Improved Exploration of the Bench Transition System in Parallel Greedy Best First Search

Takumi Shimoda, Alex Fukunaga

Graduate School of Arts and Sciences
The University of Tokyo
takumi35shimoda@yahoo.co.jp, fukunaga@idea.c.u-tokyo.ac.jp

Abstract

While parallelization of A* is fairly well-understood, parallelization of GBFS has been much less understood. Recent work has proposed PUHF, a parallel GBFS which restricts search to exploration of the Bench Transition System (BTS), which is the set of states that can be expanded by GBFS under some tie-breaking policy. However, PUHF causes threads to spend much of the time waiting so that only states which are guaranteed to be in the BTS are expanded. We propose improvements to PUHF which significantly reduce idle time and allow more rapid exploration of the BTS, resulting in better search performance.

1 Introduction

Parallelization of combinatorial search algorithms is important because while the overall performance of CPUs has been continued to increase steadily, most of this improvement is due to the increased number of cores per CPU, while single-thread performance has been much more gradual since the early 2000's (Hennessy and Patterson 2017). In order to maximize performance on modern CPUs, effective usage of the available cores is necessary.

In the case of cost-optimal search, parallelization of the standard A^*algorithm (Hart, Nilsson, and Raphael 1968) is somewhat well understood, and viable, practical approaches have been proposed. The optimality requirement imposes a relatively strong constraint on the set of states which must be expanded by any parallel A^*, so previous work has focused on approaches for expanding those required states while minimizing synchronization and communication overheads and avoiding expansion of non-required states (Fukunaga et al. 2017; Burns et al. 2010; Kishimoto, Fukunaga, and Botea 2013; Mukherjee, Aine, and Likhachev 2022).

On the other hand, for satisficing search where the object is to quickly find any valid solution path (regardless of path cost), the situation is quite different. Greedy Best First Search (GBFS) (Doran and Michie 1966) is a widely used satisficing search algorithm. However, the performance of straightforward parallelizations of GBFS is non-monotonic with respect to resource usage – there is a significant risk that using k threads can result in significantly worse performance using fewer than k threads. It has been shown experimentally that parallel GBFS can expand orders of magnitude more states than GBFS (Kuroiwa and Fukunaga 2019), and it has been shown theoretically that KPGBFS, a straightforward parallelization of GBFS, can expand arbitrarily many more states than GBFS (Kuroiwa and Fukunaga 2020).

One approach to avoid search behavior which drastically diverges from that of single-threaded GBFS is an algorithm portfolio (Huberman, Lukose, and Hogg 1997), where one of the threads executes A independently of the other threads, guaranteeing that at least 1 thread behaves very similarly to a single-threaded search. However, portfolio approaches are not a panacea. First, monotonic performance is not guaranteed. Suppose single-thread search algorithm A requires M bytes of RAM to find a solution. If we include A in a portfolio, but the other threads in the portfolio consume enough RAM that A has less than M bytes available, then A will fail. Second, such approaches which mostly isolate each component search algorithm in its own "sandbox" are orthogonal and complementary to tightly coupled search algorithms such as KPGBFS, as tightly coupled algorithms can be components of portfolios, e.g., a 32-thread portfolio consisting of four, 8-thread KPGBFS processes.

An issue with designing a parallel satisficing search is that unlike parallel cost-optimal search with A^*, there is no obvious set of states which a parallel satisficing search *must* explore in order to be considered a "correct" parallelization of the sequential algorithm. Recent advances in the theoretical understanding of GBFS provided one direction to pursue towards a principled approach to parallel satisficing search. Heusner, Keller, and Helmert (2017) identified the *Bench Transition System* (BTS), which corresponds to the set of all states that can be expanded by GBFS using at least one tie-breaking order. Limiting search to states in the BTS provides a natural constraint for parallel GBFS relative to single-thread GBFS. A performant parallel GBFS constrained to explore only the BTS could be a significant step forward in understanding parallel satisficing search.

Kuroiwa and Fukunaga (2020) proposed PUHF, a parallel GBFS which guarantees that only states which are in the BTS will be expanded. However, this assurance came with

a cost in performance. Since PUHF prevents expansion of any state unless it is certain that the state is in the BTS, threads can be forced to be idle while they wait for a state which is guaranteed to be in the BTS becomes available, resulting in significantly worse performance than KPGBFS.

To improve resource utilization, Kuroiwa and Fukunaga proposed SPUHF, which performs speculative search using $SOpen$/$SClosed$ lists which are separate from the main PUHF $Open$/$Closed$ lists. Although SPUHF outperforms PUHF, separate speculative search queues for unconstrained exploration is an *ad hoc* speedup method which compromises the original motivation for PUHF (parallel exploration of only the BTS).

In this paper, we extend this line of work on parallel GBFS by focusing on more effective exploration of the BTS. The contributions are: (1) an experimental analysis of PUHF and SPUHF, and (2) improvements to PUHF which significantly improve processor utilization without resorting to unconstrained, speculative search. We first experimentally analyze PUHF and show that a significant fraction of the cpu resources are being wasted compared to KPGBFS. We show that SPUHF successfully improves resource utilization, but that the contribution of SPUHF to exploration of the BTS is limited. Next, we focus on improving the rate at which the BTS is explored. We propose several improved, sufficient criteria for determining whether a state is guaranteed to be in the BTS. We propose and evaluate PUHF2, PUHF3, and PUHF4 which use these new criteria to search the BTS. We show that our new methods significantly improve upon the performance of PUHF, and show that a search which is constrained to search only the BTS can achieve performance comparable to the unconstrained KPGBFS.

In the rest of the paper, Section 2 reviews background and previous work. Section 3 experimentally analyzes PUHF and SPUHF. Section 4 proposes new, sufficient conditions for determining whether a state is in the BTS. Section 5 experimentally evaluates PUHF2/3/4 and compares them to PUHF and KPGBFS. Section 6 concludes with a discussion of our contributions and directions for future work.

2 Preliminaries and Background

State Space Topology State space topologies are defined following Heusner, Keller, and Helmert (2018).

Definition 1. A *state space* is a 5-tuple $\mathcal{S} = \langle S, succ, s_{init}, S_{goal} \rangle$, where S is a finite set of states, $succ : S \rightarrow 2^S$ is the successor function, $s_{init} \in S$ is the initial state, and $S_{goal} \subseteq S$ is the set of goal states. If $s' \in succ(s)$, we say that s' is a successor of s and that $s \rightarrow s'$ is a (state) transition. $\forall s \in S_{goal}, succ(s) = \emptyset$. A *heuristic* for $\mathcal{S}$ is a function $h : S \rightarrow \mathbb{N}_0$ and $\forall s \in S_{goal}, h(s) = 0$. A *state space topology* is a pair $\langle \mathcal{S}, h \rangle$, where $\mathcal{S}$ is a state space.

We call a sequence of state $\langle s_0, ..., s_n \rangle$ a *path* from s_0 to s_n, and denote the set of paths from s to s' as $P(s, s')$. p_i is the i th state in a path p and $|p|$ is the length of p. A *solution* of a state space topology is a path p from s_{init} to a goal state. We assume at least one goal state is reachable from s_{init}, and $\forall s \in S, s \notin succ(s)$.

For simplicity, we assume unit-cost state transitions.

Best-First Search Best-First Search (BFS) is a class of search algorithms that use an evaluation function $f : S \rightarrow \mathcal{R}$ and a tie-breaking strategy τ. BFS searches states in the order of evaluation function values (f-values). States with the same f-value are prioritized by τ. In Greedy Best-First Search (GBFS) (Doran and Michie 1966), $f(s) = h(s)$.

K-Parallel GBFS (KPGBFS) K-Parallel BFS (Vidal, Bordeaux, and Hamadi 2010) is a straightforward parallelization of BFS. All threads share a single $Open$ and $Closed$. Each thread locks $Open$ to remove a state s with the lowest f-value in $Open$, locks $Closed$ to check duplicates and add s to $Closed$, and locks $Open$ to add $succ(s)$ to Open. KPGBFS is KPBFS with $f(s) = h(s)$. The set of states explored by KPGBFS may be very different than those explored by GBFS (Kuroiwa and Fukunaga 2020).

Bounding the Behavior of Parallel BFS One natural objective to design a parallel best-first search is to try to avoid performance which is "much worse" than simply using a single-threaded search. Kuroiwa and Fukunaga (2022) proposed the following TB-*bound* criterion:

Definition 2. Let $\mathcal{T}$ be a state space topology. A search algorithm A is TB-*bounded* relative to a search algorithm B on $\mathcal{T}$ iff A does not expand any states which are not expanded by B with any tie-breaking strategy. A is TB-bounded relative to B iff A is TB-bounded on all state space topologies.

Bench Transition Systems In the special case of GBFS, it is possible to obtain TB-bounded behavior relative to single-threaded GBFS by applying the notion of bench transition systems (Heusner, Keller, and Helmert 2017, 2018).

Definition 3. Let $\langle \mathcal{S}, h \rangle$ be a state space topology with states S and $P(s) = \{p \in P(s, s') \mid s' \in S_{goal}\}$. The high-water mark of $s \in S$ is

$$hwm(s) := \begin{cases} \min_{p \in P(s)}(\max_{s' \in p} h(s')) & \text{if } P(s) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

The high-water mark of a set of states $S' \subseteq S$ is defined as

$$hwm(S') := \min_{s \in S'} hwm(s)$$

Definition 4. A state s of a state space topology $\langle \mathcal{S}, h \rangle$ is a *progress state* iff $hwm(s) > hwm(succ(s))$.

Definition 5. Let $\langle \mathcal{S}, h \rangle$ be a state space topology with a set of states S. Let $s \in S$ be a progress state.

The *bench level* of s is $level(s) = hwm(succ(s))$.

The *inner bench states* $inner(s)$ for s consist of all states $s'' \neq s$ that can be reached from s on paths on which all states $s' \neq s$ (including s'' itself) are non-progress states and satisfy $h(s') \leq level(s)$.

The *bench exit states* $exit(s)$ for s consist of all progress states s' with $h(s') = level(s)$ that are successors of s or of some inner bench state of s.

The *bench states* $states(s)$ for s are $\{s\} \cup inner(s) \cup exit(s)$.

The *bench induced by* s, denoted by $\mathcal{B}(s)$, is the state space with states $states(s)$, initial state s, and goal states

$exit(s)$. The successor function is the successor function of S restricted to $states(s)$ without transitions to s and from bench exit states $exit(s)$.

Definition 6. Let $\mathcal{T} = \langle \mathcal{S}, h \rangle$ be a state space topology with initial state s_{init}. The bench transition system $\mathcal{B}(\mathcal{T})$ of $\mathcal{T}$ is a directed graph $\langle V, E \rangle$ whose vertices are benches. The vertex set V and directed edges E are inductively defined as the smallest sets that satisfy the following properties:

1. $\mathcal{B}(s_{init}) \in V$
2. if $\mathcal{B}(s) \in V$, $s' \in exit(s)$, and s' is a non-goal state, then $\mathcal{B}(s') \in V$ and $\langle \mathcal{B}(s), \mathcal{B}(s') \rangle \in E$

Theorem 1 (Heusner, Keller, and Helmert (2017))**.** Let $\mathcal{T} = \langle \mathcal{S}, h \rangle$ be a state space topology with set of states S and bench transition system $\langle V, E \rangle$. For each state $s \in S$, it holds that $s \in \mathcal{B}(s')$ for some $\mathcal{B}(s') \in V$ iff there is a tie-breaking strategy with which GBFS expands s.

The bench transition system (BTS) defines the set of all states which are candidates for expansion by BFS with some tie-breaking strategy. If parallel BFS expands states which are outside of the BTS, then it is not TB-bounded relative to GBFS by Theorem 1, and vice versa.

2.1 PUHF: A BTS-Constrained Parallel GBFS

Definition 7. A search algorithm is BTS*-constrained* if it expands only states which are in the BTS.

In the rest of this paper, we will say "BTS-constrained" instead of "TB-bounded relative to sequential GBFS", as they are equivalent, the former is more concise, and this paper focuses only on parallel GBFS.

During search, we can identify states in $Open$ which are *guaranteed* to be in the BTS.

Theorem 2 (Kuroiwa and Fukunaga (2020))**.** When s is expanded by GBFS with some tie-breaking strategy, if $h(s') \leq h(s)$ where $h(s') = \min_{s'' \in succ(s)} h(s'')$, s' is also expanded by GBFS with some tie-breaking strategy.

Parallel Under High-water mark First (PUHF) (Algorithm 1) is a BTS-constrained parallel GBFS which only expands states marked as $certain$ (Kuroiwa and Fukunaga 2020). In Algorithm 1, k is the number of threads and s_i is the node expanded by thread i. States are marked $certain$ under 2 conditions: (1) states which satisfy the condition in Theorem 2 when generated (lines 21-22 are marked $certain$, and (2) if no threads are expanding a state, then the top of $Open$ is marked $certain$ (lines 6-9).

Theorem 3 (Kuroiwa and Fukunaga (2020))**.** PUHF is BTS-constrained.

Speculative PUHF (SPUHF) When a thread is available but no available states are marked as $certain$, PUHF will not expand anything and will waste that cycle (Algorithm 1 line 13). To avoid this waste, SPUHF, an extension of PUHF, adds separate speculative open/closed lists, $SOpen$ and $SClosed$ (Kuroiwa and Fukunaga 2020). All states not marked $certain$ are inserted into $SOpen$. When a thread does not have a $certain$ state available for expansion, it expands a state from $SOpen$, and its successors are inserted into $SOpen$ and $SClosed$. If the speculative expansion finds a goal, the solution path is returned. Since $SOpen$ and $SClosed$ are separate from $Open$ and $Closed$, $Open$ and $Closed$ remain BTS-constrained. SPUHF safely shares work between the BTS-constrained search using $Open$/$Closed$ and the speculative search using $SOpen$/$SClosed$, using a *evaluation cache* which caches h-values of all evaluated states. We say that a *speculative cache hit* occurs when the BTS-constrained search generates a state to evaluate (computation of $h(s'_i)$ in Algorithm 1, line 21) and finds it in the cache.

To increase the likelihood that speculative search evaluates states which later yield speculative cache hits, the expansion priority for $SOpen$ is (in decreasing order): (1) successors of states expanded from $Open$, if any (according to h-value and then the tie-breaking strategy τ), (2) all other states according to h-value, then τ.

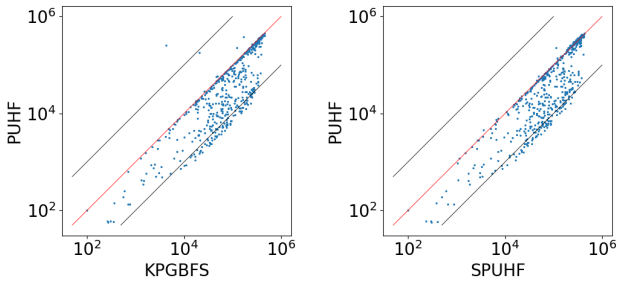
Algorithm 1: Parallel Under High-water mark First

```
1:  parent(s_init) ← NULL; certain(s_init) ← true
2:  Open ← {s_init}, Closed ← {s_init}; ∀i, s_i ← NULL
3:  for i ← 0, ..., k − 1 in parallel do
4:      loop
5:          lock(Open)
6:          if ∀j, s_j = NULL then
7:              if Open = ∅ then unlock(Open); return NULL
8:              if certain(top(Open)) = false then
9:                  certain(top(Open)) ← true
10:         if certain(top(Open)) = true then
11:             s_i ← top(Open); Open ← Open \ {s_i}
12:         unlock(Open)
13:         if s_i = NULL then continue
14:         if s_i ∈ S_goal then return s_i
15:         for s'_i ∈ succ(s_i) do
16:             lock(Closed)
17:             if s'_i ∉ Closed then
18:                 Closed ← Closed ∪ {s'_i}
19:                 unlock(Closed)
20:                 parent(s'_i) ← s_i
21:                 if h(s'_i) = min_{s∈{s_i}∪succ(s_i)} h(s) then
22:                     certain(s'_i) ← true
23:                 else
24:                     certain(s'_i) ← false
25:                 lock(Open); Open ← Open ∪ {s'_i}; unlock(Open)
26:             else
27:                 unlock(Closed)
28:         s_i ← NULL
```

3 Experimental Analysis of PUHF and SPUHF

We experimentally analyze the behavior of PUHF and SPUHF. As a baseline, we use the unconstrained parallel GBFS algorithm, KPGBFS.

Experimental Settings All experiments in Sections 3 and 5 use the satisficing instances of the Autoscale-21.11 benchmark set (42 STRIPS domains, 30 instances/domain, 1260



(a) PUHF vs. KPGBFS (b) PUHF vs. SPUHF

Figure 1: State evaluation rate comparison (states/second), 16 threads). Each point represents a problem instance. diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

total instances) (Torralba, Seipp, and Sievers 2021), an improved benchmark suite based on the IPC classical planning benchmarks. All algorithms use the FF heuristic (Hoffmann and Nebel 2001). Each run has a run time limit of 5 minutes. The experiments were executed on Amazon EC2 c5a instances on Amazon (AMD EPYC 7R32 2.8GHz processor, 16 threads / 64GB total RAM, 8 threads / 32GB RAM, 4 threads/ 16GB RAM) without hyperthreading.

PUHF vs. KPGBFS: Is the BTS a Good Set of States to Explore? First, we compare the search performance of PUHF vs. KPGBFS. The coverage on the 1260 Autoscale benchmark instances for 4/8/16 threads was 522/550/579 for KPGBFS, and 497/507/523 for PUHF, so the PUHF coverage was significantly lower than that of KPGBFS. *However, coverage alone does not tell the whole story*. Figure 6 (first column) compares the number of states expanded by PUHF vs. KPGBFS. This shows that on the problems which were solved by both PUHF and KPGBFS, PUHF tends to expand somewhat fewer states than KPGBFS. On the other hand, the comparison of run times in Figure 7 (first column) shows that PUHF is clearly *slower* than KPGBFS. Thus, from the perspective of *where* to focus the search, exploration of the BTS is a reasonable search strategy, but PUHF explores these states too slowly to be a viable alternative to KPGBFS.

Is PUHF Efficiently Using Parallel Resources? Next, we investigate how much time PUHF is wasting waiting for a $certain$ state to become available. Since the heuristic computation and state expansion account for the vast majority of the runtime of all of the parallel GBFS variants studied in this paper, comparisons of the *evaluation rates* (number of states evaluated/second) gives a good indication as to how much computational resources are wasted by being idle.

The average state evaluation rate (geometric mean) for 4/8/16 threads over all 1260 benchmark instances was 28707/48371/78817 for KPGBFS, and 18087/23749/29265 for PUHF. Figure 1a compares evaluation rates for 16 threads on problems which were solved by all methods. The evaluation rate of PUHF is significantly lower than KPGBFS. Thus, in PUHF, many threads spent much of the time waiting for a $certain$ state to become available for expansion, causing PUHF to significantly underperform KPGBFS.

Does SPUHF Improve Parallel Resource Utilization? As SPUHF always performs speculative expansion rather than waiting idly for a $certain$ state to become available, SPUHF fully utilizes all cpu resources, like KPGBFS. Figure 1b compares the state evaluation rates of SPUHF and PUHF for 16 threads. The evaluation rates for SPUHF are significantly higher than PUHF, showing that the speculative search allows SPUHF to utilize more resources than PUHF.

How much Does Speculative Search Contribute to Exploration of the BTS? In SPUHF, although the BTS-constrained portion of the search (i.e., the PUHF part which only expands $certain$ states) and the speculative search are isolated so that they do not share $Open/Closed$, they share a state evaluation cache. The speculative search contributes to the exploration of the BTS by filling the evaluation cache so that the BTS-constrained search can explore the BTS faster by using the cached h-values.

The speculative cache hit rate is the number of cache hits divided by the number of states generated by the speculative search. This hit rate is an upper bound on the contribution of the speculative search to BTS exploration, since only some fraction of the hits will satisfy the criterion for determining a state to be $certain$ (Algorithm 1 lines 21-22). On the Autoscale-21.11 benchmark set, 16 cores 5 minute limit/run, the speculative cache hit rate was 0.19. In other words, at most 19% of the states evaluated by the speculative search contributed to speeding up the BTS-constrained search.

Search Performance: Why Does SPUHF Outperform PUHF? SPUHF coverage for 4/8/16 threads was 519/540/568, significantly higher than PUHF. One possibility is that the higher coverage is achieved because the speculative search allows SPUHF to explore the BTS much more effectively than PUHF. However, as discussed above, only a small fraction of the states explored by speculative search are likely to be contributing to the exploration of the BTS. We instrumented SPUHF to identify whether solution plans were found by the speculative search (i.e., the goal state was generated while expanding a state in $SOpen$, or the non-speculative search (the goal was generated while expanding a state in $Open$ in Algorithm 1, line 14). On 16 threads, out of 568 instances solved by SPUHF, 158 were solved by the speculative search, and 410 were found by the non-speculative search. As PUHF without speculation solves 479 instances, this result, in combination with the above analysis of speculative cache hits, strongly indicates that the high coverage achieved by SPUHF is mostly due to speculative search exploring according to its own policy, and not because of its contribution to exploration of the BTS.

4 Improved PUHF

The previous section showed that PUHF has a significantly worse state evaluation rate than KPGBFS, and that although SPUHF successfully increases evaluation rate, much of the additional work being done by SPUHF is not necessarily contributing to exploration of the BTS. Therefore, we seek a better algorithm with higher state evaluation rate than PUHF but only expands states which are in the BTS.

The main idea in PUHF is to only expand states which are in the BTS, and to enforce this BTS-constrained behavior by marking states which are guaranteed to be in the BTS as *certain*. While an ideal BTS-constrained parallel GBFS would correctly identify all states in the BTS and mark them as *certain* as soon as they are generated, implementing such an ideal behavior seems nontrivial.

Instead, PUHF relies on a sufficient but not necessary condition for inclusion in the BTS, and marks as *certain* the states which are guaranteed to be in the BTS according to this sufficient criterion in Theorem 2. As a consequence, many states which are generated in line 15 which are actually in the BTS might not satisfy the check in line 22 for the criterion in Theorem 2. Such states will eventually be marked *certain* in line 9 and subsequently expanded, but this may be after their evaluation has been unnecessarily delayed for a long time.

Thus, we propose several sufficient criteria for marking a state as *certain* which seek to improve upon Theorem 2.

4.1 PUHF2

Our first improvement is motivated by the example in Figure 2. Suppose s_{init} has been expanded, and its children $s_{1,1}$-$s_{1,4}$ have been generated. All new states are initially marked as not *certain*. PUHF marks only $s_{1,1}$ as *certain*, and while $s_{1,1}$ is being expanded, $s_{1,2}$-$s_{1,4}$ remain in $Open$. As they are not *certain*, each of $s_{1,2}$-$s_{1,4}$ are expanded one at a time (only when they satisfy Algorithm 1 line 9), after $s_{1,1}$ is expanded. However, this is inefficient because as soon as expansion of $s_{1,1}$ is complete, it is clear that $s_{1,2}$-$s_{1,4}$ are all in the BTS (all of $s_{1,2}$-$s_{1,4}$ will be expanded by some tie-breaking order), so waiting idly while any of $s_{1,2}$-$s_{1,4}$ are in $Open$ is wasteful. This inefficiency is due to the fact that when $s_{1,1}$-$s_{1,4}$ were generated, only $s_{1,1}$ could be marked *certain*, and $s_{1,2}$-$s_{1,4}$ were left marked not *certain*, and their *certain* status could not be updated until Algorithm 1 line 9 applies, one at a time.

Algorithm 2 shows PUHF2. The main difference between PUHF and PUHF2 is the point in time at which it is determined whether a generated state is *certain* or not. PUHF determines whether states are *certain* immediately after evaluating them (Algorithm 1, line 21-24). In contrast, PUHF2 determines whether a state n is *certain* at the point when n

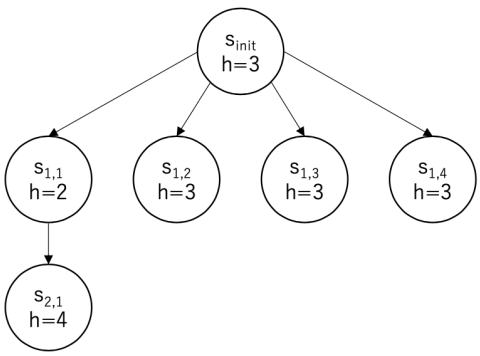


Figure 2: Example for PUHF2

Algorithm 2: PUHF2

```
1:  parent(s_init) ← NULL; certain(s_init) ← true
2:  Open ← {s_init}, Closed ← {s_init}; ∀i, s_i ← NULL
3:  for i ← 0, ..., k − 1 in parallel do
4:      loop
5:          lock(Open)
6:          if Open = ∅ then
7:              if ∀j, s_j = NULL then unlock(Open); return NULL
8:          else if h(top(Open)) ≤ min_{0≤j<k} h(s_j) then
9:              certain(top(Open)) ← true
10:         if certain(top(Open)) = true then
11:             s_i ← top(Open); Open ← Open \ {s_i}
12:         if PUHF3 then                                  ▷ see Section 4.2
13:             for s ∈ Open s.t.
                  h(s) = min_{0≤j<k} h(s_j) and certain(s) = false do
14:               certain(s) ← true
15:         unlock(Open)
16:         if s_i = NULL then continue
17:         if s_i ∈ S_goal then return s_i
18:         for s'_i ∈ succ(s_i) do
19:             lock(Closed)
20:             if s'_i ∉ Closed then
21:                 Closed ← Closed ∪ {s'_i}
22:                 unlock(Closed)
23:                 parent(s'_i) ← s_i
24:                 certain(s'_i) ← false
25:                 if PUHF4 then                          ▷ see Section 4.3
26:                     if h(s'_i) = min_{s∈{s_i}∪succ(s_i)} h(s) then
27:                         certain(s'_i) ← true
28:                 children(s_i) ← children(s_i) ∪ {s'_i}
29:             else
30:                 unlock(Closed)
31:         lock(Open)
32:         for s'_i ∈ children(s_i) do
33:             Open ← Open ∪ {s'_i}
34:         unlock(Open)
35:         s_i ← NULL
```

is being considered for evaluation (Algorithm 2, line 9). If the h-value of the top state in $Open$ is less than the h-value of all states currently being expanded in any other thread $(s_j, 0 \leq j < k)$, then $top(Open)$ is marked *certain* (in the case of idle threads, i.e., $s_j = NULL$, the h-value is treated as ∞ $h(s_j) = \infty$).

In the example in Figure 2, PUHF2 first expands $s_{1,1}$. While $s_{1,1}$ is being expanded, $s_{1,2}$-$s_{1,4}$ wait in $Open$. However, as soon as $s_{1,1}$ has been expanded, $s_{1,2}$-$s_{1,4}$ can all be expanded simultaneously.

Lemma 1. In PUHF2 (as well as PUHF and KGBFS), after a state s such that $hwm(s) \neq \infty$ is first inserted into $Open$, then at least one state in either $Open$ or the set of states currently being expanded by some thread has an h-value less than or equal to $hwm(s)$.

Proof. Suppose the opposite is true. Then it is not possible to reach a goal from s via states with h-values less than or equal to $hwm(s)$, contradicting the definition of $hwm(s)$. □

Theorem 4. PUHF2 is BTS-constrained.

Proof. This follows the proof of Theorem 3 by Kuroiwa and Fukunaga (2020). Let $\mathcal{T}$ be a state space topology and its bench transition system $\mathcal{B}(\mathcal{T}) = \langle V, E \rangle$. Let $S' = \{s \in \mathcal{B}(s') \mid \mathcal{B}(s') \in V\}$. By Theorem 1, PUHF2 is BTS-constrained iff it never expands states not in S'.

Since PUHF2 only expands *certain* states from *Open*, we show that all *certain* states in *Open* are in S'. The proof is by induction over states marked as *certain*. If $s_{min} = s_{init}$, $s_{min} \in S'$. Suppose that PUHF2 has not expanded any state not in S'. Let $s_{min} = top(Open)$ in line 9. s_{min} is marked as *certain* here if $h(s_{min}) \leq min_{0 \leq j < k} h(s_j)$.

Assume that $s_{min} \notin S'$. p, the parent of s_{min}, is marked as *certain* since p is expanded in S' by the induction hypothesis. If p is a progress state, $h(s_{min}) > hwm(succ(p))$. Otherwise, $h(s_{min}) > hwm(succ(s'))$ where $\mathcal{B}(s')$ is the bench p belongs to. Although it is possible that p is in multiple benches, we can determine $\mathcal{B}(s')$ by the path PUHF followed from s_{init} to p. In any case, there is a path to a goal whose high-water mark is lower than $h(s_{min})$, so by Lemma 1, there are states in *Open* or currently being expanded with h-value less than $h(s_{min})$. This contradicts that s_{min} has the lowest h-value in *Open* and that $h(s_{min}) \leq min_{0 \leq j < k} h(s_j)$. Therefore $s_{min} \in S'$.

□

4.2 PUHF3

The criterion for determining whether a state is *certain* can be further improved. Suppose we run PUHF2 with 2 threads to the state space in Figure 3, and thread 1 expands s_{init}, then $s_{1,1}$. Next, any of $s_{1,2}$-$s_{1,4}$ could be expanded, but suppose threads 1 and 2 expand $s_{1,2}$ and $s_{1,3}$, and $s_{2,2}$ is generated and marked *certain*, so thread 1 starts to expand $s_{2,2}$. Although $s_{1,4}$ is clearly in the BTS because they could have been selected before $s_{1,2}$ and $s_{1,3}$ by another tiebreak order, thread 2 must wait until thread 1 finishes expanding $s_{2,1}$.

In PUHF2, when there are multiple available *certain* states $\{n_1, \ldots, n_m\}$ with the same h-value $h(n_1) = \ldots h(n_m)$ in Algorithm 2 line 9, one of them is selected as s_i according to a tie-breaking policy. All of the other states $\{n_1, \ldots, n_m\} \setminus s_i$ must also be in the BTS, and should be marked *certain*, as any of them could have been selected as s_i by another tiebreak order.

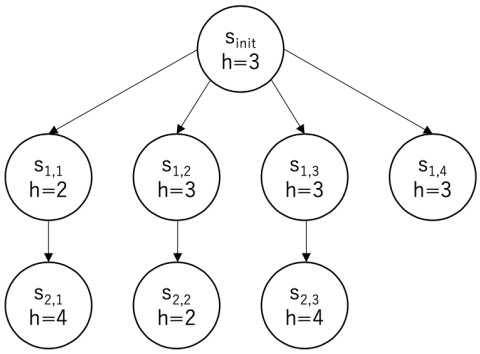


Figure 3: Example for PUHF3

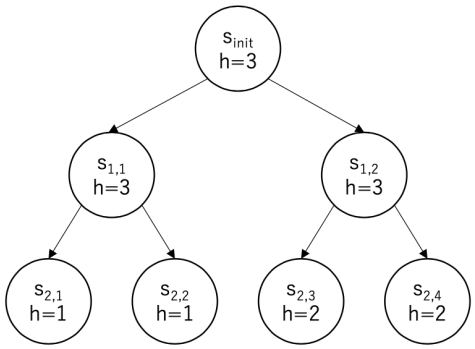


Figure 4: Example for PUHF4

PUHF3 is a modification to PUHF2 after line 12 in Algorithm 2. This marks all states in *Open* with h-value $h(s) = min_{0 \leq j < k} h(s_j)$ as *certain*. All states marked *certain* by PUHF2 are also marked *certain* by PUHF3. Since PUHF2 is BTS-constrained, and all of the states additionally marked *certain* by PUHF3 are in the BTS, PUHF3 is BTS-constrained.

4.3 PUHF4

Although PUHF3 and PUHF2 are designed to determine more states as *certain* earlier than PUHF, PUHF3 does not dominate PUHF. In Figure 4, with more than 2 threads, $s_{2,3}$, $s_{2,4}$ are *certain* according to the PUHF criterion when they are generated, but are not *certain* according to the PUHF2 and PUFH3 criteria while $s_{2,1}$, $s_{2,2}$ are being expanded.

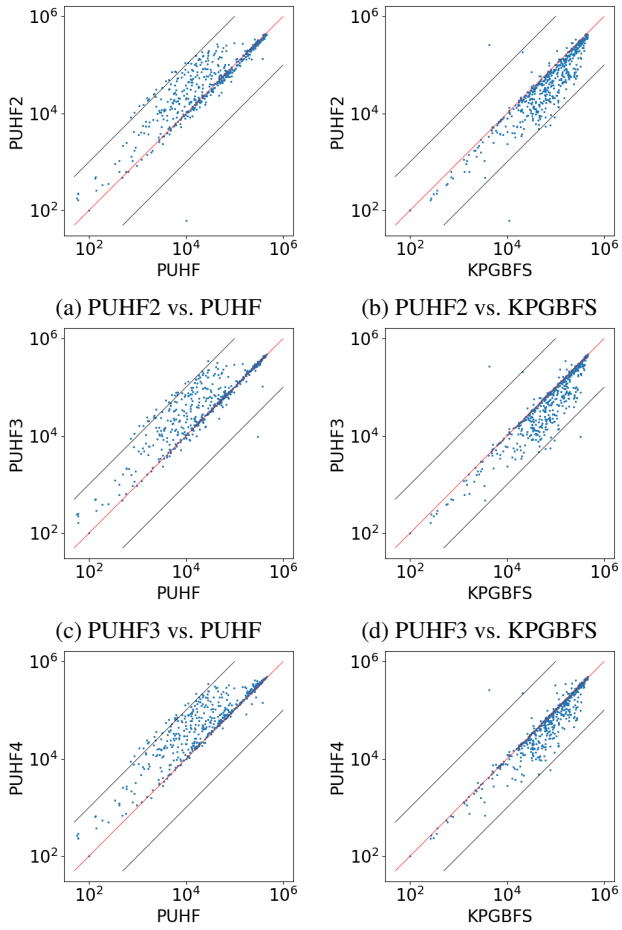
PUHF4 adds the *certain* criterion for PUHF to PUHF3. In addition to inserting the PUHF3 check to PUHF2 as explained in 4.2, PUHF4 inserts the PUHF *certain* criterion check after line 25 in Algorithm 2. In line 8, PUHF4 breaks ties to favor states which are marked *certain*. As PUHF, PUHF2, and PUHF3 are all BTS-constrained, PUHF4 is also BTS-constrained.

5 Experimental Evaluation of PUHF2, PUHF3, PUHF4

We experimentally compare our new methods with three baselines, KPGBFS, PUHF, and SPUHF on $k = 4, 8, 16$ threads. See Section 3 for details on experimental settings.

	Evaluation Rate			Coverage		
GBFS	8462 (1 thread)			470 (1 thread)		
#threads	4	8	16	4	8	16
KPGBFS	28707	48371	78817	522	550	579
PUHF	18087	23749	29265	497	507	523
SPUHF	28171	46936	77164	519	540	568
PUHF2	23178	34454	45702	518	538	567
PUHF3	24599	36477	49838	523	547	562
PUHF4	25973	40588	59682	519	536	562

Table 1: Comparison of evaluation rates (states/second, geometric mean) and coverage (# solved out of 1260) on Autoscale planning benchmark set.



(a) PUHF2 vs. PUHF (b) PUHF2 vs. KPGBFS

(c) PUHF3 vs. PUHF (d) PUHF3 vs. KPGBFS

(e) PUHF4 vs. PUHF (f) PUHF4 vs. KPGBFS

Figure 5: State evaluation rate comparison (states/second), 16 threads.diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

Evaluation Rates The average evaluation rates r over all instances are shown in Table 1. Figure 5 compares the state evaluation rates of the algorithms. Overall, PUHF2, PUHF3, and PUHF4 have a significantly higher evaluation rate than PUHF, and $r_{\text{PUHF4}} > r_{\text{PUHF3}}$, and $r_{\text{PUHF3}} > r_{\text{PUHF2}}$, showing that expanding the sufficient criterion for marking a state *certain* results in increased state evaluation rate.

The evaluation rate of PUHF2-4 *relative to KPGBFS* (i.e., $r_{\text{PUHFx}}/r_{\text{KPGBFS}}$) decreases as the number of threads k increases. For example, for PUHF3, the relative expansion rate is 0.86 for k=4 threads, 0.75 for k=8, and 0.63 for k=16. This is because as k increases, it becomes more likely that there are insufficient *certain* states marked by PUHF2-4 to occupy the available threads. Finally, PUHF2-4 all have a significantly lower evaluation rate than KPGBFS, even for k=4, so there is opportunity for further improvement.

Search Performance Table 1 shows the coverage (#instances solved) achieved by the parallel search algorithms. Figures 6 and 7 compare the # of states expanded and runtimes of PUHF2-4 vs. KPGBFS for k=4, 8, 16 threads.

The new methods (PUHF2, PUHF3, and PUHF4) achieved significantly higher coverage than PUHF, and are comparable to KPGBFS and SPUHF. Figure 6 shows that on the problems solved by both algorithms, the PUHF variants tend to expand fewer states than KPGBFS. Figure 7 shows that PUHF2-4 are significantly faster than PUHF, and are comparable to KPGBFS. In particular, PUHF3 had the highest coverage for k=4 (523 vs. KPGBFS 522), and slightly lower coverage for k=8 (547 vs. KPGBFS 550).

With k=16, a gap between KPGBFS and PUHF2-4 becomes apparent in both the coverage (Table 1) and run time (Figure 7), although the states expanded by PUHF2-4 continue to be somewhat lower than KPGBFS. The slowdown of PUHF2-4 as the number of threads increases is most likely caused by the degradation in relative expansion rate compared to KPGBFS, as discussed above.

With k=16 threads, PUHF2/3/4 solved 27/24/20 problems which were not solved by KPGBFS, and KPGBFS solved 39/41/37 problems not solved by PUHF2/3/4. With k=16 threads, PUHF2/3/4 solved 31/23/23 problems which were not solved by SPUHF, and SPUHF solved 32/29/29 problems not solved by PUHF2/3/4.

Although PUHF4 has the highest evaluation rate among the pure BTS-constrained algorithms, PUHF4 had lower coverage than PUHF2 and PUHF3 for k=8. This shows that rapid (high evaluation rate) exploration of the BTS by itself does not necessarily lead to better search performance, and that a better search strategy *within* the BTS can play an important role.

Overall, these results show that a parallel GBFS constrained to expand only states in the BTS can be a viable alternative to a standard, unconstrained parallel GBFS.

6 Discussion and Conclusion

Although previous work showed that it was possible to constrain parallel GBFS to expand only states that can be expanded by sequential GBFS with some tie-breaking order, we showed that PUHF pays a high price for this constraint. The state evaluation rate of PUHF is much lower than KPGBFS, a straightforward, unconstrained parallelization of GBFS, resulting in poor search performance. While SPUHF (Kuroiwa and Fukunaga 2020) uses speculation to boost the performance of PUHF, we showed that the performance improvement compared to PUHF is largely due to the unconstrained speculative exploration, and not because of the speedup of the BTS-constrained search enabled by evaluation caching.

We proposed PUHF2, PUHF3, and PUHF4, which use improved, sufficient criteria for determining that a state is in the BTS. We showed that a parallel GBFS constrained to expand only states in the BTS can achieve performance comparable to unconstrained KPGBFS, without the aid of unconstrained methods such as speculation. This is an important step in a principled approach to developing parallel GBFS algorithms, as we now have a viable building block with constrained search behavior which can incorporated into more complex parallel algorithms, e.g., portfolios, in future work.

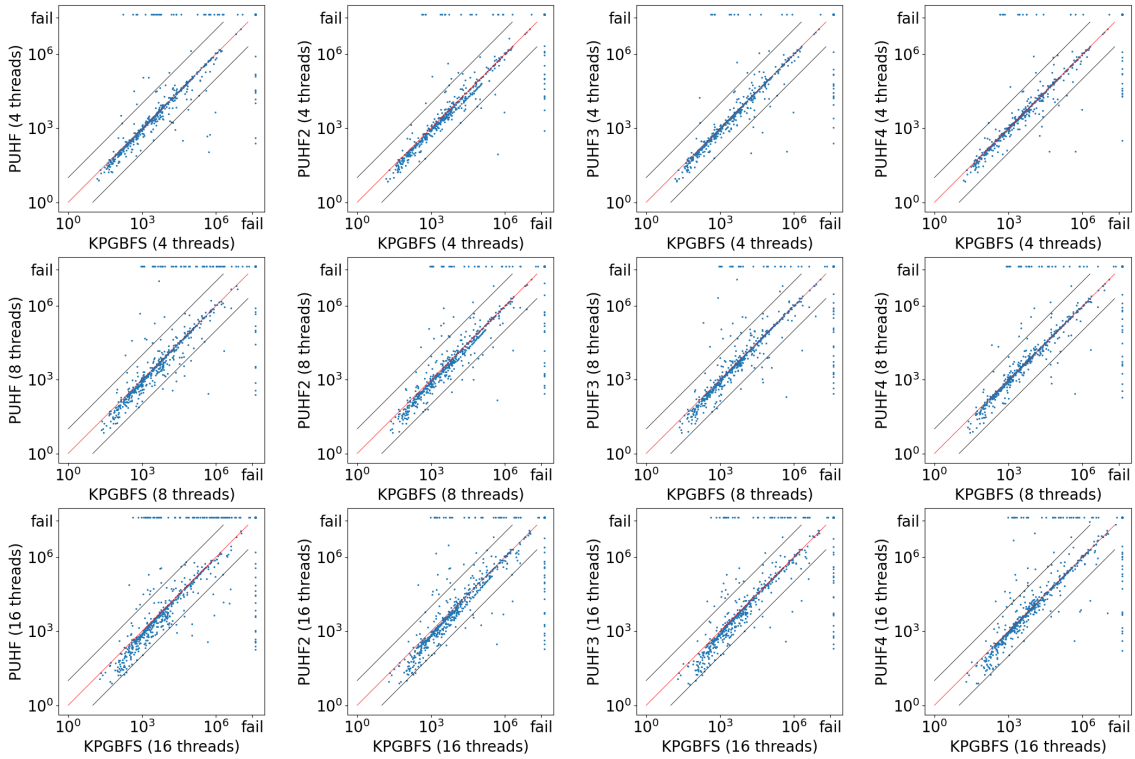


Figure 6: Number of states expanded: PUHF and PUHF2-4 vs. KPGBFS. 5 min. time limit."fail"= out of time/memory, diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

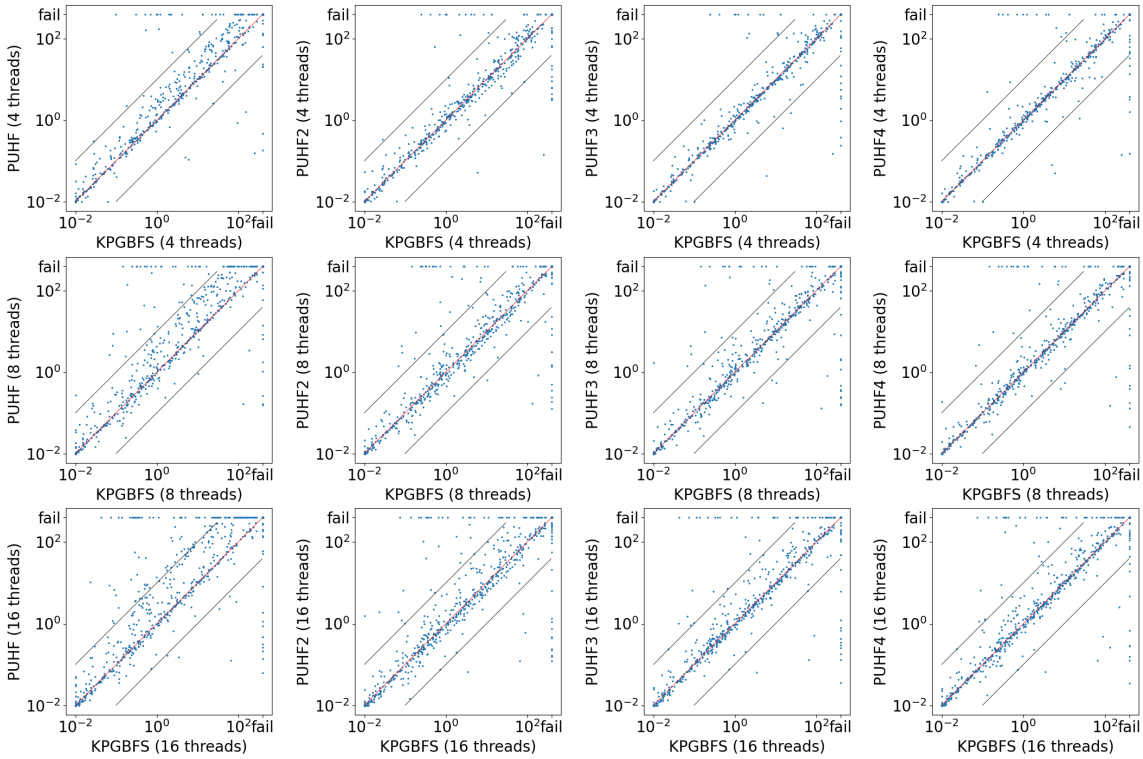


Figure 7: Run time (seconds) comparison: PUHF and PUHF2-4 vs. KPGBFS. 5 min. time limit, "fail"= out of time/memory, diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

Acknowledgments

This research was supported by JSPS KAKENHI Grant 20K11932.

References

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-First Heuristic Search for Multicore Machines. *J. Artif. Intell. Res.*, 39: 689–743.

Doran, J.; and Michie, D. 1966. Experiments with the Graph Traverser Program. In *Proc. Royal Society A: Mathematical, Physical and Engineering Sciences*, volume 294, 235–259.

Fukunaga, A.; Botea, A.; Jinnai, Y.; and Kishimoto, A. 2017. A Survey of Parallel A*. *CoRR*, abs/1708.05296.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2): 100–107.

Hennessy, J. L.; and Patterson, D. A. 2017. *Computer Architecture - A Quantitative Approach, 6th Edition*. Morgan Kaufmann. ISBN 978-0128119051.

Heusner, M.; Keller, T.; and Helmert, M. 2017. Understanding the Search Behaviour of Greedy Best-First Search. In *Proc. SOCS*, 47–55.

Heusner, M.; Keller, T.; and Helmert, M. 2018. Best-Case and Worst-Case Behavior of Greedy Best-First Search. In *Proc. IJCAI*, 1463–1470.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Intell. Res.*, 14: 253–302.

Huberman, B.; Lukose, R. M.; and Hogg, T. 1997. An Economics Approach to Hard Computational Problems. *Science*, 275(5296): 51–54.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2013. Evaluation of a simple, scalable, parallel best-first search strategy. *Artif. Intell.*, 195: 222–248.

Kuroiwa, R.; and Fukunaga, A. 2019. On the Pathological Search Behavior of Distributed Greedy Best-First Search. In *Proc. ICAPS*, 255–263.

Kuroiwa, R.; and Fukunaga, A. 2020. Analyzing and Avoiding Pathological Behavior in Parallel Best-First Search. In *Proc. ICAPS*, 175–183. AAAI Press.

Mukherjee, S.; Aine, S.; and Likhachev, M. 2022. ePA*SE: Edge-Based Parallel A* for Slow Evaluations. In Chrpa, L.; and Saetti, A., eds., *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, 136–144. AAAI Press.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In *Proc. ICAPS*, 376–384. AAAI Press.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Adaptive K-Parallel Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. In *Proc. SOCS*, 100–107.