Parallel Breadth-First Heuristic Search on a Shared-Memory Architecture

Yang Zhang and **Eric A. Hansen**
Dept. of Computer Science and Engineering
Mississippi State University
Mississippi State, MS 39762
{*fz15, hansen*}*@cse.msstate.edu*

Abstract

We consider a breadth-first approach to memory-efficient graph search and discuss how to parallelize it on a shared-memory architecture that uses multithreading to achieve parallelism. The approach we develop for parallelizing breadth-first search uses *layer synchronization*, in which threads expand all nodes in one layer of the breadth-first search graph before considering any nodes in the next layer. We show that this allows simple and effective techniques for termination detection, dynamic load balancing, and concurrent access to shared data structures. The Fifteen Puzzle is used as an initial test domain.

Introduction

Parallel formulations of state space search algorithms have been extensively studied for decades (Rao & Kumar 1987; Kumar & Rao 1987; Kumar, Ramesh, & Rao 1988; Grama & Kumar 1999; Grama *et al.* 2003). Most work has focused on parallelization of depth-first search and best-first search. There has been comparatively little work on parallelization of breadth-first search. In large part, this is because the memory requirements of breadth-first search are traditionally greater than for its alternatives, and often impractical. However recent work shows that a breadth-first search strategy can be implemented in a memory-efficient way by using a divide-and-conquer method of solution recovery that only requires storing nodes that are on the search frontier (Korf *et al.* 2005). If lower and upper bounds are used to limit the size of the frontier, as in branch-and-bound search, a breadth-first frontier can be smaller than a best-first frontier, and *breadth-first heuristic search* requires less memory than A* in solving search problems with unit edge costs (Zhou & Hansen 2006).

In this paper, we consider how to parallelize breadth-first search. We consider search in implicit graphs with uniform edge costs, and describe a general approach to parallelization that applies to both blind breadth-first search and breadth-first heuristic search. We show that the layered structure of a breadth-first search graph allows a simple and effective approach to parallelization, especially on a shared-memory architecture using multithreading. The key idea of this approach, which we call *layer synchronization*, is for threads to expand all nodes in one layer before considering any nodes in the next layer. We show that layer synchronization leads to simple and effective approaches to several key designs in parallel graph search, including distributed termination detection, dynamic load balancing, and concurrent access to shared data structures.

Parallel processing offers two advantages. First, it can reduce running time. Second, parallel systems often provide significantly more physical memory. Focus on the second advantage is more often associated with distributed-memory parallelism, however. On a shared-memory architecture, even a sequential program can utilize any additional physical memory, and the primary focus of parallelization is to reduce running time. Since we consider a shared-memory parallel architecture in this paper, we focus on using parallelization to reduce search time rather than to increase available memory.

The paper begins with a brief review of previous work on breadth-first search and its parallelization. Layer synchronization is then introduced as an approach to parallelizing breadth-first search. This is followed by discussion of how to design a Closed list that allows concurrent access and an Open list that allows effective load balancing. Several other issues, including termination detection, are also discussed, and experimental results for the Fifteen Puzzle are reported.

Background

We begin with brief reviews of a breadth-first approach to heuristic search, previous approaches to parallelizing breadth-first search, and shared-memory parallelization.

Breadth-first heuristic search

Frontier search is a memory-efficient approach to graph search that only stores the Open list, and saves memory by not storing the Closed list (Korf *et al.* 2005). It uses special techniques to prevent regeneration of previously closed nodes that are no longer in memory. Because the traditional traceback method of solution recovery requires all closed nodes to be in memory, it also uses a divide-and-conquer method of solution recovery. In divide-and-conquer solution recovery, each search node keeps track of an intermediate node along the best path that leads to it. When the goal node is selected for expansion, the intermediate node associated with it must be on an optimal solution path. This intermediate node is used to divide the original search problem into

two subproblems — the problem of finding an optimal path from the start node to the intermediate node, and the problem of finding an optimal path from the intermediate node to the goal node. These subproblems are then solved recursively by the same search algorithm until all nodes along an optimal solution path for the original problem are identified.

Frontier search is a general strategy for memory-efficient graph search that can be applied to A*, Dijkstra's single-source shortest-path algorithm, and breadth-first search. Zhou and Hansen (2006) show that frontier breadth-first branch-and-bound search is more memory-efficient than frontier A* because a breadth-first frontier is usually smaller than a best-first frontier. They call this approach *breadth-first heuristic search*. They also describe a simple approach to preventing regeneration of previously closed nodes that can only be used in breadth-first search. In this approach, called *layered duplicate detection*, the Closed list is stored in layers, one for each g-cost, and the shallowest layers are deleted to recover memory. In undirected graphs, it is only necessary to keep the single most recent layer of closed nodes in memory in order to detect all duplicates and prevent regeneration of already closed nodes. In directed graphs, it may be necessary to store more than one previous layer of closed nodes in memory, where the number needed to completely prevent regeneration of closed nodes depends on the structure of the graph. Even if only one previous layer of a directed search graph is stored in memory, Zhou and Hansen show that the number of times a node can be regenerated is bounded by the depth of the search, which means there is little node regeneration overhead in practice. In the rest of this paper, we consider how to parallelize breadth-first heuristic search with layered duplicate detection. But our approach to parallelization applies to any breadth-first search algorithm.

Parallelization of breadth-first search

One application area in which parallel breadth-first search has been previously studied is model checking (Lerda & Sisto 1999; Brim *et al.* 2001; Heyman *et al.* 2002; Barnat, Brim, & Chaloupka 2003). Verification of a model requires exhaustive search of a state space. Because best-first search does not have an advantage over breadth-first search in exhaustive exploration, breadth-first search is widely used.

Distributed-memory parallel model checking allows larger models to be verified by making more physical memory available, as well as by reducing running time. Barnat et al. (2003) propose a distributed-memory parallel LTL model checker that uses breadth-first search combined with nested depth-first search for cycle detection. They use a layer synchronized approach that is similar to the approach we develop, although they do not consider dynamic load balancing or several other issues discussed in this paper. Distributed-memory parallelization of breadth-first search has also been used for symbolic model checking, although without layer synchronization (Heyman *et al.* 2002). Memory-efficient graph search using divide-and-conquer solution recovery, which we consider in this paper, is not yet used in model checking. However, for an initial exploration of this idea, see (Lamborn & Hansen 2006).

Korf and Schultze (2005) use breadth-first frontier search with external memory to perform an exhaustive search of the Fifteen Puzzle state space. They consider disk-based search, whereas we consider in-memory search, and the issues are different. They emphasize the advantages of multithreading for search algorithms with intensive disk I/O.

Parallel breadth-first search has also been studied in areas such as analysis of explicitly stored graphs (Yoo *et al.* 2005). But here, the structure and shape of the graph are known in advance, and the issues are quite different.

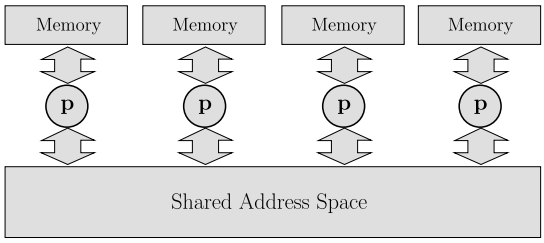

Figure 1: Shared-Address-Space Concept

Multithreading Parallelism

The concept of shared-memory parallelism historically refers to multiple processors that truly share physical memory. With current hardware, it is probably more accurate to refer to *shared-address-space* parallelism (Grama *et al.* 2003). The term shared-address-space emphasizes a logical view of the underlying architecture. The physical organization may include distributed memory interconnected via a high-speed network. If the system gives the programmer a global view of the memory space, and communication over the interconnection network is invisible to the programmer, the system is said to be a shared-address-space architecture. Figure 1 illustrates this concept. If the time it takes a processor to access any memory location in the system is identical, the system is referred to as a *uniform memory access*, or UMA, architecture. If the time for accessing different memory locations in the system is not identical, then it is called a *non-uniform memory access*, or NUMA, architecture. Many modern shared-memory machines are NUMA architectures. On a NUMA architecture, it can be useful for application programs to enforce data locality because accessing remote memory may be slower than accessing local memory. In the rest of this paper, we assume the shared-address-space concept when we refer to shared-memory parallelism.

Threads are the preferred model for programming shared-address-space machines. A thread is a stream of control flow inside a process. Threads are much faster to manipulate and schedule than processes. Threads also have a logical memory model similar to the shared-address-space architecture. Each thread has its own private stack space but interacts with other threads through the memory space of the process. Thread interactions within the shared data region require explicit synchronization. A key to achieving good parallel performance is minimizing synchronization overhead and its frequency.

Layer Synchronization

In this section, we describe the overall approach we adopt for parallelizing breadth-first heuristic search. In subsequent sections, we fill in the details.

In our approach, a graph is searched in parallel by several threads using a distributed Open list and a single shared Closed list. The part of the Open list belonging to each thread is a private copy that is only accessed by that thread. All generated nodes, whether open or closed, are stored in a Closed list that is shared among all threads. Duplicate detection is performed by threads concurrently querying and updating this shared Closed list.

A key aspect of our approach is that the Open and Closed lists are indexed by layer, and the search is *layer synchronized*. By layer synchronization, we mean that the threads must expand all nodes in one layer before expanding any nodes in the next layer. Layer synchronization is necessary in the memory-efficient approach to search we adopt, in which previous layers of the search graph can be deleted to recover memory. If the graph being searched is undirected, for example, a layer of the Closed list can be deleted as soon as all of the nodes in its successor layer have been expanded. If we were to allow the threads to simultaneously expand nodes in many different layers, all of these layers of the Closed list would need to be kept in memory.[1]

We use a single *control thread* to initiate the search and coordinate the activities of the other threads, which are called *working threads*. The control thread begins the breadth-first search. As soon as the number of open nodes in a layer exceeds a threshold that is equal to or greater than the number of working threads available, the open nodes in the layer are evenly partitioned among the working threads, which continue the search in parallel. Once a working thread finishes expanding the nodes in the current layer of its Open list, it is not allowed to expand nodes in the next layer until all of the other working threads are finished expanding nodes in the current layer. If one thread finishes much sooner than others, dynamic load balancing can be used to redistribute the open nodes in the layer so that a thread does not remain idle too long. This is discussed in a later section.

After the control thread spawns the working threads, it goes to sleep until awoken by a working thread that no longer has open nodes to expand. When the control thread sees that all working threads have no more open nodes in the current layer to expand, it signals all the working threads to proceed to the next layer. Before expanding the next layer, a previous layer of the breadth-first graph is deleted in order to recover memory.

When breadth-first search with layer synchronization is used, distributed termination detection becomes straightforward. The first solution found by any thread is guaranteed to be optimal. Once a solution is found, the search is terminated and divide-and-conquer solution recovery proceeds in the usual way.

[1] Indexing the Closed list by layer also saves memory by allowing a smaller node data structure that does not include g-cost, since all nodes in the same layer share the same g-cost.

Concurrent Closed List

In graph search, the Closed list is usually implemented as a hash table. In parallel graph search, the hash table must support concurrent access and allow synchronization. We consider some alternative designs and show how layer synchronization can simplify design of a concurrent hash table.

An obvious way to design a concurrent hash table is to use multiple locks to synchronize access. This has several drawbacks, however. First, locks take space, and guarding each entry with a (composite) lock can increase space overhead substantially. Second, use of so many locks can make it difficult or even prohibitive to resize a hash table since doing so requires obtaining all locks in the table. Third, read operations occur more frequently than other hash table operations, and using a composite read/write lock, a read request cannot be granted when a thread is adding or removing an entry's contents. Given that read operations occur so frequently, a non-blocking "read" would be more useful.

A state-of-the-art concurrent hash table is Doug Lea's concurrent hash map data structure for the Java concurrency package (Lea 2006). In Lea's design, the hash table is partitioned into multiple internal segments. Each segment is a complete hash table of its own that is guarded by a mutex lock. This design makes the space overhead for locks insignificant. It also allows individual segments to be resized instead of having to resize the entire hash table. Note that each segment can also be viewed as a coarsened hash entry in a traditional hash table. An item is first hashed to a segment and then it is hashed to a segment entry. Since different items are hashed into different segments under a uniform hash function, different threads can usually operate on different segments without blocking each other. The number of segments can be adjusted to maximize thread concurrency. Also, by cleverly arranging and cloning segment entries, the "read/add," "read/remove," and "read/resize" operations can be performed at the same time by different threads even on the same segment.[2] Thus, this design allows a completely non-blocking "read" operation.

Our concurrent hash table design adopts the same internal segment approach used by Lea. By utilizing the layered search structure, however, we are able to produce a non-blocking "read" operation more easily (without the need to have garbage collection or to manage complex memory regions). Concurrent execution of "read/add" operations is allowed the same way as in Lea's design. Concurrent "read/remove" operations can be addressed more simply. In breadth-first heuristic search, the only place the algorithm removes items from the Closed list is when a layer is deleted. During layer expansion, no "remove" operations occur. This temporal access pattern is enough to guarantee that the "read" and "remove" operations for the Closed list cannot overlap. Concurrent "read/resize" operations can also be effectively addressed by leveraging the layered search structure. The "resize" operation will only occur during "add" operations to the Closed list, and can be deferred until all "add" operations for a layer have been performed

[2] However, this also relies on Java's garbage collection capability.

and a previous layer is being deleted. Essentially, layer synchronization partitions the search into two phases. The first phase is the node expansion phase where each thread expands nodes in the current layer. The second phase is the layer deletion phase where each thread deletes unnecessary previous layers from memory. The node expansion phase only allows "read" and "add" operations to the Closed list while the layer deletion phase only allows "remove" and "resize" operations. When "resize" is triggered by the "add" operation, the "resize" request is only flagged and the actual action is deferred until the layer deletion phase. This may increase the hash table load factor until the actual "resize" operation. However, such flagged "resize" operations do not happen frequently and the cost is well amortized into the whole process.

Finally, we note that we use chaining to resolve hash collisions in order to maximize concurrency and space utilization. An open-addressing approach to resolving collisions may result in large areas of the hash table being visited, which can incur synchronization overhead. It is therefore more difficult to design a non-blocking "read" operation. Also in order to keep the hash table reasonably loaded to reduce access time, much of the space in the table is wasted in an open-addressing design.

Chunk Based Open List

The Open list is distributed and each thread has its own private copy. Since duplicate detection requires a global list, all graph nodes are hashed into the Closed list as soon as they are generated. In other words, the Closed list contains all open and closed nodes, and duplicate detection involves checking the Closed list. If the open nodes are also stored in a distributed Open list, there is memory overhead for nodes that coexist in both the Open and Closed lists for some time. This can be alleviated by only storing pointers in the Open list. The key issues we address in design of the Open list are efficient dynamic load balancing and memory utilization.

We choose not to implement the Open list as a hash table. Since it is not used for duplicate detection, fast access is not necessary, and, since a hash table is not designed to be split easily and efficiently, a hash table is not a suitable data structure for dynamic load balancing. Instead, we implement the Open list as a linked list. The split and splice operations of a linked list have constant-time complexity, which facilitates dynamic load balancing. However a traditional linked list incurs considerable space overhead because it requires storing a pointer with every list node. In addition, a traditional linked list does not exhibit strong data locality. For these reasons, we introduce a "coarsened linked list" that we call "chunk list." A chunk list is a list of chunks, where each chunk is a fix-size deque (a double ended queue). A chunk list is a hybrid of a linked list and an array. On the chunk level, operations such as split and splice are the same as for conventional linked lists. But inside a chunk, elements are aggregated compactly and have good data locality. Because each chunk can hold a large number of elements, the memory overhead for list pointers is negligible.

In parallel graph search, a private layered Open list is maintained for each thread, consisting of the current and next layer, where each layer is a chunk list of node pointers, as illustrated in Figure 2.

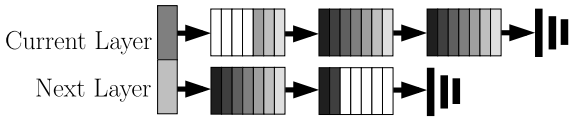

Figure 2: Chunk-based Open List

Dynamic Load Balancing

Parallel graph search requires balancing the workload for different processes so that processing power is not wasted by leaving some processes idle. Load balancing is typically dynamic because the irregular shape of the graph makes it difficult to statically partition the workload in a balanced way.

The structure of our distributed Open list is designed to facilitate dynamic load balancing, which is achieved by splitting and splicing chunks in the Open list among different threads. All working threads are arranged in a ring topology in which each thread has a predetermined neighbor thread. Each thread has a status flag that indicates whether or not it still has nodes to expand. When its Open list is empty, a thread sets its status flag, signals the control thread, and goes to sleep. Each thread is responsible for monitoring its neighbor's status flag. If it notices that its neighbor is idle, and a significant number of chunks remain in the current layer of its Open list to be expanded, it can migrate some of these chunks to its neighbor. It does so by splitting some chunks from the current layer of its Open list, putting them on its neighbor's Open list, and waking up its neighbor. The time complexity of dynamic load balancing is roughly linear in the number of chunks in the Open list. Because each chunk usually contains a large number of nodes, this is negligible compared to the complexity of node expansion. The pseudocode on the following page shows how the control (Algorithm 2) and working (Algorithm 1) threads cooperate in expanding the nodes in a layer, including dynamic load balancing.

This dynamic load balancing scheme requires virtually no shared resources. The control thread merely coordinates working threads and does not itself distribute work. Therefore the control thread is typically not a bottleneck. Layer synchronization, in which breadth-first search must explore *all* the nodes in the current layer before the next layer is considered, considerably simplifies the task of dynamic load balancing. In parallel best-first search, load balancing serves two purposes. It makes sure the processes are busy and it makes sure they are doing useful work. To ensure the latter, it must evenly distribute the most-promising nodes among processes. Because a ring virtual topology has low connectivity, it is usually not fast enough to distribute the most-promising nodes in an asynchronous parallel best-first search (Grama *et al.* 2003). For breadth-first search with layer synchronization, however, this is not a problem. The sole purpose of dynamic load balancing is to keep the pro-

```
 1: loop
 2:    while layer-done ≠ true do
 3:       while open list not empty do
 4:          if neighbor's flag = true then
 5:             if worth to migrate work then
 6:                split and splice open list
 7:                neighbor's flag ← false
 8:                wake up neighbor
 9:             end if
10:          end if
11:          if solution found then
12:             search-done ← true
13:             wake up control thread
14:             terminate all working threads and exit
15:          else
16:             do node expansion
17:          end if
18:       end while
19:       flag ← true {this thread runs out of open list}
20:       wake up control thread and sleep
21:       {eventually this thread will be woken up
           and possibly receive some additional
           states in its own open list}
22:    end while
23:    layer-done ← false
24:    if depth > upper-bound then
25:       exit
26:    end if
27:    prune previous Closed list layer
28:    go on to next layer
29: end loop
```

Algorithm 1: Working Thread

```
 1: sequential expansion until open list is big enough
 2: layer-done ← false
 3: search-done ← false
 4: all working threads' flag ← false
 5: distribute open list and start all working threads
 6: while search-done ≠ true do
 7:    go to sleep {will be woken up later}
 8:    if all working threads' flag = true then
 9:       layer-done ← true
10:       all working threads' flag ← false
11:       wake up all working threads
12:    end if
13: end while
```

Algorithm 2: Control Thread

cesses busy. Using layer synchronization, there can be no useless work. For this reason, the ring topology is sufficient for distributing work in parallel breadth-first search, at least for the relatively small-scale parallelism considered in this paper. To scale up to large scale parallelism, a topology with higher connectivity and a hierarchical structure may be more suitable. Such a change in load balancing topology should be relatively easy to incorporate into the search framework.

Preliminary Performance Analysis

We used the Fifteen Puzzle as an initial test domain for our implementation of parallel breadth-first heuristic search. Table 1 shows timing results for eight of the nine most difficult of the one hundred instances of the Fifteen Puzzle used as a test suite by Korf (1985). We ran the search algorithm on a Sun Fire 880 with eight 750MHz Ultrasparc III processors running the Solaris 10 operating system. All programs were compiled using GNU gcc version $4.0.1$. We used the "hoard" multiprocessor memory allocator (Berger *et al.* 2000; Berger 2006) (version $2.1.2$d) developed by Emery Berger at the University of Massachusetts at Amherst. In Table 1, the number of threads refers to the number of working threads and does not include the control thread. The measured time is the search wall time and includes initialization and finalization of all serial resources. The chunk size for the Open list is $10,240$, which means that a chunk holds $10,240$ puzzle state pointers. As Table 1 shows, parallelization of the search algorithm results in close to linear speedup.

In a multithreading program, most parallel overhead is due to synchronization in the shared-address-space. Additional overhead sometimes includes redundant work and process idling. But since our approach uses layer synchronization, there can be no redundant work. And using our dynamic load balancing scheme, there is little parallel overhead due to process idling. Thus, most parallel overhead appears to be due to concurrent access of the Closed list, and we have focused on making this as efficient as possible. Since we run our algorithm on a NUMA architecture, the memory locality problem could also affect performance. Our algorithm does not currently leverage memory locality. Finally, the performance of a multithreading programs is sensitive to-low level system and OS support, and we are trying to further improve this aspect of our implementation.

Conclusion and Future Work

We have presented a parallel formulation of breadth-first heuristic search for a shared-memory architecture. By using layer synchronization, in which threads expand all nodes in one layer of the breadth-first search graph before expanding any nodes in the next layer, we were able to simplify the algorithm in several ways and reduce parallel overhead. For example, a layer-synchronized search makes it impossible to do useless work and has a simpler termination condition. It allows design of a simpler concurrent hash table that includes a simple non-blocking read of the hash table. Since nodes in any layer of the search graph are not prioritized, it also simplifies dynamic load balancing. Using our chunk list

#	states expanded	1 thread	2 threads	3 threads	4 threads	5 threads	6 threads
60	767,633,572	29210.3	15063.8	10260.4	7916.1	6337.9	5381.7
82	549,571,484	21218.5	10957.9	7657.9	5713.1	4637.8	3933.5
66	275,077,252	8595.0	4399.8	2990.4	2256.4	1860.7	1550.9
49	243,792,202	7356.2	3874.9	2657.4	2027.5	1646.8	1383.8
17	219,026,812	4319.5	2294.1	1592.8	1218.9	996.0	848.9
59	158,917,399	3120.9	1645.1	1134.1	867.0	701.9	599.4
56	141,184,414	2702.0	1425.9	990.2	756.2	617.4	535.2
53	177,293,989	3312.1	1782.5	1237.6	941.5	773.2	663.5

Table 1: Timing results (in wall clock seconds) for eight of Korf's Fifteen Puzzle cases.

data structure, we were able to design a simple but efficient load balancing scheme.

This is preliminary work and we are continuing to improve the algorithm. Improvements that we are currently exploring include methods for compressing the Open and Closed lists and more efficient memory-management techniques. We also plan to test the algorithm on additional problems. Eventually, we will consider a layer-synchronized approach to parallel graph search in a distributed-memory environment.

Acknowledgments

Yang Zhang thanks his PhD advisor, Ed Luke, for helpful advice and support.

References

Barnat, J.; Brim, L.; and Chaloupka, J. 2003. Parallel breadth-first search LTL model-checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, 106–115.

Berger, E. D.; McKinley, K. S.; Blumofe, R. D.; and Wilson, P. R. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. Cambridge, Massachusetts.

Berger, E. D. 2006. The hoard allocator website. http://www.hoard.org/. Accessed March 19, 2006.

Brim, L.; Černá, I.; Krčál, P.; and Pelánek, R. 2001. Distributed LTL model checking based on negative cycle detectioin. In *Lecture Notes in Computer Science*, volume 2245, 96–107.

Grama, A., and Kumar, V. 1999. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering* 11(1).

Grama, A.; Gupta, A.; Karypis, G.; and Kumar, V. 2003. *Introduction to Parallel Computing: Second Edition*. Addison-Wesley. ISBN 0-201-64865-2.

Heyman, T.; Geist, D.; Grumberg, O.; and Schuster, A. 2002. A scalable parallel algorithm for reachability analysis of very large circuits. *Formal Methods in Systems Design* 21:317–338.

Korf, R., and Schultze, P. 2005. Large-scale parallel breadth-first search. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05)*, 1380–1385.

Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *Journal of the ACM* 52(5):715–748.

Korf, R. 1985. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Kumar, V., and Rao, V. N. 1987. Parallel depth-first search on multiprocessors part II: Analysis. *International Journal of Parallel Programming* 16(6):501–519.

Kumar, V.; Ramesh, K.; and Rao, V. N. 1988. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence (AAAI-88)*.

Lamborn, P., and Hansen, E. 2006. Memory-efficient graph search in planning and model checking. In *Proceedings of the Doctoral Consortium of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*.

Lea, D. 2006. The Java concurrency package. Online at http://gee.cs.oswego.edu/dl/concurrency-interest/index.html. Accessed March 19, 2006.

Lerda, F., and Sisto, R. 1999. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop*.

Rao, V. N., and Kumar, V. 1987. Parallel depth-first search on multiprocessors part I: Implementation. *International Journal of Parallel Programming* 16(6):479–499.

Yoo, A.; Chow, E.; Henderson, K.; and McLendon, W. 2005. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the ACM/IEEE Supercomputing 2005 Conference*, 25–35.

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