Best-First Heuristic Search for Multi-Core Machines

Ethan Burns[1] and **Seth Lemons**[1] and **Rong Zhou**[2] and **Wheeler Ruml**[1]
[1]Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
eaburns, seth.lemons, ruml at cs.unh.edu

[2]Embedded Reasoning Area
Palo Alto Research Center
Palo Alto, CA 94304 USA
rzhou at parc.com

Abstract

To harness modern multi-core processors, it is imperative to develop parallel versions of fundamental algorithms. In this paper, we present a general approach to best-first heuristic search in a shared-memory setting. Each thread attempts to expand the most promising open nodes. By using abstraction to partition the state space, we detect duplicate states without requiring frequent locking. We allow speculative expansions when necessary to keep threads busy. We identify and fix potential livelock conditions in our approach, verifying its correctness using temporal logic. In an empirical comparison on STRIPS planning, grid pathfinding, and sliding tile puzzle problems using an 8-core machine, we show that A* implemented in our framework yields faster search than improved versions of previous parallel search proposals. Our approach extends easily to other best-first searches, such as Anytime weighted A*.

1 Introduction

It is widely anticipated that future microprocessors will not have faster clock rates, but rather more computing cores per chip. Tasks for which there do not exist effective parallel algorithms will suffer a slowdown relative to total system performance. In artificial intelligence, heuristic search is a fundamental and widely-used problem solving framework. In this paper, we develop a parallel version of best-first search, a popular method underlying algorithms such as A* [Hart *et al.*, 1968].

In best-first search, two sets of nodes are maintained: *open* and *closed*. Open contains the search frontier: nodes that have been generated but not yet expanded. In A*, open nodes are sorted by f value, the estimated lowest cost for a solution path going through that node. Closed contains all previously expanded nodes, allowing the search to detect duplicated states in the search space and avoid expanding them multiple times. One challenge in parallelizing best-first search is avoiding contention between threads when accessing the open and closed lists. We will use a technique called *parallel structured duplicate detection* (PSDD), originally developed by Zhou and Hansen (2007) for parallel breadth-first search, in order to dramatically reduce contention and allow threads to enjoy periods of synchronization-free search. PSDD requires the user to supply an abstraction function that maps multiple states to a single abstract state, called an $n$block.

In contrast to previous work, we focus on general best-first search. Our algorithm is called Parallel Best-NBlock-First (PBNF).[1] It extends easily to domains with non-uniform and non-integer move costs and inadmissible heuristics. Using PSDD with best-first search in an infinite search space can give rise to livelock, where threads continue to search but a goal is never expanded. We will discuss how these conditions can be avoided in PBNF using a method we call *hot $n$blocks*, as well as our use of formal methods to validate its effectiveness. We study the empirical behavior of PBNF on three popular search domains: STRIPS planning, grid pathfinding, and the venerable sliding tile puzzle. We compare against several previously proposed algorithms, as well as novel improvements of them, using a dual quad-core Intel machine. Our results show that PBNF yields faster search than all other algorithms tested.

2 Previous Work

Early work on parallel heuristic search investigated depth-first approaches [Powley *et al.*, 1990]. But because it does not keep a closed list, depth-first search cannot detect duplicate states and is thus doomed to failure on domains with many duplicate states, such as grid pathfinding and some planning domains.

The simplest approach to parallel best-first search is to have mutual exclusion locks (mutexes) for the open and closed lists and require each thread to acquire the lock before manipulating the corresponding structure. We call this search 'parallel A*' (PA*). As we see below, this naive approach performs worse than serial A*. Parallel Retracting A* (PRA*) [Evett *et al.*, 1995] attempts to avoid contention by assigning separate open and closed lists to each thread. A hashing scheme is used to assign nodes to the appropriate thread when they are generated. (Full PRA* also includes a retraction scheme that reduces memory use in exchange for increased computation time; we do not use that feature in this paper.) The choice of the hashing function is crucial to the

[1]Peanut Butter 'N' (marshmallow) Fluff, also known as a fluffernutter, is a well-known children's sandwich in the USA.

performance of the algorithm, since it determines the way that work is distributed. Note that with PRA* each thread needs a synchronized open list or message queue that other threads can add nodes to. While this is less of a bottleneck than having a single global, shared open list, we will see below that it can still be expensive. We present two variations of this algorithm, the primary difference being hashing function. While PRA* uses a simple representation-based node hashing scheme , APRA* makes use of a state space abstraction. We define an abstraction function with the goal of limiting the number of successors, thus limiting the number of other threads' open lists a given thread will insert nodes into. Abstract states are distributed evenly among all threads in hopes that open nodes will always be available to each thread.

One way of avoiding contention altogether is to allow one thread to handle synchronization of the work done by the other threads. K-Best-First Search (KBFS) [Felner *et al.*, 2003] expands the best k nodes at once, each of which can be handled by a different thread. In our implementation, a master thread takes the k best nodes from open and gives one to each worker. The workers expand their nodes and the master checks the children for duplicates and inserts them into open. This allows open and closed to be used without locking, but requires the master thread to wait for all workers to finish their expansions before handing out new nodes to adhere to a strict k best first ordering. The approach will not scale if node expansion is fast compared to the number of processors.

2.1 Parallel Structured Duplicate Detection

The intention of PSDD is to avoid the need to lock on every node generation. It builds on the idea of structured duplicate detection (SDD), which was originally developed for external memory search [Zhou and Hansen, 2004]. SDD uses an *abstraction function*, a many-to-one mapping from states in the original search space to states in an abstract space. The abstract node to which a state is mapped is called its *image*. An $n$block is the set of nodes in the state space that have the same image in the abstract space. We will use the terms 'abstract state' and '$n$block' interchangeably. The abstraction function creates an *abstract graph* of nodes that are images of the nodes in the state space. If two states are successors in the state space, then their images are successors in the abstract graph.

For efficient duplicate detection, we can equip each $n$block with its own open and closed lists. Note that two nodes representing the same state s will map to the same $n$block b. When we expand s, its children can map only to b's successors in the abstract graph. These $n$blocks are called the *duplicate detection scope* of b because they are the only $n$blocks whose open and closed lists need to be checked for duplicate states when expanding nodes in b.

In parallel SDD (PSDD), the abstract graph is used to find $n$blocks whose duplicate detection scopes are disjoint. These $n$blocks can be searched in parallel without any locking. An $n$block b is considered to be *free* iff none of its successors are being used. Free $n$blocks are found by explicitly tracking $\sigma(b)$, the number of $n$blocks among b's successors that are in use by another processor. An $n$block can only be acquired when its $\sigma = 0$. PSDD only uses a single lock, controlling manipulation of the abstract graph, and it is only acquired by threads when finding a new free $n$block to search.

Zhou and Hansen [2007] used PSDD to parallelize breadth-first heuristic search [Zhou and Hansen, 2006b]. In each thread of the search, only the nodes at the current search depth in an $n$block are searched. When the current $n$block has no more nodes at the current depth, it is swapped for a free $n$block that does have open nodes at this depth. If no more $n$blocks have nodes at this depth, all threads synchronize and then progress to the next depth. An admissible heuristic is used to prune nodes below the current solution upper bound.

2.2 Improvements to PSDD

As implemented by Zhou and Hansen, PSDD uses the heuristic estimate of a node only for pruning; this is only effective if a tight upper bound is already available. To cope with situations where a good bound is not available, we have implemented a novel variation of PSDD that uses iterative deepening (IDPSDD) to increase the bound. As we report below, this approach is not effective in domains such as grid pathfinding that do not have a geometrically increasing number of nodes within successive f bounds.

Another drawback of PSDD is that breadth-first search cannot guarantee optimality in domains where operators have differing costs. In anticipation of these problems, Zhou and Hansen [2004] suggest two possible extensions to their work, best-first search and a speculative best-first layering approach that allows for larger layers in the cases where there are few nodes (or $n$blocks) with the same f value. To our knowledge, we are the first to implement and test these algorithms. Best-first PSDD (BFPSDD) uses f value layers instead of depth layers. This means that all nodes that are expanded in a given layer have the same (lowest) f value. BFPSDD provides a best-first search order, but may incur excessive synchronization overhead if there are few nodes in each f layer (as in grid pathfinding or when using a weighted heuristic). To ameliorate this, we enforce that at least m nodes are expanded before abandoning a non-empty $n$block. (Zhou and Hansen credit Edelkamp and Schrödl [2000] with this idea.) When populating the list of free $n$blocks for each layer, all of the $n$blocks that have nodes with the current layer's f value are used or a minimum of k $n$blocks are added where k is four times the number of threads. This allows us to add additional $n$blocks to small layers in order to amortize the cost of synchronization. The value four gave better performance than other values tried. In addition, we tried an alternative implementation of BFPSDD that used a range of f values for each layer. This implementation did not perform as well and we do not present results for it. With either of these enhancements, threads may expand nodes with f values greater than that of the current layer. Because the first solution found may not be optimal, search continues until all remaining nodes are pruned by the incumbent solution.

3 Parallel Best-NBlock-First (PBNF)

Ideally, all threads would be busy expanding $n$blocks that contain nodes with the lowest f values. To achieve this,

```
1. while there is an nblock with open nodes
2.     lock; b ← best free nblock; unlock
3.     while b is no worse than the best free nblock or
4.         we've done fewer than m expansions
5.       n ← best open node in b
6.       if f(n) > f(incumbent), prune all open nodes in b
7.       else if n is a goal
8.         if f(n) < f(incumbent)
9.           lock; incumbent ← n; unlock
10.      else for each child c of n
11.        insert c in the open list of the appropriate nblock
```

Figure 1: A sketch of basic PBNF search, showing locking.

we combine PSDD's duplicate detection scopes with an idea from the Localized A* (LA*) algorithm of Edelkamp and Schrödl [2000]. LA*, which was designed to improve the locality of external memory search, maintains sets of nodes that reside on the same memory page. Decisions of which set to process next are made with the help of a heap of sets ordered by the minimum f value in each set. By maintaining a heap of free $n$blocks ordered on their best f value, we can approximate our ideal parallel search. We call this algorithm Parallel Best-NBlock-First (PBNF).

In PBNF, threads use the heap of free $n$blocks to acquire the free $n$block with the best open node. A thread will search its acquired $n$block as long as it contains nodes that are better than those of the $n$block at the front of the heap. If the acquired $n$block becomes worse than the best free one, the thread will attempt to release its current $n$block and acquire the better one. There is no layer synchronization, so the first solution found may be suboptimal and search must continue until all open nodes have f values worse than the incumbent. Figure 1 shows pseudo-code for the algorithm.

Because PBNF is only approximately best-first, we can introduce optimizations to reduce overhead. It is possible that an $n$block has only a small number of nodes that are better than the best free $n$block, so we avoid excessive switching by requiring a minimum number of expansions. Our implementation also attempts to reduce the time a thread is forced to wait on a lock by using the `try_lock` function whenever possible. Rather than sleeping if a lock cannot be acquired, `try_lock` immediately returns failure. This allows a thread to continue expanding its current $n$block if the lock is busy. Both of these optimizations can introduce additional 'speculative' expansions that would not have been performed in a serial best-first search.

3.1 Livelock

The greedy free-for-all order in which PBNF threads acquire free $n$blocks can lead to livelock in domains with infinite state spaces. Because threads can always acquire new $n$blocks without waiting for all open nodes in a layer to be expanded, it is possible that the $n$block containing the goal will never become free. This is because we have no assurance that all $n$blocks in its duplicate detection scope will ever be unused at the same time. To fix this, we have developed a method called 'hot $n$blocks' where threads altruistically release their $n$block if they are interfering with a better $n$block. We call this enhanced algorithm 'Safe PBNF.'

We define the *interference scope* of an $n$block b to be those $n$blocks whose duplicate detection scopes overlap with b's. In Safe PBNF, whenever a thread checks the heap of free $n$blocks, it also ensures that its $n$block is better than any of those in its interference scope. If it finds a better one, it flags it as 'hot.' Any thread that finds a hot $n$block in its interference scope releases its $n$block in an attempt to free the hot $n$block. For each $n$block b, $\sigma_h(b)$ tracks the number of hot $n$blocks in b's interference scope. If $\sigma_h(b) \neq 0$, b is removed from the heap of free $n$blocks. This ensures that a thread will not acquire an $n$block that is preventing a hot $n$block from becoming free.

There are three cases to consider when attempting to set an $n$block b to hot with an undirected abstract graph: 1) none of the $n$blocks in the interference scope of b are hot, so b can be set to hot; 2) a better $n$block in the interference scope of b is already hot, so b must not be set to hot; and 3) an $n$block b' that is worse than b is in the interference scope of b and is already hot. In this case b' must be un-flagged as hot (updating σ_h values appropriately) and in its place b is set to hot. (This reset is also done if threads ever notice a hot $n$block worse than themselves.) Directed graphs have two additional cases: 4) an $n$block b' has b in its interference scope, b' is hot and b' is worse than b, then un-flag b' as hot and set b to hot; 5) an $n$block b' has b in its interference scope, b' is hot and b' is better than b, then do not set b to hot, leave b' hot. This scheme ensures that there are never two hot $n$blocks interfering with one another and that the $n$block that is set to hot is the best $n$block in its interference scope. As we verify below, this approach guarantees the property that if an $n$block is flagged as hot it will eventually become free. Full pseudo-code for Safe PBNF is given in Appendix A.

A Formal Model

To help ensure that the hot $n$block method works properly, we have constructed a formal model using the temporal logic TLA+ [Lamport, 2002]. The model describes an abstract version of the hot $n$block procedure in which the abstract graph is connected in a ring (each $n$block is connected to two adjacent $n$blocks). Additionally we modelled the hot $n$blocks method using a directed abstract graph with eight $n$blocks. The search procedure itself is not modeled, since this is not required to prove the desired properties. During a search action in the model, a thread can optionally set any $n$block that it is interfering with to hot if that $n$block is not already hot, if nothing in its interference scope is hot and if it is not in the interference scope of another hot $n$block. These cases correspond to the five cases mentioned above, and the final one is only required if the abstract graph is directed. After searching, a thread will release its $n$block and try to acquire a new free $n$block.

While this model is a greatly abstracted version of the Safe PBNF algorithm, it is actually able to show a stronger property. Since the model does not take into account the f values of nodes within each $n$block, a thread has the ability to set *any* $n$block it interferes with to hot (assuming that no hot interference is created), rather than being restricted to "good" $n$blocks. Using this model, we hope to prove that any $n$block

that is set to hot (regardless of the $n$block's best f value) will eventually be added to the heap of free $n$blocks. We have used the TLC bounded model checker [Yu *et al.*, 1999] to show that livelock arises in the plain (unsafe) algorithm and that the hot $n$block method is effective at fixing the livelock for all cases of up to 12 $n$blocks and three threads on the ring abstraction and 8 nblocks and three threads on the directed abstraction.

4 Empirical Evaluation

We have implemented and tested the parallel heuristic search algorithms discussed above on three different benchmark domains: grid pathfinding, the sliding tile puzzle, and STRIPS planning. The algorithms were programmed in C++ using the POSIX threading library and run on dual quad-core Intel Xeon E5320 1.86GHz processors with 16Gb RAM, except for the planning results, which were written in C and run on dual quad-core Intel Xeon X5450 3.0GHz processors limited to roughly 2GB of RAM. All open lists and free lists are binary heaps, and closed lists are hash tables. PRA* and APRA* use queues for incoming nodes, and a hash table is used to detect duplicates in both open and closed. For grids and sliding tiles, we used the jemalloc library [Evans, 2006], a special multi-thread-aware malloc implementation, instead of the standard glibc (version 2.7) malloc, because we have performed experiments demonstrating that the latter scales poorly above 6 threads. We configured jemalloc to use 32 memory arenas per CPU. In planning, a custom memory manager was used which is also thread-aware and uses a memory pool for each thread. For the following experiments we show the performance of each algorithm with its best parameter settings (e.g., minimum number of expansions and abstraction granularity) which we determined by experimentation.

4.1 Grid Pathfinding

We tested on grids 2000 cells wide by 1200 cells high, with the start in the lower left and the goal in the lower right. Cells are blocked with probability 0.35. We test two cost models (discussed below) and both four-way and eight-way movement. The abstraction function we used maps blocks of adjacent cells to the same abstract state, forming a coarser abstract grid overlaid on the original space. For this domain we are able to tune the size of the abstraction and our results show the best abstraction size for each algorithm where it is relevant. SafePBNF, PBNF, BFPSDD use 64 minimum expansions. They and APRA* use 6400 nblocks. PSDD uses 625 nblocks. Each plot includes a horizontal line representing the performance of a serial A* search.

Four-way Unit Cost: In the unit cost model, each move has the same cost. The upper right plot in Figure 2 shows the performance of previously proposed algorithms for parallel best-first search on unit-cost four-way movement path planning problems. The y-axis represents elapsed wall-clock time. Error bars indicate 95% confidence intervals on the mean and algorithms in the legend are ordered on their average performance. The figure in the upper right, shows only algorithms that are above the A* line, while the figure in the upper left shows the more competitive algorithms. While PSDD seems to scale reasonably as threads are added, the lack of a tight upper bound hurts its performance. We have implemented the IDPSDD algorithm, but the results are not shown on the grid pathfinding domains. The non-geometric growth in the number of states when increasing the cost bound leads to very poor performance with iterative deepening.

The upper left plot in Figure 2 shows our novel APRA*, BFPSDD, PBNF and Safe PBNF algorithms on the same unit-cost four-way problems. PBNF and Safe PBNF are superior to any of the other algorithms, with steadily decreasing solution times as threads are added and an average speed-up over serial A* of greater than 4x when using eight threads. The checking of interference scopes in Safe PBNF adds a small time overhead. The BFPSDD algorithm also gives good results on this domain, surpassing the speed of APRA* after 3 threads. APRA*'s performance gets gradually worse for more than four threads.

Four-way Life Cost: Moves in the life cost model have a cost of the row number of the state where the move was performed—moves at the top of the grid are free, moves at the bottom cost 1200. This differentiates between the shortest and cheapest paths. The bottom left plot in Figure 2 shows these results. PBNF and Safe PBNF have the best performance for two threads and beyond. The BFPSDD algorithm has the next best performance, following the same general trend as PBNF. The APRA* algorithm does not seem to improve its performance beyond four threads.

Eight-way Unit Cost: In our eight-way movement path planning problems, horizontal and vertical moves have cost one, but diagonal movements cost $\sqrt{2}$. These real-valued costs make the domain different from the previous two path planning domains. The top middle panel shows that PBNF and Safe PBNF give the best performance. While APRA* is initially better than BFPSDD, it does not scale and is slower than BFPSDD at 6 or more threads.

Eight-way Life Cost: This model combines the eight-way movement and the life cost models; it is the most difficult path planning domain presented in this paper. The bottom middle panel shows that the two PBNF variants give the best performance when using multiple threads. BFPSDD gives an overall performance profile similar to PBNF, but consistently slower. The APRA* algorithm gives slightly faster solution speeds then BFPSDD, but it fails to scale after 5 threads.

4.2 Sliding Tile Puzzle

The sliding tile puzzle is a common domain for benchmarking heuristic search algorithms. For these results, we use forty-three of the easiest Korf 15-puzzle instances (ones that were solvable by A* in 15GB of memory) because they are small enough to fit into memory, but are difficult enough to differentiate algorithmic performance.

We found that a smaller abstraction which only considers the position of the blank and 1-tile did not produce a sufficient number of abstract states for PBNF, IDPSDD, and APRA* to scale as threads were added, so we used one which takes into account the blank, the 1-tile, and the 2-tile. This is because the smaller abstraction does not provide enough free $n$blocks at many points in the search, and threads are forced to contend heavily for the free list. BFPSDD, however, did better with

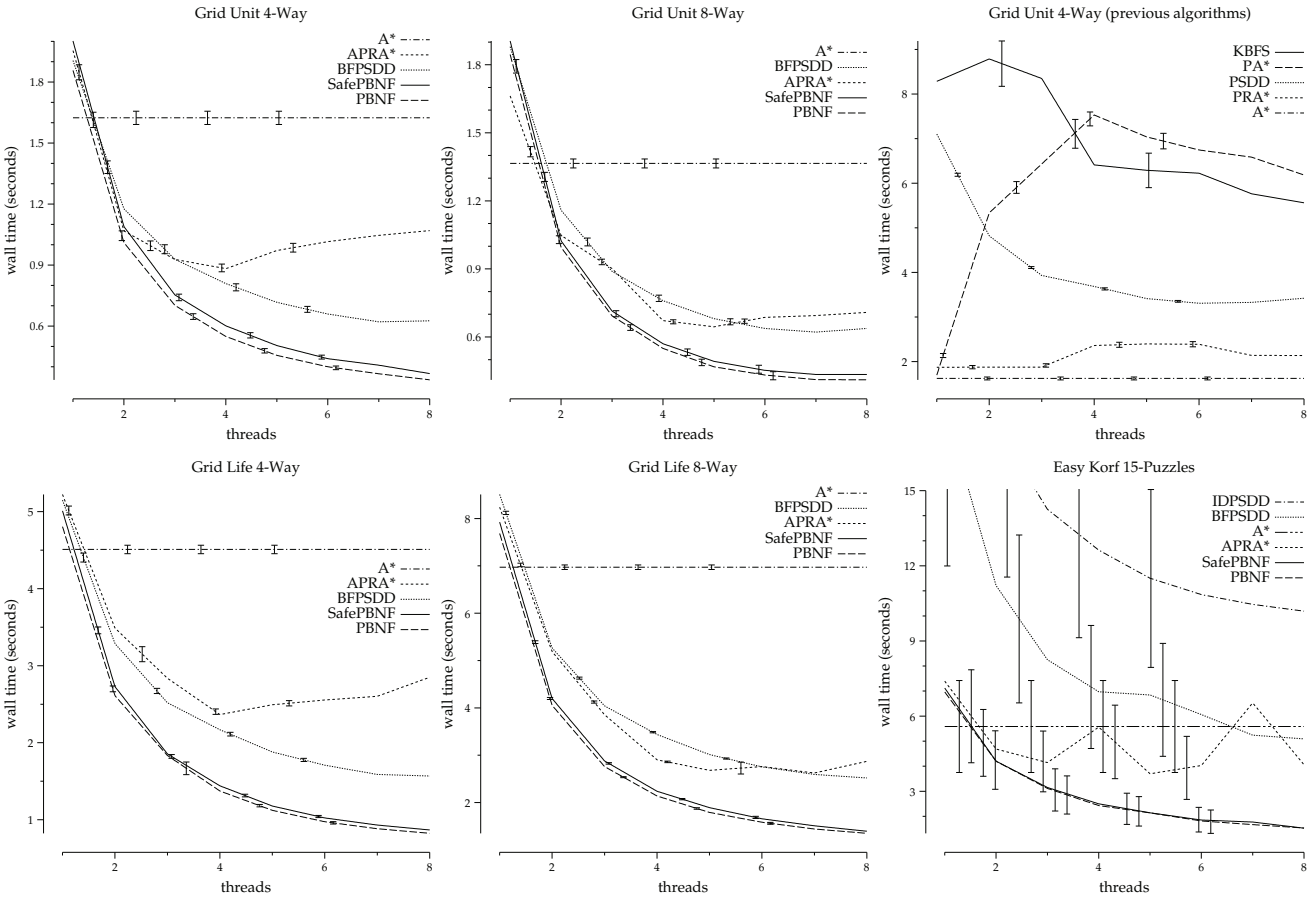

Figure 2: Results on grid path planning and the sliding tiles puzzle.

the smaller abstraction, presumably because it did not require switching between $n$blocks as often because of the narrowly defined layer values. In addition, BFPSDD uses 64 minimum expansions, while PBNF and Safe PBNF only use 32.

The bottom right panel in Figure 2 shows the results for BFPSDD, IDPSDD, APRA*, PBNF and Safe PBNF. The two variants of PBNF show the best performance consistently. The APRA* algorithm has very unstable performance, but often performs better than A*. We found that APRA* had a more difficult time solving some of the larger puzzle instances, consuming much more memory at higher numbers of threads. BFPSDD's performance was poor, but it improves consistently with the number of threads added and eventually gets faster than A*. The IDPSDD algorithm performed much worse than the other algorithms on average, but it exhibited a smooth performance increase as more threads were added.

4.3 STRIPS Planning

In addition to the path planning and sliding tiles domains, the algorithms were embedded into a domain-independent optimal sequential STRIPS planner using regression and the max-pair admissible heuristic of Haslum and Geffner [2000]. Figure 3 presents the results for APRA*, PSDD, BFPSDD, PBNF, and serial A* (for comparison.) A value of 'M' indicates that the program ran out of memory. The PSDD algorithm was given the optimal solution cost as an upper bound to perform pruning in the breadth-first heuristic search. The best result on each problem is marked in bold. PSDD with optimal pruning performed better than PBNF by 3% on one problem. On average at seven threads, Safe PBNF takes 66% of the time taken by PSDD. Interestingly, while plain PBNF was often a little faster than the safe version, it failed to solve two of the problems within our time bound. This is most likely due to livelock, but could also simply be because the hot $n$blocks fix allows Safe PNBF to follow a different search order than PBNF. BFPSDD at 7 threads performs better on average than PSDD, but does not always outperform it. It only gives better speed than either variant of PBNF in the one case where PSDD also does so, aside from the cases where PBNF runs out of memory. We see APRA* following the same trend as we have seen elsewhere, improving at some points, but doing so erratically and often getting worse as threads are added. It also suffered from a large memory footprint, probably because nodes are not checked against the closed list immediately, but only once they are transferred from the queue to the open list.

The right-most column shows the time that was taken by the PBNF and PSDD algorithms to generate the abstraction function. The abstraction is generated dynamically on a per-problem basis and, following Zhou and Hansen [2007], this time was not taken into account in the solution times presented for these algorithms. The abstraction function is gen-

	threads	logistics-6	blocks-14	gripper-7	satellite-6	elevator-12	freecell-3	depots-7	driverlog-11	gripper-8
A*	1	2.3	5.2	118	131	336	199	M	M	M
APRA*	1	1.5	7.1	60	96	213	150	301	322	528
	3	0.76	5.5	51	49	269	112	144	103	M
	5	1.2	3.8	41	66	241	61	M	M	M
	7	0.84	3.7	28	49	169	40	M	M	M
PNBF	1	1.3	6.3	40	68	157	186	M	M	230
	3	0.72	3.8	16	34	56	64	M	M	96
	5	0.58	2.7	11	21	35	44	M	M	61
	7	**0.53**	2.6	**8.6**	17	**27**	**36**	M	M	**48**
Safe PBNF	1	1.2	6.2	40	77	150	127	156	154	235
	3	0.64	2.7	17	24	54	47	63	60	98
	5	0.56	2.2	11	17	34	38	43	39	64
	7	0.62	**2.0**	9.2	14	**27**	37	**35**	**31**	52
BFPSDD	1	2.1	7.8	42	62	152	131	167	152	243
	3	1.1	4.3	18	24	59	57	67	62	101
	5	0.79	3.9	12	20	41	48	48	43	71
	7	0.71	3.4	10	14	32	45	43	35	59
PSDD	1	1.2	6.4	66	62	163	126	160	156	388
	3	0.78	3.6	29	24	63	54	73	63	172
	5	0.68	3.0	22	17	43	46	58	42	121
	7	0.64	2.9	19	**13**	37	44	55	34	106
Abst.	1	0.42	7.9	0.8	1.0	0.7	17	3.6	9.7	1.1

Figure 3: Wall time on STRIPS planning problems.

erated by greedily searching in the space of all possible abstraction functions [Zhou and Hansen, 2006a]. Because the algorithm needs to evaluate one candidate abstraction for each of the unselected state variables, it can be trivially parallelized by having multiple threads working on different candidate abstractions.

5 Discussion and Possible Extensions

We have shown that previously proposed algorithms for parallel best-first search can be much slower than running A* serially. We presented a novel hashing function for PRA* that takes advantage of the locality of a search space and gives superior performance. While APRA* does not perform as well as the best algorithms in our study, it does have the advantage of being very simple to implement. We also found that the original breadth-first PSDD algorithm does not give competitive behavior without a tight upper bound for pruning. We implemented a novel extension to PSDD, BFPSDD, that gives reasonable performance on all domains we tested. Our experiments, however, demonstrate that the new PBNF algorithm gives robust results and almost always outperforms the other algorithms.

The PBNF algorithm outperforms the PSDD algorithm because of the lack of layer-based synchronization and a better utilization of heuristic cost-to-go information. Another less obvious reason why PBNF may perform better is because a best-first search can have a larger frontier size than the breadth-first heuristic search used by the PSDD algorithm. This larger frontier size will tend to create more $n$blocks containing open search nodes there will be more disjoint duplicate detection scopes with nodes in their open lists and, therefore, the possibility of increased parallelism.

Some of our results show that, even for a single thread, PBNF can outperform a serial A* search (see Figure 3). This can be attributed to the speculative behavior of the PBNF algorithm. Since PBNF uses a minimum number of expansions before testing if it should switch to an $n$block with better f values, it will search some sub-optimal nodes that A* would not search. In order to get optimal solutions, PBNF acts as an anytime algorithm; it stores incumbent solutions and prunes until it can prove that it has an optimal solution. Zhou and Hansen show that this approach has the ability to perform better than A* [Hansen and Zhou, 2007] because of upper bound pruning. We are currently exploring the use of a weighted heuristic function with the PBNF algorithm to increase the amount of speculation. In our preliminary tests with a PBNF variant modeled after Anytime weighted A* [Hansen and Zhou, 2007] using a weight of 1.2, we were able to solve some 15-puzzles twice as fast as standard Safe PBNF and even solve some for which PBNF ran out of memory. The use of PBNF as a framework for additional best-first heuristic searches is an exciting area for future work.

6 Conclusion

We have presented Parallel Best-NBlock-First, a parallel best-first heuristic search algorithm that combines the duplicate detection scope idea from PSDD with the heap of sets and speculative expansion ideas from LA*. PBNF approximates a best-first search ordering while trying to keep all threads busy. To perform PBNF safely in parallel, it is necessary to avoid potential livelock conditions. For this purpose, we presented a 'hot $n$block' method and used model checking to verify its correctness. In an empirical evaluation on STRIPS planning, grid pathfinding, and the sliding tile puzzle, we found that the PBNF algorithm is most often the best among those we tested across a wide variety of domains. It is also easy to use the PBNF framework to implement additional best-first search algorithms, including weighted A* and anytime heuristic search.

Acknowledgements

We gratefully acknowledge support from NSF grant IIS-0812141 and helpful suggestions from Jordan Thayer.

References

[Edelkamp and Schrödl, 2000] Stefan Edelkamp and Stefan Schrödl. Localizing A*. In *AAAI-2000*, pages 885–890.

[Evans, 2006] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of BSDCan 2006*.

[Evett *et al.*, 1995] Matthew Evett, Ambuj Mahanti, Dana Nau, James Hendler, and James Hendler. PRA*: Massively parallel heuristic search. *J. Par. and Dist. Computing*, 25:133–143, 1995.

[Felner *et al.*, 2003] Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: K best-first search. *Annals of Math. and A. I.*, 39:2003.

[Hansen and Zhou, 2007] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *JAIR*, 28:267–297, 2007.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, July 1968.

[Haslum and Geffner, 2000] Patrik Haslum and Hctor Geffner. Admissible heuristics for optimal planning. In *ICAPS*, pages 140–149, 2000.

[Lamport, 2002] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. 2002.

[Powley *et al.*, 1990] Curt Powley, Chris Ferguson, and Richard E. Korf. Parallel heuristic search: two approaches. In V. Kumar, P. S. Gopalakrishnan, and L. N. Kanal, eds, *Parallel algorithms for machine intelligence and vision*, pp 42–65. Springer, 1990.

[Yu *et al.*, 1999] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pp 54–66. Springer, 1999.

[Zhou and Hansen, 2004] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *Proc. of AAAI-2004*.

[Zhou and Hansen, 2006a] R. Zhou and E. Hansen. Domain-independent structured duplicate detection. In *Proc. of AAAI-2006*, pages 1082–1087, 2006.

[Zhou and Hansen, 2006b] Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. *AIJ*, 170(4-5):385 – 408, 2006.

[Zhou and Hansen, 2007] Rong Zhou and Eric A. Hansen. Parallel structured duplicate detection. In *Proc. of AAAI-2007*.

A Pseudo-code for Safe PBNF

search(initial node)
1. insert initial node into open
2. for each $p \in$ *processors*, *threadsearch*()
3. while threads are still running, *wait*()
4. return *incumbent*

threadsearch()
1. $b \leftarrow$ NULL
2. while not done
3. $b \leftarrow$ *nextnblock*(b)
4. *exp* $\leftarrow 0$
5. while $\neg$*shouldswitch*(b, exp)
6. $n \leftarrow$ best open node in b
7. if $n >$ *incumbent* then prune n
8. if n is a goal then
9. if $n <$ *incumbent* then
10. lock; *incumbent* $\leftarrow n$; unlock
11. else if n is not a duplicate then
12. *children* $\leftarrow$ *expand*(n)
13. for each *child* $\in$ *children*
14. insert *child* into open of appropriate nblock
15. *exp* $\leftarrow$ *exp* $+ 1$

shouldswitch(b, exp)
1. if b is empty then return true
2. if *exp* $<$ *min-expansions* then return false
3. *exp* $\leftarrow 0$
4. if *best*(*freelist*) $< b$ or *best*(*interferenceScope*(b)) $< b$ then
5. if *best*(*interferenceScope*(b)) $<$ *best*(*freelist*) then
6. *sethot*(*best*(*interferenceScope*(b)))
7. return true
8. lock
9. for each $b' \in$ *interferenceScope*(b)
10. if *hot*(b') then *setcold*(b')
11. unlock
12. return false

sethot(b)
1. lock
2. if $\neg$*hot*(b) and $\sigma(b) > 0$
3. and $\neg\exists i \in$ *interferenceScope*(b) : $i < b \wedge$ *hot*(i) then
4. *hot*(b) $\leftarrow$ true
5. for each $n' \in$ *interferenceScope*(b)
6. if *hot*(n') then *setcold*(n')
7. if $\sigma(n') = 0$ and $\sigma_h(n') = 0$
8. and n' is not empty then
9. *freelist* $\leftarrow$ *freelist* $\setminus \{n'\}$
10. $\sigma_h(n') \leftarrow \sigma_h(n') + 1$
11. unlock

setcold(b)
1. *hot*(b) $\leftarrow$ false
2. for each $n' \in$ *interferenceScope*(b)
3. $\sigma_h(n') \leftarrow \sigma_h(n') - 1$
4. if $\sigma(n') = 0$ and $\sigma_h(n') = 0$ and n' is not empty then
5. if *hot*(n') then
6. *setcold*(n')
7. *freelist* $\leftarrow$ *freelist* $\cup \{n'\}$
8. wake all sleeping threads

release(b)
1. for each $b' \in$ *interferenceScope*(b)
2. $\sigma(b') \leftarrow \sigma(b') - 1$
3. if $\sigma(b') = 0$ and $\sigma_h(b') = 0$ and b' is not empty then
4. if *hot*(b') then
5. *setcold*(b')
6. *freelist* $\leftarrow$ *freelist* $\cup \{b'\}$
7. wake all sleeping threads

nextnblock(b)
1. if b has no open nodes or b was just set to hot then lock
2. else if *trylock()* fails then return b
3. if $b \neq$ NULL then
4. *bestScope* $\leftarrow$ *best*(*interferenceScope*(b))
5. if $b <$ *bestScope* and $b <$ *best*(*freelist*) then
6. unlock
7. return b
8. *release*(b)
9. if ($\forall l \in nblocks : \sigma(l) = 0$) and *freelist* is empty then
10. *done* $\leftarrow$ true
11. wake all sleeping threads
12. while *freelist* is empty and $\neg$*done*, sleep
13. if *done* then $n \leftarrow$ NULL
14. else
15. $n \leftarrow$ *best*(*freelist*)
16. for each $b' \in$ *interferenceScope*(n)
17. $\sigma(b') \leftarrow \sigma(b') + 1$
18. unlock
19. return n