Thread-Ordered Parallel Greedy Best First Search

Dawson Tomasz[1], **Rishi Veerapaneni**[2], **Maxim Likhachev**[2], **Richard Valenzano**[1]

[1] Toronto Metropolitan University
[2] Carnegie Mellon University
dawson.tomasz@proton.me, {rveerapa, maxim}@cs.cmu.edu, rick.valenzano@torontomu.ca

Abstract

While parallelizing Greedy Best-First Search (GBFS) can accelerate planning, existing approaches offer no runtime guarantees: they are not necessarily faster than sequential GBFS, and counterintuitively, adding more threads can possibly degrade runtime. To address this unpredictable scaling, we introduce Thread-Ordered Parallel-GBFS (TOP-GBFS). Inspired by MonoBeam — which guarantees monotonically improving solution costs as beam width increases — TOP-GBFS is a novel parallelization designed to ensure that search time never increases as thread count grows. Under standard assumptions, we prove that TOP-GBFS bounds the total number of expansions to a constant factor of sequential GBFS and guarantees monotonically non-increasing runtimes. We empirically validate this monotonicity and demonstrate that TOP-GBFS remains competitive with existing parallel GBFS implementations on standard PDDL benchmarks.

1 Introduction

Much of the performance increase of compute in recent years has been due to more cores while sequential execution speeds have improved more gradually (Hennessy and Patterson 2011). And so, consistent and scalable parallel Best First Search (BFS) is increasingly important in order to fully make use of modern multi-core machines in heuristic search.

Optimal BFS algorithms like A* have been successfully parallelized with quite consistent scaling as more threads are used (Kishimoto, Fukunaga, and Botea 2013; Phillips, Likhachev, and Koenig 2014). This has proved more difficult when parallelizing *satisficing* algorithms (Kuroiwa and Fukunaga 2019, 2020; Shimoda and Fukunaga 2023) — *ie.* those that seek to find a solution quickly at the expense of any cost bounds — such as *Greedy Best First Search (GBFS)* (Doran and Michie 1966). This is because the OPEN and CLOSED lists of the different threads of a GBFS parallelization can "pollute" or "block" each other thereby pushing the overall search to consider parts of the state-space that sequential GBFS never would. As a result, adding a new thread can completely change the course of the search. This means that in some cases, having additional cores may degrade performance by an arbitrary amount when compared to using fewer threads (Kuroiwa and Fukunaga 2019, 2020).

Previous analysis of this behavior (Kuroiwa and Fukunaga 2019) thus identified two desirable properties that parallelizations of GBFS should aim to achieve: 1) only expand states which sequential GBFS could possibly expand, and 2) have a constant factor bound on the number of additional expansions done compared to standard single-core GBFS. In this work, we consider a different desirable property: that the runtime of a GBFS parallelization is *monotonically non-increasing* in the number threads made available to the algorithm. This property is inspired by the single-core beam-search algorithm *MonoBeam* (Lemons et al. 2022) which guarantees monontonically non-increasing solution costs as the beam width increases. In the context of GBFS parallelizations, we view this property as a form of stability and predictability, since it means that all available hardware resources can be safely used with the confidence that the additional threads will not incur a performance penalty.

To that end, we introduce a novel parallelization of GBFS called *Thread-Ordered Parallel-GBFS (TOP-GBFS)* which explicitly aims at achieving monotonic runtimes. We formally analyze the algorithm using a model of its parallel behavior and show that it expands a bounded number of states compared to a single-core GBFS, and that the runtime is monotonically non-increasing in the number of threads used. Finally, we empirically evaluate TOP-GBFS to show it is competitive with existing GBFS parallelizations, while also exhibiting the desired monotonicity property in practice.

2 Background

In this section we briefly cover state-spaces, heuristic search algorithms, existing parallel GBFS algorithms, and the pitfalls which can beset parallel GBFS.

2.1 State-spaces

A *state-space task* T is a 4-tuple $T = \langle S, s_I, succ, S_G \rangle$, where S is a set of *states*, s_I is an *initial state*, $S_G \subseteq S$ is a set of *goal states*, and $succ : S \mapsto 2^S$ is a *successor generator function*. The objective of a search problem is to find a plan, which is a sequence of states $(s_0, \ldots, s_k)$ where $s_k \in S_G$, $s_0 = s_I$, and $s_i \in succ(s_{i-1})$ for every $0 < i \leq k$. As our focus is on satisficing search, we ignore plan and transition costs in this work.

An algorithm for finding a plan to a given state-space task is called a *search algorithm*. We focus on algorithms that use a *heuristic function* to help guide the search. Formally, a heuristic function is defined as $h : S \mapsto \mathbb{R}^{\geq 0}$, and we refer to the pair $\langle T, h \rangle$ of a state-space task T and heuristic h as a *state-space topology*. Typically, a heuristic estimates the distance from a state to a goal state. We use the term *sequential algorithm* to refer to such an algorithm that runs on a single CPU core or thread, and a *parallel algorithm* as one that may use multiple cores or threads.

2.2 Sequential Search Algorithms

Best First Search (BFS) is a family of search algorithms which uses a priority queue, the *open list*, ordered by an evaluation function $f : S \mapsto \mathbb{R}^{\geq 0}$ and tie-breaking scheme $\tau : S \mapsto \mathbb{R}$ to conduct a search. At each search step, the state in the open list, s, with the lowest f-value (with ties broke by τ) will have its successors added to the open list. s will then be removed from the open list and put into the *closed list*, which records states that have already been visited[1].

Greedy Best First Search (GBFS) is a special case of BFS where $f(s) = h(s)$ (Doran and Michie 1966). When a state s has its successor set generated and placed into the open list, we say s has been *expanded*. An expansion is a single search step in a sequential search and the *sequence of expansions* is the list of states in the order they were expanded during the search. Note that the open list and closed list are referred to as OPEN and CLOSED, respectively, in the text below.

In standard descriptions of Best-First Search, ties between the f-cost of states in OPEN can be broken using a tie-breaking scheme τ, or otherwise arbitrarily. For our analysis, we will further require that the search algorithm be *deterministic*, meaning there are no arbitrary selections and so the combination of the evaluation function f and tie-breaking scheme τ, is guaranteed to produce a single, unique expansion sequence. This can be achieved if the tie-breaking scheme τ imposes a strict total order over all generated states. Specifically for any two distinct states s_1 and s_2 where $f(s_1) = f(s_2)$, τ ensures one state is always strictly preferred to the other. We note that deterministic tie-breaking is trivially achieved using LIFO or FIFO. As such, all sequential search algorithms in this paper are assumed to be deterministic.

Beam search is an incomplete BFS algorithm which is parameterized with a *beam width* $w \geq 1$ (Bisiani 1992). At each search step, beam search will generate all successors of the current OPEN and take the "best" w states of them (based on f-value) to be included in the next OPEN. MonoBeam is a variant of beam search which guarantees monotonic solution costs as w increases (Lemons et al. 2022). MonoBeam serves as an inspiration for this work and is detailed in Section 3.

2.3 Related Work on Parallel Search Algorithms

Prior work on optimal and bounded suboptimal algorithms like A* and Weighted A* has demonstrated that they can be parallelized with highly robust scaling. State distribution methods, such as Hash-Distributed A* (Burns et al. 2010; Snir 1998) and the partitioning scheme of PBNF (Kishimoto, Fukunaga, and Botea 2013), efficiently divide work across threads and scale to thousands of processors. For domains with slow state expansions, algorithms like PA*SE, ePA*SE, and GePA*SE (Phillips, Likhachev, and Koenig 2014; Mukherjee, Aine, and Likhachev 2022; Mukherjee and Likhachev 2023) parallelize specific phases within A* to accelerate search while maintaining optimality bounds.

In the satisficing space, considerable effort has focused on parallelizing GBFS. K-Parallel Best First Search (KP-BFS) is a parallel BFS implementation that maintains a global OPEN and CLOSED that are shared among the k threads (Vidal, Bordeaux, and Hamadi 2010). Access to OPEN and CLOSED are protected by a mutex the ensure exclusive access . K-Independent OPEN-BFS (KIO-BFS) is a parallel BFS where each thread keeps a local OPEN but share CLOSED (Kuroiwa and Fukunaga 2020). At each search step, threads choose the minimum f-valued state in their local OPEN if it is not empty; otherwise they choose the globally lowest f-valued state across all open lists. Again, open lists and CLOSED are guarded by a mutex.

Recent literature has focused on mimicking the state-space exploration of sequential GBFS and bounding expansions relative to sequential GBFS. Shimoda and Fukunaga (2023, 2025) introduced Parallel Under Higher Watermark First (PUHF) and later One Bench at a Time (OBAT). OBAT guarantees a constant factor bound on the number of expansions it performs compared to sequential GBFS, and both OBAT and PUHF guarantee only expanding states that GBFS could possibly expand under any tie-breaking scheme. It was found that meeting these guarantees can incur significant overhead, which can be mitigated by decoupling successor generation and heuristic evaluation (Shimoda and Fukunaga 2024).

Parallel GBFS as a parallel portfolio has also been investigated (Valenzano et al. 2010). Parallel search portfolios run independent searches simultaneously, avoiding any need for partitioning or thread coordination at the potential expense of duplicate effort between threads.

2.4 Parallel Search Pathologies

In order to understand how the performance of a parallel GBFS algorithm deviated from that of a sequential GBFS run, Kuroiwa and Fukunaga (2020) introduced the following notions for comparing the number of expansions required by two search algorithms, A and B:

Definition 1. *Given a state-space task T, a search algorithm A is t-bounded relative to search algorithm B on T iff A performs no more than t-times as many expansions as B. A is t-pathological relative to B on T iff A is not t-bounded relative to B on T.*

Definition 2. *A is t-bounded relative to B iff it is t-bounded on all possible state-space topologies.*

Definition 3. *A is t-pathological relative to B iff for all $t > 0$ there exists a state-space topology T such that A is t-pathological relative to B on T.*

[1] We ignore re-openings in this work since our focus is on solving problems quickly, regardless of solution quality.

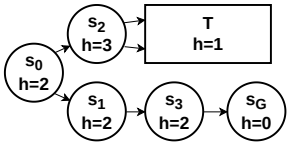

Figure 1: KP-GBFS is not *t-bounded* relative to GBFS. The figure is adapted from Kuroiwa and Fukunaga (2020).

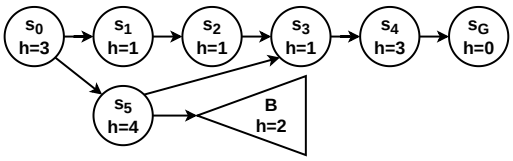

Figure 2: KIO-GBFS is not *t-bounded* relative to GBFS. The figure is adapted from Kuroiwa and Fukunaga (2020).

Intuitively, A is t-bounded relative to B if the amount of effort needed to solve a problem using A is never more than a constant factor larger than B, and A is t-pathological relative to B if it may be arbitrarily worse than B on certain state-space tasks. Kuroiwa and Fukunaga (2020) further introduced *TB-boundedness*, which considers the subset of the state-space task possibly expanded by the sequential variant of the algorithm and whether the parallel implementation is confined to the same set.

Definition 4. *Let T be a state space task. A search algorithm A is TB-bounded with respect to search algorithm B on T iff A does not expand any states which are not expanded by B under any tie-breaking mechanism. A is TB-bounded relative to B if it is TB-bounded on every possible state-space topology.*

OBAT is both *t-bounded* and *TB-bounded* relative to sequential GBFS (Shimoda and Fukunaga 2025). KP-GBFS and KIO-GBFS, are not *TB-bounded* and are *t-pathological* relative to sequential GBFS (Kuroiwa and Fukunaga 2020).

We now briefly sketch out why KIO-GBFS and KP-GBFS are t-pathological with respect to sequential GBFS using examples similar to those given in the original work. In the case of KP-GBFS, this is due to the fact that a shared open list of KP-GBFS means threads can 'pollute' each other by drawing them into regions of the state space they otherwise would not search. This can be seen in Figure 1, which shows a state-space task on which sequential GBFS would expand $\langle s_0, s_1, s_3 \rangle$ and then select s_G and terminate. Now consider running KP-GBFS on the same state space with $k = 2$. First s_0 gets expanded; then s_1 and s_2 get expanded in parallel; then both threads will expand states in the T component as their states have the smallest h-values. The fact that KP-GBFS is t-pathological relative to sequential GBFS follows from the fact that T can be made arbitrarily large, and thus the bound will not hold for any constant t.

Whereas KP-GBFS is t-pathological relative to sequential GBFS because of its use of a shared OPEN, KIO-GBFS is t-pathological relative to sequential GBFS because of its use of a shared CLOSED, which can act as a barrier to search. For example, consider Figure 2. Sequential GBFS would necessarily expand the sequence $\langle s_0, s_1, s_2, s_3, s_4 \rangle$ at which point it would select the goal, s_G, and terminate. KIO-GBFS with 2 threads would expand s_0 first; then, s_1 and s_5 would get expanded; and on the third search step, s_2 and s_3 would be expanded. On the next step, since s_3 is in CLOSED, both threads will end up expanding the component B. The fact that KIO-GBFS is t-pathological relative to sequential GBFS thus follows since $|B|$ may be arbitrarily large (Kuroiwa and Fukunaga 2020).

3 Thread-Ordered Parallel-GBFS

Notice that the pathological behaviour of KP-GBFS and KIO-GBFS described above occurs because the data structures shared by the threads can change the expansion order of states depending on how many threads are sharing them. In essence, 'additional' threads can impact the expansion sequence of the 'prior' thread(s), possibly leading to the runtime increasing with the number of threads.

In this section, we describe *Thread-Ordered Parallel-GBFS (TOP-GBFS)*, which is a new parallelization of GBFS designed specifically to address this issue. We begin by reviewing MonoBeam — which ensures monotonicity in beam search — before adapting the insights of MonoBeam to the case of parallel GBFS. TOP-GBFS will be used in cases where the number threads is irrelevant, while TOP-GBFS$_k$ indicates a variant with specifically k threads.

3.1 Non-Monotonicity in Beam Search

Recall that on every iteration of beam search, the entire open list is expanded simultaneously and only the "best" w states are kept. Lemons et al. (2022) identified that increasing w did not necessarily improve solution quality despite the fact that doing so seemingly makes the search inherently less greedy. In particular, they identified that this non-monotonicity arises, in part, from *cuckoo* states. These are states with incorrectly low heuristic values that only get generated at larger beam widths. Such states can dominate the beam and thereby drag the search into local minima it would have avoided when using a smaller beam width. Too, they showed that the use of a wider beam may close a state earlier than a narrower beam otherwise would. Doing so can block the search from finding a path it would have found with a narrower beam, leading to a second cause for non-monotonicity.

MonoBeam was introduced to address these issues (Lemons et al. 2022). MonoBeam strictly orders its operations by decomposing the beam into individual "slots". Slot i in the beam can only be filled by states generated from a parent slot j, where $0 \leq j \leq i$. Moreover, duplicates are permitted if a lower slot needs to expand a state that was already expanded by a higher slot. Together, these techniques ensure that the front of the beam will contain the same states as they would when using a smaller beam width, meaning the smaller width search is replicated within a search that uses a larger beam width. As such, the expanded cuckoo states and state closures of higher beams cannot derail the search of lower beams.

Algorithm 1: TOP-GBFS Thread Routine for Thread i

Require: List of open lists $[OPEN_0, ..., OPEN_i]$, Thread index i, Global Table $CLOSED$

```
1:  while not SOLVED and not UNSOLVABLE do
2:      s ← SELECT([OPEN_0, ..., OPEN_i], i)
3:      if s = NULL then
4:          if i = 0 then
5:              signal UNSOLVABLE
6:              return
7:          continue
8:      if is_goal(s) then
9:          signal SOLVED
10:         return trace_path(s)
11:     Lock(OPEN_i), Lock(CLOSED)
12:     OPEN_i ← (OPEN_i \ {s})
13:     if CLOSED[s] > i then
14:         CLOSED[s] ← i
15:         OPEN_i ← OPEN_i ∪ SUCC(s)
16:     Unlock(OPEN_i), Unlock(CLOSED)
```

Algorithm 2: An example SELECT(·) method

Require: List of open lists $[OPEN_0, ..., OPEN_i]$, Thread index i

```
1: for j ← i downto 0 do
2:     if not OPEN_i = ∅ then
3:         return arg min_{s'∈OPEN_j} ⟨h(s'), τ(s')⟩
4: return NULL
```

3.2 Encouraging Monotonicity in Parallel GBFS

As mentioned above, one reason that KP-GBFS and KIO-GBFS do not get strictly faster with additional threads is because threads may negatively interfere with each other's searches. A portfolio-based parallelization will avoid this behaviour, but may result in a substantial amount of duplicate work. Instead, we take inspiration from MonoBeam: KP-GBFS is not t-bounded essentially because the open lists for the different threads end up being polluted by cuckoo states, drawing the search into additional local minima, while the shared closed list of KIO-GBFS blocks the search just like it does when using a larger beam width.

While MonoBeam orders expansions over consecutive slots in the beam, in TOP-GBFS$_k$, the ordering is over the k threads, $\langle t_0, \ldots, t_{k-1} \rangle$. Any thread t_i is then not allowed access to any progress made by a lower priority thread (*i.e.* some t_j where $j > i$). This means that just as MonoBeam ensures that any increase in beam width does not affect how the slots at the front of the beam are populated, TOP-GBFS ensures that increasing the number of threads does not affect the expansion sequence of 'prior' threads.

TOP-GBFS maintains a global $CLOSED$ list, and each thread t_i has an associated open list $OPEN_i$. The algorithm begins by adding the initial state s_I to $OPEN_0$ and leaving the open lists for all other threads empty. The global $CLOSED$ list is initialized as an empty map. Pseudocode for each thread is shown in Algorithm 1. Locking is only explicitly done for 'write' operations while 'read' operations are allowed to be done concurrently. Note that in our algorithm description below, we say a thread t_i *dominates* t_j if $i < j$; and the *dominant set* of thread t_j is the set of threads $\langle t_0, ..., t_{j-1} \rangle$ that dominate t_j.

The pseudocode shows that each thread iteratively expands states until any one thread finds a solution. A thread t_i can select states for expansion from its own open list, $OPEN_i$, or from the open list of a thread from its dominant set. This is done by the call to SELECT(·) on line 2. Importantly, a thread can only add successors to (line 15) and remove states from (line 12) its own open list. Because threads select, but never remove, states from their dominant set, we refer to selecting a state from a dominant thread as *peeking*. This unidirectional visibility ensures that a thread t_i cannot access states that were only found by lower priority threads. Much like the beam slots in MonoBeam, this prevents a thread t_i from polluting the open lists of its dominant set with cuckoo-like states, meaning TOP-GBFS avoids this cause of t-pathological behaviour.

To keep track of which threads have expanded a state, the closed list maps a state to the smallest index of all threads that have expanded it (line 14). For simplicity, we assume that if s is not in $CLOSED$, then $CLOSED[s] = \infty$. Threads can re-expand states closed by threads they dominate, meaning the closure of a state by t_i will not impact the behaviour of a more dominant thread (line 13). This is much the same as the duplicate checking mechanism in MonoBeam, and will allow TOP-GBFS to avoid the closed list blocking behaviour that causes KIO-GBFS to be t-pathological.

Algorithm 1 actually defines a family of algorithms that may differ on their implementation of SELECT(·). A simple example, for illustration only, is given in Algorithm 2. Starting with $OPEN_i$, this method iterates through the available open lists (line 1). When a non-empty list is found, the "best" state in that list is returned. In its current form, Algorithm 2 allows all threads to expand the same state if the less dominant threads select first. This can be mitigated by using different tie-breaking schemes when a threads peek from their dominant sets; we discuss the practicalities of implementing TOP-GBFS in Section 5. Ultimately, as long as the thread ordering is respected, there is flexibility in how threads peek from their dominant sets.

Notice that t_0 only has access to $OPEN_0$ when selecting states for expansion, and only considers a state closed if t_0 itself closed it. As such, if SELECT(·) is defined so that t_0 always selects greedily from its open list, t_0 will be performing a standard sequential GBFS. TOP-GBFS uses its strict ordering of threads to balance the twin desires of never making things worse with additional threads, and maintaining an ability to avoid duplicated work. We will see in the following sections how this ordering ensures runtime monotonicity.

4 Theoretical Analysis

The core property of TOP-GBFS is that adding a k^{th} thread does not alter the behaviour of the prior threads: the first $k-1$ threads will expand the exact same states in the same time as they would in a k-thread instance, provided the k^{th} thread

does not find a solution faster. In this section we formalize this relationship and present our theoretical findings about TOP-GBFS. We first discuss the deterministic model used in our analysis; we then present our theoretical results; lastly, we discuss the practical limitations of our findings.

4.1 Modeling Parallel Search

In parallel search algorithms, thread scheduling and other system noise may lead to non-deterministic behaviour. Therefore, modeling parallel search demands making assumptions which yield a deterministic model of computation (JáJá 1992; Kuroiwa and Fukunaga 2020).

To this end, we introduce the *deterministic expansion model*, which generalizes the idealized parallel framework of Kuroiwa and Fukunaga (2020). Like their model, we assume multiple threads execute simultaneously with zero memory contention, synchronization overhead, or scheduling interference. Whereas Kuroiwa and Fukunaga assume uniform expansion times for all states, we permit non-uniform expansion times. Specifically, we assume a deterministic function $\delta : S \mapsto \mathbb{R}^{\geq 0}$ that maps each state s to a constant expansion time $\delta(s)$, which remains invariant regardless of when or by which thread s is expanded.

In the analysis below, we assume the expansion time for any state is bounded by strictly positive constants such that $0 < \delta_{min} \leq \delta(s) \leq \delta_{max}$ for all $s \in S$ (which holds trivially for a finite state space). We also assume the deterministic expansion model, and that SELECT$(\cdot)$ ensures that thread t_0 always selects states greedily from $OPEN_0$ (meaning t_0 performs a sequential GBFS). Notably, SELECT$(\cdot)$ may be defined arbitrarily for the other threads, provided it satisfies the strict thread ordering requirements of TOP-GBFS. We note that with deterministic tie-breaking, the call to SELECT$(\cdot)$ will be deterministic, in that it will always select the same state for expansion given the same current contents of the closed and open lists.

4.2 Search Properties of TOP-GBFS

In this section, we show our primary theoretical results: we first establish constant factor expansion bounds, and then establish the runtime monotonicity of TOP-GBFS as more threads are used for the search. We begin by formalizing the idea that a thread in TOP-GBFS does not impact the execution threads of its dominant set. This is because the state expansion order of a thread — and the runtimes of those expansions — does not change when more threads are added. To show this, we let $\pi_i^k(r)$ denote the list, in chronological order, of the sequence of states expanded by thread t_i up to wall-clock time $r \geq 0$ in some execution of TOP-GBFS$_k$. We now formalize this property as follows:

Lemma 1. *For any i and k where $1 \leq i < k$, let R_i and R_k be the runtimes of TOP-GBFS$_i$ and TOP-GBFS$_k$, respectively. Then at any time $r \leq \min(R_i, R_k)$ and any thread t_j where $0 \leq j < i$, the lists $\pi_j^i(r)$ and $\pi_j^k(r)$ are identical.*

Proof. Let r be the smallest value for which there exists some thread j where $\pi_j^i(r)$ and $\pi_j^k(r)$ differ. Since r is the smallest such value, they can only differ by the elements at the ends of these lists. There are two possible such cases.

First, the two lists may be of equal length, but have a different last element. This would mean that thread t_j selected a different next state for expansion in TOP-GBFS$_i$ and TOP-GBFS$_k$. However, immediately before r, the open lists for all threads up to and including t_j must be identical for both algorithms. This holds since the expansion sequences are identical for all threads in both algorithms and the threads that dominate t_j cannot add states to the open lists in the dominant set. By a similar argument, the set of states with a $CLOSED$ value of j or less must be identical. It is therefore a contradiction that t_j selected a different state in TOP-GBFS$_i$ and TOP-GBFS$_j$, since the SELECT$(\cdot)$ used for t_j was assumed to be deterministic.

The second possibility is that one of $\pi_j^i(r)$ or $\pi_j^k(r)$ is longer than the other. This would require that the additional threads has sped up or slowed down the execution of thread j. However, this is not possible by the deterministic execution model. Thus the statement holds by contradiction. □

We can now establish a worst-case expansion bound of TOP-GBFS$_k$ relative to a sequential GBFS.

Theorem 2. *TOP-GBFS$_k$ is* t-bounded *relative to GBFS for any $k \geq 1$, with $t = k\frac{\delta_{max}}{\delta_{min}}$.*

Proof. Let E_1 be the number of expansions taken by standard sequential GBFS, and let R_1 be its runtime. By Lemma 1, the highest priority thread t_0 performs a sequential GBFS regardless of k. The maximum possible time taken for t_0 to find a solution is $E_1 \cdot \delta_{max}$. Because t_0 terminates the search at or before $E_1 \cdot \delta_{max}$, no other thread can run for longer than $E_1 \cdot \delta_{max}$. Since the minimum possible expansion time is δ_{min}, any individual thread can perform at most $\frac{R_1}{\delta_{min}} \leq E_1 \frac{\delta_{max}}{\delta_{min}}$ expansions before t_0 terminates. Thus, even if the remaining $k-1$ threads perform entirely unproductive expansions, TOP-GBFS$_k$ will collectively perform at most $E_1 + (k-1)E_1\frac{\delta_{max}}{\delta_{min}} \leq kE_1\frac{\delta_{max}}{\delta_{min}}$ total expansions. □

We can similarly bound how much the total expansion count can scale when adding an additional thread.

Theorem 3. *TOP-GBFS$_{k+1}$ is* t-bounded *relative to TOP-GBFS$_k$ for any $k \geq 1$, with $t = 1 + \frac{\delta_{max}}{k\delta_{min}}$.*

Proof. Let E_k denote the total number of expansions performed by TOP-GBFS$_k$ on task T. This implies that the worst-case maximum runtime for TOP-GBFS$_k$ is $\frac{E_k\delta_{max}}{k}$.

When using TOP-GBFS$_{k+1}$, Lemma 1 guarantees that the additional thread t_k cannot alter the execution of the original k threads. In the worst case, t_k will spend all its time doing entirely unproductive expansions at the fastest possible rate, contributing at most $\frac{E_k\delta_{max}}{k\delta_{min}}$ extra expansions before the search terminates. Thus, the total expansion count satisfies: $E_{k+1} \leq E_k + \frac{E_k\delta_{max}}{k\delta_{min}} = \left(1 + \frac{\delta_{max}}{k\delta_{min}}\right) E_k$ □

Lastly, Lemma 1 implies that the wall-clock runtime of TOP-GBFS is monotonically non-increasing as more processing threads are allocated to the task.

Theorem 4. *If R_k and R_{k+1} are the runtimes on search task T for TOP-GBFS$_k$ and TOP-GBFS$_{k+1}$, respectively, then $R_{k+1} \leq R_k$.*

Proof. Consider two cases. First, if T is unsolvable, R_k is the time at which t_0 signals unsolvability. By Lemma 1, t_{k+1} cannot alter the execution of t_0 and cannot falsely find a goal, meaning t_0 will signal unsolvability at exactly the same time. Thus, $R_{k+1} = R_k$.

Second, if T is solvable, let R_k be the time it takes for some thread t_i to expand a goal state in TOP-GBFS$_k$. In TOP-GBFS$_{k+1}$, Lemma 1 guarantees that if the search does not terminate prior to R_k, thread t_i will still expand that goal state at exactly time R_k. However, the introduction of t_k introduces the possibility of expanding a goal state at an earlier time prior to R_k. The search will thus terminate either early or at exactly R_k, proving $R_{k+1} \leq R_k$. □

We note that TOP-GBFS does not restrict itself to the same search sub-space as sequential GBFS because the state selection policy for threads is arbitrary (barring t_0); that is to say, TOP-GBFS$_k$ is not necessarily TB-bounded relative to GBFS, if further assumptions on SELECT$(\cdot)$ are not made.

4.3 Limitations

If the assumptions of our deterministic expansion model are dropped — such as introducing system noise, memory contention, or non-deterministic expansion times — neither the expansion bounds nor the runtime monotonicity from Section 4.2 will hold. Consider the task in Figure 3. GBFS would expand the sequence $s_0, s_1, s_3, s_4, B, s_5, s_G$ where B is a set of states reachable from s_1. TOP-GBFS$_2$ may expand, in perfect lockstep, the sequence $(s_0), (s_1, s_2), (s_3, s_4), (s_4, s_5), (s_i, s_G)$, where $s_i \in B$. The non-determinism that can arise from dropping our assumptions comes from the possibility that thread t_0 expands s_4 before t_1. In this case, t_1 gets blocked by the closure done by t_0 (line 13 of Algorithm 1) and so will expand states in T instead of expanding s_4, s_5, s_G. Allowing T to be arbitrarily large breaks our expansion bounds.

Similarly, for runtime monotonicity, consider running TOP-GBFS$_3$ on Figure 3. It could be that due to the non-ideal environment, t_0 completes 4 expansions before threads t_1 or t_2 expand anything, yielding a possible partial expansion sequence $(s_0), (s_1), (s_3), (s_4), (s_i, s_2), (s_j, s_k, s_l)$, where $s_i, s_j \in B$, and $s_k, s_l \in T$. If threads t_1 and t_2 keep expanding component T, the runtime of TOP-GBFS$_3$ will exceed that of TOP-GBFS$_2$.

Thus, TOP-GBFS may not necessarily be monotonic in practice. However, it does seem "more monotonic" than other GBFS parallelizations, as shown in the next section.

5 Empirical Results

All experiments were conducted on dual 64-core AMD EPYC 7742 processors operating at a base clock of 2.25 GHz, with a total of 256 GB of RAM. Algorithms were tested with 1, 2, 4, 6, 8, 10, 12, 14, 16 and 32 threads, and were granted as many cores as threads, along with 2 GB of RAM per thread. In all cases, algorithms were given a 5-minute per-problem time limit, and used the unit cost FF heuristic (Hoffmann and Nebel 2001).

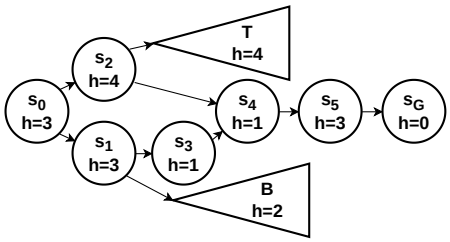

Figure 3: Non-determinism due to state s_4.

The evaluation uses 15 domains from the Autoscale benchmark suite (Torralba, Seipp, and Sievers 2021). We selected the 10 domains with the lowest coverage and the 5 domains with the highest coverage for single-threaded GBFS as reported by Shimoda and Fukunaga (2025). Optimal track problems were used to ensure high coverage, shifting the focus to expansions and runtime analysis.

5.1 Algorithms

Our code[2] is based on the codebase developed by Shimoda and Fukunaga (2025). For KP-GBFS and OBAT, denoted KP and OBAT respectively, the implementations by Shimoda and Fukunaga (2025) were used largely unchanged. For KIO-GBFS, denoted KIO, we added our own implementation following the description in Section 2. Notably, all algorithms, including TOP-GBFS$_k$, were given a fixed 128 MB lossy heuristic cache that overwrites entries in the event of a collision.

For TOP-GBFS$_k$, two major issues can arise when implementing the selection policy: 1) with each thread potentially peeking from its dominant set, there could be $k - i$ threads reading from each $OPEN_i$ every step creating significant lock contention; and 2) if something like Algorithm 2 is used, there might be many duplicate selections as everyone is selecting greedily from every open list. In our implementation, to balance threads helping each other while avoiding contention, the dominant set gets peeked every 100 expansions or if $OPEN_i = \emptyset$, and a state only gets selected from the dominant set if it has a better heuristic value than what is on top of $OPEN_i$. To mitigate duplicate selections, different tie-breaking schemes are used when threads peek from their dominant set.

5.2 Results

Runtime For a given task, we expect TOP-GBFS to exhibit monotonic runtime scaling across thread counts ($R_i \leq R_j$ if $i > j$). To investigate this, Table 1 reports runtime inversions — instances where this expectation is violated. For example, running increasing numbers of threads on a problem and getting runtimes of 5, 6, 3, 7, 2, and 1 seconds contains four inversions ($5 \not\geq 6$, $5 \not\geq 7$, $6 \not\geq 7$, $3 \not\geq 7$). Table 1 reports the sum of inversions across all problems for all algorithms. A minimum runtime difference of 0.01 seconds is

[2]Available at https://github.com/dawson-brown/SOCS-26

Domain	KP	TOP	OBAT	KIO
airport	139	61	365	48
barman	52	24	100	89
childsnack	141	88	158	89
data-network	111	66	60	74
depots	57	37	54	86
driverlog	13	31	63	16
floortile	36	2	121	3
hiking	55	36	19	100
mprime	129	64	109	67
parcprinter	11	0	50	0
rovers	12	5	25	9
storage	20	47	16	76
termes	10	30	322	34
thoughtful	172	105	232	98
woodworking	82	97	72	126
Total Inversions	**1040**	**693**	**1766**	**915**
Inversion Rate	5.14%	3.42%	8.72%	4.52%

Table 1: Number of thread scaling inversions per domain.

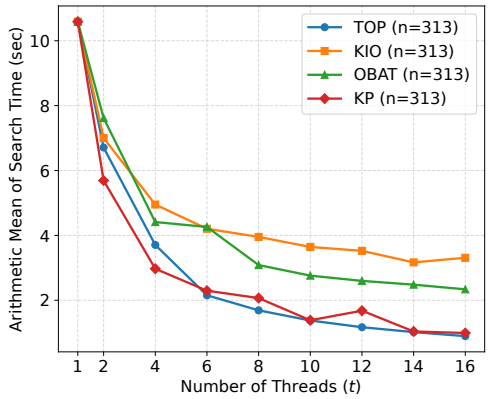

Figure 4: Average search time per thread count. The 32 thread variants were omitted for space.

required to register an inversion to account for system noise (we justify this threshold below).

As seen in the *inversion rate* (the fraction of pairwise comparisons that were out of order), all algorithms were largely monotonic, with OBAT being the notably worst performer according to this metric. However, TOP clearly had the fewest inversions, empirically agreeing with our theoretical monotonicity.

Figure 4 illustrates how average runtime scales with thread count for the four algorithms. The average for each algorithm is calculated over the intersection of instances (n) successfully solved by all algorithms and thread configurations. Again, all algorithms were largely monotonic in runtime. KP and TOP enjoyed the greatest average speedup as more threads were used.

Coverage Figure 5 summarizes coverage vs. time for the four algorithms, with each line denoting a different number of threads. We note that since we are plotting the total success rate, the expectation is that increasing the number of threads will improve coverage on average. This contrasts with measuring per instance monotonic improvement (which we did in the previous section).

We see in Figure 5 that KIO had the best maximum coverage of 387 with 32 threads and was also largely monotonic in coverage, only seeing a reduction in coverage when going from 8 to 10, and from 12 to 14 threads. TOP had a maximum coverage of 377 with 32 threads; TOP was almost entirely monotonic in coverage, only going from 8 threads to 10 saw reduced coverage. KP had the least monotonic coverage, with 4 to 6, 12 to 14, and 14 to 16 threads all seeing reduced coverage; KP's maximum coverage, also with 32 threads, was 376. OBAT had the worst maximum coverage of 367 at 16 threads (likely due to its restrictive expansion policy); OBAT was monotonic in coverage, except when doubling from 16 to 32 threads, which is particularly surprising and perhaps speaks to the increased thread starvation that can arise in more constrained searches like OBAT.

To better understand why TOP did not perform better in coverage despite strong monotonicity, we more thoroughly compare it to KIO by looking at expansion rate and the impact of the cache on search. These two algorithms had much greater expansion rates than KP and OBAT, in part due to the local search spaces reducing contention on shared resources. Figure 6 compares the expansion rates of TOP and KIO with 32 threads. Perhaps surprisingly, despite the near equal coverage for the algorithms with 16 threads, TOP saw much faster expansions on problems solved by both. This is in part due to the cache performance of each. On the jointly solved instances, TOP hit the cache 67% of the time, while KIO hit the cache 10% of the time. We hypothesize this is because TOP peeked work every 100 expansions and so threads were often drawn into their dominant sets' search spaces. Indeed, 0.5% of TOP expansions were on peeked states, while 0.06% of KIO expansions were on stolen states.

This suggests to us that the SELECT(·) policy of TOP requires careful tuning. The suggested TOP configuration spent a lot of time doing duplicate work, evidenced by the cache hit rate. So although it expanded many states very quickly, those states were not sufficiently diverse to ensure new and shorter solution paths could be consistently found. TOP may benefit from less peeking, or less greedy peeking. Some preliminary testing to this end indicated that peeking every 10,000 and 100,000 expansions had similar coverage results to peeking every 100 expansions. Investigating alternative selection policies is an interesting avenue for future work. It is lastly worth noting that having a heuristic cache at all explicitly violates the assumptions we made for our theoretical results as it directly leads to expansions not taking constant time. It is possible that this made the circumstances of non-monotonicity from Section 4.3 more likely to occur, hence the unexpected coverage results.

6 Discussion and Conclusion

All four of the algorithms we tested usually exhibited the monotonicity that we wanted and expected from a parallel search algorithm — both in terms of search time and coverage. Having said that, TOP-GBFS was the "most" mono-

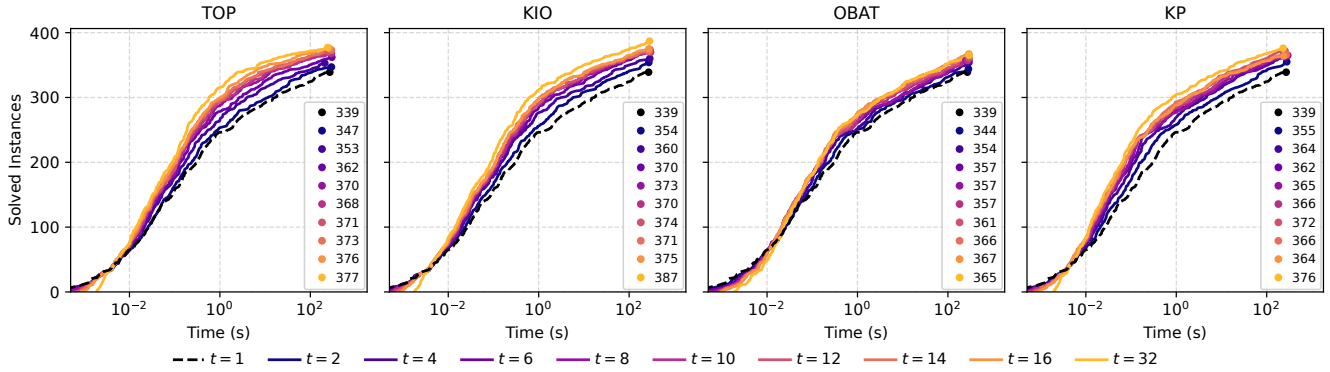

Figure 5: Coverage vs. Runtime. The legend indicates maximum coverage with t threads.

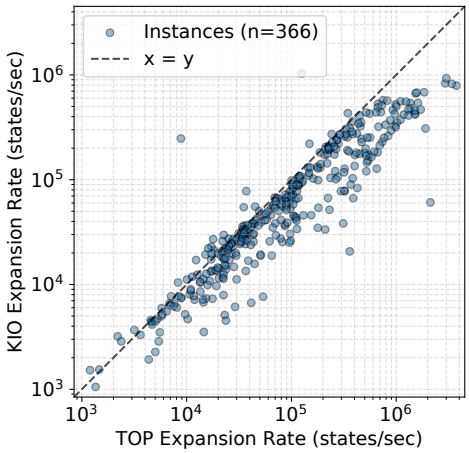

Figure 6: KIO vs TOP expansion rate with 32 threads.

tonic, suggesting its more robust against the pathological search behaviour identified in Section 2.4.

The pathologies in Section 2.4 hinge on the existence of large *local minima* in the space (connected regions with incorrectly low h-values). We hypothesize that domains which exhibit large local minima will have a greater potential for pathological search behaviours and so TOP-GBFS may see greater benefits on such domains. Cohen and Beck examined 6 such domains (nomystery, rovers[3], tpp, maintenance, parking, and freecell) (2018); testing TOP-GBFS on them is interesting future work.

The deterministic expansion model is not entirely realistic. Analyzing TOP-GBFS and other parallel algorithms with better models is left for future work. Specifically, Read-Queue/Write-Queue models can better model contention (Gibbons, Matias, and Ramachandran 1998); alternatively, expansion times could be sampled from a distribution.

Unlike OBAT and PUHF, TOP-GBFS is not *TB-bounded* relative to GBFS. It may be theoretically interesting future work to develop TB-bounded TOP-GBFS variants. It is also worth noting that OBAT and PUHF benefit greatly from separate generation and evaluation (Shimoda and Fukunaga 2024), which was not used here.

The thread ordering of TOP-GBFS is not limited to GBFS and can operate as a more general framework. The thread-priority could theoretically wrap any deterministic, sequential search algorithm while retaining the same monotonic runtime scaling. Whether these thread-ordered mechanisms can be adapted to optimal or bounded sub-optimal algorithms remains an open question for future work.

Beyond changing the underlying sequential search, the flexibility in when and how threads share states from their open lists is an interesting opportunity for exploration. The tested configuration of TOP-GBFS saw a lot of duplicate effort between threads. GBFS exploration techniques may serve as the basis for sensible TOP-GBFS configurations. Xie, Müller, and Holte (2014) suggested only exploring in GBFS when the search has failed to make heuristic progress after a specified number of expansions. Threads could adhere to a similar protocol to determine when they should peek at another open list. Concerning *how* states peek from each other, Cohen and Beck (2018) explored the value of randomized restarts as a GBFS enhancement. In TOP-GBFS, this could be simulated by peeking the highest h-valued state instead of the lowest.

Lastly, this approach may be particularly suitable to classical planning where the complexity of the heuristic calculation dwarfs any multi-threading overhead. This can make our idealized models better approximations of reality than they would in other paradigms. Many standard heuristic search domains are characterized by extremely fast expansions (such as the sliding tile or pancake puzzles) and so lock contention stands to be a much greater fraction of the overall runtime. In such domains, TOP configurations will likely require less frequent peeking to remain efficient.

Ultimately, while GBFS parallelizations do generally appear to exhibit monotonic behaviour, TOP-GBFS does show the most predictable scaling, while still being competitive with the state-of-the-art. Finding the right balance between our theoretical guarantees and practical overheads will be key to extending this framework to a wider variety of domains and algorithms.

[3]Although rovers was in our benchmark suite, the instances we used were too easy.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Ontario Graduate Scholarship (OGS).

This work used Bridges-2 at Pittsburgh Supercomputing Center through allocation CIS240221 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

Bisiani, R. 1992. Beam search. *Encyclopedia of artificial intelligence*.

Burns, E.; Lemons, S.; Ruml, W.; and Zhou, R. 2010. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39: 689–743.

Cohen, E.; and Beck, J. C. 2018. Local Minima, Heavy Tails, and Search Effort for GBFS. In *International Joint Conference on Artificial Intelligence*, 4708–4714.

Doran, J. E.; and Michie, D. 1966. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437): 235–259.

Gibbons, P. B.; Matias, Y.; and Ramachandran, V. 1998. The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2): 733–769.

Hennessy, J. L.; and Patterson, D. A. 2011. *Computer architecture: a quantitative approach*. Elsevier.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14: 253–302.

JáJá, J. 1992. *An Introduction to Parallel algorithms*.

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

Kuroiwa, R.; and Fukunaga, A. 2019. On the pathological search behavior of distributed greedy best-first search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, 255–263.

Kuroiwa, R.; and Fukunaga, A. 2020. Analyzing and avoiding pathological behavior in parallel best-first search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 175–183.

Lemons, S.; López, C. L.; Holte, R. C.; and Ruml, W. 2022. Beam search: Faster and monotonic. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 222–230.

Mukherjee, S.; Aine, S.; and Likhachev, M. 2022. ePA*SEe: Edge-based Parallel A* for slow evaluations. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 136–144.

Mukherjee, S.; and Likhachev, M. 2023. GePA* SE: Generalized edge-based parallel A* for slow evaluations. In *Proceedings of the International Symposium on Combinatorial Search*, volume 16, 153–157.

Phillips, M.; Likhachev, M.; and Koenig, S. 2014. PA* SE: Parallel A* for slow expansions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 208–216.

Shimoda, T.; and Fukunaga, A. 2023. Improved exploration of the bench transition system in parallel greedy best first search. In *Proceedings of the International Symposium on Combinatorial Search*, volume 16, 74–82.

Shimoda, T.; and Fukunaga, A. 2024. Separate Generation and Evaluation for Parallel Greedy Best-First Search. In *Proceedings of ICAPS 2024 Workshop on Heuristics and Search for Domain-Independent Planning (HSDIP)*.

Shimoda, T.; and Fukunaga, A. 2025. Parallel greedy best-first search with a bound on expansions relative to sequential search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 26668–26677.

Snir, M. 1998. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press.

Torralba, Á.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. In Goldman, R. P.; Biundo, S.; and Katz, M., eds., *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 376–384.

Valenzano, R.; Sturtevant, N.; Schaeffer, J.; Buro, K.; and Kishimoto, A. 2010. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, 177–184.

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 *Proceedings of the International Symposium on Combinatorial Search*, volume 1, 100–107.

Xie, F.; Müller, M.; and Holte, R. 2014. Adding local exploration to greedy best-first search in satisficing planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28.