threatens to run out triggers the *memoryPanic* function. In our case we have implemented it as clearing out half of our local fringe of search nodes and the loop detection information. While this does mean we loose parts of the search space, the alternative would be to immediately return without a plan. Additionally, with the restarting mechanism we introduced in section 3.2.4 CrowdHTN retains completeness even in this case.

Messages and dying workers In 4.3.2 we explained why we can no longer assume that messages will always be delivered and how Mallob implements a return message mechanism to catch this. Now we will cover in detail how we respond to each kind of return message. Afterwards we will look at more complicated cases of workers dying and reappearing and show how they do not affect correctness of our planner
In malleable CrowdHTN, workers directly exchange three kinds of messages. These are *work requests*, *work packages*, i.e. a positive answer to a *work request*, and *negative answers*, i.e. the worker receiving the *work request* does not have any work to share. Getting a returned *work request* is equivalent to receiving a *negative answer*. We treat it the same way and the worker sends out another *work request* to a random other worker. When we receive a returned *work package*, we re-add it to the back of our local fringe. This ensures that no information is lost. If multiple *work packages* are returned to a worker, their order at the back of the fringe may change. We do not expect any significant impact from this, as the number of *work packages* which are in flight is limited by the number of PEs overall which is extremely low compared to the number of search nodes in a hierarchical planning problem. This leaves a returned *negative answer* as the last type to handle. We can simply ignore this type of return message.
Return messages are only one case of dying workers affecting our communication. In addition to this, a worker may die, be terminated and the PE is then, through rebalancing, reassigned to the same job. Any messages meant for the original worker will be received by the new worker, as it belongs to the same job. For *work requests* this is not a problem, the new worker will respond like any other. Receiving *work packages* and *negative answers* will change our behavior. If the worker sent out a *work request* before receiving the *work package* meant for the previous worker, it may end up receiving two *work packages*. Similarly, receiving a *negative answer* while the worker has a *work request* on the way prompts the worker to send out an additional *work request*. In both cases, we integrate any additional *work packages* into the local fringe to not cut off parts of the search space and keep working.

5.2 Efficiently Handling Version Increases

In our malleable CrowdHTN implementation, version increases show up in a number of ways. They are necessitated by the global loop detection introduced in section 3.2 and further allow our DFS based planner to achieve completeness as explained in section 3.3. Due to the distributed fashion in which CrowdHTN operates, version updates are not perfectly synchronized and workers may be at different versions. We will now outline how correctness is ensured and how versions are propagated efficiently.
While versions between workers may differ, we must ensure that especially work packages of different versions are not mixed as to not duplicate parts of the search space. This is ensured by attaching the worker version to any outgoing messages. Upon receiving a message, a worker first decodes the version. If this incoming version is higher than the internal version, the internal version is updated, the local fringe and loop detection cleared and the message is then handled according to this new internal state. If the incoming version is lower, depending on message type it is ignored (e.g. for work packages) or responded to normally (e.g. for work requests). Including the version with all messages has an additional use when integrating new PEs. As

they start out empty and without a way to know the current version, they will immediately send out a work request to a random other worker and receive both the current version and potentially their first work package, requiring no special handling.
Including the version in each message is already sufficient to propagate the version to all workers. However, if we disable global loop detection there are no regular broadcasts from the root to the other PEs. Additionally, the work represented by a search node and it's children may be arbitrarily large. While this reduces the amount of messages sent and is one of the strengths of work stealing in parallel TOHTN planning, it also results in a potentially slow propagation of version increases, having many workers perform outdated work. To counter this problem, whenever a version increase happens at the root PE we start a version broadcast along the binary tree structure of PEs. This ensures that all PEs adopt the new version in a timely manner.

5.3 Global Loop Detection

In section 3.2.4 we introduced a distributed loop detection mechanism based on regularly shared bloom filters. This leaves us with two problems, first performing the associated allreduction while the PEs assigned to the job may change at any time and secondly performing the restarts which are required if the bloom filter fills up.
In both cases we will make use of the specific way in which Mallob organizes the PEs assigned to a job which we have already explained in section 4. The two properties we rely on are the fact that PEs are internally structured as a binary tree with parent and child information available to us and the fact that the root PE will remain assigned to a job during the job's full duration.

Performing the Reduction The all-reduction of our loop detection data is initiated by the root PE and performed in three phases.

- Initiating the reduction
- Aggregating information upwards
- Broadcasting the aggregated information

The root PE is responsible for initializing the all-reduction. It does so by starting a broadcast, sending an initialization message to all it's children. Upon receiving a reduction initialization message, a PE both forwards the message to its own children and prepares the local loop detection data. At the leaves, this data can immediately be sent upwards whereas inner nodes wait until they have received data from all children before performing their local aggregation and forwarding the result upwards. As we combine the bloom filters via a bitwise or operation the message size stays constant throughout. Once the root has received data from all children it once again starts a broadcast, this time containing the aggregated data.
Starting the reduction with the initial broadcast allows us to easily coordinate all PEs even as PEs may assigned to our job may change at any moment. Similarly, we have to deal with PEs leaving at any time. This may be communicated to us either through getting our initial broadcast returned as no receiver is available or by having the *appl_suspend()* function called on us by Mallob. In both cases we simply substitute the message of the missing PE with a response that simulates empty data. We note that, due to changing PEs, the sets of PEs which broadcast their data and which receive the aggregated data may be different. Furthermore, neither of these two sets needs to correspond to the actual tree of PEs assigned to the job at any given time, as this set may change during the broadcast.

Loop Detection Induced Restarts In section 3.2.4 we introduced a global loop detection mechanism based on regularly shared bloom filters. One of the problems this induces is that we need to induce restarts to increase the size of the bloom filter in order to avoid increasing false positive rates. The main problem here is that for different PEs the global bloom filter will fill up at different times. This is due to the fact that different PEs may be assigned to our job for different spans in time which may be further disjointed as PEs are suspended and subsequently reassigned to a job. However, to uphold our guarantees we want to restart all our PEs as soon as a single PE needs to do so.
To solve this, we rely on the fact that the root PE is guaranteed to remain assigned to a job during the job's full lifetime. Due to this, the root PE takes part in every single loop detection data exchange and it's global filter will contain at least as much data as any other PE's filter. This fact allows us to only ever check on the root PE whether the global bloom filter is full and institute a restart if needed. Doing so lets us avoid any problems that would stem from all PEs performing such checks, such as multiple PEs instituting restarts at the same time.

5.4 Improving the Search Node Exploration Algorithm

One of the main improvements we made to the internal workings of CrowdHTN is to reduce the number of search nodes ever explicitly represented. The main idea behind this optimization is that if the next task we need to resolve is an action, then our search node has only one possible child. As such, this search node does not represent a choice point in our search and we do not need to ever explicitly instantiate it.
More formally speaking, let $t = t_1, \ldots, t_n$ be our sequence of open tasks. Then let $t' = t_1, \ldots, t_k$ be the longest prefix of t which consists of only actions. If $k = n$, we create the next search node by applying all actions, checking preconditions and applying effects as we go. If $k < n$, we create the next search node by applying all actions in t' and then additionally resolving abstract task t_{k+1}.
In addition to reducing the size of our fringe by reducing the overall number of search nodes we create, we specifically hope to save both memory and run time by reducing the number of created and represented world states. This is due to the fact that in our old algorithm resolving tasks $t_1, \ldots, t_k$ would have necessitated the creation of k world states which would also not be shared as in our previously introduced scheme. Reducing the number of search nodes has additional benefits regarding loop detection. Using hash sets we reduce the memory footprint as there are fewer nodes inserted in the visited nodes set and using bloom filters the inherent probability for false positives is less of a problem the fewer nodes we check against the filter.
Lastly, we have used the definition of reductions to further reduce our memory footprint. Remember that each search node is identified by both world state s and open tasks tn. If the first open task in tn is compound, then any child nodes will share the same world state as applying a reduction only ever changes the open tasks. We replicate this in our program by having both search nodes use the same world state instance.

5.5 Efficiently Hashing Nodes of the Search Graph

As we described in section 3.2.3, we hash a search node by hashing all of it's open tasks as well as the full world state. While the size of the world state and thus the time required to hash a world state is bound by the number of ground predicates no such limit exists regarding the open tasks. In other words, hashing both world state and open tasks has an unbounded run time which limits the effectiveness of all hash based loop detection mechanisms. In this

section we will describe how we manage to reduce the time required to hash the open tasks to $\mathcal{O}(h \cdot \max\{\#\text{ subtasks of } r | r \in \mathit{reductions}\})$ where a single predicate can be hashed in $\mathcal{O}(h)$. Let n_1, n_2 be search nodes with n_2 a child of n_1. Let t_1, t_2 be their respective sequences of open tasks with t_1 containing at least 1 abstract task which can be resolved by a reduction r with m subtasks $t_{r_1}, \ldots, t_{r_m}$. Furthermore, let $t_1 = t_{1_1}, \ldots, t_{1_k}$ with $t_{1_1}, \ldots, t_{1_l}$ the longest prefix of only actions. Then $t_2 = t_{r_1}, \ldots, t_{r_m}, t_{1_{l+2}}, \ldots, t_{1_k}$, i.e. the subtasks of r concatenated with all of t_1 except the prefixed actions and the first abstract task. Assuming we compute our order dependent hash over a task network t by going from back to front, for hashing t_2 we can reuse the hash of $t_{1_{l+2}}, \ldots, t_{1_k}$, only computing the hash of $t_{r_1}, \ldots, t_{r_m}$.
Storing the hash with each open task does increase our memory footprint. We do consider the trade-off worth it as the hash is small and it allows us to transform our hash function from an unbound to a bound run time.

6 Experimental Evaluation

In this section we will perform an evaluation of our new CrowdHTN implementation as it is integrated into Mallob, the new features we added into CrowdHTN and how it behaves in a malleable environment.
We start out by presenting our experimental setup in 6.1, including the naming scheme we use to identify different versions and configurations of CrowdHTN. After, we first offer a comparison of the performance of CrowdHTN as a standalone program and improved CrowdHTN integrated into Mallob. A short comparison of CrowdHTN with state of the art sequential search-based planners is also included.
This is followed by more detailed discussions of the different improvements and features we added. We will review the effect of the improvements on the implementation level and discuss the behavior of different search algorithms in CrowdHTN. Then we see how the performance changes as we enable bloom filters for loop detection, probabilistic restarts and distributed loop detection one after the other. We conclude with a discussion on the scaling behavior and the performance of CrowdHTN in a malleable environment, followed by an overall recap.
The overall results can also be seen in tables 9 for the sequential planners and standalone CrowdHTN and 10 for CrowdHTN integrated into Mallob.

6.1 Experimental Setup

In our evaluation we reuse the reduced IPC benchmark set we introduced in [14]. We use them, as they remain the de-facto standard for evaluating hierarchical planners [56, 35, 34, 14]. The selection consists of 120 out of the 892 instances used in the IPC 2020, using 5 instances per domain and using 900 seconds per run instead of the 30 minutes in the IPC to make evaluation of many different planner configurations more feasible.
We define the run time of a planner as follows:

- From start until a plan is printed for standalone planners. This includes time spent parsing and grounding
- The wallclock time measured by Mallob for our malleable CrowdHTN, again including parsing

Planners are scored according to both the IPC score and coverage. The IPC score is defined as 1 if a plan was found in less than 1 second, 0 if no plan was found and for $0 < t < T = 900$ as

$$1 - \frac{log(t)}{log(T)}$$

Our tests were done on two machines. The first is a server with an Intel Xeon Gold 6138 processor with 4 sockets, 20 cores per socket and 2 threads per core clocked 2.00G Hz with around 750GB of RAM and running Ubuntu 20.04. We will call it PC1. The second is a server with an AMD EPYC 7702 processor with 1 socket with 64 cores and 2 threads per core clocked 2.00 GHz with around 1TB of RAM and running Ubuntu 20.04. We will call it PC2. The numbers shown in tables 9 and 10 were all obtained on PC1.

Naming Scheme As our CrowdHTN planner contains multiple configuration options, we use a succinct naming scheme to identify them in the evaluation. The name for a CrowdHTN configuration has the following structure:

Cr ⟨CrowdHTN version⟩ ⟨#of PEs⟩ ⟨loop detection method⟩ ⟨presence of restarts⟩

A list of possible values for each category as well as their meaning is shown in table 3. The use of a global bloom filter always implies the use of probabilistic restarts. Unless noted otherwise, all configurations of CrowdHTN use randomized DFS.

Table 3: List of parameters identifying a CrowdHTN configuration

Parameter	Value	Meaning
CrowdHTN Version	O	Old CrowdHTN, standalone
	N	New CrowdHTN, integrated with Mallob
Loop Detection Method	Hs	Hash Set
	Bl	Local Bloom Filter
	Bg	Global Bloom Filter
	No	No loop detection
Presence of Restarts	R	Time dependent restarts are used
	/	No time dependent restarts are used

6.2 Comparing New to Old CrowdHTN and Sequential Planners

When comparing our new implementation of CrowdHTN with the old CrowdHTN, we do see an overall higher IPC score while retaining coverage for our best version where all new features are active. However, when comparing old CrowdHTN with the new implementation using hash sets for loop detection, we note a loss in both coverage and IPC score. A plot of their respective performances is shown in figure 6.
We suspect that the performance degradation comes from our integration into Mallob. CrowdHTN as a work stealing planner sends a high number of messages. As we have seen in the implementation section, to uphold the guarantees of Mallob we do not directly communicate and instead write messages into separate buffers for Mallob to receive and send on which we suspect as one area of lost performance. Additional small overhead may be due to the fact that Mallob performs additional scheduling and rebalancing work in the background.
However, the improvements we added to CrowdHTN do make up for these losses. Additionally, our new implementation of CrowdHTN generally achieves a higher IPC score per coverage, i.e., if a plan is found it is found fast. If this only happened in badly performing configurations, we would suspect that only easy problems with a high score are solved anymore. However, this correlation holds for all configurations of new CrowdHTN, even those exceeding the performance of the old version.
Compared to sequential planners PANDA and HyperTensioN, results are mixed. Looking at PANDA, compared to old CrowdHTN we manage to catch up in domains Hiking, Monroe-Fully-Observable and Snake while staying ahead on Blocksworld-HPDDL, Minecraft-Player and Rover-GTOHP. However, overall CrowdHTN still looses out to the more informed PANDA planner.
When looking at HyperTensioN, CrowdHTN retains the advantage of higher coverage as its parallel nature which can be said to correspond to trying the same search multiple times makes it less hit-or-miss than HyperTensioN itself. At the same time, improvements in time to plan mean that CrowdHTN almost catches up to HyperTensioN regarding IPC score.

6.3 Optimizations in CrowdHTN

In 5.4 we described an improvement to our progression search which let us reduce the number of search nodes and world states we instantiate. To evaluate the impact of this improvement

we created an instrumented version of CrowdHTN which let us track this information during planning. We ran this version of CrowdHTN on our benchmark on PC2, using 1 PE, DFS and hash set based loop detection while giving it 300 seconds per instance. We tracked the metadata for all instances, whether a plan was found or not. In addition to the information on actions and world states, we tracked the number of search nodes which were duplicates. The results are shown in table 4.
Compared to the other measures, the ratio of tasks which are actions is relatively consistent between domains. It varies from about one third to about two thirds of all tasks. The ratio is lowest for the Logistics-Learned-ECAI-16 domain at 28.2% and highest for Rover-GTOHP at 66.83%.
When it comes to shared world states, results vary more. On 11 out of the 24 test domains, a world state is on average shared by less than 10 search nodes. This is lowest for the Snake domain with only 1.4 search nodes per world state. On the other hand for 6 out of our 24 domains more than 1000 search nodes share one world state, going as far as $\sim 3.5 \times 10^8$ search nodes per world state for the Transport domain.
To sum these improvements up, our improved search node exploration is a clear benefit on all domains, reducing the number of search nodes we need to represent by at least 28%. Sharing world states is more mixed. While sharing is extreme on some domains, it does come at the cost of an additional pointer indirection which may be especially harmful on domains with little sharing as there is no benefit to offset this cost.
Regarding loop detection, the results are similarly varied as with state sharing. On 7 out of 24 domains no duplicate nodes were encountered at all and on another 5 domains less than 1% of nodes were duplicates. At the other end of the spectrum we have Minecraft-Player and Logistics-Learned-ECAI-16 with $\sim$ 30% and Factories-simple with $\sim$ 44% of duplicate nodes. We will return to these numbers in the evaluation of different loop detection techniques in 6.5.

6.4 Search Algorithms

In section 3.1 we presented four search algorithms that we implemented for CrowdHTN. Those algorithms are random DFS, heuristic DFS, A*-like and BFS. We have tested all four algorithms on PC1 using our test instance set using 4 PEs and a local bloom filter without restarts for loop detection. The results of this test are visualized in figure 7, a summary of coverage and

Figure 6: Instances solved per time for old and new CrowdHTN

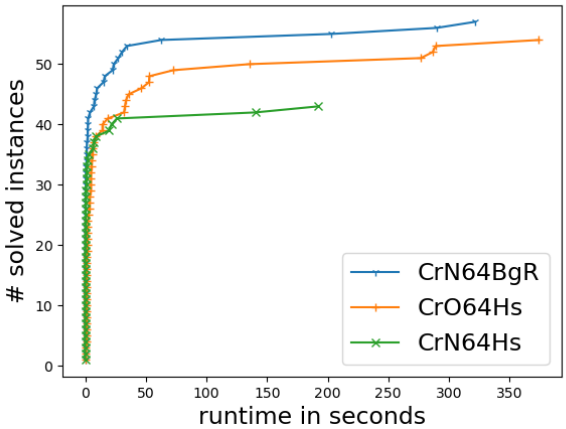


Table 4: Metadata about progression search on our benchmark

Domain	Action%	Nodes per World State	Loop%
AssemblyHierarchical	49.71	4793.8	0.006
Barman-BDI	33.56	10.4	4.866
Blocksworld-GTOHP	50.80	6.1	0.000
Blocksworld-HPDDL	49.34	83.4	1.690
Childsnack	53.94	28800.6	0.000
Depots	45.11	7.3	2.790
Elevator-Learned-ECAI-16	47.89	44.9	1.663
Entertainment	49.49	7.2	0.095
Factories-simple	31.09	2.3	43.815
Freecell-Learned-ECAI-16	44.42	2.5	0.000
Hiking	65.24	2.6	12.508
Logistics-Learned-ECAI-16	28.20	4.7	30.348
Minecraft-Player	32.21	4.4	29.896
Minecraft-Regular	31.34	3.4	0.000
Monroe-Fully-Observable	48.65	114.0	1.457
Monroe-Partially-Observable	48.29	106.6	3.418
Multiarm-Blocksworld	47.06	28.9	8.240
Robot	50.00	46207.3	0.002
Rover-GTOHP	66.83	3.5	0.000
Satellite-GTOHP	34.66	52.5	0.013
Snake	47.16	1.4	6.948
Towers	49.99	5409.3	0.000
Transport	36.98	347788104.5	0.000
Woodworking	49.83	21708.8	0.442

IPC score is presented in table 5.
Overall, random DFS performed best, followed by heuristic DFS, A*-like search and finally BFS. The best performing algorithm, random DFS, solved almost twice as many instances and has twice the IPC score of the worst performing algorithm, BFS. Additionally, we observe a hit-or-miss behavior in both our DFS implementations where plans are either found almost immediately or not at all. Out of the 50 instances solved by random DFS, only 18 were solved in more than 1 and out of these 18 only 9 were solved in more than 10 seconds. BFS on the other hand solves 14 out of 30 instances in more than a second and 11 of these 14 in over 10 seconds. As such, while overall worse performing it does seem to scale better with runtime. Overall, BFS seems unsuited for TOHTN planning due to the extremely high branching factor of the problems involved.
Comparing our two DFS-based approaches, we see that random DFS performs better than heuristic DFS guided by our heuristic from section 3.1.3. We attribute this to the fact that we consciously limited our heuristic to information on the hierarchy available from the lifted instance to reduce the time spent on precomputation. Others, such as [35] argue that heuristics must utilize both hierarchy and world state information. Our findings corroborate this theory.

Figure 7: Instances solved per time for CrowdHTN using DFS, heuristic DFS, A*-like search and BFS

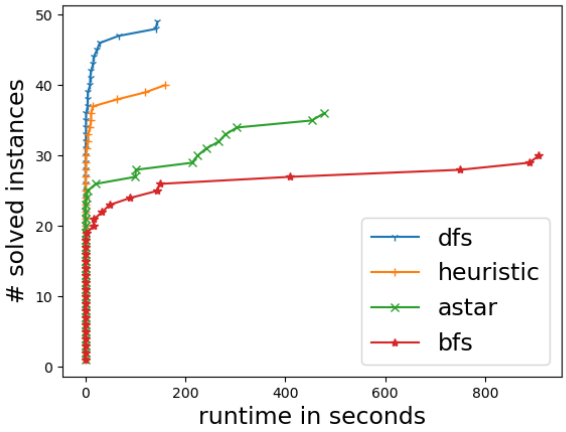


Table 5: Coverage and IPC score of our search algorithms using 4 PEs and a local bloom filter

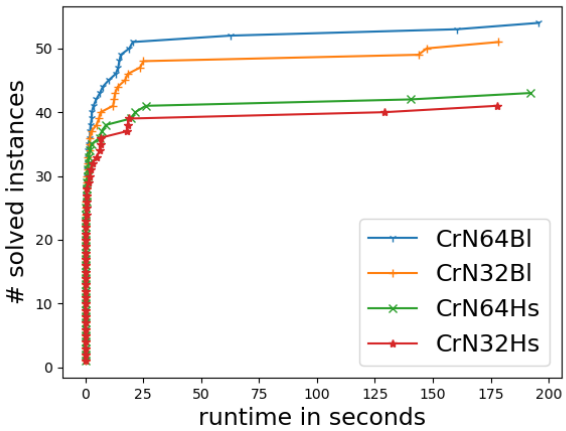
Algorithm	Coverage	IPC Score
Random DFS	41.7%	43.09
Heuristic DFS	33.3%	35.60
A*-like	38.3%	27.13
BFS	25.0%	21.87

6.5 Bloom Filters in Loop Detection

The next feature we tested was the new loop detection based on bloom filters. For our tests we set $k = 4$ and limited our false positive probability to 0.001. The results of this test are shown in figure 8. We see that our bloom filter outperforms the hash set on 32 and 64 PEs, increasing the IPC score by ~ 5.5 and coverage by about 6% when switching from hash set to bloom filter. The gains are so large that 32 PEs using a bloom filter outperform 64 PEs using

a hash set.
The gains are strongest on the domain Monroe-Fully-Observable and also visible on Snake. However, we also note that our hash set based loop detection achieves good performance on the Logistics-Learned-ECAI-16 domain while no configuration using bloom filters was able to solve a single instance on this domain. A similar but weaker effect happens in Factories-simple. Looking at the data in table 4 we see that Logistics-Learned-ECAI-16 and Factories-simple are two of the domains with the highest rate of duplicate nodes. While domains on which hash sets clearly outperform bloom filters all have a very high rate of duplicate nodes, not all domains with many duplicate nodes benefit hash set based loop detection. Minecraft-Player contains almost 30% duplicate nodes and CrowdHTN performs equal in both modes.

Figure 8: Instances solved per time with hash set and bloom filter based loop detection



6.6 Probabilistic Restarts

In our next test, we enabled the probabilistic restarts we introduced to guarantee completeness for our bloom filter based loop detection. With runs lasting 900 seconds, we expect $\sum_{t=1}^{899} \frac{1}{t} \approx 7.38$ restarts per run with 5 of these restarts happening within the first 90 seconds.
First, we compare CrowdHTN using local bloom filters with and without restarts. The result is shown in figure 9. Overall, probabilistic restarts seem to have a positive effect on coverage and IPC score which is more pronounced on a lower number of PEs. In this way they somewhat mitigate the hit-or-miss nature of our planner.
While we see a big difference on 32 PEs, there is little difference on 64 PEs where restarts come with a slight benefit to coverage and a small loss in IPC score. We suspect that, while restarts may increase our chances of finding a plan, they do decrease the chance of finding a plan fast, as they interrupt our search and are especially common right at the beginning. Additionally, there is little difference between restarts on 32 and 64 PEs. We will go further into the specific scaling behavior of CrowdHTN in the benchmark on scalability in 6.8.

6.7 Global Loop Detection

The last new feature we introduced into CrowdHTN is the ability to perform distributed loop detection. A plot of CrowdHTN with distributed loop detection and CrowdHTN using local loop detection with probabilistic restarts is shown in figure 10. With distributed loop detection

enabled we get a further small increase in both coverage and IPC score. Among all configurations using bloom filters, CrowdHTN on 64PEs and using global loop detection has the highest coverage and the overall highest IPC score. Similarly, the configuration on 32 PEs is the second best overall, slightly outperforming the 64 PE versions with and without restarts.

6.8 Scalability of CrowdHTN

As CrowdHTN is a search-based planner, it has a hit-or-miss characteristic to its performance. This can make it hard to see how its overall performance scales. Many instances are already efficiently solved on a low number of PEs while other instances will remain out of reach even on a high number of PEs. However, over our full benchmark we can still see an increase in overall performance as seen in figure 11, even if the effect is relatively weak.
To better visualize the scaling behavior of CrowdHTN we will now focus on the Monroe-Fully-Observable domain. It has instances which are somewhat reliably solved for any number of PEs while not being trivial. We ran a separate benchmark of all 20 instances that come with this domain on PC2, testing various configurations of CrowdHTN with local loop detection and no restarts versus global loop detection with restarts. The results are listed in table 6.

Figure 9: Instances solved per time with a local bloom filter with and without restarts

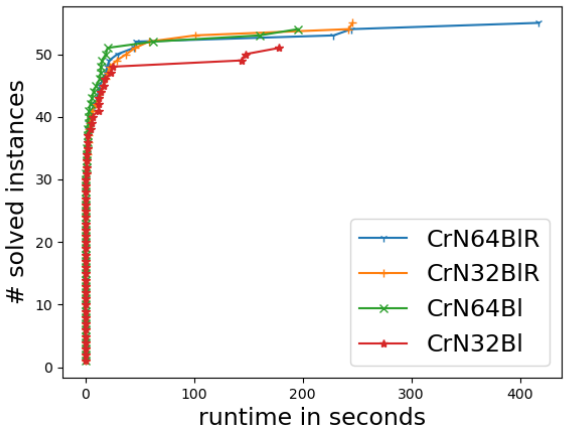
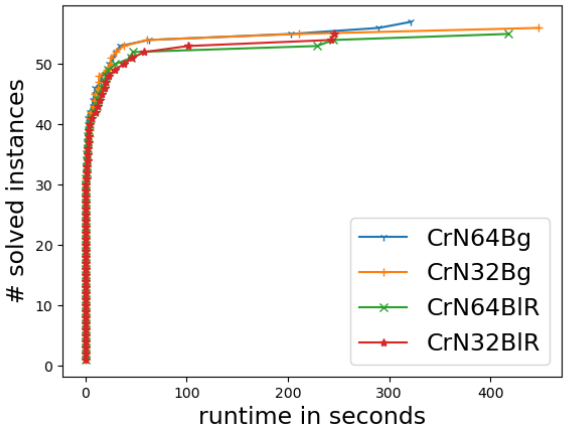
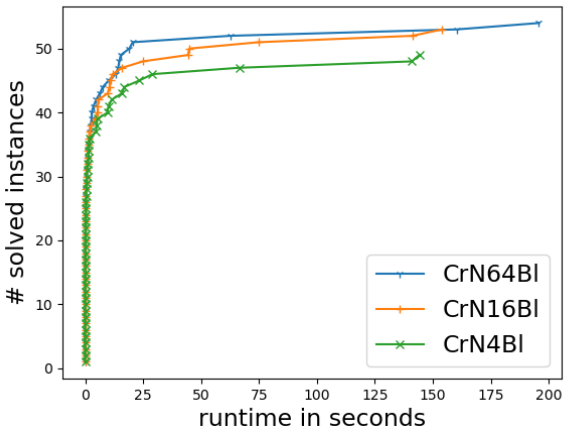


Figure 10: Instances solved per time with a local bloom filter and restarts with global loop detection



Overall we note the clearly increasing IPC score as the number of PEs is increased. Increasing the number of PEs from 4 to 16 has a bigger effect than quadrupling it again to 64. We assume that on this test benchmark the chance of encountering a plan is sufficiently high that we run into diminishing returns as the number of PEs is increased further. Interestingly, CrowdHTN without restarts seems to scale more strongly than CrowdHTN with restarts, as far as the IPC score is concerned. We attribute this to the fact that the IPC score values short run times especially high. Restarts are most frequent during the early phase of planning and may stop CrowdHTN from finding plans very fast.
To reduce the impact of randomness, we launched on additional test using instance 11 of the Monroe-Fully-Observable domain. We used CrowdHTN with global loop detection and restarts active, running it 100 times with a time limit of 90 seconds. The distribution of run times is shown in figure 12 while success rate and average run times are listed in table 7. All three configurations have high success rate, at 93%, 97% and 98% respectively. The addition of 8 more PEs correlates with an approximately 10 second decrease in average run time. This corresponds to a percentage decrease in run times of 20% when going from 16 to 24 PEs and of another 30% when going from 24 to 32 PEs, increases in PEs of 50% and 33% respectively. In reality, gains are even higher as these average run times ignore the cases were the 16 PE configuration failed to find a plan at all.

Figure 11: Instances solved per time for CrowdHTN using DFS and a local bloom filter on 4, 16 and 64 PEs



6.9 Malleable CrowdHTN

To test the behavior of CrowdHTN under malleable conditions, we extended our previous test on scalability. We ran another test on Monroe-Fully-Observable instance 11 using 32 PEs. However, every 20 seconds we injected a second unsolvable job with a time limit of 10 seconds. This means that our normal test oscillates between 32 and 16 PEs every 10 seconds, having an average of 24 PEs available. We compare success rate, average time to plan and the overall distribution of run times with moldable CrowdHTN on 24 PEs. The results are listed in table 8. Figure 13 shows a box plot of run times per solver.
Moldable CrowdHTN on 24 PEs reliably solves this problem in 90 seconds with a success rate of 97% and an average time to plan of 36.62 seconds. In the ideal case, our malleable CrowdHTN would replicate this behavior. However, we see that malleable CrowdHTN achieves only 69% success rate with an average time to plan of 31.84 seconds. Looking at the box plot visualizing

Table 6: Evalutating CrowdHTN on 20 instances of the Monroe-Fully-Observable domain

	CrN4Bl		**CrN16Bl**		**CrN64Bl**		**CrN4Bg**		**CrN16Bg**		**CrN64Bg**	
	Time	IPC	Time	IPC	Time	IPC	Time	IPC	Time	IPC	Time	IPC
01	0.2	1.00	0.1	1.00	0.4	1.00	0.1	1.00	0.7	1.00	0.1	1.00
02	161.5	0.25	30.7	0.50	1.3	0.96	115.6	0.30	10.5	0.65	22.6	0.54
03	/	0.00	5.7	0.74	8.4	0.69	250.5	0.19	4.0	0.80	30.5	0.50
04	0.4	1.00	0.3	1.00	0.2	1.00	0.3	1.00	0.3	1.00	0.4	1.00
05	49.9	0.43	55.6	0.41	22.6	0.54	23.7	0.53	30.2	0.50	26.6	0.52
06	183.9	0.23	60.1	0.40	3.5	0.82	129.0	0.29	61.7	0.39	18.2	0.57
07	18.5	0.57	7.2	0.71	3.2	0.83	5.3	0.75	2.0	0.90	4.0	0.80
08	98.1	0.33	48.0	0.43	4.4	0.78	108.0	0.31	101.7	0.32	12.5	0.63
09	62.3	0.39	17.7	0.58	26.0	0.52	70.8	0.37	13.5	0.62	14.3	0.61
10	122.1	0.29	33.9	0.48	23.8	0.53	80.4	0.35	61.9	0.39	11.0	0.65
11	148.9	0.26	60.5	0.40	15.7	0.60	148.6	0.26	62.8	0.39	15.0	0.60
12	137.5	0.28	47.9	0.43	19.4	0.56	223.4	0.20	34.7	0.48	25.6	0.52
13	19.4	0.56	2.7	0.85	1.5	0.94	7.9	0.70	0.6	1.00	1.7	0.92
14	171.3	0.24	61.1	0.40	36.1	0.47	279.5	0.17	36.2	0.47	42.1	0.45
15	/	0.00	6.5	0.73	4.3	0.79	/	0.00	5.0	0.76	0.8	1.00
16	/	0.00	6.3	0.73	2.0	0.90	7.1	0.71	6.2	0.73	4.8	0.77
17	/	0.00	1.6	0.93	11.2	0.64	/	0.00	/	0.00	1.8	0.91
18	16.7	0.59	40.0	0.46	10.3	0.66	47.2	0.43	9.0	0.68	37.7	0.47
19	1.9	0.90	5.1	0.76	1.8	0.91	15.7	0.60	15.0	0.60	5.7	0.74
20	21.7	0.55	2.9	0.84	2.5	0.86	12.5	0.63	6.0	0.74	4.6	0.78
		7.88		12.78		15.01		8.81		12.43		13.98

Figure 12: Distribution of solving times on Monroe-Fully-Observable instance 11

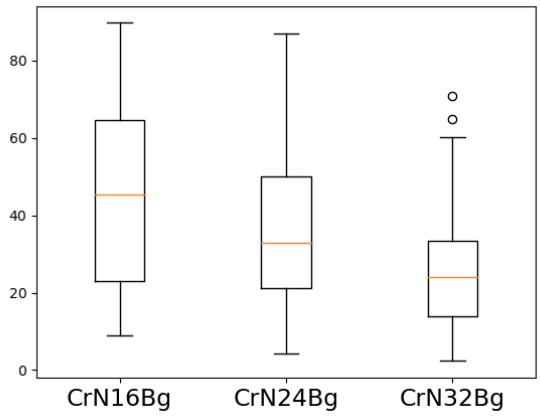


Table 7: Success rate and average, minimum and maximum run times of CrowdHTN on Monroe-Fully-Observable instance 11

Configuration	Success Rate	Average time to plan
CrN16Bg	93%	45.73
CrN24Bg	97%	36.62
CrN32Bg	98%	25.98

the distribution of run times, we see that malleable CrowdHTN and moldable CrowdHTN on 24 PEs share the distribution of run times of up to about 50 seconds. Malleable CrowdHTN is missing tail end of the distribution, though, finding no plans beyond the 60 second mark.
We suspect that this behavior is due to the way restarts are implemented in CrowdHTN and with how disappearing workers are handled. As we restart with probability $\frac{1}{t}$ at second t, we expect about 5 restarts during a 90 second run with 4 of these restarts taking place in the first 30 seconds. Additionally, we handle disappearing PEs by sending the root of their local search space to a random other PE. As in our experiment half of PEs are lost each time another job is introduced, a high number of those messages may be sent to other disappearing PEs, leading to an unexpectedly high loss of information. Restarts seem to mitigate this at the beginning of the search as they are still frequent.
Overall, while loosing performance we still managed to solve a large share of instances.

Figure 13: Distribution of solving times for malleable and moldable CrowdHTN

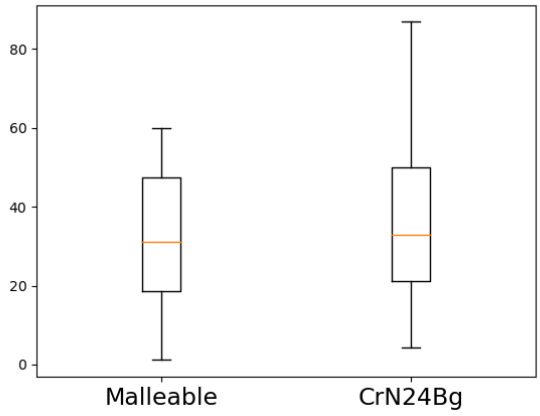


Table 8: Coverage and average run time of malleable and moldable CrowdHTN

Configuration	Success rate	Average time to plan
Malleable, average 24 PEs	69%	31.84
CrN24Bg	97%	36.62

6.10 Discussion

In this section, we compare our parallel TOHTN planner CrowdHTN as it is integrated into Mallob to the old standalone version of CrowdHTN as well as sequential planners PANDA and HyperTensioN. As shown in the previous sections, the integration of CrowdHTN into Mallob comes with some amount of overhead. We show that CrowdHTN has it's own specific strengths and manages to outperform HyperTensioN in coverage while coming close in overall score.
We see the following main results

- Our improved node exploration algorithm manages to reduce the number of instantiated nodes from around one to two thirds total.
- Bloom filters in loop detection improve both coverage and IPC score overall with the bigger impact on IPC score.

- Distributed loop detection and restarts improve coverage and IPC score for smaller numbers of PEs while increasing coverage but potentially decreasing IPC score for a large number of PEs. This may be due to the fact that very fast times to plan are lost in early restarts.

Regarding our upgrades to CrowdHTN, we see that the improved node exploration algorithm leads to a big reduction in overall search nodes encountered. Furthermore, bloom filters and distributed loop detection along with restarts both manage to improve the performance of CrowdHTN, with the bigger gain in IPC score coming from the introduction of bloom filters and distributed loop detection with restarts having a bigger impact on overall coverage.
In addition to this, we see that overall scaling in CrowdHTN is hard to demonstrate due to the hit-or-miss nature of the planner. However, on well-suited domains such as Monroe-Fully-Observable we demonstrate good scaling behavior of CrowdHTN.
Regarding malleability, we see that performance is partially preserved but that loss of information due to frequent reshuffling of a large fraction of PEs can present a problem that the current restarting technique is unequipped to handle.

Table 9: Domain-wise comparison of sequential planners PANDA, HyperTensioN and parallel planner Crowd in its standalone version

	PANDA		**HyTN**		**CrO4Hs**		**CrO64Hs**	
Domain	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov
AssemblyHierarchical	**1.0**	20%	**1.0**	20%	**1.0**	20%	0.98	20%
Barman-BDI	2.34	60%	**4.0**	80%	1.79	40%	1.74	40%
Blocksworld-GTOHP	**4.49**	100%	2.01	60%	2.0	40%	2.49	60%
Blocksworld-HPDDL	1.27	40%	**3.98**	80%	3.21	80%	3.01	80%
Childsnack	2.64	80%	**4.0**	80%	2.6	80%	2.37	80%
Depots	3.0	60%	**4.0**	80%	3.63	80%	3.6	80%
Elevator-Learned-ECAI-16	3.07	100%	3.0	60%	**4.06**	100%	3.86	100%
Entertainment	**4.0**	100%	0.0	0%	0.0	0%	0.0	0%
Factories-simple	**2.0**	40%	1.0	20%	1.91	40%	1.86	40%
Freecell-Learned-ECAI-16	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Hiking	3.55	80%	**4.0**	80%	2.0	40%	1.7	60%
Logistics-Learned-ECAI-16	2.08	60%	2.0	40%	2.57	60%	**2.79**	80%
Minecraft-Player	0.88	40%	**2.0**	40%	1.73	40%	1.62	40%
Minecraft-Regular	3.6	80%	**4.0**	80%	3.14	80%	2.99	80%
Monroe-Fully-Observable	**2.69**	100%	0.0	0%	1.68	100%	2.07	100%
Monroe-Partially-Observable	**1.53**	80%	0.0	0%	0.93	20%	0.82	20%
Multiarm-Blocksworld	**1.0**	20%	**1.0**	20%	**1.0**	20%	0.98	20%
Robot	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%
Rover-GTOHP	2.75	60%	**4.45**	100%	3.7	100%	3.26	80%
Satellite-GTOHP	**3.62**	100%	0.0	0%	0.0	0%	0.0	0%
Snake	4.45	100%	**5.0**	100%	3.07	80%	2.84	80%
Towers	**2.6**	60%	2.0	40%	2.2	60%	2.15	60%
Transport	**2.86**	80%	1.85	40%	0.0	0%	0.0	0%
Woodworking	**2.0**	40%	0.35	20%	0.0	0%	0.0	0%
Instances: 120	59.4	64%	51.6	45%	44.2	47%	43.1	48%

Table 10: Domain-wise comparison of parallel planner CrowdHTN in various configurations

	CrN32Hs		**CrN64Hs**		**CrN4Bl**		**CrN16Bl**		**CrN32BL**		**CrN64Bl**		**CrN32BlR**		**CrN64BlR**		**CrN32Bg**		**CrN64Bg**	
Domain	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov	IPC	Cov
AssemblyHierarchical	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	**1.22**	40%
Barman-BDI	1.0	20%	1.97	40%	**2.0**	40%	**2.0**	40%	1.98	40%	1.97	40%	1.83	40%	1.97	40%	1.89	40%	1.97	40%
Blocksworld-GTOHP	2.0	40%	2.0	40%	2.0	40%	2.0	40%	2.0	40%	2.39	60%	2.32	60%	2.2	60%	**2.77**	60%	2.39	60%
Blocksworld-HPDDL	**3.05**	80%	3.0	80%	3.03	80%	3.02	80%	3.0	80%	2.98	80%	2.82	80%	2.88	80%	2.88	80%	2.83	80%
Childsnack	2.86	80%	2.83	80%	**2.96**	80%	2.93	80%	2.87	80%	2.82	80%	2.57	80%	2.64	80%	2.56	80%	2.77	80%
Depots	3.83	80%	**3.95**	80%	3.92	80%	3.86	80%	3.86	80%	3.85	80%	3.77	80%	3.82	80%	3.79	80%	3.87	80%
Elevator-Learned-ECAI-16	4.24	100%	4.14	100%	4.33	100%	**4.36**	100%	4.31	100%	4.29	100%	4.02	100%	4.11	100%	4.1	100%	4.22	100%
Entertainment	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Factories-simple	**2.0**	40%	**2.0**	40%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%	1.0	20%
Freecell-Learned-ECAI-16	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Hiking	1.0	20%	2.0	40%	2.0	40%	2.0	40%	2.0	40%	2.0	40%	2.0	40%	2.89	60%	2.76	60%	**3.0**	60%
Logistics-Learned-ECAI-16	2.0	40%	**3.01**	80%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Minecraft-Player	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%
Minecraft-Regular	1.0	20%	1.98	40%	**3.6**	80%	3.58	80%	3.54	80%	3.5	80%	3.29	80%	3.47	80%	3.38	80%	3.47	80%
Monroe-Fully-Observable	0.72	20%	0.0	0%	1.92	60%	3.3	100%	2.69	80%	**4.08**	100%	3.68	100%	3.72	100%	3.34	100%	3.6	100%
Monroe-Partially-Observable	0.0	0%	0.0	0%	**1.0**	20%	0.0	0%	**1.0**	20%	**1.0**	20%	**1.0**	20%	**1.0**	20%	**1.0**	20%	**1.0**	20%
Multiarm-Blocksworld	**1.0**	20%	**1.0**	20%	0.0	0%	**1.0**	20%	**1.0**	20%	**1.0**	20%	0.99	20%	**1.0**	20%	**1.0**	20%	**1.0**	20%
Robot	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%	**2.0**	40%
Rover-GTOHP	4.43	100%	4.4	100%	**4.56**	100%	4.55	100%	4.51	100%	4.47	100%	4.39	100%	4.35	100%	4.46	100%	4.38	100%
Satellite-GTOHP	0.0	0%	0.0	0%	0.0	0%	**1.0**	20%	0.0	0%	0.0	0%	0.97	20%	0.0	0%	0.97	20%	**1.0**	20%
Snake	2.0	40%	2.0	40%	3.1	80%	3.89	100%	2.85	80%	3.4	80%	3.94	100%	3.98	100%	4.02	100%	**4.33**	100%
Towers	2.57	60%	2.56	60%	**2.66**	60%	2.65	60%	2.62	60%	2.61	60%	2.61	60%	2.56	60%	2.52	60%	2.6	60%
Transport	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	**1.0**	20%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Woodworking	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%	0.0	0%
Instances: 120	38.7	36%	41.8	39%	43.1	41%	46.1	44%	44.2	42%	47.4	45%	46.2	46%	46.6	46%	47.4	47%	48.6	48%

7 Conclusion

In this work, we have investigated and improved methods of parallel TOHTN planning and designed a malleable TOHTN planner based on work stealing. We will summarize our work in 7.1 and provide a short outlook in 7.2.

7.1 Recap

In hierarchical planning we repeatedly decompose a set of initial tasks until we have obtained a sequence of actions we can directly apply to achieve our initial goal. Malleability is the ability of a program to handle a varying amount of available resources during execution. We set out to improve the performance and scalability of our parallel planner CrowdHTN by experimenting with different search algorithms and designing a distributed loop detection scheme. In addition to this, we showed that current loop detection schemes are insufficient to guarantee completeness for our planner and argue for restarts to ensure correctness. To adapt our parallel planner into a malleable environment, we present our design of CrowdHTN integrated into the malleable scheduler and load balancer Mallob.

We evaluate our improved planner on a subset of the benchmark used in the IPC 2020. There we find that both bloom filters and distributed loop detection improve the overall performance and see that our planner shows clear scaling. In addition, we find that our restart scheme has positive performance implications beyond guaranteeing completeness.
While our planner guarantees completeness and work stealing makes it easy to efficiently use new PEs in a malleable environment, we see performance degradation due to information loss when a large number of available PEs is frequently reshuffled.

Overall, scalable and performant TOHTN planning seems possible but will need more work to improve the search itself as the high branching factor of TOHTN problems limits the potential of a brute force approach as used in CrowdHTN. Similarly, malleable applications based on work stealing in general and malleable TOHTN planning specifically seem worthwhile as the problems with information loss get addressed.

7.2 Future Work

In this work we have shown how current loop detection mechanisms are insufficient to ensure planner completeness and how our own global loop detection scheme with restarts can improve overall performance. We will now give a short overview of ideas we encountered that may improve the capabilities of loop detection and further increase the performance of our distributed loop detection and restarting scheme. This is followed by ideas as to how information loss in a malleable environment may be better addressed.

Advanced dynamic pruning In section 3.3 we show that the current state of the art in loop detection can detect some but not all cases of recursion as it is present in hierarchical planning problems. We further show that this not only negatively affects the performance of our planners but may make planners based on heuristic DFS incomplete.
To deal with such cases, we could take the idea of detecting duplicate search nodes and generalize it into a notion of dynamic pruning of uninteresting nodes. Uninteresting here means that exploring this node will give us no new information about our planning problem. A duplicate

search node is uninteresting as we have explored it before, obtaining all information that is available. However, a search node may also be uninteresting if it's open tasks is built such that we are guaranteed to perform unnecessary work when exploring it. Such a set of open tasks may contain a sequence of k times task t, where the resolution of t may only create actions and more instances of t and affect at most l predicates. If $k > 2^l$, then extending this sequence by more instances of t is of no benefit. We could only create more instances of t and resolving all open t via actions would lead us through duplicate world states, performing unnecessary work. More research would need to be put into detecting such cases to perform more intelligent pruning of search nodes. If done successfully, heuristic DFS may regain completeness if combined with this new scheme.

Global loop detection We show that a global loop detection scheme positively impacts the performance of our parallel TOHTN planner. So far we have only used a simple heuristic to determine when a search node is entered into the global filter, namely if it was twice encountered locally. Better informed heuristics to determine when a node is likely interesting for other PEs could further improve the performance of our distributed loop detection scheme. Additionally, we could track search nodes whose subgraph has been fully explored and keep them in an additional global filter. This would increase the memory footprint as nodes would have to be kept around until we backtrack past them but such information could even be kept across restarts to further improve performance over time.

Intelligent restarts In both 3.2.4 and 3.3 we argue for the use of probabilistic restarts to guarantee the completeness of CrowdHTN. We perform restarts with probability $\frac{1}{t}$ at second t. This is only one of the possible ways to use restarts.
Restarts and restarting strategies to increase solver performance have played a role in SAT solving since the 1990s. [40] suggested the use of iterative sampling in AI planning systems, repeatedly exploring random paths up to a depth limit and restarting if no solution was found. Their work was adapted by [18] who employed the iterative sampling strategy for SAT solving. As a general restart strategy for increased performance, the Luby sequence was developed [42] and has by now been adapted by SAT solvers [36]. By now a wide array of different restart strategies have been tried in SAT solvers. There are both static restart strategies, such as uniform intervals and based Luby schemes and dynamic strategies which incorporate run time information into their decision [10].
We are eager to see how incorporating such restart schemes may affect the performance of hierarchical planners.

Malleable communication schemes As we have seen in our evaluation, our malleable TOHTN planner struggles with the frequent reshuffling of PEs. This could be addressed either by changing Mallob s.t. message loss can be avoided completely or by designing new communication schemes that ensure no information is lost. This may be done by limiting communication to use the edges of the binary tree in which Mallob organizes the worker of a job and having parents guarantee that no information of their children is lost. Additional work would be needed to ensure that the guarantees of randomized work stealing still hold up in such an environment.

References

[1] Alford, Ron, Gregor Behnke, Daniel Höller, Pascal Bercher, Susanne Biundo and David W Aha: *Bound to plan: Exploiting classical heuristics via automatic translations of tail-recursive HTN problems.* In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[2] Balyo, Tomáš, Peter Sanders and Carsten Sinz: *Hordesat: A massively parallel portfolio SAT solver.* In *International Conference on Theory and Applications of Satisfiability Testing*, pages 156–172. Springer, 2015.

[3] Behnke, Gregor, Daniel Höller and Pascals Bercher: *International Planning Competition 2020 On Hierarchical Task Network (HTN) Planning.* http://gki.informatik.uni-freiburg.de/competition/results.pdf, 2020.

[4] Behnke, Gregor, Daniel Höller and Susanne Biundo: *On the complexity of HTN plan verification and its implications for plan recognition.* In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, pages 25–33, 2015.

[5] Behnke, Gregor, Daniel Höller and Susanne Biundo: *totSAT-Totally-ordered hierarchical planning through SAT.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[6] Behnke, Gregor, Daniel Höller and Susanne Biundo: *Tracking branches in trees-A propositional encoding for solving partially-ordered HTN planning problems.* In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 73–80. IEEE, 2018.

[7] Behnke, Gregor, Daniel Höller, Alexander Schmid, Pascal Bercher and Susanne Biundo: *On succinct groundings of HTN planning problems.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9775–9784, 2020.

[8] Bender, Michael A, Martin Farach-Colton, Rob Johnson, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane and Erez Zadok: *Don't thrash: how to cache your hash on flash.* In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, 2011.

[9] Bercher, Pascal, Ron Alford and Daniel Höller: *A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations.* In *IJCAI*, pages 6267–6275, 2019.

[10] Biere, Armin and Andreas Fröhlich: *Evaluating CDCL restart schemes.* Proceedings of Pragmatics of SAT, pages 1–17, 2015.

[11] Blazewicz, Jacek, Mikhail Y Kovalyov, Maciej Machowiak, Denis Trystram and Jan Weglarz: *Preemptable malleable task scheduling problem.* IEEE Transactions on Computers, 55(4):486–490, 2006.

[12] Bloom, Burton H: *Space/time trade-offs in hash coding with allowable errors.* Communications of the ACM, 13(7):422–426, 1970.

[13] Blumofe, Robert D and Charles E Leiserson: *Scheduling multithreaded computations by work stealing.* Journal of the ACM (JACM), 46(5):720–748, 1999.

[14] Bretl, Colin, Niko Wilhelm Dominik Schreiber and Peter Sanders: *Parallel and Distributed TOHTN Planning.*

[15] Broder, Andrei and Michael Mitzenmacher: *Network applications of bloom filters: A survey.* Internet mathematics, 1(4):485–509, 2004.

[16] Buisson, Jérémy, Françoise André and Jean-Louis Pazat: *A framework for dynamic adaptation of parallel components.* In *International Conference ParCo*, volume 33, page 65, 2005.

[17] Cirne, Walfredo and Francine Berman: *A model for moldable supercomputer jobs.* In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 8–pp. IEEE, 2001.

[18] Crawford, James M and Andrew B Baker: *Experimental results on the application of satisfiability algorithms to scheduling problems.* In *AAAI*, volume 2, pages 1092–1097, 1994.

[19] Erol, Kutluhan, James Hendler and Dana S Nau: *HTN planning: Complexity and expressivity.* In *AAAI*, volume 94, pages 1123–1128, 1994.

[20] Erol, Kutluhan, James Hendler and Dana S Nau: *Complexity results for HTN planning.* Annals of Mathematics and Artificial Intelligence, 18(1):69–93, 1996.

[21] Fan, Bin, Dave G Andersen, Michael Kaminsky and Michael D Mitzenmacher: *Cuckoo filter: Practically better than bloom.* In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[22] Fan, Li, Pei Cao, Jussara Almeida and Andrei Z Broder: *Summary cache: a scalable wide-area web cache sharing protocol.* IEEE/ACM transactions on networking, 8(3):281–293, 2000.

[23] Feitelson, Dror G: *Job scheduling in multiprogrammed parallel systems.* 1997.

[24] Froleyks, Nils, Marijn Heule, Markus Iser, Matti Järvisalo and Martin Suda: *SAT competition 2020.* Artificial Intelligence, 301:103572, 2021.

[25] Fukunaga, Alex, Adi Botea, Yuu Jinnai and Akihiro Kishimoto: *Parallel A* for State-Space Search.* In *Handbook of Parallel Constraint Reasoning*, pages 419–455. Springer, 2018.

[26] Georgievski, Ilche and Marco Aiello: *HTN planning: Overview, comparison, and beyond.* Artificial Intelligence, 222:124–156, 2015.

[27] Georgievski, Ilche, Faris Nizamic, Alexander Lazovik and Marco Aiello: *Cloud ready applications composed via HTN planning.* In *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 81–89. IEEE, 2017.

[28] González, José Carlos, José Carlos Pulido and Fernando Fernández: *A three-layer planning architecture for the autonomous control of rehabilitation therapies based on social robots.* Cognitive Systems Research, 43:232–249, 2017.

[29] Heisinger, Maximilian, Mathias Fleury and Armin Biere: *Distributed cube and conquer with paracooba.* In *International Conference on Theory and Applications of Satisfiability Testing*, pages 114–122. Springer, 2020.

[30] Heule, Marijn, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo and Nils Froleyks: *International SAT competitoin 2021 results.* https://satcompetition.github.io/2021/results.html, 2021.

[31] Höller, Daniel and Gregor Behnke: *Loop Detection in the PANDA Planning System.* In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, pages 168–173, 2021.

[32] Höller, Daniel, Gregor Behnke, Pascal Bercher and Susanne Biundo: *Language Classification of Hierarchical Planning Problems.* In *ECAI*, pages 447–452, 2014.

[33] Höller, Daniel, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier and Ron Alford: *HDDL: An extension to PDDL*

for expressing hierarchical planning problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9883–9891, 2020.

[34] Höller, Daniel and Pascal Bercher: *Landmark generation in HTN planning.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11826–11834, 2021.

[35] Höller, Daniel, Pascal Bercher, Gregor Behnke and Susanne Biundo: *HTN planning as heuristic progression search.* Journal of Artificial Intelligence Research, 67:835–880, 2020.

[36] Huang, Jinbo et al.: *The Effect of Restarts on the Efficiency of Clause Learning.* In *IJCAI*, volume 7, pages 2318–2323, 2007.

[37] Hungershofer, Jan: *On the combined scheduling of malleable and rigid jobs.* In *16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213. IEEE, 2004.

[38] Kishimoto, Akihiro, Alex Fukunaga and Adi Botea: *Scalable, parallel best-first search for optimal sequential planning.* In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19, pages 201–208, 2009.

[39] Kumar, Vipin and V Nageshwara Rao: *Parallel depth first search. part ii. analysis.* International Journal of Parallel Programming, 16(6):501–519, 1987.

[40] Langley, Pat: *Systematic and nonsystematic search strategies.* In *Artificial Intelligence Planning Systems*, pages 145–152. Elsevier, 1992.

[41] Lin, Sun, Zhu Anshi, Li Bo and Fan Xiaoshi: *HTN Guided Adversarial Planning for RTS Games.* In *2020 IEEE International Conference on Mechatronics and Automation (ICMA)*, pages 1326–1331. IEEE, 2020.

[42] Luby, Michael, Alistair Sinclair and David Zuckerman: *Optimal speedup of Las Vegas algorithms.* Information Processing Letters, 47(4):173–180, 1993.

[43] Magnaguagno, Maurício C, Felipe Rech Meneguzzi and LAVINDRA DE SILVA: *HyperTensioN: A three-stage compiler for planning.* In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS), 2020, França.*, 2020.

[44] Mali, Amol Dattatraya and Subbarao Kambhampati: *Encoding HTN Planning in Propositional Logic.* In *AIPS*, pages 190–198, 1998.

[45] Mohr, Felix, Marcel Wever and Eyke Hüllermeier: *ML-Plan: Automated machine learning via hierarchical planning.* Machine Learning, 107(8):1495–1515, 2018.

[46] Muñoz-Avila, Hector and David Aha: *On the role of explanation for hierarchical case-based planning in real-time strategy games.* In *Proceedings of ECCBR-04 Workshop on Explanations in CBR*, pages 1–10. Citeseer, 2004.

[47] Nau, Dana, Yue Cao, Amnon Lotem and Hector Munoz-Avila: *SHOP: Simple hierarchical ordered planner.* In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973, 1999.

[48] Nau, Dana S: *Current trends in automated planning.* AI magazine, 28(4):43–43, 2007.

[49] Ontanón, Santiago and Michael Buro: *Adversarial hierarchical-task network planning for complex real-time games.* In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[50] Padia, Kalpesh, Kaveen Herath Bandara and Christopher G Healey: *Yarn: Generating Storyline Visualizations Using HTN Planning.* In *Graphics Interface*, pages 26–33, 2018.

[51] Ramoul, Abdeldjalil, Damien Pellier, Humbert Fiorino and Sylvie Pesty: *Grounding of HTN planning domain.* International Journal on Artificial Intelligence Tools, 26(05):1760021, 2017.

[52] Rao, V Nageshwara and Vipin Kumar: *Parallel depth first search. part i. implementation.* International Journal of Parallel Programming, 16(6):479–499, 1987.

[53] Sanders, Peter: *Lastverteilungsalgorithmen fur parallele tiefensuche.* PhD thesis, Dissertation, Karlsruhe, Universität, 1996, 1997.

[54] Sanders, Peter and Dominik Schreiber: *Decentralized online scheduling of malleable NP-hard jobs.* In *European Conference on Parallel Processing*, pages 119–135. Springer, 2022.

[55] Sanders, Peter and Jochen Speck: *Efficient parallel scheduling of malleable tasks.* In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 1156–1166. IEEE, 2011.

[56] Schreiber, Dominik: *Lilotane: A lifted SAT-based approach to hierarchical planning.* Journal of Artificial Intelligence Research, 70:1117–1181, 2021.

[57] Schreiber, Dominik, Damien Pellier, Humbert Fiorino et al.: *Tree-REX: SAT-based tree exploration for efficient and high-quality HTN planning.* In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 382–390, 2019.

[58] Schreiber, Dominik and Peter Sanders: *Scalable SAT solving in the cloud.* In *International Conference on Theory and Applications of Satisfiability Testing*, pages 518–534. Springer, 2021.

[59] Sirin, Evren, Bijan Parsia, Dan Wu, James Hendler and Dana Nau: *HTN planning for web service composition using SHOP2.* Journal of Web Semantics, 1(4):377–396, 2004.

[60] Sonmez, Ozan, Hashim Mohamed, Wouter Lammers, Dick Epema et al.: *Scheduling malleable applications in multicluster systems.* In *2007 IEEE International Conference on Cluster Computing*, pages 372–381. IEEE, 2007.

[61] Tucker, Andrew and Anoop Gupta: *Process control and scheduling issues for multiprogrammed shared-memory multiprocessors.* In *Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 159–166, 1989.

[62] Weld, Daniel S: *An introduction to least commitment planning.* AI magazine, 15(4):27–27, 1994.

[63] Xie, Kun, Yinghua Min, Dafang Zhang, Jigang Wen and Gaogang Xie: *A scalable bloom filter for membership queries.* In *IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference*, pages 543–547. IEEE, 2007.