Parallel Best-First Search: Optimal and Suboptimal Solutions

Ethan Burns[1] and **Seth Lemons**[1] and **Wheeler Ruml**[1] and **Rong Zhou**[2]
[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 nodes. By using abstraction to partition the state space, we detect duplicate states while avoiding lock contention. We allow speculative expansions when necessary to keep threads busy. We identify and fix potential livelock conditions. 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 performance than previous parallel search proposals. We also demonstrate that our approach extends easily to other best-first searches, such as weighted A* and anytime heuristic search.

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, Nilsson, and Raphael 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 their 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 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). It extends easily to domains with non-uniform, non-integer move costs and inadmissible heuristics. This algorithm was introduced by Burns et al. (2009), who discuss its performance using A*. In this paper, we review the algorithm and its performance and then show in detail its extension to weighted A* and anytime heuristic search. 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 optimal solutions faster than all other algorithms tested. We also show that PBNF is able to find bounded suboptimal solutions, with its advantages increasing as problem difficulty increases.

Previous Work

The most basic 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. Burns et al. (2009) show that this naive approach to parallelizing A* does not perform very well. 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 for bounded-memory operation; we ignore that feature in this paper.) We use a novel hashing scheme for PRA* based on state-space abstraction, we call this implementation APRA* ('A' because of the abstraction based hashing scheme). This method was found to give better performance than a trivial hashing algorithm (Burns et al. 2009). 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, Burns et al. (2009) show that it can still be expensive.

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'll 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 equip each $n$block with its own open and closed lists. 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 their successors that are in use by another processor. An $n$block can only be acquired when its σ is zero. 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 2006). In each thread, 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 cost-to-go estimate is used to prune nodes below the current solution upper bound.

Variations of 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. This can be handled by using iterative deepening, but it has been shown previously that this does not perform well on domains that do not have a geometrically increasing number of nodes within successive f bounds (Burns et al. 2009). 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 f value ties. 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. To ameliorate this, one can enforce that at least m nodes are expanded before abandoning a non-empty $n$block. (Zhou and Hansen credit (Edelkamp and Schrodl 2000) with this idea.) Our implementation of the BFPSDD algorithm performs a minimum of m expansions per $n$block. 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. Following Burns et al. (2009) we use the value four times the number of threads for k. This allows us to add additional $n$blocks to small layers in order to amortize the cost of synchronization. With 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.

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

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

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, we combine PSDD's duplicate detection scopes with an idea from the Localized A* (LA*) algorithm of Edelkamp and Schrodl (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. It can, however, be used to prune an $n$block's entire open list when the minimum f value is greater than the cost of the incumbent. Figure 1 shows pseudo-code, indicating where locking is necessary.

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 `try_lock` 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 'speculative' expansions that would not have been performed in a serial best-first search.

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. We have no assurance that all $n$blocks in its duplicate detection scope will 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.

The method for setting $n$blocks to hot is designed so 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. This guarantees the property that if an $n$block is flagged as hot it will eventually become free. Complete pseudo-code for safe PBNF is given in the appendix and further details are discussed in earlier work (Burns et al. 2009).

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. Our list of algorithms includes APRA*, BFPSDD, and safe PBNF. Other algorithms were tested in earlier work (Burns et al. 2009). 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 a dual quad-core Intel Xeon X5450 3.0GHz processors limited to roughly 2GB of RAM. 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 the latter is known to scale 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.

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 we determined that 6400 nblocks gives good performance for all relevant algorithms. We use the value 64 for the minimum number of expansions (where relevant) which we also determined to give good performance.

Four-way Unit Cost: In the unit cost model, each move has the same cost. The upper left plot in Figure 2 shows the APRA*, BFPSDD, and safe PBNF algorithms on unit-cost four-way movement path planning problems. Error bars indicate 95% confidence intervals on the mean and algorithms in the legend are ordered on their average performance. Each plot includes a horizontal line, which represents the mean performance of a serial A* search. From this figure we see that safe PBNF is superior to any of the other algorithms, with steadily decreasing solution times as threads are added. 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. While APRA* does not perform as well as safe PBNF and BFPSDD, it has the advantage of being very simple to implement.

Four-way Life Cost: Moves in the life cost model have cost equal to 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, and the shortest path is likely not the cheapest. The bottom left plot in Figure 2 shows these results. Safe PBNF has 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 performs the worst, as it doesn't 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 right panel shows that safe PBNF gives the best performance quite clearly. While APRA* is initially better than BFPSDD, it does not scale and is slower than BFPSDD at 6 or more threads.

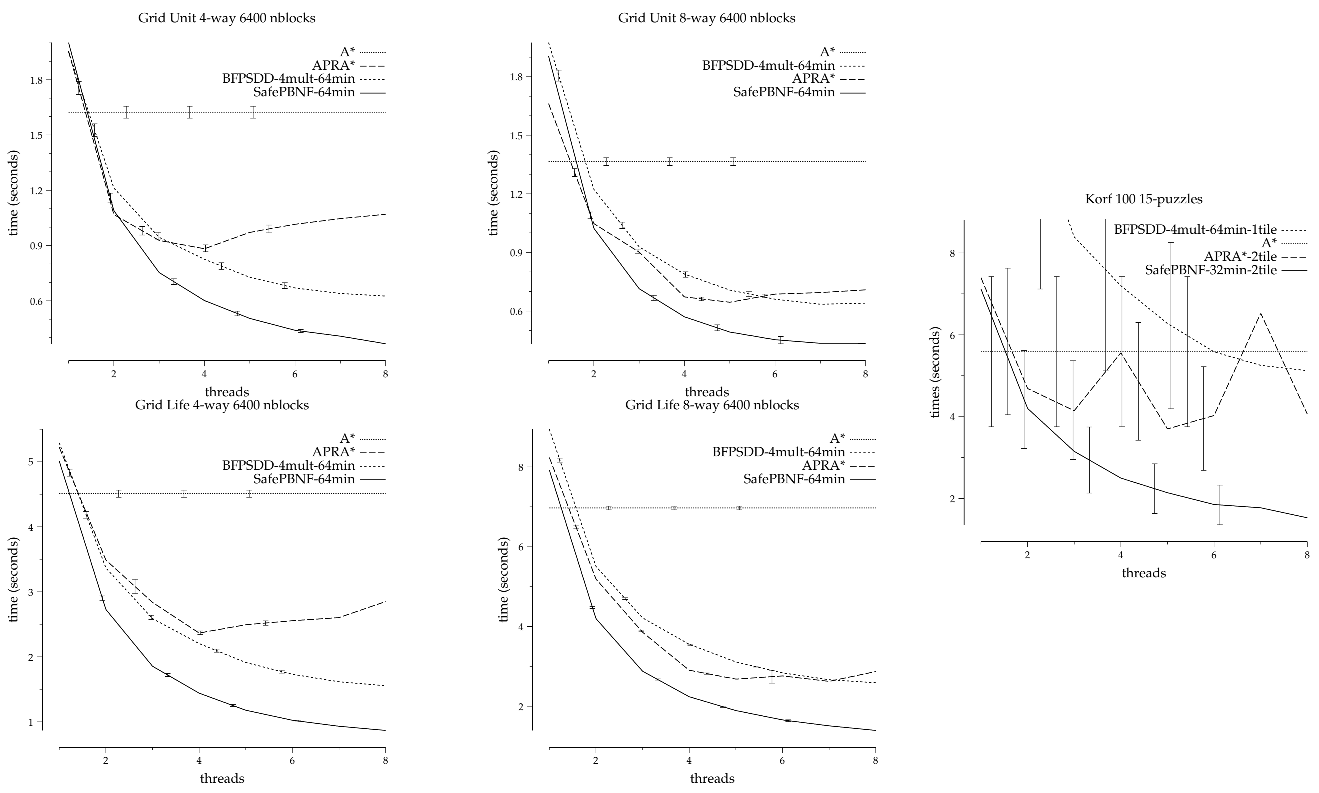

Figure 2: Results on grid path planning and the sliding tiles puzzle

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. Figure 2 shows that safe PBNF gives the best performance for all thread counts beyond two. BFPSDD gives performance similar to PBNF, but consistently slower. The APRA* algorithm does fairly well in the eight-way life cost model showing solution speeds which are on par with that of BFPSDD, but it again scales poorly after 5 threads.

Sliding Tiles

The sliding tiles puzzle is a common domain for benchmarking heuristic search algorithms. For these results, we use fourty-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 (240 nblocks on the 15-puzzle) did not produce a sufficient number of abstract states for PBNF 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 (3360 nblocks). 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 the smaller abstraction, presumably because it did not require switching between $n$blocks as often because of the narrowly defined layer values. The PBNF algorithm used 32 for the minimum expansions parameter and BFPSDD used 64 minimum expansions. These values gave the best performance for their respective algorithms on this domain.

The right-most panel in Figure 2 shows the results for BFPSDD, APRA*, and safe PBNF. Safe PBNF shows 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*.

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 (2000). Figure 3 presents the results for APRA*, PSDD and PBNF, which were identified as the most competitive based on the results presented above, and serial A* for comparison. PSDD 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

	A*	APRA*				Safe PBNF				PSDD				
Problem	1	1	3	5	7	1	3	5	7	1	3	5	7	Abst.
logistics-6	2.30	1.46	0.76	1.22	0.84	1.17	0.64	0.56	**0.62**	1.20	0.78	0.68	0.64	0.42
blocks-14	5.19	7.12	5.50	3.78	3.65	6.21	2.69	2.20	**2.02**	6.36	3.57	2.96	2.87	7.90
gripper-7	118	59.8	51.1	41.0	27.5	39.6	16.9	11.2	**9.21**	65.7	29.4	21.9	19.2	0.83
satellite-6	131	95.5	48.5	65.9	48.8	77.0	24.1	17.3	13.7	61.5	23.6	16.7	**13.3**	0.98
elevator-12	336	213	269	241	169	150	53.5	34.2	**27.0**	162.8	62.7	43.3	36.7	0.67
freecell-3	199	150	112	60.5	39.9	127	47.1	38.1	**37.0**	126.3	53.8	45.5	43.7	16.6
depots-7	MEM	301	144	MEM	MEM	156	63.0	42.9	**34.7**	160	73.0	57.7	54.7	3.59
driverlog-11	MEM	322	103	MEM	MEM	154	60.0	38.8	**31.2**	156	63.2	41.9	34.0	9.68
gripper-8	MEM	528	MEM	MEM	MEM	235	98.2	63.7	**51.5**	388	172	121	106	1.11

Figure 3: Computation time on STRIPS planning problems, in seconds, for various numbers of threads.

	Threads					
w	1	2	4	5	7	8
1.1	1.01	0.95	0.92	0.96	1.02	0.95
1.2	1.23	1.20	1.12	1.13	1.20	1.11
1.4	1.24	1.17	1.10	1.10	1.19	1.12
1.8	0.90	0.83	0.82	0.79	0.88	0.86

Figure 4: AwPBNF speedup over standard PBNF.

fix allows safe PNBF to follow a different search order than PBNF.

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. In the current implementation, the abstraction generation algorithm is implemented serially but it should be trivial to parallelize and, therefore, execute much more quickly.

The PSDD algorithm was given the optimal solution cost as an upper bound to perform pruning in the breadth-first heuristic search. These times can be used as a lower bound on the time BFPSDD (with $m = 0$ and $k = 1$) would take, because it expands the same nodes without the overhead of sorting or re-expanding. The PBNF algorithm finds its own upper bound from suboptimal solutions and therefore will give the performance shown in this figure without first requiring the cost of the optimal solution.

Anytime Search

Hansen and Zhou (2007) show that using wA* as an anytime algorithm (searching until an optimal solution is found; giving a stream of incumbent solutions at increasing qualities) can lead to speedup over A* for some weight values in certain domains. Figure 4 shows results for an anytime adaptation of PBNF which we call AwPBNF (anytime weighted PBNF). The values in this table represent the speedup of AwPBNF over PBNF for some of the easiest Korf 15-Puzzles using 3360 nblocks and 32 minimum expansions per nblock. For weight values of 1.2 and 1.4, the AwPBNF algorithm performs up to 24% better than the standard PBNF algorithm. The performance increase over PBNF seems to decrease as more threads are added, this may be due to extra speculation as more of the frontier nodes are searched in parallel.

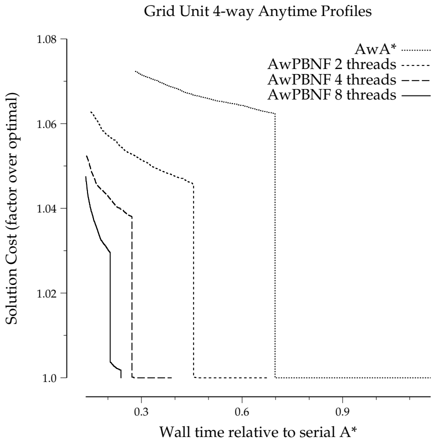

Figure 5: Grid pathfinding: lower hull performance profile.

Figure 5 shows a comparison of the performance of Anytime weighted A* (AwA*) (Hansen and Zhou 2007) and Anytime weighted PBNF at 2, 4 and 8 threads on four-way unit cost grid pathfinding. In these results AwPBNF uses 10000 nblocks and 64 minimum expansions. Each data point in this figure represents the best performance of the respective algorithm over all of the following weights: 1.1, 1.2, 1.4, 1.8. From this figure we can see that AwPBNF gives similar performance to AwA*, however as threads are added the profile shifts to the left and the upper tail gets smaller as better solutions are found more quickly.

Bounded Suboptimal Search

Sometimes it is acceptable or even preferable to search for a solution which is not optimal. Suboptimal solutions can often be found much more quickly than optimal ones and with lower memory consumption. When a suboptimal search is performed it is usually desirable to have a bound on the suboptimality of the solution found. Weighted A* guarantees that suboptimality will be bounded by the weight used. It is possible to modify PBNF, PSDD, and PRA* to use weights to find suboptimal solutions, but a strict f' ordering is not followed. This causes the first solution found to, possibly, be outside the bound. Much like the original versions of these algorithms, we must prove the quality of our solution by either exploring or pruning all nodes.

Let s be the current incumbent solution and w the suboptimality bound. A node n can clearly be pruned if $f(n) \geq g(s)$. It can also be pruned if $w \cdot f(n) \geq g(s)$. We only need to retain n if it is on the optimal path to a solution that is a factor of w better than s.

Theorem 1 *We can prune a node n if $w \cdot f(n) \geq g(s)$ without sacrificing w-admissibility.*

Proof: If the incumbent is w-admissible, we can safely prune any node, so we consider the case where $g(s) > w \cdot g(\mathit{opt})$, where *opt* is an optimal goal. Note that without pruning, there always exists a node p in some open list (or being generated) that is on the best path to *opt*. By the admissibility of h and the definition of p, $w \cdot f(p) \leq w \cdot f^*(p) = w \cdot g(\mathit{opt})$. If the pruning rule discards p, that would imply $g(s) \leq w \cdot f(p)$ and thus $g(s) \leq w \cdot g(\mathit{opt})$, which contradicts our premise. Therefore, an open node leading to the optimal solution will not be pruned if the incumbent is not w-admissible. A search that does not terminate until open is empty will not terminate until the incumbent is w-admissible or it is replaced by an optimal solution. $\square$

We make explicit a useful corollary:

Corollary 1 *We can prune a node n if $f'(n) \geq g(s)$ without sacrificing w-admissibility.*

Proof: Clearly $w \cdot f(n) \geq f'(n)$, so Theorem 1 applies. $\square$
With this corollary, we can use a pruning shortcut: when the open list is sorted on increasing f' and the node at the front has $f' \geq g(s)$, we can prune the entire open list.

As before, when all open lists are empty, we can terminate with the guarantee that our current solution is within the suboptimality bound. The time spent proving that the incumbent solution is within the bound, however, poses a significant disadvantage against wA*. Our method may require many re-expansions of nodes early on in a path because speculation led us to them through a non-w-admissible route. This effect gets more important as weight increases and makes it difficult to perform competitively on easy problems at high weights.

Evaluation

We implemented and tested weighted versions of A*, APRA* (wPRA*), BFPSDD (wBFPSDD), and PBNF (wPBNF). All algorithms prune nodes based on f' and $w * f$ criteria. All parallel algorithms prune whole open lists on f'. Duplicates which have been expanded are dropped in serial wA*, regardless of value, in grids, as discussed by Likhachev, Gordon, and Thrun (2003). We do not use duplicate dropping with wA* in the sliding tiles domain because these problems do not have as many duplicates and have fewer paths to the goal. We found that duplicate dropping makes wA* perform worse in the sliding tiles domain.

Speedup versus wA* is plotted in Figure 6 showing number of threads and weight used. A 10000 nblock abstraction was used in path planning, 3360 nblocks were used in tiles. wPBNF used 64 min expansions on path planning, 32 on tiles. wBFPSDD used a multiplier of 4 and 64 min expansions on path planning, 32 on tiles. From Figure 6 we see that all of the algorithms lose their advantage over wA* as weights increase, presumably because the overhead of threads and contention is great compared to the low number of nodes expanded. It is interesting that wPBNF consistently

wPBNF: Unit Four-way Path Planning

	Threads					
w	1	2	4	5	7	8
1.1	0.83	1.50	2.68	3.20	3.94	4.25
1.2	0.75	1.37	2.36	2.77	3.33	3.48
1.4	0.51	1.00	1.60	1.81	1.94	2.00
1.8	0.53	0.61	0.66	0.66	0.68	0.67
3.0	0.44	0.43	0.42	0.42	0.41	0.41

wBFPSDD: Unit Four-way Path Planning

	Threads					
w	1	2	4	5	7	8
1.1	0.84	1.25	1.87	2.10	2.39	2.40
1.2	0.76	1.11	1.61	1.80	1.98	1.93
1.4	0.51	0.75	1.10	1.15	1.19	1.16
1.8	0.45	0.48	0.50	0.48	0.43	0.39
3.0	0.42	0.41	0.39	0.37	0.33	0.30

wAPRA*: Unit Four-way Path Planning

	Threads					
w	1	2	4	5	7	8
1.1	0.94	1.51	1.96	1.88	1.72	1.68
1.2	0.87	1.38	1.78	1.69	1.57	1.50
1.4	0.55	0.90	1.20	1.11	1.08	0.96
1.8	0.51	0.66	0.57	0.48	0.50	0.40
3.0	0.57	0.59	0.39	0.31	0.36	0.27

wPBNF: Korf 100 15-Puzzle instances

	Threads					
w	1	2	4	5	7	8
1.4	0.79	1.50	2.02	2.42	3.02	3.27
1.8	0.71	0.90	1.62	1.82	1.85	2.10
2.0	0.51	0.88	1.68	1.86	1.64	2.09
3.0	1.01	0.77	1.17	1.00	1.05	0.97
5.0	0.56	0.60	0.76	0.72	0.67	0.64

wBFPSDD: Korf 100 15-Puzzle instances

	Threads					
w	1	2	4	5	7	8
1.4	0.87	1.12	1.52	1.61	1.92	1.98
1.8	0.53	0.83	1.16	1.07	1.40	1.27
2.0	0.60	0.71	0.94	0.92	1.00	0.95
3.0	0.47	0.58	0.64	0.55	0.55	0.49
5.0	0.38	0.46	0.42	0.40	0.36	0.32

wAPRA*: Korf 100 15-Puzzle instances

	Threads					
w	1	2	4	5	7	8
1.4	0.65	1.07	0.98	1.40	1.51	1.22
1.8	0.64	1.12	1.00	1.30	1.18	0.89
2.0	0.64	1.27	1.02	1.19	0.90	0.89
3.0	0.58	0.89	0.65	0.62	0.57	0.40
5.0	0.54	0.76	0.48	0.45	0.34	0.31

Figure 6: Speed-up over serial weighted A*.

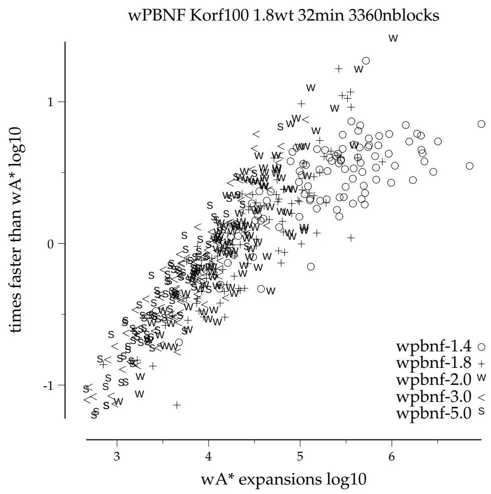

Figure 7: wPBNF speedup over wA* v.s. wA* expansions.

wPBNF: Easy Korf 15-Puzzle instances

	Threads						
w	1	2	4	8	16	32	64
1.0	1.21	2.26	4.44	8.72	13.9	17.1	7.68
1.4	0.72	1.44	2.75	5.09	8.07	9.52	2.55
1.8	0.69	1.06	1.79	3.03	3.52	2.98	1.05
2.0	0.58	1.28	1.63	2.61	3.10	2.62	0.85
3.0	1.61	1.81	2.12	2.66	2.72	1.90	0.71

Figure 8: Speedup over wA* of wPBNF on a Sun T5440.

performs better than wPRA* or wBFPSDD. Also, wPBNF and wBFPSDD consistently improve as threads are added for all but the largest weight values. wPRA* increases at first, but its performance drops off after a few threads, much like the results for optimal searches.

Figure 7 shows a comparison of wPBNF to weighted A*. The points in this scatter plot represent time taken by wA* divided by the time taken by wPBNF (at 8 threads with 3360 nblocks and 32 minimum expansions) on the y axis and number of nodes expanded by wA* on the x axis. Each different set of glyphs represents a different weight value used for both wPBNF and wA*. The data in this figure shows that, while wPBNF does not outperform wA* on easier problems, the benefits of wPBNF over wA* increase as problem difficulty increases. The speed gain for the 1.4 weight value appears to increase with problem difficulty and levels off just under 10 times faster than wA*. This behavior is ideal since the machine used for this experiment had eight cores. Speeds at greater weights do not reach 10 times faster than wA*, however there are a few instances that seem to have super linear speeds. These can be explained by the speculative expansions that wPBNF performs. The poor behavior of wPBNF for easy problems is most likely due to the overhead of the abstraction and contention. wPBNF outperforms wA* more often at low weights (where the problems require more expansions) and less often at higher weights (where the problems will require fewer expansions).

We are beginning to experiment with PBNF on the SPARC architecture, using a Sun SPARC Enterprise T5440 with 4 1.2GHz T2+ processors (8 cores per processor, 8 threads per core), 64 GB of RAM. Our preliminary results on some of the easiest Korf 15-puzzles using a Sun T5440 are given in Figure 8. This figure shows the speedup over wA* of wPBNF (with 3360 nblocks and 32 minimum expansions per nblock) at various weights and threads. From this data we can see that wPBNF continues to perform better up to 32 threads at low weights. At 64 threads the performance drops off. We expect that, as we learn more about this platform, we will see better scalability and more of an advantage over wA* at more threads.

Discussion

We presented empirical results for PRA*, BFPSDD, and safe PBNF, testing their abilities to return optimal and bounded suboptimal solutions. It is clearly shown that PBNF outperforms the other parallel algorithms tested in all cases. We also illustrated that PBNF can perform better than weighted A* when returning bounded suboptimal solutions, and that its advantages grow as threads are added and as problem difficulty increases.

The PBNF algorithm outperforms BFPSDD because of the lack of layer based synchronization and a better utilization of heuristic cost-to-go information. Another 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 which have nodes in them. With more $n$blocks containing search nodes there will be more disjoint duplicate detection scopes with open nodes and, therefore, the possibility of increased parallelism.

Conclusions

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. 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. We have also demonstrated that PBNF, PRA* and BFPSDD can be modified to use weighted heuristic values for suboptimal solutions that can outperform their optimal counterparts and can find solutions faster than weighted A* in some situations. Evidence that we present suggests that wPBNF can be preferable to wA* for more difficult problems.

Acknowledgements

We gratefully acknowledge support from NSF grant IIS-0812141, the DARPA CSSG program and helpful suggestions from Jordan Thayer. Some results and algorithms discussed here are were presented by Burns et al. (2009).

References

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-first heuristic search for multi-core machines. In *Proceedings of* *the 14th International Joint Conference on Artificial Intelligence (IJCAI-09).*

Edelkamp, S., and Schrodl, S. 2000. Localizing A*. In *AAAI 2000*, 885–890.

Evans, J. 2006. A scalable concurrent malloc(3) implementation for freebsd. In *Proceedings of BSDCan 2006*.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1995. PRA* - massively-parallel heuristic-search. *Journal of Parallel and Distributed Computing* 25(2):133–143.

Hansen, E. A., and Zhou, R. 2007. Anytime heuristic search. *JAIR* 28:267–297.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics* SSC-4(2):100–107.

Haslum, P. 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Formal analysis. Technical Report CMU-CS-03-148, Carnegie Mellon University School of Computer Science.

Zhou, R., and Hansen, E. A. 2004. Structured duplicate detection in external-memory graph search. In *AAAI 2004*.

Zhou, R., and Hansen, E. 2006. Breadth-first heuristic search. *AI* 170(4–5):385–408.

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI 2007*.

Pseudocode

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 $exp < min\text{-}expansions$ 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; return $b$
7.      $release(b)$
8.  if $(\forall l \in nblocks : \sigma(l) = 0)$ and $freelist$ is empty then
9.      $done \leftarrow$ true
10.     wake all sleeping threads
11. while $freelist$ is empty and $\neg done$, sleep
12. if $done$ then $n \leftarrow$ NULL
13. else
14.     $n \leftarrow best(freelist)$
15.     for each $b' \in interferenceScope(n)$
16.         $\sigma(b') \leftarrow \sigma(b') + 1$
17. unlock
18. return $n$
```