Separate Generation and Evaluation for 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

Parallelization of Greedy Best First Search (GBFS) has been difficult because straightforward parallelization can result in search behavior which differs significantly from sequential GBFS, exploring states which would not be explored by sequential GBFS with any tie-breaking strategy. Recent work has proposed a class of parallel GBFS algorithms which constrains 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, enforcing this constraint is costly, as such BTS-constrained algorithms are forced to spend much of the time waiting so that only states which are guaranteed to be in the BTS are expanded. We propose an improvement to parallel search which decouples state generation and state evaluation and significantly improves state evaluation rate, resulting in better search performance.

1 Introduction

Parallelization of combinatorial search algorithms is important in order to maximize search algorithm performance on modern, multi-core CPUs. 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* (all nodes with f-values less than the optimal path cost $C*$ *must* be expanded), 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 (Burns et al. 2010; Kishimoto, Fukunaga, and Botea 2013; Phillips, Likhachev, and Koenig 2014; Fukunaga et al. 2017).

For satisficing search where the object is to quickly find any valid solution path (regardless of path cost), parallelization is not well understood. 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 than 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).

Unlike parallel cost-optimal search, 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 theoretical analysis of GBFS has yielded a promising direction for determining which states should be expanded. Heusner, Keller, and Helmert (2017) identified the *Bench Transition System* (BTS), the set of all states that can be expanded by GBFS under some tie-breaking policy (conversely, if a state is not in the BTS, there does not exist any tie-breaking strategy for GBFS which will expand that state). Limiting search to states in the BTS provides a natural constraint for parallel GBFS.

Recent work has proposed parallel GBFS algorithms which expand states from $Open$ only if some constraint is satisfied. Kuroiwa and Fukunaga (2020) proposed PUHF, a parallel GBFS which guarantees that only states which are in the BTS will be expanded. However, this guarantee comes at 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 until a state which is guaranteed to be in the BTS become available, resulting in significantly worse performance than KPGBFS. Improved versions of PUHF with looser constraints (which still guarantee that only states in the BTS are expanded) have been proposed (Shimoda and Fukunaga 2023), but these still have a significantly lower state evaluation rate than parallel GBFS without expansion constraints.

In this paper, we propose Separate Generation and Evaluation (SGE), which decouples state expansion and evaluation so that instead of waiting for a single thread to fully expand a state (generating and evaluating its successors), multiple threads can be used to evaluate the successors. We show that this significantly improves the state evaluation rate in parallel GBFS with expansion constraints.

The rest of the paper is structured as follows. Section 2 reviews background and previous work. Section 3 presents Constrained Parallel GBFS (CPGBFS), which unifies KPGBFS and all previous versions of PUHF as instances of a class of search algorithms with various state expansion constraints. Section 4 discusses the state expansion bottlenecks in CPGBFS. Section 5 proposes SGE. Section 6 experimentally evaluates SGE and compares them to PUHF and KPGBFS. Section 7 concludes with a discussion 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 4-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 states $\langle s_0, ..., s_n \rangle$ 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 $\forall s \in S, s \notin succ(s)$.

Best-First Search Best-First Search (BFS) is a class of search algorithms that use an evaluation function $f : S \rightarrow \mathbb{R}$ 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, baseline 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 $succ(s)$ 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 from those explored by GBFS. Kuroiwa and Fukunaga (2020) showed that straightforward parallelizations of GBFS with shared $Open$ and/or $Closed$, including KPGBFS, can expand arbitrarily more states than GBFS.

Bench Transition Systems Heusner et al. (2017) defined *bench transition system* (BTS) in order to characterize the behavior of GBFS, building upon the definition of high-water marks by Wilt and Ruml (2014). The BTS is defined as the set of all states which can be expanded by GBFS with some tie-breaking policy, i.e., a state s is in the BTS if there exists some tie-breaking policy under which s is expanded. Conversely, states not in the BTS will not be expanded by GBFS under any tie-breaking policy.

BTS-Constrained Search Restricting the search to only expand states which are in the BTS is a natural constraint for parallel GBFS.

Definition 2. A search algorithm is BTS*-constrained* if it expands only states which are in the BTS (Shimoda and Fukunaga 2023).

PUHF: A BTS-Constrained Parallel GBFS Kuroiwa and Fukunaga (2020) proposed Parallel Under High-water mark First (PUHF), a BTS-constrained parallel GBFS. PUHF marks states which are guaranteed to be in the BTS as *certain*, and only expands states marked as certain. The criterion used by PUHF to mark states as certain was a restrictive, sufficient (but not necessary) condition for being in the BTS. Recently, looser sufficient conditions for marking states as certain were proposed, resulting in PUHF2–4, which significantly improved performance over PUHF (Shimoda and Fukunaga 2023).

3 Constrained Parallel GBFS

As described above, the original proposing PUHF and PUHF2–4 presented these algorithms as marking states guaranteed to be in the BTS as *certain*, and only expanding nodes marked as *certain*. The stage in the algorithm where states were marked as *certain* were specific to the specific algorithm (PUHF, PUHF2–4). However, the the similarities and differences among KPGBFS, PUHF, and PUHF2–4 can be clarified by reframing this behavior in terms of satisfying an algorithm-specific expansion constraint.

We present a unified framework, Constrained Parallel GBFS (CPGBFS) (Algorithm 1), which subsumes KPGBFS, PUHF, and PUHF2–4. CPGBFS is a schema for a class of parallel search algorithms based on KPGBFS, which only expands nodes which satisfy some algorithm-specific constraint in line 7, where $satisfies(s)$ is a function which returns $true$ if and only if s satisfies the algorithm-specific expansion constraint.

KPGBFS is a special case of CPGBFS where $satisfies(s)$ always returns $true$. The previously proposed BTS-constrained search algorithms (PUHF and PUHF2–4) are instances of CPGBFS where the $satisfies(s)$ function implements a check for the sufficient constraint which guarantees that s is in the BTS – the specific implementation details of $satisfies$ depend on whether the algorithm is PUHF, PUHF2, PUHF3, or PUHF4. In addition, algorithm-specific auxiliary computations related to $satisfies$ are omitted for clarity.

4 State Expansion Bottlenecks in Constrained Parallel Search

Previous work has shown that CPGBFS (all of the PUHF variants) have a significantly lower state evaluation rate than unconstrained parallel search (KPGBFS). There are two, related reasons for the low state evaluation rate in CPGBFS: (1) the expansion constraint, and (2) eager evaluation policy.

Expansion Constraint Bottleneck Unconstrained parallel search algorithms such as KPGBFS will unconditionally

Algorithm 1: CPGBFS: Constrained Parallel GBFS Template

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

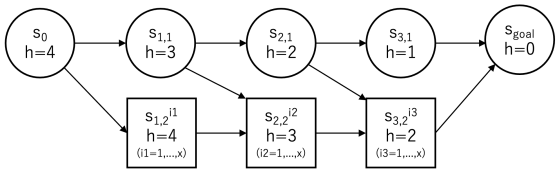

Figure 1: Example for SGE

expand the top states in $Open$. Threads in unconstrained parallel search algorithms are only idle when waiting for a mutex lock for the shared $Open$ and $Closed$ structures. If the shared $Open$/$Closed$ data structures are implemented efficiently (e.g., using sharding to internally partition the data structures), then waiting mutual exclusion overhead can be greatly reduced.

In contrast, constrained parallel GBFS algorithms such as PUHF force threads to be idle in order to guarantee that only states which satisfy the search constraints are expanded – even if there is a state in $Open$ and the mutex lock is available, CPGBFS can not expand the top state $top(Open)$ unless the expansion constraint is satisfied ($satisfies(top(Open)) = true$).

One approach to decreasing idle time is to improve the expansion constraint to be closer to a necessary constraint, as the previously proposed expansion constraints are all sufficient (overly restrictive) constraints for a state to be in the BTS. Previous work used this approach to improving the expansion constraint for PUHF, resulting in significantly improved evaluation rate (Shimoda and Fukunaga 2023). While this approach can be effective, finding safe improvements to the expansion constraint can be nontrivial.

Furthermore, even an ideal constraint (e.g., a necessary and sufficient expansion constraint for the BTS), would not eliminate idle threads. For example, in Figure 1 the only states in the BTS are the circular nodes (s_0,$s_{1,1}$, $s_{2,1}$, $s_{3,1}$, s_{goal}), so any BTS-constrained algorithm can only expand one of these states at a time, while all other threads must wait.

Batch Successor Insertion Bottleneck A second, closely related bottleneck is caused by the requirement that CPGBFS needs to insert successor states into $Open$ in a single batch. Consider Figure 1. In a standard, single-thread implementation of best-first search with eager evaluation, the expansion of $s_{1,1}$ includes (1) generating $succ(s_{1,1})$ and (2) evaluating all states in $succ(s_{1,1}) = s_{2,1}, s^1_{2,2}, \ldots, s^x_{2,2}$ with a heuristic evaluation function, and (3) inserting $succ(s_{1,1})$ in $Open$. In many cases, computing the heuristic evaluation function consumes the majority of time spent expanding the state, and the full expansion of a single state such as $s_{1,1}$ can take a significant amount of time due to the evaluation of all of its successors.

A constrained parallel search which seeks to expands a similar set of nodes as GBFS has an additional requirement not present in single-threaded search: the successors of a state s are all simultaneously inserted in $Open$ only after all successors of state s are evaluated (Algorithm 1, lines 21–24). This ensures that the successors of s are expanded in best-first order – without this batch insertion (e.g., if states were inserted one at a time directly into $Open$ immediately after being evaluated), a state s' with a worse f-value than its sibling s'' might be inserted into $Open$ and expanded by another thread before s'' has been evaluated into $Open$.

In the case of unconstrained parallel search such as KPGBFS, the overall evaluation rate is not significantly affected by whether the successors are inserted in a single batch or one at a time, because available threads can freely expand the top states from $Open$.

However, in CPGBFS, the combination/interaction of the expansion constraint and the batch successor insertion requirement causes a significant bottleneck. For example, in Figure 1, in Algorithm 1, if a thread starts to expand $s_{1,1}$, all other threads must stop and wait until all of $succ(s_{1,1})$ have been fully evaluated and inserted into $Open$.

5 Separate Generation and Evaluation (SGE)

In this section, we propose SGE, an approach for increasing the state evaluation rate in constrained best-first search algorithms such as PUHF. SGE alleviates the batch successor insertion bottleneck described above.

Continuing the example from the previous section, in the case of Figure 1, instead of waiting idly while one thread expands $s_{1,1}$ (which includes computing all of the heuristic values for $s_{2,1}, s^1_{2,2}, .., s^x_{2,2}$), it would be more efficient to

parallelize the evaluation of $s_{2,1}, s^1_{2,2}, .., s^x_{2,2}$ among available threads.

Algorithm 2: Constrained Parallel GBFS with SGE Template

```
1:  Open ← {s_init}, Closed ← {s_init}; ∀i, s_i ← NULL
2:  for i ← 0, ..., k − 1 in parallel do
3:      loop
4:          lock(Unevaluated)
5:          if Unevaluated ≠ ∅ then
6:              s_i ← top(Unevaluated)
7:              Unevaluated ← Unevaluated \ {s_i}
8:              unlock(Unevaluated)
9:              evaluate(s_i)      ▷ using a hashtable to prevent reevaluation of states
10:             if all siblings of s_i has been evaluated then
11:                 lock(Open), lock(Closed)
12:                 for s'_i ∈ siblings of s_i do
13:                     if s'_i ∉ Closed then
14:                         Closed ← Closed ∪ {s'_i}
15:                         Open ← Open ∪ {s'_i}
16:                 unlock(Open), unlock(Closed)
17:         else
18:             unlock(Unevaluated), lock(Open)
19:             if Open = ∅ then
20:                 unlock(Open)
21:                 if ∀j, s_j = NULL then
22:                     return NULL
23:             else if satisfies(top(Open)) = true then
24:                 s_i ← top(Open); Open ← Open \ {s_i}
25:                 unlock(Open)
26:             if s_i = NULL then continue
27:             if s_i ∈ S_goal then
28:                 return Path(s_i)
29:             lock(Unevaluated)
30:             Unevaluated ← Unevaluated ∪ succ(s'_i)
31:             unlock(Unevaluated)
32:         s_i ← NULL
```

We propose Separate Generation and Evaluation (SGE), which parallelizes state evaluations in addition to expansions. The main idea is to decompose the expansion of state s into separate units of work which can be parallelized: (1) successor generation, which generates $succ(s)$, the successors of s, and (2) successor evaluation, which evaluates $succ(s)$.

Algorithm 2 shows Constrained Parallel GBFS with SGE. After a thread selects a state for expansion from the shared $Open$, it generates $succ(s)$, and inserts $succ(s)$ into the shared $Unevaluated$ queue. The evaluation of states in $Unevaluated$ is done in parallel, taking precedence over selection of states for expansion (a thread will select a state for expansion from $Open$ only if $Unevaluated$ is currently empty (Algorithm 2, line 5)).

Evaluated states are *not* immediately inserted into $Open$. Instead, we insert all of the successors of s simultaneously into $Open$, after they have all been evaluated (lines 10-16). This is so that the parallel search is able to prioritize $succ(s)$ similarly to GBFS (otherwise, $succ(s)$ can be expanded in a completely different order than by GBFS, which we are trying to prevent).

Consider the search behavior of BTS-constrained parallel search such as PUHF with SGE on the search space in Figure 1. First, a thread pops s_0 from $Open$ ($satisfied(s_0) = true$), and generates its successors ($s_{1,1}, s^1_{1,2}, .., s^x_{1,2}$), which are inserted in $Unevaluated$. Available threads will pop these successors from $Unevaluated$ and evaluate them. When all successors of s_0 have been evaluated, they are all inserted in $Open$. Next, some thread removes $s_{1,1}$ ($satisfied(s_{1,1}) = true$), generates its successors ($s_{2,1}, s^1_{2,2}, .., s^x_{2,2}$), and inserts them in $Unevaluated$. While generating the successors of $s_{1,1}$, other threads may try to pop a state from $Open$, but since the top state at that time ($s^i_{1,2}$) is not in the BTS ($satisfied(s^i_{1,2}) = false$), $Open$ will remain untouched. After the successors of $s_{1,1}$ have been inserted in $Unevaluated$, the available threads will remove them from $Unevaluated$ and evaluate them. The search continues similarly until s_{goal} is found. Each time a state's successors are generated, multiple available threads evaluate the successors in parallel. This clearly improves thread utilization compared to BTS-constrained search without SGE, which only uses one thread to expand (generate and evaluate successors of) each state so that only 1 thread is active throughout the search space in Figure 1.

The basic idea of decoupling state generation and state evaluation is similar to that of ePA*SE, which decouples state generation and *edge evaluations* in a parallel A* (Mukherjee, Aine, and Likhachev 2022), where an edge evaluation is the computation required to evaluate the application of an operator, e.g., collision checking using a simulation model in robot motion planning). In both cases, using only a single thread to fully process a state expansion causes a bottleneck forcing other threads to be idle, as search algorithm constraints for constrained parallel GBFS and A* requires waiting until that expansion is fully processed, so decomposing the expansion process and parallelizing it among the available threads leads to better processor utilization. Because the requirements and objectives of GBFS (satisficing search) and A* (cost-optimal search) differ, the implementation of SGE is somewhat simpler (using an $Unevaluated$ queue instead of dummy/real edges as in ePA*SE).

SGE and Thread Utilization Consider a situation where all threads are currently simultaneously available. In unconstrained parallel BFS with k threads, the top k nodes in $Open$ can be simultaneously expanded, so all k threads will be utilized. On the other hand, in CPGBFS *without* SGE, if there are currently a states which satisfy the expansion constraint, then the number of states which can be simultaneously expanded is $\min(a, k)$, and $k - \min(a, k)$ threads will be idle until more states in $Open$ satisfy the expansion constraint. With SGE, although there will be a brief time when only $\min(a, k)$ threads are active (the generation phase), the number of active threads while the successors of the a states are being evaluated will be $\min(ab, k)$, where b is the average number of previously unevaluated successors per state.

SGE and search behavior The state expansion order of a parallel search algorithm A with SGE will differ from the expansion order of A without SGE. Although it is nontrivial to precisely characterize the difference between the expan-

sion order of an algorithm with and without SGE, a simple approximation is that executing a parallel search algorithm A with SGE on k threads is somewhat similar to executing A without SGE on $m < k$ threads, where each of the m "threads" is faster than each of the actual k threads.

As a simple example, consider searching a state space which is a tree with uniform branch factor 2, where the heuristic evaluation function computation is the computational bottleneck, and assume that $Open$ currently contains many nodes. With $k = 16$ threads, KPGBFS will expand 16 states at a time – each thread expands 1 state, where the expansion includes generation and heuristic evaluation of the state's 2 successors. In contrast, KPGBFS with SGE would be evaluating 16 states at a time – each thread, after quickly generating the successors of a state, would then be assigned to evaluate 1 successor state, essentially the same as KPGBFS without SGE expanding 8 states, i.e., similar to KPGBFS without SGE running on 8 threads.

SGE and expansion constraints The state expansion constraints (i.e., the $satisfies$ check) for the various CPGBFS algorithms known to date are defined based on: (a) comparisons between the h-value of a state's parent and the h-values of the siblings of s (for PUHF), or (b) $h(s)$ vs. the h-values of other states currently being expanded (for PUHF2, PUHF3, PUHF4). Therefore, distributing the evaluation of the successors of s among multiple threads has no impact on the correctness of the expansion constraint (i.e., the guarantee that the node being expanded is in the BTS).

Overall effect on search performance SGE can be applied to any parallel best-first search algorithm such as PUHF and KPGBFS. The impact on performance will depend on the extent to which state expansion/evaluation is a performance bottleneck. In the case of a constrained algorithm such as PUHF, SGE should allow higher utilization of threads that would otherwise be idle (waiting for a state which satisfies expansion constraints), resulting in significantly higher state evaluation rates, which should in turn result in better overall search performance (more problems solved within a given time limit).

On the other hand, in the case of an unconstrained algorithm such as KPGBFS, the evaluation rate should be minimally affected, but search performance may be slightly affected because of the different state expansion order due to SGE.

6 Experimental Evaluation of SGE

In this section, we experimentally evaluate SGE. We emphasize that the goal of this experimental study is to evaluate and understand SGE as an implementation technique which can be applied to a wide range of parallel best-first search algorithms. Thus, we focus on comparing PUHF2 vs. PUHF2$_S$ to evaluate the effect of SGE on constrained parallel best-first search, and we compare KPGBFS vs. KPGBFS$_S$ to evaluate the effect of SGE on unconstrained parallel best-first search. Comparing constrained parallel search vs. unconstrained parallel search (e.g., PUHF2$_S$ vs. KPGBFS) is beyond the scope of this study.

To evaluate SGE, we use the satisficing instances of the Autoscale-21.11 benchmark set (42 STRIPS domains, 30 instances/domain, 1260 total instances) (Torralba, Seipp, and Sievers 2021), an improved benchmark suite based on the IPC classical planning benchmarks. All search algorithms use the FF heuristic (Hoffmann and Nebel 2001). Each run has a run time limit of 5 minutes and 3 GB RAM/thread (e.g., 24 GB total for a 8-thread run) limit on a Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz processor.

All tie-breaking is First-In-First-Out.

We evaluate all algorithms on $k \in \{4, 8, 16\}$ threads. We also report some results for single-threaded GBFS as a baseline.

Table 1a and Figure 2 compares the state evaluation rates of the algorithms. Table 1b and Figure 3 compares the number of states expanded by the algorithms. Table 1c and Figure 4 compares search time by the algorithms. Table 1d shows the speedup compared to single-threaded GBFS.

Table 2a shows the number of instances solved by each algorithm, and Table 2b shows the number of problems solved by algorithm x but not by algorithm y for $x, y \in \{\text{KPGBFS}, \text{KPGBFS}_S, \text{PUHF2}, \text{PUHF2}_S\}$.

In Tables 1a-1d, we include only the 412 instances solved by all algorithms so that means can be computed. Figures 2–4 show all instances – instances which were not solved (due to either timeout or memory exhaustion) are shown as either $x = fail$ or $y = fail$.

Evaluation Rates First, comparing the state evaluation rates of KPGBFS and PUHF2 (Table 1a, Figure 2), we confirm that as shown in previous work (Shimoda and Fukunaga 2023), PUHF2 has a significantly lower evaluation rate than KPGBFS. The only difference between KPGBFS and PUHF2 is that KPGBFS can always expand the top state in $Open$ while PUHF2 only expands the top node in $Open$ if a more restrictive constraint is satisfied in Algorithm 1. Thus, the lower state evaluation rate in PUHF2 is due to threads waiting idly when $top(Open)$ does not satisfy the constraint.

Comparing the state evaluation rates (states/second) of PUHF2$_S$ and PUHF2 (Table 1a, Figure 2), we see that for $k \in \{4, 8, 16\}$ threads, PUHF2$_S$ has a significantly higher evaluation rate than PUHF2. This shows that SGE successfully achieves the goal of improving the evaluation rate for constrained parallel best-first search.

Comparing KPGBFS and KPGBFS$_S$, the evaluation rate of KPGBFS$_S$ is somewhat lower than that of KPGBFS for $k \in \{4, 8, 16\}$ threads. Thus, management of the overhead of a separate $Unevaluated$ queue introduced by SGE imposes a noticeable penalty – in the comparison of KPGBFS vs KPGBFS$_S$ in Figure 2, the effect of this overhead is noticeable for the problems with the highest evaluation rate (top right).

Number of States Expanded The number of states expanded to solve an instance measures search efficiency. For all values of k, PUHF2$_S$ expanded fewer states than PUHF2, and KPGBFS$_S$ expanded fewer states than KPGBFS, so SGE has the effect of reducing the search required to solve problem instances for both constrained and unconstrained parallel GBFS.

#threads	1 thread	4 threads	8 threads	16 threads
GBFS	5791		-	
KPGBFS	-	18502	33944	58790
KPGBFS$_S$	-	17683	32074	53880
PUHF2	-	14994	22716	33632
PUHF2$_S$	-	16350	26126	40097

(a) State evaluation rate (geometric mean)

#threads	1 thread	4 threads	8 threads	16 threads
GBFS	1123		-	
KPGBFS	-	1896	2559	3400
KPGBFS$_S$	-	1601	2016	2557
PUHF2	-	1631	2009	2422
PUHF2$_S$	-	1456	1634	1945

(b) Number of states expanded (geometric mean)

#threads	1 thread	4 threads	8 threads	16 threads
GBFS	1.2605		-	
KPGBFS	-	0.672	0.497	0.385
KPGBFS$_S$	-	0.588	0.401	0.302
PUHF2	-	0.723	0.595	0.498
PUHF2$_S$	-	0.601	0.426	0.334

(c) Search time (geometric mean)

#threads	1 thread	4 threads	8 threads	16 threads
GBFS	1.0		-	
KPGBFS	-	3.17	10.22	12.69
KPGBFS$_S$	-	3.06	8.29	12.38
PUHF2	-	2.38	5.22	6.67
PUHF2$_S$	-	3.81	7.02	11.03

(d) Speedup (Search time[GBFS]/Search time , arithmetic mean)

Table 1: Comparison on Autoscale-21.11 IPC-based planning benchmark set. Means for 412 instances solved by all algorithms

A possible explanation for this is that as explained in Section 5, an algorithm with SGE tends to behave as if there were fewer threads, and previous work has shown that as the number of threads increases, the search efficiency of parallel search tends to decrease (Kuroiwa and Fukunaga 2019), so behaving as if there were fewer threads leads to better search efficiency.

Search Time and Speedup The wall-clock runtime required to find a solution (search time) is a measure of overall performance.

As a result of higher state evaluation rate and comparable search efficiency, PUHF2$_S$ has significantly lower search time than PUHF2. On the other hand, KPGBFS$_S$ only runs slightly faster than KPGBFS, because the improved number of states expanded due to SGE is offset by the lower evaluation rate.

Speedup compared to single-threaded search (GBFS) measures how successful a method is as a *parallelization* technique. Comparing PUHF2$_S$ and PUHF2, we observe that PUHF2$_S$ achieves almost k-fold speedup relative to single-threaded GBFS, while the speedup for PUHF2 is far from k.

#threads	1 thread	4 threads	8 threads	16 threads
GBFS	448		-	
KPGBFS	-	510	534	567
KPGBFS$_S$	-	507	529	565
PUHF2	-	500	517	547
PUHF2$_S$	-	507	524	551

(a) Number of instances solved

	KPGBFS	KPGBFS$_S$	PUHF2	PUHF2$_S$
KPGBFS	-	34	41	43
KPGBFS$_S$	32	-	38	33
PUHF2	21	20	-	20
PUHF2$_S$	27	19	24	-

(b) For $k = 16$ threads, number of problems solved by algorithm (row) but not by algorithm (column), e.g., KPGBFS solved 41 problems not solved by PUHF2

Table 2: Coverage results on Autoscale-21.11 IPC-based planning benchmark set.

Coverage Table 2a shows that applying SGE improves the number of instances solved by PUHF2, but slightly lowers the number of instances solved by KPGBFS. Table 2b shows that for $k = 16$ threads, each algorithm (PUHF2, PUHF2$_S$, KPGBFS, KPGBFS$_S$) solves some instances not solved by other algorithms (similar for 4 and 8 threads, not shown due to space).

7 Discussion and Conclusion

We proposed SGE, an approach to increase state evaluation rates in constrained parallel search algorithm by separating successor generation and evaluation. We showed SGE significantly increases the state evaluation rate of PUHF2, resulting in improved overall performance for PUHF2$_S$ compared to PUHF2. Preliminary experiments with PUHF and PUHF3 have yielded similar results, suggesting that SGE consistently yields performance improvements for parallel GBFS algorithms with state expansion constraints. For unconstrained search such as KPGBFS, SGE results in a decrease in evaluation rate, as there are no bottlenecks in unconstrained search, and the additional complexity of SGE imposes an overhead.

We also showed that SGE results in more efficient search (fewer expansions to solve an instance) for both PUHF2 (constrained search) as well as for KPGBFS (unconstrained search). While we hypothesize that this is because SGE with k threads behaves as if there were fewer, faster threads as explained in Section 5, a deeper experimental investigation of the expansion order with and without SGE is an avenue for future work.

The batch successor insertion state bottleneck addressed by SGE (Section 4) is a byproduct of implementing a parallel search algorithm which is limited to expanding a set of states similar to the states expanded by single-threaded GBFS with a standard eager evaluation policy, where states are evaluated immediately after they are generated and be-

fore being inserted in $Open$. In lazy (deferred) evaluation (Richter and Helmert 2009), where states are not evaluated before insertion into $Open$ and are inserted into $Open$ based on their parent's f-value (and later evaluated when they are expanded), the batch successor insertion bottleneck would not apply. Search with lazy evaluation behaves quite differently than search with eager evaluation, and parallel satisficing search with lazy evaluation is an avenue for future work.

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.

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.

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.

Mukherjee, S.; Aine, S.; and Likhachev, M. 2022. ePA*SE: Edge-Based Parallel A* for Slow Evaluations. In Chrpa, L.; and Saetti, A., eds., *Proc. SOCS*, 136–144.

Phillips, M.; Likhachev, M.; and Koenig, S. 2014. PA*SE: Parallel A* for Slow Expansions. In *Proc. ICAPS*, 208–216.

Richter, S.; and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *Proc. ICAPS*, volume 19, 273–280.

Shimoda, T.; and Fukunaga, A. 2023. Improved Exploration of the Bench Transition System in Parallel Greedy Best First Search. In *Proc. SOCS*, 74–82.

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

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.

Wilt, C. M.; and Ruml, W. 2014. Speedy Versus Greedy Search. In *Proc. SOCS*, 184–192.

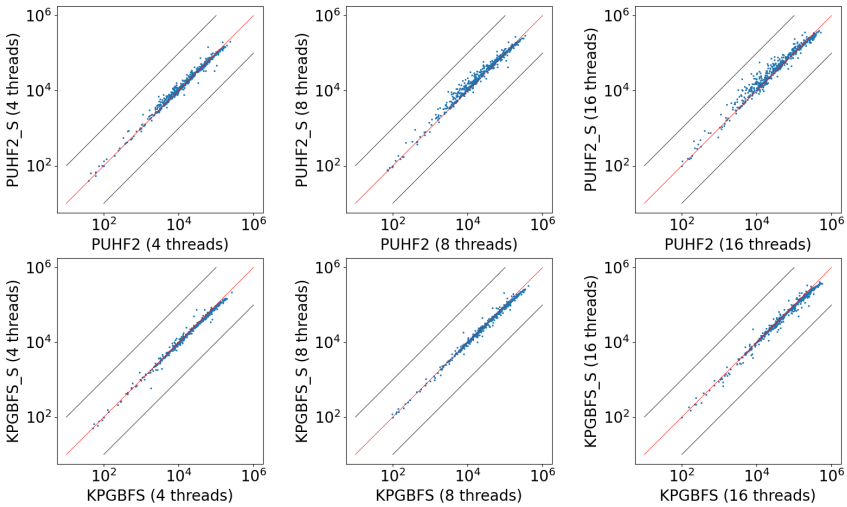

Figure 2: State evaluation rates (states/second), diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

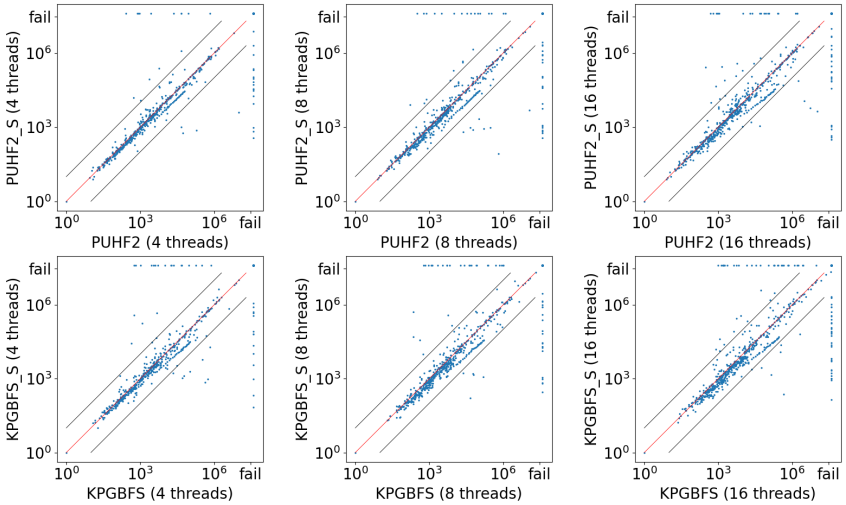

Figure 3: Number of states expanded, "fail"= out of time/memory, diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$

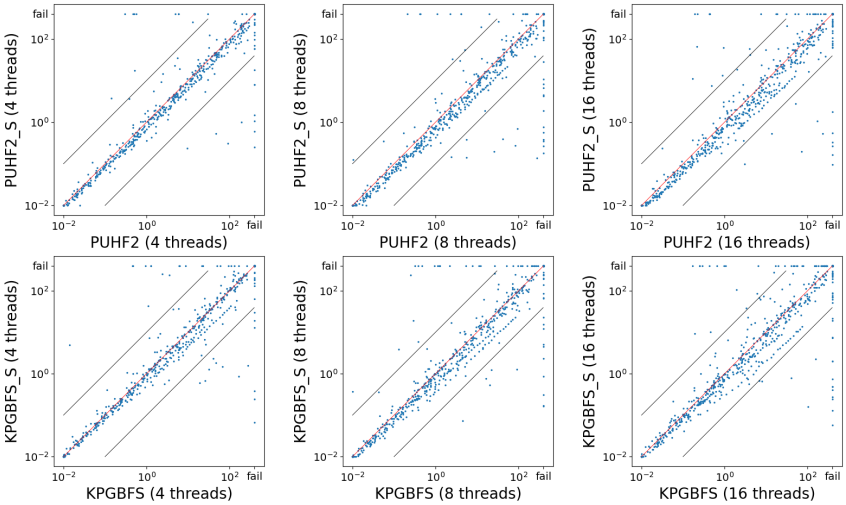

Figure 4: Search time (seconds) "fail"= out of time/memory, diagonal lines are $y = 0.1x$, $y = x$, and $y = 10x$