Suboptimal and Anytime Heuristic Search on Multi-Core Machines

Ethan Burns and **Seth Lemons** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
eaburns, seth.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

In order to scale with modern processors, planning algorithms must become multi-threaded. In this paper, we present parallel shared-memory algorithms for two problems that underlie many planning systems: suboptimal and anytime heuristic search. We extend a recently-proposed approach for parallel optimal search to the suboptimal case, providing two new pruning rules for bounded suboptimal search. We also show how this new approach can be used for parallel anytime search. Using temporal logic, we prove the correctness of our framework, and in an empirical comparison on STRIPS planning, grid pathfinding, and sliding tile puzzle problems using an 8-core machine, we show that it yields faster search performance than previous proposals.

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. Many modern AI planning systems are based upon suboptimal or anytime heuristic search. In this paper, we develop parallel algorithms for these important problems.

In a best-first heuristic 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 $f(n) = g(n) + h(n)$. Closed contains all previously expanded nodes, allowing the search to detect duplicated states in the search space and avoid expanding them multiple times. The main 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 Best-NBlock-First (PBNF), originally demonstrated by Burns et al. (2009) for parallelizing A*. As we discuss in detail below, PBNF requires a many-to-one abstraction function that maps search states to abstract nodes called $n$blocks. The sparse connectivity of this abstract graph is used to divide labor among threads without requiring frequent locking.

We make three central contributions. First, we provide a proof of correctness for the PBNF framework, demonstrating its liveness and completeness in the general case. Second, we show how to adapt PBNF for bounded suboptimal search, quickly finding w-admissible solutions (within a factor of w of optimal). We provide two new pruning criteria for parallel suboptimal search and prove that they retain w-admissibility. Third, we demonstrate how suboptimal PBNF leads naturally to an effective anytime search algorithm. Our new algorithms are empirically compared to serial and alternative parallel algorithms on a dual quad-core Intel machine using three benchmark domains: STRIPS planning, grid pathfinding, and the sliding tile puzzle. We find that weighted and anytime PBNF surpass alternative proposals, achieving substantial speed-up over serial search with an advantage that increases with problem difficulty.

Background

There have been several proposed approaches to parallel heuristic search. 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.) Burns et al. (2009) increased the performance of this method by using an abstraction function before hashing in order to reduce the connectivity of the communication graph—they call this abstraction-based algorithm APRA*. Note that each thread needs a synchronized message queue that other threads can add nodes to. While this is less of a bottleneck than having a single shared open list, we will see that it can still be expensive.

Best-First PSDD

Zhou and Hansen (2007) parallelized breadth-first heuristic search (Zhou and Hansen 2006b) using a scheme they call parallel structured duplicate detection (PSDD). This algorithm uses an abstraction function to create 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. We use the terms 'abstract node' and '$n$block' interchangeably. Each $n$block has 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

```
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.           incumbent ← n
10.      else for each child c of n
11.        insert c in the open list of the appropriate nblock
```

Figure 1: A sketch of the PBNF search framework.

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 generating the children of nodes in b. PSDD takes advantage of the fact that $n$blocks whose duplicate detection scopes are disjoint can be expanded in parallel without any locking.

An $n$block b is considered to be free for expansion 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 thread. 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.

PSDD is based on breadth-first search and cannot exploit heuristic guidance unless a tight upper bound is available. Burns et al. (2009) present an adaptation of PSDD to best-first search called best-first PSDD (BFPSDD) which outperformed standard PSDD in all domains which were tested. The search is divided into f value layers, and in each thread of the search, only the nodes in the current layer in an $n$block are searched. If no more $n$blocks have nodes in this layer, all threads synchronize and then progress to the next layer. To ensure that there are a sufficient number of nodes in each layer to outweigh the synchronization overhead, at least m nodes are expanded before abandoning a non-empty $n$block. Also, when populating the list of free $n$blocks for each layer, at least k $n$blocks are added, where k is four times the number of threads. 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.

Parallel Best-NBlock-First (PBNF)

Ideally, all threads would be busy expanding $n$blocks that contain nodes with the lowest f values. PBNF approximates this by maintaining a heap of free $n$blocks ordered on their best f value. 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 (or until m nodes have been expanded). There is no layer synchronization. Figure 1 shows pseudo-code, indicating the single lock required. While incumbent information is shared between threads, atomic updates can be used to avoid the addition of a second lock.

Our implementation 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. As with the minimum number of expansions, this optimization can introduce 'speculative' expansions that would not have been performed in a serial best-first search.

The greedy free-for-all order in which basic 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, Burns et al. (2009) developed an enhanced version called 'Safe PBNF' that uses a method called 'hot $n$blocks', where threads altruistically release their $n$block if they are interfering with a better $n$block. The *interference scope* of an $n$block b is 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.

Correctness of PBNF

Given the complexity of parallel shared-memory algorithms, it can be reassuring to have proofs of correctness. Because we will be using the PBNF framework, we will verify that PBNF exhibits various desirable properties.

Soundness Soundness holds trivially because no solution is returned that does not pass the goal test.

Deadlock There is only one lock in PBNF and the thread that currently holds it never attempts to acquire it a second time, so deadlock cannot arise.

Livelock Burns et al. (2009) used a model checker to find an example of livelock in plain PBNF. They were unable to find an example of livelock in Safe PBNF when using up to three threads and 12 $n$blocks in an undirected ring-shaped abstract graph and up to three threads and eight $n$blocks in a directed graph. This leaves open the question of whether livelock can occur in other situations. We have modeled Safe PBNF in the temporal logic TLA+ and proved by hand that a hot $n$block will eventually become free regardless of the number of threads or the abstract graph. Because the full proof in TLA+ notation is eight pages long, we present an English summary. First, we need a helpful lemma:

Lemma 1 *If an $n$block n is hot, there is at least one other $n$block in its interference scope that is in use by a thread. Also, n is not interfering with any other hot $n$blocks.*

Proof: Initially no $n$blocks are hot. This can change only while a thread searches or when it releases an $n$block. During a search, a thread can only set n to hot if it has acquired an $n$block m that is in the interference scope of n. Additionally, a thread may only set n to hot if it does not create any interference with another hot $n$block. During a release, if n is hot, either the final acquired $n$block in its interference scope is released and n is no longer hot, or n still has at least one acquired $n$block in its interference scope. □

Now we are ready for the key theorem:

Theorem 1 *If an $n$block n becomes hot, it will eventually be added to the free list and will no longer be hot.*

Proof: We will show that the number of acquired $n$blocks in the interference scope of a hot $n$block n is strictly decreasing. Therefore, n will eventually become free.

Assume an $n$block n is hot. By Lemma 1, there is a thread p that has an $n$block in the interference scope of n, and n is not interfering with or interfered by any other hot $n$blocks. Assume that a thread q does not have an $n$block in the interference scope of n. There are four cases:

1. p searches its $n$block. p does not acquire a new $n$block and therefore the number of $n$blocks preventing n from becoming free does not increase. If p sets an $n$block m to hot, m is not in the interference scope of n by Lemma 1. p will release its $n$block after it sees that n is hot (see case 2).
2. p releases its $n$block and acquires a new $n$block m from the free list. The number of acquired $n$blocks in the interference scope of n decreases by one as p releases its $n$block. Since m, the new $n$block acquired by p, was on the free list, it is not in the interference scope of n.
3. q searches its $n$block. q does not acquire a new $n$block and therefore the number of $n$blocks preventing n from becoming free does not increase. If q sets an $n$block m to hot, m is not in the interference scope of n by Lemma 1.
4. q releases its $n$block (if it had one) and acquires a new $n$block m from the free list. Since m, the new $n$block acquired by q, was on the free list, it is not in the interference scope of n and the number of $n$blocks preventing n from becoming free does not increase. □

We can now prove the progress property that we really care about:

Theorem 2 *A node n with minimum f value will eventually be expanded.*

Proof: We consider n's $n$block. There are three cases:

1. The $n$block is being expanded. Because n has minimum f, it will be at the front of *open* and will be expanded.
2. The $n$block is free. Because it holds the node with minimum f value, it will be at the front of the free list and selected next for expansion, reducing to case 1.
3. The $n$block is not on the free list because it is in the interference scope of another $n$block that is currently being expanded. When the thread expanding that $n$block checks its interference scope, it will mark the better $n$block as hot. By Theorem 1, we will eventually reach case 2. □

Completeness This follows easily from liveness:

Corollary 1 *If the heuristic is admissible or the search space is finite, a goal will be returned if one is reachable.*

Proof: If the heuristic is admissible, we inherit the completeness of serial A* by Theorem 2. Otherwise, note that nodes are only re-expanded if their g value has improved, and this can happen only a finite number of times in a finite search space, so a finite number of expansions will suffice to exhaust the search space. □

Bounded Suboptimal Search

Sometimes it is acceptable or even preferable to search for a solution that is not optimal. Suboptimal solutions can often be found much more quickly and with lower memory requirements. Weighted A*, a variant of A* that searches on $f'(n) = g(n) + w \cdot h(n)$, is probably the most popular suboptimal search. It guarantees that, for a weight w, the solution returned will be w-admissible.

Pruning Poor Nodes

It is possible to modify APRA*, BFPSDD, and PBNF to use weights to find suboptimal solutions, but, as in the optimal case, parallelism implies that a strict f' search order will not be followed and we must prove the quality of our solution by either exploring or pruning all nodes. Thus finding effective pruning rules is critical for performance.

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 3 *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*. 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 the optimal solution will not be pruned if the incumbent is not w-admissible. Therefore a search that does not terminate until open is empty will not terminate until the incumbent is w-admissible. □

We make explicit a useful corollary:

Corollary 2 *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 3 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.

Pruning Duplicate Nodes

When searching with an inconsistent heuristic, as in weighted A*, it is possible for the search to find a better path to an already-expanded state. Likhachev, Gordon, and Thrun (2003) proved that, provided the heuristic is consistent, weighted A* will still return a w-admissible solution if

these duplicate states are pruned during search. This ensures that each state is expanded at most once during the search. Unfortunately, their proof depends on expanding in exactly best-first order, which is violated by several of the parallel search algorithms we consider here. However, we can still prove that some duplicates can be dropped. Consider the expansion of a node n that re-generates a duplicate state d that has already been expanded. We propose the following weak duplicate dropping criterion: the new copy of d can be pruned if the old $g(d) \leq g(n) + w \cdot c^*(n, d)$, where $c^*(n, d)$ is the cost of the path from n to d.

Theorem 4 *Even if the weak dropping rule is applied, there will always be a node p from an optimal solution path on* open *such that $g(p) \leq w \cdot g^*(p)$.*

Proof: We proceed by induction over iterations of search. The theorem clearly holds after expansion of the initial state. For the induction step, we note that node p can only come off *open* by being expanded. If its child p_i that lies along the optimal path is added to *open*, the theorem holds. The only way it won't be is if there exists a previous duplicate copy p'_i and the pruning rule holds, i.e., $g(p'_i) \leq g(p_{i-1}) + w \cdot c^*(p_{i-1}, p_i)$. By the inductive hypothesis, $g(p_{i-1}) \leq w \cdot g^*(p_{i-1})$, and by definition $g^*(p_{i-1}) + c^*(p_{i-1}, p_i) = g^*(p_i)$, so we have $g(p'_1) \leq w \cdot g^*(p'_1)$. □

Note that the use of this technique prohibits using the global minimum f value as a lower bound on the optimal solution's cost, because g values can now be inflated by up to a factor of w. However, if s is the incumbent and we search until the global minimum f' value is $\geq g(s)$, as in a serial weighted A* search, then w-admissibility is assured:

Corollary 3 *If the minimum f' value is $\geq g(s)$, where s is the incumbent, then we have $g(s) \leq w \cdot g^*(\text{opt})$*

Proof: Recall node p from Theorem 4. $g(s) \leq f'(p) = g(p) + w \cdot h(p) \leq w \cdot (g^*(p) + h(p)) \leq w \cdot g^*(opt)$. □

Algorithms

We implemented and tested weighted versions of the parallel search algorithms discussed above: wAPRA*, wBFPSDD, and wPBNF. All algorithms prune nodes based on the $w \cdot f$ criterion presented in Theorem 3 and prune entire open lists on f' as in Corollary 2. Additionally, in the grid pathfinding domain all parallel algorithms drop duplicate nodes using the criteria of Theorem 4. Search terminates when all nodes have been pruned by the incumbent solution.

We also considered an additional algorithm that has not been evaluated in prior work. There has been much work on complex data structures that retain correctness under concurrent access. We implemented a simple parallel A* search, which we call Lock-free A*, in which all threads access the same concurrent priority queue and concurrent hash table. We implemented the concurrent priority queue data structure of Sundell and Tsigas (2003). For the closed list, we used a concurrent hash table data structure developed by Click (2008) and implemented by Dybnis (2009)

Evaluation

We have implemented and tested the parallel bounded suboptimal 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 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 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 that 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) that we determined by experimentation. On grids and sliding tiles, $n$block data structures are created lazily, so only the visited part of abstract graph needs to fit in memory.

STRIPS Planning We used a domain-independent optimal sequential STRIPS planner employing regression and the max-pair admissible heuristic of Haslum and Geffner (2000). The abstraction function 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 generated 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 work on different candidate abstractions.

Figure 2 shows the performance of the parallel search algorithms in terms of speed-up over serial weighted A*. wAPRA* was not able to solve some problems due to excessive memory consumption (marked M in the table). It also exhibited the poorest speed-up. All algorithms had better speed-up at 7 threads than at 2. wPBNF seems to have the largest and most consistent speed-up. Its performance is sometimes slower than weighted A* (speed-up < 1) on short-running problems, but is generally larger for the more difficult instances or lower weights. On one problem, freecell-3, serial weighted A* performs much worse as the weight increases. Interestingly, wPBNF and wBFPSDD do not show this pathology, and thus record speed-ups of up to 1,700 times.

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 tested three different action models: 1) Four-way unit cost, in which each move has the same cost. 2) Four-way life cost, which captures the intuition that goal are often quickly achieved if cost is no object. Moves in the life cost model have cost equal to the row where the move was performed:

		wPBNF				wBFPSDD				wAPRA*			
		1.5	2	3	5	1.5	2	3	5	1.5	2	3	5
2 Threads	logistics-8	2.68	2.27	**4.06**	1.00	1.86	2.12	1.14	0.86	1.01	0.98	0.64	1.14
	blocks-16	0.93	0.54	0.48	1.32	0.34	0.19	0.16	**2.42**	1.16	0.82	**2.42**	0.25
	gripper-7	2.01	1.99	1.99	**2.02**	1.91	1.89	1.86	1.84	0.77	0.76	0.76	0.75
	satellite-6	2.02	1.53	5.90	3.04	1.71	2.22	**7.50**	2.80	0.71	0.84	0.67	0.79
	elevator-12	2.02	2.08	**2.21**	2.15	1.76	1.76	1.81	2.18	0.70	0.68	0.72	0.73
	freecell-3	2.06	0.84	8.11	10.7	1.42	0.54	16.9	**55.8**	1.12	1.18	0.84	1.03
	depots-13	2.70	**4.49**	0.82	0.81	1.48	1.58	0.18	0.16	0.74	1.12	0.32	0.29
	driverlog-11	0.85	0.19	0.69	0.62	0.85	0.11	0.19	0.21	**0.86**	0.33	0.20	0.19
	gripper-8	2.06	2.04	**2.08**	2.07	2.00	1.96	1.97	1.98	0.61	M	M	M
7 Threads	logistics-8	**7.10**	6.88	1.91	0.46	3.17	3.59	0.11	0.1	3.17	2.77	2.23	1.02
	blocks-16	**2.87**	0.7	0.37	1.26	0.49	0.22	0.01	0.32	2.67	1.06	1.02	0.13
	gripper-7	**5.67**	5.09	5.07	5.18	4.33	4.28	4.14	4.05	1.76	1.75	1.74	1.80
	satellite-6	4.42	2.85	2.68	**5.89**	3.13	2.31	3.01	1.05	1.10	0.96	1.18	0.97
	elevator-12	**6.32**	6.31	7.10	3.68	3.78	4.04	3.95	0.94	0.96	1.03	1.01	1.01
	freecell-3	7.01	2.31	131	**1721**	2.12	0.70	44.5	137	2.48	0.76	2.93	2.96
	depots-13	**3.12**	1.80	0.87	0.88	1.88	1.87	0.13	0.12	M	2.58	0.12	0.11
	driverlog-11	**1.72**	0.43	0.67	0.42	1.26	0.21	0.30	0.23	M	M	M	M
	gripper-8	**5.85**	5.31	5.40	5.44	4.62	4.55	4.55	4.51	M	M	M	M

Figure 2: Speed-up over serial weighted A* on STRIPS planning problems for various weights.

moves at the top of the grid are free, moves at the bottom cost 1200. This strongly differentiates between the shortest path, which takes expensive moves directly to the goal, and the cheapest path, which prefers deviations toward the top of the grid, where moves are cheaper. 3) Eight-way unit cost, where horizontal and vertical moves have cost one, but diagonal movements cost $\sqrt{2}$. The real-valued costs and abundance of different paths to the same state make the domain very different from the previous two.

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. A 10,000 $n$block abstraction was used, and the min expansions parameter was set to 64. Duplicate states that have already been expanded are dropped in the serial wA* algorithm, as discussed by Likhachev, Gordon, and Thrun (2003).

Results are presented in the upper portion of Figure 3, again in terms of speed-up versus serial weighted A*. In contrast to Figure 2, columns represent number of threads and rows represent various weights. In general, the parallel algorithms show increased speed-up as threads are added, and decreased speed-up as the weight is increased. All algorithms have difficulty in the unit 8-way problems because they cannot as many drop duplicates as serial wA*. wPBNF is consistently the fastest algorithm. Lock-free A* is not shown in the table—its fastest speed-up on unit 4-way grids was 0.45 (more than twice the time of serial).

Sliding Tile Puzzles The sliding tiles puzzle is a common domain for benchmarking heuristic search algorithms. We used forty-three of the easiest Korf 15-puzzle instances that were solvable by A* in 15GB of memory. We did not use duplicate dropping in this domain, as we found it made the searches perform worse. These problems do not have as many duplicates as grids and have fewer paths to the goal. An abstraction that takes into account the blank, 1-tile, 2-tile was used, yielding 3,360 $n$blocks. The min expansions parameter was set to 32.

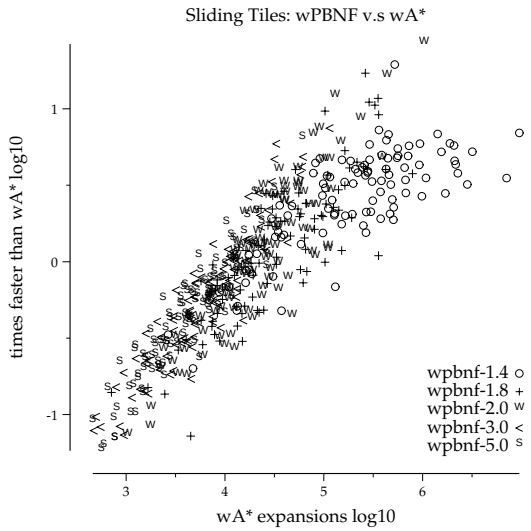

Figure 4: wPBNF speed-up over wA* as a function of problem difficulty.

Results are presented in the lower portion of Figure 3. We see that all of the algorithms lose their advantage over wA* as weights increase, presumably because the overhead of threads and contention is too great compared to the low number of nodes expanded. wPBNF consistently performs better than wPRA* or wBFPSDD. Also, wPBNF and wBFPSDD consistently improve in performance as threads are added for all but the largest weight values. wPRA* increases at first, but its performance drops off after a few threads.

To confirm our understanding of the effect of problem size on speed-up, Figure 4 shows a comparison of wPBNF to weighted A* on all 100 Korf 15-puzzles. Each point represents a run on one instance at a particular weight, the y axis represents wPBNF speed-up relative to serial wA* (on a log scale), and the x axis represents the number of nodes expanded by wA* (also on a log scale). Different glyphs represents different weight values used for both wPBNF and

		wPBNF					wBFPSDD					wAPRA*				
	w	1	2	4	5	8	1	2	4	5	8	1	2	4	5	8
Unit 4	1.1	0.83	1.49	2.69	3.14	**4.04**	0.87	1.31	1.97	2.21	2.57	0.93	1.50	2.09	1.99	1.76
	1.2	0.76	1.38	2.44	2.77	**3.38**	0.83	1.24	1.84	2.06	2.22	0.88	1.40	1.92	1.83	1.58
	1.4	0.52	0.99	1.60	1.75	**1.95**	0.78	1.15	1.57	1.68	1.53	0.55	0.92	1.26	1.22	1.05
	1.8	0.53	0.61	0.67	0.68	0.65	0.72	**0.73**	0.67	0.63	0.47	0.51	0.68	0.63	0.51	0.44
Life 4	1.1	1.02	1.84	3.45	4.13	**5.43**	1.05	1.63	2.48	2.85	3.52	0.92	1.44	2.41	2.42	2.20
	1.2	1.02	1.86	3.44	4.12	**5.40**	1.05	1.61	2.45	2.80	3.48	0.93	1.42	2.45	2.43	2.23
	1.4	1.05	1.91	3.50	4.12	**5.23**	1.05	1.62	2.40	2.71	3.21	0.92	1.70	2.51	2.47	2.25
	1.8	0.40	0.96	1.82	1.94	**2.18**	0.99	1.39	1.81	1.90	1.80	0.19	0.34	0.57	0.46	0.72
Unit 8	1.1	0.57	1.08	1.70	1.79	1.75	0.69	1.06	1.47	1.53	1.62	0.67	1.05	1.67	**1.87**	1.58
	1.2	0.29	0.28	0.26	0.25	0.24	0.38	0.42	0.39	0.36	0.29	**0.78**	0.64	0.43	0.36	0.26
	1.4	0.15	0.14	0.13	0.13	0.12	0.26	0.34	0.30	0.27	0.22	**0.77**	0.51	0.33	0.28	0.19
	1.8	0.14	0.13	0.12	0.12	0.11	0.29	0.28	0.24	0.22	0.18	**0.77**	0.51	0.33	0.28	0.20
Tiles	1.4	0.65	1.12	1.65	1.92	**2.62**	0.64	0.87	1.07	1.32	1.54	0.63	1.03	0.90	1.70	1.35
	1.7	0.41	0.76	1.37	1.50	1.49	0.60	0.76	0.99	1.06	1.14	0.61	0.97	0.98	**1.79**	1.06
	2.0	0.37	0.62	1.10	1.34	**1.46**	0.43	0.47	0.62	0.63	0.66	0.61	1.23	0.82	1.37	0.96
	3.0	0.74	0.62	**0.90**	0.84	0.78	0.34	0.39	0.46	0.42	0.32	0.57	0.86	0.54	0.68	0.45

Figure 3: Average speed-up over serial weighted A* for various numbers of threads.

wA*. The 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 weight 1.4 runs levels 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 speed-up greater than 10. 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 are easier.

Anytime Search

A popular alternative to bounded suboptimal search is anytime search, in which a highly suboptimal solution is returned quickly and then improved solutions are returned over time until the algorithm is terminated (or the optimal solution is proved). The two most popular anytime heuristic search algorithms are Anytime weighted A* (AwA*) (Hansen and Zhou 2007), in which a weighted A* search is just allowed to continue, pruning when the unweighted $f(n) \geq g(s)$, and Anytime Repairing A* (ARA*) (Likhachev, Gordon, and Thrun 2004), in which the weight is lowered when a solution meeting the current suboptimality bound has been found and a special inconsistent list is kept that allows the search to expand a node at most once during the search at each weight.

We used the PBNF framework to implement Anytime weighted PBNF (AwPBNF). Because PBNF inherently continues searching after the first solution is found, it serves very naturally as an anytime algorithm in the style of Anytime weighted A*. In the planning domain, we also implemented Anytime weighted BFPSDD (AwBFPSDD) and Anytime weighted APRA* (AwAPRA*) for comparison.

Evaluation

The implementation and empirical set-up was identical to that used for suboptimal search.

STRIPS Planning Figure shows the speed-up of the parallel anytime algorithms over serial anytime weighted A*. All algorithms were run until an optimal solution was proved. (For a weight of 5, AwA* ran out of memory on blocks-14, so our speed-up values at that weight for that instance are lower bounds.) For all algorithms, speed-up over serial generally increased with more threads and a higher weight. The PBNF algorithm generally gives the fastest performance, just as in suboptimal search.

Hansen and Zhou (2007) show that AwA* can lead to speedup over A* for some weight values in certain domains. Finding a suboptimal solution quickly allows f pruning that keeps the open list short and quick to manipulate, resulting in faster performance even though AwA* expands more nodes than A*. We found a similar phenomenon in the corresponding parallel case. Figure 6 shows speedup over unweighted optimal PBNF when using various weights for the anytime algorithms. A significant fraction of the values are over 1, representing a speed-up when using the anytime algorithm instead of the standard optimal parallel search. In general, speed-up seems more variables as the weight increases. For a weight of 1.5, AwPBNF always provides a speed-up.

Grid Pathfinding To fully evaluate anytime algorithms, it is necessary to consider their performance profile, ie, the expected solution quality as a function of time. While this can be easily plotted, it ignores the fact that the anytime algorithms considered in this paper all have a free parameter, namely the weight or schedule of weights used to accelerate the search. For ARA*, we considered several different weight schedules: $\{7.4, 4.2, 2.6, 1.9, 1.5, 1.3, 1.1, 1\}$, $\{4.2, 2.6, 1.9, 1.5, 1.3, 1.1, 1.05, 1\}$, $\{3, 2.8, \ldots, 1.2, 1\}$, $\{5, 4.8, \ldots, 1.2, 1\}$, for all other algorithms we consider weights of 1.1, 1.2, 1.4, and 1.8.

The two left most panels in Figure 7 shows the "raw" performance profiles for ARA* and AwPBNF on unit cost four-

		AwPBNF				AwBFPSDD				AwAPRA*			
		1.5	2	3	5	1.5	2	3	5	1.5	2	3	5
2 Threads	logistics-6	1.06	1.35	1.94	**1.98**	0.68	0.91	0.91	0.56	0.93	0.89	1.37	1.20
	blocks-14	1.91	1.99	13.22	22.4	1.02	1.18	7.71	11.9	1.33	8.67	**52.2**	6.65
	gripper-7	**2.05**	1.96	1.99	1.95	1.94	1.89	1.94	1.82	0.80	0.79	0.78	0.76
	satellite-6	1.58	1.96	**1.98**	1.91	1.85	1.87	1.49	1.80	0.75	0.79	0.79	0.80
	elevator-12	2.01	2.07	**2.13**	2.07	1.74	1.74	1.75	1.69	0.67	0.67	0.67	0.68
	freecell-3	1.93	1.06	2.78	**6.23**	1.45	1.46	1.97	3.08	1.30	1.32	4.71	1.44
	depots-7	1.94	2.00	2.01	**4.10**	1.44	1.45	1.32	2.40	1.22	1.29	1.26	2.69
	driverlog-11	1.95	**2.10**	1.99	0.77	1.73	1.78	1.59	1.41	1.16	1.20	1.14	1.21
	gripper-8	2.04	2.05	**2.09**	2.06	2.01	2.00	1.98	1.96	M	M	M	M
7 Threads	logistics-6	2.04	2.46	4.19	**4.21**	1.02	1.35	1.37	0.92	1.41	1.32	1.76	1.80
	blocks-14	3.72	22.4	25.7	7.20	1.60	1.96	12.1	19.9	2.49	15.2	**99.2**	170
	gripper-7	**5.61**	5.05	5.03	5.06	4.30	4.24	4.16	3.96	1.70	1.72	1.69	1.71
	satellite-6	**5.96**	4.66	5.74	4.70	4.10	3.54	4.16	3.88	1.27	1.27	1.28	1.29
	elevator-12	6.18	6.03	**6.20**	6.05	3.71	3.74	3.73	3.38	0.94	0.92	0.94	0.93
	freecell-3	3.54	1.50	**15.3**	11.5	1.78	1.82	2.59	4.14	3.57	3.71	11.8	4.28
	depots-7	5.74	5.52	5.48	**10.8**	2.02	1.96	1.92	3.68	M	M	M	M
	driverlog-11	5.78	**5.83**	5.73	2.18	2.58	2.86	2.57	2.34	M	M	M	M
	gripper-8	**5.82**	5.36	5.39	5.39	4.62	4.55	4.57	4.50	M	M	M	M

Figure 5: Speed-up of anytime search to optimality over serial AwA* on STRIPS planning using various weights.

		AwPBNF				AwBFPSDD				AwAPRA*			
		1.5	2	3	5	1.5	2	3	5	1.5	2	3	5
7 Threads	logistics-6	1.48	1.84	**2.36**	2.27	0.68	0.93	0.71	0.54	1.12	1.08	1.08	0.98
	blocks-14	1.24	1.22	0.21	0.03	0.87	0.18	0.16	0.16	**1.46**	1.46	1.42	0.94
	gripper-7	**1.07**	0.99	0.99	1.00	0.93	0.95	0.93	0.92	0.99	1.03	1.01	0.99
	satellite-6	**1.10**	0.87	1.08	0.88	0.88	0.77	0.91	0.90	0.99	1.00	1.01	1.02
	elevator-12	**1.06**	1.04	1.04	1.03	0.77	0.78	0.76	0.73	1.02	1.00	1.00	1.00
	freecell-3	1.05	0.44	0.99	0.29	0.64	0.64	0.20	0.14	1.13	**1.16**	0.82	0.10
	depots-7	**1.20**	1.15	1.15	1.08	0.54	0.53	0.52	0.49	M	M	M	M
	driverlog-11	1.16	1.15	**1.19**	0.43	0.53	0.58	0.54	0.50	M	M	M	M
	gripper-8	**1.06**	0.99	0.99	1.00	0.99	0.98	0.99	0.97	M	M	M	M

Figure 6: Speed-up of anytime search to optimality over PBNF on STRIPS planning problems using various weights.

way grid pathfinding problems. We can see that the performance of ARA* is similar with all weight schedules. The performance of AwPBNF, however, varies greatly for different weights. For weights 1.2 and 1.1 the algorithm finds the optimal solution immediately and merely spends the remaining time proving it. For weights 1.4 and 1.8 the algorithm finds a stream of incumbents of gradually increasing quality.

In order to compare algorithms, we make the assumption that, in any particular application, the user will attempt to find the parameter setting giving good performance for the timescale they are interested in. Under this assumption, we can plot the performance of each anytime algorithm by computing, at each time, the best performance that was achieved at that time by any of the parameter settings tried. We refer to this concept as the 'lower hull' of the profiles, because it takes the minimum over the profiles for each parameter setting.

The right four panels in Figure 7 present the lower hulls of both serial and parallel algorithms on both grid pathfinding and sliding tile puzzles. In each panel, the y axis represents solution cost (relative to optimal) and the x axis represents wall time (relative to the performance of serial A*). Both serial and parallel algorithms are plotted. The profiles start when the algorithm first returns a solution and ends when the algorithm has proved optimality.

In the unit four-way grids (top center panel), we see that, although ARA* has a smooth anytime profile, AwA* converges much faster. While AwPNBF's profile resembles AwA*'s, it successfully exploits multiple threads and dominates the other algorithms. Due to performance issues in harder grid domains, AwA* is not shown in further grid plots. In the unit eight-way problems (bottom center panel), duplicate handling is critical. While AwPBNF takes longer to find an initial solution, the solution it finds is very good and it quickly converges to optimal. In the life four-way (top right panel) grids, ARA* performs poorly, and AwPBNF continues to scale well. Overall, AwPBNF is very effective on these domains.

Sliding Tile Puzzles The lower right panel of Figure 7 presents lower hulls for the anytime algorithms on 43 of the easiest sliding tile puzzles. (Memory constraints prevent A* from finding optimal solutions to the full benchmark set.) For ARA* we used the same weight schedules as in grids, however, for all other algorithms, we used weights of 1.4, 1.7, 2.0, 3.0 and 5.0. We see from the figure that although serial AwA* finds a suboptimal solution sooner, AwPBNF quickly becomes competitive and then converges to optimal much faster.

Conclusions

PBNF approximates a best-first search ordering while trying to keep all threads busy. We proved the correctness of

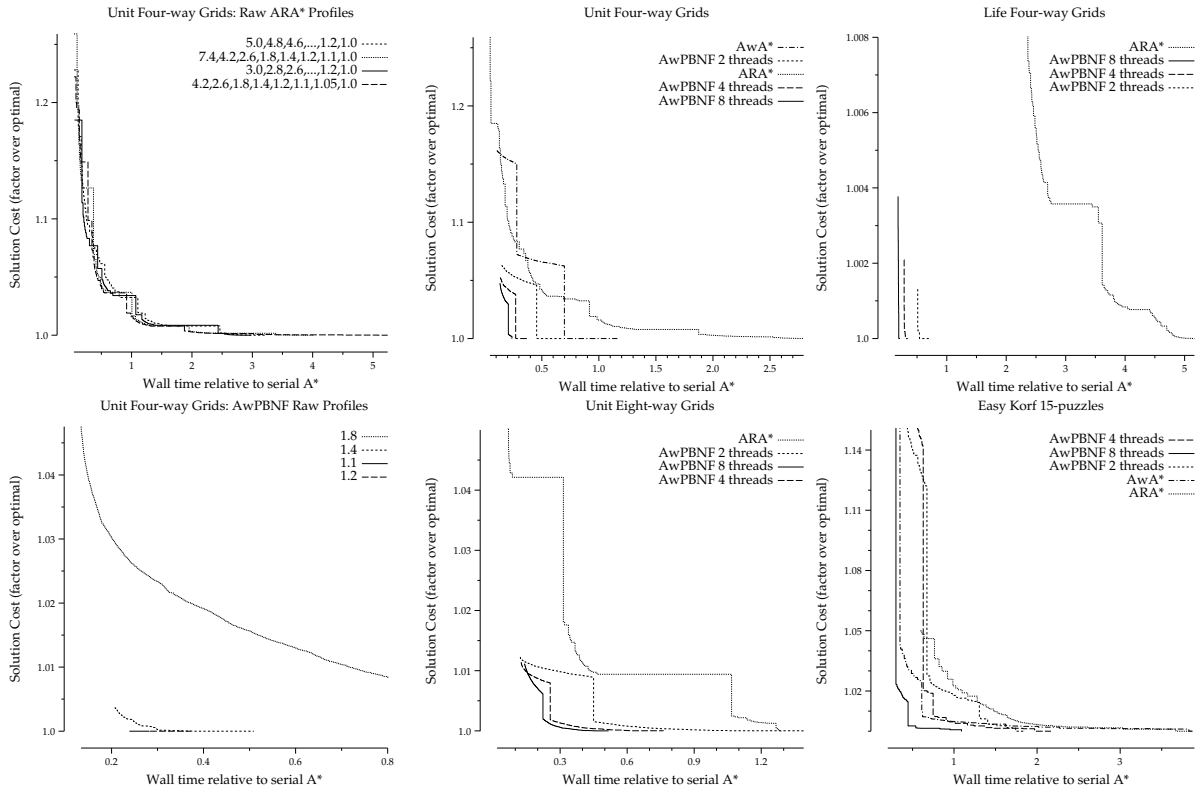

Figure 7: Lower hull anytime profiles.

the PBNF search framework and used it to derive new suboptimal and anytime algorithms. In a comprehensive empirical comparison with suboptimal and anytime variations of other proposed algorithms, we found that the PBNF algorithms exhibit better parallel speed-up and faster absolute performance. Its advantage over serial search grows with problem difficulty. The wPBNF algorithm outperforms its closest competitor, wBFPSDD, because of the lack of layer-based synchronization. Our results with Anytime wPBNF confirmed the observation of Hansen and Zhou (2007) that anytime weighted search can sometimes find and prove optimal solutions faster than plain optimal search. We conclude that suboptimal and anytime PBNF are promising methods to help planning systems scale with the increasingly parallel machines being built.

Acknowledgements

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

References

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-first heuristic search for multi-core machines. In *IJCAI-09*.

Click, C. 2008. Towards a non-blocking coding style. http://www.azulsystems.com/events/javaone_2008/-2008_CodingNonBlock.pdf.

Dybnis, J. 2008. Non-blocking data structures. http://code.google.com/p/nbds/.

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.

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

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *ICAPS*, 140–149.

Likhachev, M.; Gordon, G.; and Thrun, S. 2004. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of NIPS 16*.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Formal analysis. CMU TR-CS-03-148

Sundell, H., and Tsigas, P. 2003. Fast and lock-free concurrent priority queues for multi-thread systems. In *Parallel and Distributed Processing Symposium, 2003*.

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

Zhou, R., and Hansen, E. 2006a. Domain-independent structured duplicate detection. In *AAAI-06*, 1082–1087.

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

Zhou, R., and Hansen, E. A. 2007. Parallel structured duplicate detection. In *AAAI-07*.