Parallel Best-First Search: The Role of Abstraction

Ethan Burns and **Sofia Lemons** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
eaburns, sofia.lemons, ruml at cs.unh.edu

Rong Zhou
Embedded Reasoning Area
Palo Alto Research Center
Palo Alto, CA 94304 USA
rzhou at parc.com

Abstract

To harness modern multicore 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 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 in detail 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. 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 study the empirical behavior of PBNF on three popular search domains: STRIPS planning, grid pathfinding, and the venerable sliding tile puzzle. We show how abstraction can be used to improve an alternate parallel search algorithm called HDA* (Kishimoto, Fukunaga, and Botea 2009). When comparing PBNF to our extended version of HDA*, our results show that PBNF often yields optimal solutions faster. In addition, we show that parallel search can obtain bounded suboptimal solutions more quickly than serial weighted A*, with the advantage of parallelism 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 parallizing 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. Kishimoto, Fukunaga, and Botea (2009) show that the PRA* algorithm can be improved by using asynchronous communication, they call the algorithm with this modification hash distributed A*, or HDA*. We have created a novel hashing scheme for HDA* based on state-space abstraction, we call this implementation AHDA* ('A' because of the abstraction based hashing scheme). We show experimental results that demonstrate that using abstraction with the PRA* algorithm can be more beneficial than using asynchronous communication.

Parallel Structured Duplicate Detection

Here we describe the parallel structured duplicate detection algorithm that was the basis for PBNF. 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 $\sigma = 0$. 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 2006). 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 cost-to-go estimate is used to prune nodes below the current solution upper bound.

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 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).

```
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.

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. We 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.

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. Other algorithms were tested in earlier work (Burns et al. 2009), however they were shown to be less competitive. All 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.

Abstraction with PRA*

We begin by looking at the benefit of using abstraction with the parallel retracting A* algorithm. The upper left panel of figure 2 shows the results of an experiment running PRA* on a set of grid pathfinding instances both with and without abstraction and with and without asynchronous communication. The heuristic used was the Manhattan distance to the goal location. Each line in the figure gives the mean wall clock time in seconds versus the number of threads on a set of twenty instances. The error bars show the 95% confidence interval on the means and the legend is sorted in order of average performance. We can see from this figure that the two variants of PRA* without abstraction gave significantly worse performance than the two variants that did use abstraction. We can also see that the benefit of adding abstraction was greater than the benefit of using asynchronous communication. This is evident because PRA* with synchronous communication and abstraction (labeled "sync. and abst. (APRA*)") had better wall clock performance than the variant with asynchronous communication and without abstraction (labeled "async. (HDA*)"). Note that this latter variant is exactly the HDA* algorithm.

On average the variant with both asynchronous communication and abstraction (labeled "async. and abst. (AHDA*)") which we call AHDA* performed the best. In the grid pathfinding domain the abstraction function that was used separates the grid into a coarser grid of abstract states. When using abstraction to distribute newly generated nodes among the various threads many of the successors will belong to the same abstract state as their parent. When this happens, no communication is required and the nodes can be checked for duplicates and added directly to the expanding thread's open list. Generally hash functions are designed to avoid collisions in a hash table. If a hash function is used instead of an abstraction function to distribute the nodes it will be rare that successors are assigned to the currently expanding thread. Additionally, it will be rare that the generated successors map to the same thread as their siblings. This means that communication will often be required between different threads for each node that is generated.

Grid Pathfinding

We tested on grids 5000 cells wide by 5000 cells high, with the start in the upper left and the goal in the lower right. We test two cost models (discussed below) and both four-way and eight-way movement. Cells are blocked with probability 0.35 in four-way movement boards and with a probability of 0.45 in eight-way movement boards. 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. All algorithms used the Manhattan distance to the goal as their heuristic.

The plots discussed below show the speedup of each algorithm as the number of threads are increased (x-axis) over a serial A* search (y-axis). Error bars indicate 95% confidence intervals on the mean and algorithms in the legend are ordered on their average performance. The diagonal line labeled "Perfect speedup" shows a perfect linear speedup where the performance increase over serial A* is the same as the number of threads searching. A more practical reference point for speedup is shown by the "Achievable speedup" line. On a perfect machine with n processors, running n independent A* searches will take the same amount of time as a single A* search. On a real machine, however, there are hardware considerations that prevent this perfect speedup. The line labeled "Achievable speedup" shows the speedup over serial A* which is achieved by running multiple independent A* searches in parallel. This can be thought of as a soft upper bound on the achievable speedup for the given machine on the given domain.

Four-way Unit Cost: In the unit cost model, each move has the same cost. The top center plot in Figure 2 shows the AHDA*, and safe PBNF algorithms on unit-cost four-way movement path planning problems. On average the safe PBNF algorithm gave better speedup than AHDA* in this domain. At eight threads safe PBNF showed a speedup that was approximately six times that of serial A* search.

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 5000, and the shortest path is likely not the cheapest. The bottom left plot in Figure 2 shows these results. Again, on average safe PBNF gave more speedup than AHDA*, however, at six and seven threads AHDA* outperformed safe PBNF.

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 had the best mean speedup even though it was outperformed by AHDA* at five, six and seven threads. In this domain AHDA* have a very erratic performance, sometimes greatly decreasing its speedup over serial search as more threads were added.

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 center panel shows the results for this domain. We can see that the AHDA* algorithm gave the best performance on average, although it was outperformed by safe PBNF at eight threads.

Sliding Tiles

The sliding tiles puzzle is a common domain for benchmarking heuristic search algorithms. In this section we present results of an experiment on 250 15-puzzles that were solvable by A* in 3 million expansions. The abstraction used by the safe PBNF algorithm, in this domain, ignored the position

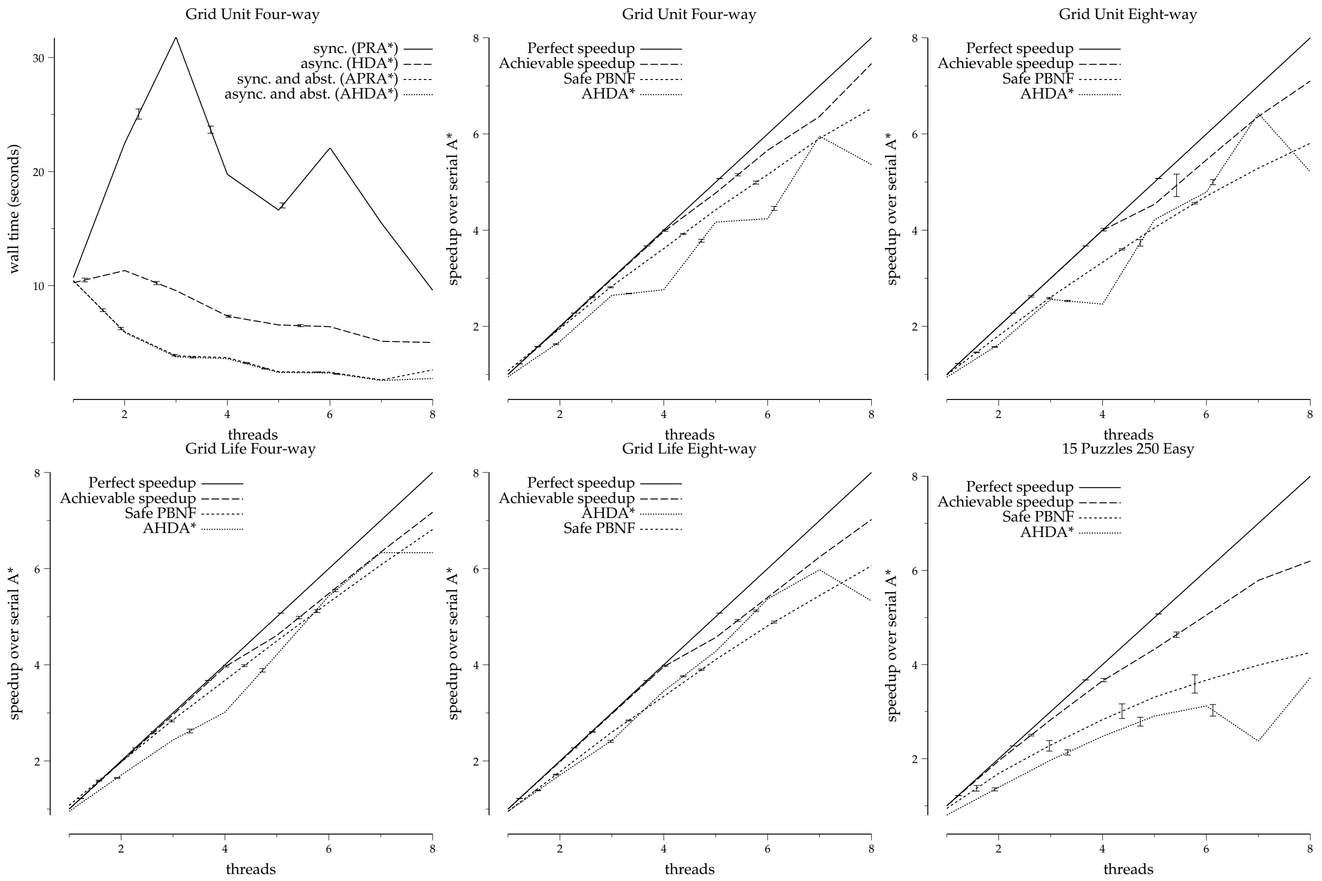

Figure 2: Results on grid path planning and the sliding tiles puzzle.

of all tiles except the blank, 1 and 2 tiles. For AHDA*, we found that the performance was better with an abstraction that ignored all except the 1, 2 and 3 tile. Since the blank tile was ignored in the abstraction it was more ofter the case that AHDA* was able to map child nodes to the same thread that expanded the parent. When this occurs, no communication is required and therefore the search performs better. The heuristic used by all algorithms was the Manhattan distance heuristic. The bottom right panel in Figure 2 shows the results for safe PBNF and AHDA*. Safe PBNF gave the best performance consistently, peaking at 4x speedup over serial A*.

STRIPS Planning

algorithms were embedded into a domain-independent optimal sequential STRIPS planner using regression and the max-pair admissible heuristic of Haslum and Geffner (2000). Table 1 presents the results for AHDA*, PSDD and PBNF. Entries in the table are **bold** if they are within 10% of the best entry for the given domain. The PSDD algorithm was given the optimal solution cost as an upper bound to perform pruning in the breadth-first heuristic search. Safe PBNF and AHDA* find their 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.

The right-most column shows the time that was taken by the 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.

Overall, we see that safe PBNF gave the best performance at seven threads across all except two domains (logistics-6 and freecell-3). In these two domains, AHDA* found solutions in the least amount of time at seven threads.

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, and AHDA* to find suboptimal

Problem	A* 1	AHDA* 1	AHDA* 3	AHDA* 5	AHDA* 7	Safe PBNF 1	Safe PBNF 3	Safe PBNF 5	Safe PBNF 7	PSDD 1	PSDD 3	PSDD 5	PSDD 7	Abst.
logistics-6	2.30	1.44	0.70	0.48	**0.40**	1.17	0.64	0.56	0.62	1.20	0.78	0.68	0.64	0.42
blocks-14	5.19	7.13	5.07	2.25	**2.13**	6.21	2.69	**2.20**	**2.02**	6.36	3.57	2.96	2.87	7.90
gripper-7	118	59.5	34.0	16.0	12.7	39.6	16.9	11.2	**9.21**	65.7	29.4	21.9	19.2	0.83
satellite-6	131	95.5	33.6	24.1	18.2	77.0	24.1	17.3	**13.7**	61.5	23.6	16.7	**13.3**	0.98
elevator-12	336	206	96.8	67.7	57.1	150	53.5	34.2	**27.0**	162.8	62.7	43.3	36.7	0.67
freecell-3	199	148	93.6	38.2	**27.4**	127	47.1	38.1	37.0	126.3	53.8	45.5	43.7	16.6
depots-7	M	300	126	51.0	39.1	156	63.0	42.9	**34.7**	160	73.0	57.7	54.7	3.59
driverlog-11	M	316	85.2	51.3	49.0	154	60.0	38.8	**31.2**	156	63.2	41.9	**34.0**	9.68
gripper-8	M	533	239	97.6	76.3	235	98.2	63.7	**51.5**	388	172	121	106	1.11

Table 1: Computation time on STRIPS planning problems, in seconds, for various numbers of threads.

solutions. Since parallelism is used, a strict f' ordering is not followed by these algorithms and therefore the first solution found may 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 $f(n) \geq g(s)$. But according to the following theorem, 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. This is a much stronger rule.

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 $g(s) > w \cdot g(opt)$, 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*. Let f^* be the cost of an optimal solution. By the admissibility of h and the definition of p, $w \cdot f(p) \leq w \cdot f^*(p) = w \cdot g(opt)$. If the pruning rule discards p, that would imply $g(s) \leq w \cdot f(p)$ and thus $g(s) \leq w \cdot g(opt)$, which contradicts our premise. Therefore, an open node leading to an 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. □

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. □
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*, AHDA* (wAHDA*), and safe PBNF (wPBNF.) All algorithms prune nodes based on f' and $w * f$ criteria. Both 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 2 showing number of threads and weight used. Table entries that are in **bold** are entries that are not significantly different ($p < 0.05$ using a Wilcoxon signed-rank test) from the best value for the same weight in the given domain.

The left half of Table 2 shows the results of an experiment run on the same four-way unit-cost grid pathfinding problems that were presented for optimal search. In grid pathfinding, the wPBNF algorithm gave the best speedup for weights of 1.1 and 1.2 where it achieved up to 5x the performance of serial weighted A*. At a weight of 1.4 both wPBNF and wAHDA* gave similar speedups at eight threads. wAHDA* outperformed wPBNF at a weight of 1.8 where wPBNF was unable to find solutions faster than serial weighted A* at any number of threads.

The right half of Table 2 shows the results of an experiment run on Korf's 100 15-puzzle instances. From this table, we see that both wPBNF and wAHDA* performed comparably for weights of 1.4 and 1.7. At a weight of 2, wPBNF gave the best performance with six threads and wAHDA* gave the least performance decrease over serial search with two threads.

From Table 2 we see that both wAHDA* and wPBNF decrease their advantage over wA* as weights increase, presumably because the overhead of threads and contention is too great compared to the very low number of nodes expanded. To confirm our understanding of the effect of problem size on speedup, Figure 3 shows a comparison of wPBNF to weighted A* on all of the 100 Korf 15-puzzle instances. Each point represents a run on one instance at a particular weight, the y-axis represents wPBNF speedup relative to serial wA*, and the x-axis represents the number of

	Unit Four-way Grids								Korf's 100 10-Puzzles							
	wPBNF				wAHDA*				wPBNF				wAHDA*			
	1.1	1.2	1.4	1.8	1.1	1.2	1.4	1.8	1.4	1.7	2.0	3.0	1.4	1.7	2.0	3.0
1	0.98	0.91	0.51	0.73	0.87	0.79	0.32	0.56	0.68	0.44	0.38	0.69	0.61	0.60	0.59	0.54
2	1.74	1.65	1.07	0.87	1.35	1.17	0.63	0.84	1.35	0.81	1.00	0.63	1.18	1.11	1.32	**0.78**
3	2.47	2.33	1.62	0.89	1.90	1.69	1.30	**1.30**	1.48	0.97	0.85	0.56	1.53	**1.30**	1.40	0.73
4	3.12	2.92	2.13	0.90	2.04	2.10	1.57	**1.30**	1.70	1.20	0.93	0.60	1.91	**1.57**	1.55	0.74
5	3.76	3.52	2.48	0.91	1.77	2.08	1.79	0.97	2.04	1.38	0.97	0.74	**2.33**	1.70	1.27	0.66
6	4.30	3.99	2.80	0.89	3.23	3.03	2.18	**1.33**	2.16	1.30	**1.19**	0.67	**2.28**	**1.72**	1.24	0.52
7	4.78	4.40	**3.01**	0.88	3.91	3.78	2.56	**1.30**	**2.55**	**1.46**	1.04	0.62	2.71	1.50	1.03	0.44
8	**5.09**	**4.66**	**3.11**	0.87	3.79	3.64	**3.02**	1.13	**2.71**	**1.71**	1.10	0.60	**2.70**	**1.51**	1.24	0.44

Table 2: Speed-up over serial weighted A*.

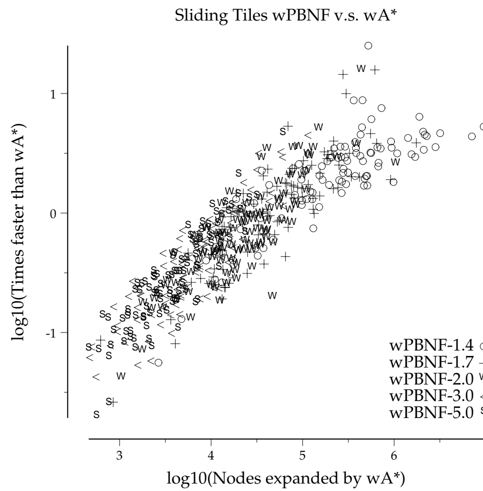

Figure 3: wPBNF speedup versus problem difficulty.

nodes expanded by wA*. Different glyphs represents different weight values used for both wPBNF and wA*. The figure shows that, while wPBNF did not outperform wA* on easier problems, the benefits of wPBNF over wA* increased as problem difficulty increased. The speed gain for the instances that were run at a weight of 1.4 (the lowest weight tested) leveled off just under 10 times faster than wA*. This is because the machine has eight cores. There are a few instances that seem to have speedup greater than 10x. These can be explained by the speculative expansions that wPBNF performs which may find a bounded solution faster than weighted A* due to the pruning of more nodes with f' values equal to that of the resulting solution. The poor behavior of wPBNF for easy problems is most likely due to the overhead described above. This effect of problem difficulty means that wPBNF outperformed wA* more often at low weights, where the problems required more expansions, and less often at higher weights, where the problems were completed more quickly.

Conclusions

We presented empirical results for AHDA*, and safe PBNF, testing their abilities to return optimal and bounded suboptimal solutions. It is clearly shown that PBNF outperformed AHDA* in the optimal search setting. In the bounded suboptimal case, the weighted versions of PBNF and AHDA* were more competitive. We also illustrated that the advantage of parallel search seems to grow as problem difficulty increases.

Acknowledgements

We gratefully acknowledge support from NSF grant IIS-0812141 and the DARPA CSSG program.

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*.

Edelkamp, S., and Schrödl, S. 2000. Localizing a*. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, 885–890. AAAI Press.

Evans, J. 2006. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proc. 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.

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., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the Fifth Internationas Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS-00)*, 140–149.

Kishimoto, A.; Fukunaga, A.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS-09)*.

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 *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*.

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

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*.