Lock-Free Multiway Search Trees as Priority Queues in Parallel Branch and Bound Applications

Michael Spiegel and Paul F. Reynolds, Jr.

February 7, 2011

Department of Computer Science
Technical Report CS-2011-01
School of Engineering and Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740

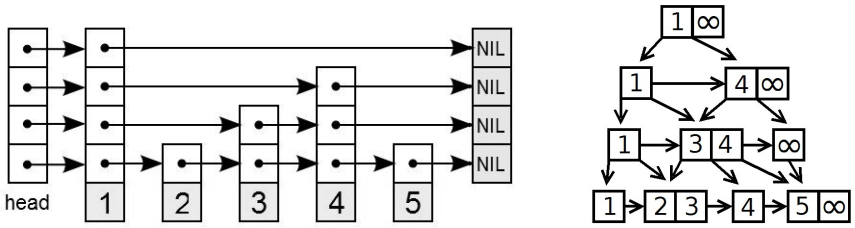
Abstract

The lock-free skip tree is a cache-conscious concurrent data structure for many-core systems that shows significant performance improvements over the state of the art in concurrent data structure designs for those applications that must contend with the deleterious effects of the memory wall. In a previous study using a series of synthetic benchmarks, the lock-free skip tree was found to improve peak throughput by x1.8 to x2.3 relative to a state of the art lock-free skip list implementation when the working set size exceeds cache size. In this work, we study a class of application benchmarks that can be used to characterize the relative merits of the lock-free skip tree as compared to the lock-free skip list. In a series of four parallel branch-and-bound applications, two of the applications are x2.3 and x3.1 faster when using the skip tree as a concurrent priority queue as compared to the lock-free skip list priority queue. On a shared-memory supercomputer architecture the two branch-and-bound applications are x1.6 and x2.1 faster with the skip tree versus the skip list running at 80 hardware threads. Based on the four application benchmarks and a synthetic branch-and-bound application, a set of guidelines is offered for selecting the lock-free skip tree to use as a centralized priority queue in parallel branch-and-bound applications. This technical report is an extended version of a manuscript of the same title that has been submitted for publication.

1 Introduction

Cache-conscious concurrent data structures are essential for scalability and performance on modern computer architectures. The need for cache-conscious concurrent algorithms stems from the constraints that are forcing a trend from implicitly parallel architectures towards explicitly parallel architectures. The three constraints driving this trend are the power wall, the instruction-level parallelism (ILP) wall, and the memory wall [1]. While the implications of the power wall and the ILP wall lie primarily in the hardware domain, the implications of the memory wall can be seen in the design of efficient algorithms for concurrent applications.

The lock-free skip tree is a cache-conscious concurrent data structure for many-core systems that shows significant performance improvements over the state of the art in concurrent data structure designs for those applications that must contend with the deleterious effects of the memory wall. In a series of synthetic benchmarks, the lock-free skip tree implementation performs up to x2.3 better in read-dominated workloads compared to the state of the art lock-free skip list implementation with only a 13% maximum penalty across all workloads [2].



(a) Skip list. (b) Lock-free skip tree.

Figure 1: Skip list and skip tree isomorphisms.

A class of applications has been selected to characterize the merits of the lock-free skip tree under conditions where the working set size exceeds cache size. Parallel branch and bound applications use a concurrent priority queue to conduct a best-first search through a set of candidate solutions. In the literature, a significant portion of papers that improve concurrent priority queue designs are published in the context of parallel branch-and-bound applications.

Four NP-hard problems have been selected to solve using a parallel branch-and-bound technique: N puzzle, graph coloring, asymmetric traveling salesman, and 0-1 knapsack. A synthetic branch and bound application is constructed to study the effects of three application characteristics on the skip tree priority queue: the distribution of solutions in the search space, the computation time of the lower bound, and the branching factor of the application. Based on the four applications and the synthetic benchmark, a set of guidelines is offered for selecting the lock-free skip tree to use as a centralized priority queue.

2 Lock-Free Skip Tree Algorithm

Many data structures have been designed with concurrent, lock-free, cache conscious, or randomized properties. The lock-free skip tree shares some structural properties with these data structures. Several related data structures are summarized in Table 1 and their properties of interest are enumerated.

The skip tree was introduced by Messeguer [10] as a generalization of the skip list for concurrent operations. A skip tree is a randomized multiway search tree such that each new element is assigned a height from a geometric distribution. Elements are stored in the leaves of the tree, and the interior nodes of the tree serve as partitions. We modify the skip tree structure in three primary aspects to generate the lock-free skip tree algorithm [2]. First, the path length invariant of the skip tree is eliminated. Second, 'link' pointers are introduced to allow nodes to split independently of their parent nodes. Third, the lock-free skip tree does not require that neighboring elements in the tree's interior serve as partitions on the tree. The algorithm maintains consistency by defining a *reachability* relation from the root of the tree to any potential element stored in the tree for all possible states of the tree. Optimal paths through the tree can be temporarily eliminated by deletion operations, and eventually restored using online node compaction. Figure 1a shows an example of a skip list and Figure 1b shows

Data Structure	Concurrent	Cache-conscious	Randomized	Sorted Order
Lock-free skip list [3, 4]	Y	N	Y	Y
Treap/Randomized search tree [5, 6]	N	N	Y	Y
Cache-oblivious B-tree [7]	Y	N	Y[a]	Y
Hopscotch hash table [8]	Y	Y	N	N
HAT-trie [9]	N	Y	N	Y
Lock-free skip tree [2]	Y	Y	Y	Y

[a] randomized version does not support deletions

Table 1: Comparison of Related Data Structures

the corresponding lock-free skip tree.

3 Application Benchmarks

In a previous paper, a series of synthetic benchmarks showed the lock-free skip tree outperforming the lock-free skip list in large working set sizes as measured by the peak number of operations per second as the quantity of concurrent threads is varied [2]. The disadvantage of synthetic operations is that they do not have any explicit meaning. As such, some caution must be exercised when projecting the synthetic benchmarks onto a specific application domain. In this technical report, we identify a class of problems that can be used to characterize the relative merits of the lock-free skip tree as compared to the lock-free skip list. We selected four NP-hard problems to solve using a parallel branch-and-bound technique: N puzzle, graph coloring, asymmetric traveling salesman, and 0-1 knapsack. In a series of four parallel branch-and-bound applications, two of the applications are x2.3 and x3.1 faster when using the skip tree as a concurrent priority queue as compared to the lock-free skip list priority queue. In a shared-memory supercomputer architecture the two branch-and-bound applications are x1.6 and x2.1 faster with the skip tree versus the skip list running at 80 hardware threads. Based on the four application benchmarks and the synthetic benchmark, a set of guidelines are determined for selecting the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application.

We study the relative performance of a parallel branch-and-bound solver on four NP-hard problems using a lock-free skip list and lock-free skip tree as concurrent priority queues. The four benchmarks are evaluated on a Sun Fire T1000 and an Intel Xeon L5430. The Sun Fire has 8 UltraSPARC T1 cores at 1.0 GHz and 32 logical processors. The cores share a 3 MB level-2 unified cache. The operating system version on the Sun Fire T1000 is Solaris 10. The Xeon L5430 has 4 cores at 2.66 GHz and 8 logical processors. Each pair of cores shares a 6 MB level-2 unified cache. The operating system distribution is CentOS release 5.3 with Linux kernel 2.6.29-2. The Sun Fire T1000 benchmarks are run using the 64-bit build of the HotSpot Java virtual machine version 1.6.0_18 with command-line arguments "-Xmx7G -XX:+UseParallelGC -XX:+UseParallelOldGC". The Intel Xeon benchmarks are run using the 64-bit build of the HotSpot Java virtual machine version 1.6.0_20 with command-line arguments "-Xmx43G -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:ParallelGCThreads=8."

To evaluate the relative performance of the lock-free skip tree as a concurrent priority queue, we have created a synthetic branch-and-bound application. The synthetic application is a simplification of a real branch-and-bound application. It has been designed to test three hypotheses of the branch-and-bound applications: (1) the distribution of lower bounds of the candidates in the search space affects the performance of the skip tree; (2) the computation time of the lower bound affects the performance of the skip tree; and (3) the branching factor of the application affects the performance of the skip tree. Based on the four application benchmarks and the synthetic benchmark, we provide a set of guidelines for selecting the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application versus the lock-free skip list. Finally, at the conclusion of this technical report we present results from analyzing the parallel branch-and-bound benchmarks on two shared-memory supercomputer architectures.

4 Candidate Application Benchmarks

In selecting the class of problems for this report, we were looking for a set of concurrent algorithms that relies heavily on a linearizable data structure that preserves the sorted set or sorted map abstraction. Our first choice was the Apache Hadoop implementation of Google's MapReduce framework. MapReduce is a programming model for processing and generating large datasets. Users specify the computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks [11]. The MapReduce specification guarantees that within each of the reduce partitions, the intermediate key/value pairs are processed in sorted order. One possible implementation of this guarantee is to

use a ConcurrentSkipListMap to collect key/value pairs as they are generated in the map phase. But we discovered that the Apache Hadoop implementation does not use a ConcurrentSkipListMap or ConcurrentSkipListSet.

Our second choice was the Apache Cassandra distributed database project [12]. Apache Cassandra uses a multidimensional key/value data model. A column represents a single key/value pair. Columns are grouped together into sets called column families. A set of column families can be grouped together into a super column family. Applications can specify the sort order of columns within a column family or a super column family. Column families and super column families are implemented using ConcurrentSkipListMaps. We replaced the ConcurrentSkipListMap instances in Cassandra 0.6.0 with ConcurrentSkipTreeMap instances. The performance of the two data structures was compared using the stress.py tool in the Cassandra source distribution. stress.py is intended for benchmarking and load testing a Cassandra cluster. At first we found no difference in performance using the skip list or the skip tree. Some investigation revealed that column families are kept at relatively small sizes in memory before they are stored persistently on disk. After modifying the max column family size parameter setting, again no performance difference was exhibited between the skip list and the skip tree. A runtime profile of the Cassandra server during the execution of stress.py shows that the most time-consuming work phases are disk I/O on the server and network I/O between the client and server.

Our experiences with Apache Hadoop and Apache Cassandra led us to revise the prerequisites for the application benchmarks to focus on a class of problems that are solved with a single algorithmic technique that does not rely on tuning configuration parameters to maximize I/O bandwidth and minimize I/O latency. We settled on NP-hard problems that can be solved using a parallel branch and bound algorithm. Parallel branch and bound algorithms can be implemented using a concurrent priority queue. In the literature, a significant portion of papers that improve concurrent priority queue designs are published in the context of parallel branch-and-bound applications. We selected four NP-hard problems to solve using a parallel branch-and-bound technique: N puzzle, graph coloring, asymmetric traveling salesman, and 0-1 knapsack.

For each of the NP-hard problems, we selected an algorithm for estimating the lower bound of each candidate solution that had been published as the best solution at some point in time, and is used in the current literature as a reference strategy against which novel approaches are measured. Many improvements to the lower bound estimate are made by improving the set of heuristics which can be applied to the candidate solution [13]. The scope of this report is the lock-free data structures and not to focus on the heuristic techniques. A sophisticated heuristic can reduce search space for a branch-and-bound application. For our purposes, the advanced heuristics lead to a change in the space of initial configurations that have large working set sizes that can be contained within the memory available on the machine. In acknowledging the shift in configurations that have large working set sizes, the critical observation is that a non-empty set of configurations exists that can benefit from a cache-conscious priority queue, regardless of the technique used to estimate the lower bounds.

5 Parallel Branch-and-Bound Algorithms

Branch and bound is an algorithmic technique for finding optimal solutions of optimization problems. Given a finite set of candidate solutions X and an objective function $f(X) \rightarrow \mathbb{R}$, an optimal solution $x^\star \in X$ exists such that $f(x^\star) = \min\{f(x)|x \in X\}$. The goal of the branch and bound technique is to systematically traverse the space of candidate solutions in order to find an optimal solution. It is assumed that a splitting procedure exists that accepts an input set of candidates S and returns two or more smaller sets $S_1, S_2, \ldots, S_n$ whose union covers S. Each child of the input set is typically distinguished by the imposition of one or more constraints upon S to select the subset of candidates S_i. The second assumption of the branch and bound algorithm is that there exists a method for estimating the lower and upper bounds of the objective function values in a set of candidate solutions. The branch and bound method was first proposed by [14] for solving integer linear programming problems at British Petroleum [15].

The bounding procedure takes advantage of the fact that if the lower bound for some set A of can-

didate solutions is greater than the upper bound of another set B, then A may be discarded from the search space. In practice, the bounding procedure is implemented with global state that keeps track of the current best solution that has been identified so far. The branch and bound algorithms studied in this section perform a best-first search. Candidate solutions are searched in increasing order of their lower bound. Best-first selection is optimal with respect to the number of decomposed subproblems, with no ties occurring among lower bounds, and the branching and bounding operations do not depend on previous history [16]. We have selected the best-first search strategy to analyze the performance of the lock-free skip tree versus the lock-free skip list when used as a concurrent priority queue. The disadvantage of a best-first search is the memory requirement for storing all unexplored candidates in the search tree until an optimal solution is discovered. If an optimal solution can be discovered before running out of memory, then it is found relatively quickly as compared to a depth-first search.

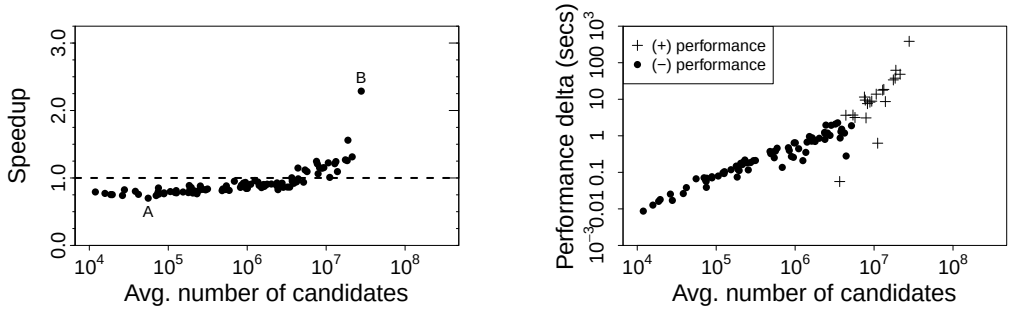
[17] identify three main approaches to designing parallel branch and bound algorithms. The first approach uses a sequential search strategy, but improves the performance of calculating the lower bound estimate with a concurrent implementation of this calculation. The second approach builds a single centralized search tree with multiple worker threads concurrently extracting candidate solutions from the tree and inserting children solutions into the tree. The third approach builds several search trees in parallel with some method of coordination among the search trees to balance the workload across all trees. The second approach has been the most popular approach to designing a parallel branch and bound application [18, 19, 20, 21, 22].

The java.util.concurrent ConcurrentSkipListSet is a lock-free implementation of a quiescent priority queue backed by a skip list data structure [4]. Quiescent consistency is defined such that the operations of any processors separated by a period of quiescence should appear to take effect in their real-time order. A linearizable priority queue would prohibit the following situation from occurring: Thread 1 inserts element x into the priority queue and then inserts element y into the priority queue, such that $y < x$. Concurrent with both insert operations, Thread 2 is performing an extract minimum operation. Thread 2 returns the value x. In order to enforce linearizability, one strategy is to redefine the extract minimum operation to select from elements that have completed insertion before the beginning of the extract min operation. This strategy can be implemented with timestamps per each node that note the logical time of insertion into the queue [23]. The quiescent priority queue is sufficient for branch-and-bound applications, where the only necessary requirement is that when $y < x$, then y is extracted from the queue before x, given a period of quiescence has expired.

6 N puzzle

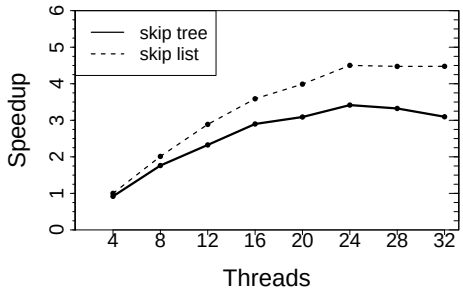
The N puzzle is a game played on an $m \times m$ grid containing n square tiles, where $n = m^2 - 1$. Each tile is assigned a unique number from $1 \ldots n$. One tile in the grid is always empty, called the 'blank' tile, and depending on its position, the blank has two, three, or four adjacent tiles. A move in the game swaps the positions of the blank tile and a single neighboring tile. The tiles are initially set to some random configuration. The goal is to rearrange the tiles into a specific goal configuration. Typically the goal configuration consists of the tiles in row-wise consecutive order, with the blank tile designated to appear either first or last in the sequence. The initial board configurations with the largest improvement of the lock-free skip tree as compared to the lock-free skip list exhibits a x2.1 runtime improvement on the Sun Fire and a x2.4 improvement on the Intel Xeon. [24] provides more information concerning the history of the N puzzle problem. An excellent survey of NP-complete single-player games has been written by [25].

Half of the starting configurations of the N puzzle cannot reach the goal state [26]. A parity argument can be made using the parity of permutations plus the parity of the Manhattan distance moved by the blank tile. The Eight Puzzle contains $\frac{9!}{2}$ or $181,440$ possible configurations. The solutions for all legal starting configurations of the Eight Puzzle have been solved using brute force [27]. The average length of a shortest path between two states is about 22 moves, and the maximum distance between any pair of states is 31 moves. The Fifteen Puzzle contains $\frac{16!}{2}$ or over 10^{13} possible states. The Fifteen

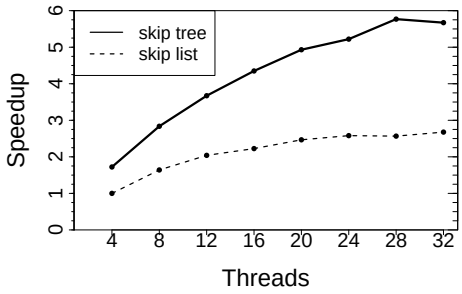


(a) relative speedup of lock-free skip tree

(b) absolute speedup of lock-free skip tree



(c) problem 'A' instance



(d) problem 'B' instance

Figure 2: Fifteen Puzzle on Sun Fire T1000

Puzzle is used as the NP-complete example in the publication that introduced the Iterative-Deepening A* (IDA*) algorithm [28]. One hundred randomly generated Fifteen Puzzle instances are used in the IDA* paper and their solutions are included in the publication. The average optimal solution length of these one hundred instances is about fifty-three moves. The Iterative-Deepening A* algorithm performs a depth-first search. We show that a parallel branch-and-bound solver can solve all one hundred instances presented in the IDA* paper using a breadth-first search.

The lower bound calculation used as a baseline for the parallel branch-and-bound solver is the sum of the Manhattan distances of each tile's current position relative to the tile's goal position. Two admissible heuristics are used to improve the lower bound: the linear-conflict heuristic and the last moves heuristic. Each heuristic can improve upon the lower bound when a specific set of conditions mandates additional state transitions beyond those accounted for in the Manhattan distance lower bound. Additional care must be taken to ensure that multiple admissible heuristics do not interact unfavorably to produce a lower bound estimate that is not admissible (ie. never overestimates the lower bound). The best upper bound on the N puzzle problem is independent of the initial board state. The upper bound serves no useful purpose to limit the possible search space. Assuming that N is a fixed value for the N puzzle problem, calculation of the lower bound is $O(1)$.

The linear-conflict heuristic can increase the lower bound estimation when two tiles are in their goal row or column, but are reversed relative to their goal positions [29]. For example, if the top row of the puzzle contains the tiles $(2\,1)$ in that order, then one of the tiles must move down into the next row, in order to allow the other tile to move into the correct position. The last moves heuristic exploits the necessity of the blank tile to be the last tile that reaches the goal position, which is the upper-left corner in this case. The last move must either move the 1 tile to the right, or the 4 tile down. If the 1 tile is not in the left-most column, and the 4 tile is not in the top row, then two moves can be added to the Manhattan distance and still preserve admissibility [30]. If either the 1 tile or the 4 tile is in a linear-conflict state, then the last moves heuristic cannot be applied.

Figures 2a and 2b show the relative speedup of the parallel branch-and-bound solver applied to

one hundred instances of the Fifteen Puzzle executed on a Sun Fire T1000. The basis for comparison in these graphs is the runtime performance of the solver when the java.util.concurrent lock-free skip list is used as a priority queue. The vertical axis represents the relative speedup of the solver when the lock-free skip tree algorithm is used. The horizontal axis represents the average number of partial solutions generated in the search space until the goal state was reached. Worker threads in the parallel branch-and-bound algorithm will not process partial solutions in the same order across repeated trials, and so therefore the average of the number of partial solutions is reported.

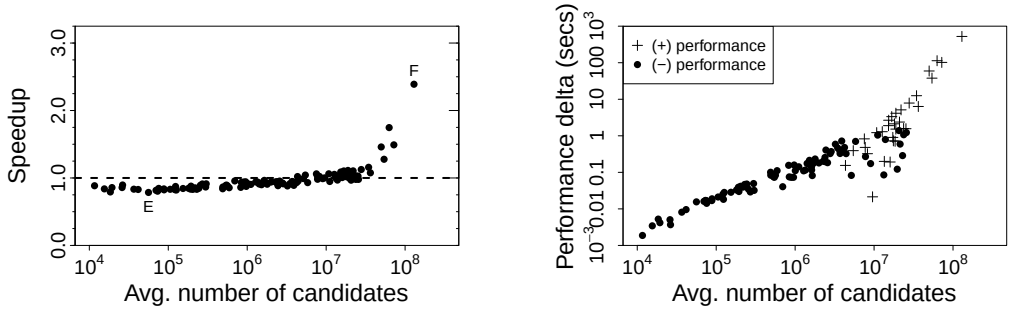
Instances of the Fifteen Puzzle with a relatively small working set appear on the left side of the graphs, and instances with a relatively large working set appear on the right side of the graphs. Figure 2a shows the relative speedup of the skip tree versus the skip list. For initial board configurations with a relatively small working set size, the skip list outperforms the skip tree. For initial board configurations with the largest working set sizes that could be contained in memory on this machine, the skip tree outperforms the skip list by as much as x2.3. Figure 2b shows the absolute difference in the runtime of the skip tree and the skip list. The vertical axis denotes the log of the execution time with the skip list subtracted from the execution time with the skip tree. Skip tree performance improvements are plotted with the '$+$' symbol and skip tree performance declines are plotted with the '$\bullet$' symbol. The absolute performance graphs illustrate the trend that an initial board configuration with a small working set size will execute in less time than an initial board configuration with a large working set size. As a corollary, any relative performance deficiencies of the skip tree at small working set sizes correspond to a small performance penalty in absolute units.

The initial board configurations with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'A' and 'B' in Figure 2a. The speedup of configurations 'A' and 'B' relative to the number of available processors is shown in Figures 2c and 2d. All other graphs are shown utilizing the maximum number of available processors. Initial board 'A' is solved in 53 moves with an average of 5.8×10^4 partial solutions generated in the search space. Initial board 'B' is solved in 55 moves with an average of 2.8×10^7 partial solutions generated in the search space. There is no speedup of the skip tree relative to the skip list in board configuration 'A' when running on a single core (4 threads). As the thread count increases, the application speedup is smaller using the skip tree as compared to using the skip list. Initial board 'B' on a single core is solved using the skip tree with a speedup of x1.7 as compared to the skip list. Running on all eight cores, the speedup of the skip list relative to a single core is x2.7 and the speedup of the skip tree relative to the skip list on a single core is x5.7.

Performance results of the Fifteen Puzzle solver on a quad core Intel Xeon are shown in Figure 3. There are more initial board configurations in these results as compared to the Sun Fire T1000, as this Intel Xeon had more memory available, implying that configurations with a larger working set size could be solved. The relative speedup of the skip tree to the skip list follows a similar trend on the Intel Xeon benchmarks as seen earlier on the Sun Fire benchmarks. The initial board configurations with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'E' and 'F'. There is no speedup of the skip tree relative to the skip list in board configuration 'E' when running on a single thread. Initial board 'F' on a single core is solved using the skip tree with a speedup of x1.7 as compared to the skip list. Running with eight threads, the speedup of the skip list relative to a single thread is x7.3 and the speedup of the skip tree relative to the skip list on a single thread is x17.8.

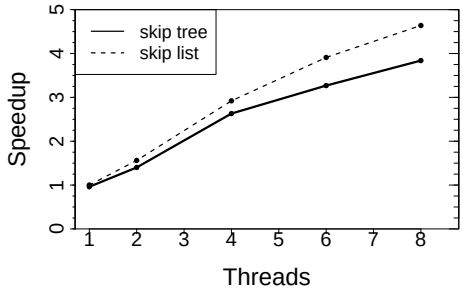
7 Graph coloring

A graph G is an ordered pair $G = (V, E)$, where V is a finite set of vertices, and E is a finite set of unordered pairs of vertices representing edges. A legal vertex-coloring of graph $G = (V, E)$ is a function $c : V \to \mathbb{N}$, in which any two incident vertices $u, v \in V$ are assigned different colors, meaning $\{u, v\} \in E \implies c(u) \neq c(v)$. The function c is called the coloring function. A graph G for which there exists a vertex-coloring which requires k colors is called k-colorable. The smallest number k for which there exists a k-coloring of graph G is called the chromatic number of graph G and is denoted by $\chi(G)$ [31]. The initial board configurations with the largest improvement of the lock-free skip tree

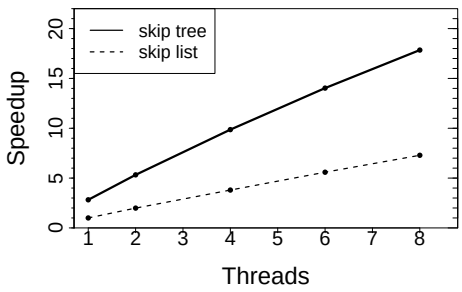


(a) relative speedup of lock-free skip tree

(b) absolute speedup of lock-free skip tree



(c) problem 'E' instance



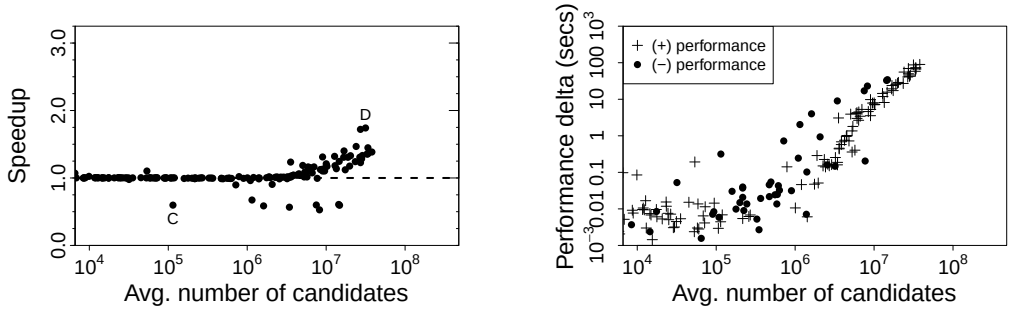
(d) problem 'F' instance

Figure 3: Fifteen Puzzle on quad core Intel Xeon

as compared to the lock-free skip list exhibits a x1.5 runtime improvement on the Sun Fire and a x3.1 improvement on the Intel Xeon.

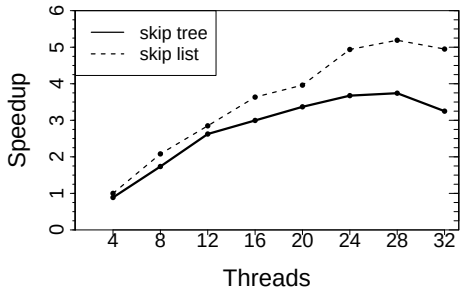
The saturation largest-first (SLF) algorithm, also known as DSATUR algorithm, solves the graph coloring problem by iteratively selecting the uncolored vertex v with the highest saturation degree in G and then branching on the legal colorings of vertex v [32]. The saturation degree of a vertex v in a partially colored graph G is defined as the number of distinctly colored vertices adjacent to v. The DSATUR algorithm has been considered a "*de facto* standard among exact graph coloring algorithms" [33]. An initial upper bound is estimated using a greedy iterative variation of the SLF algorithm, whereby the smallest possible color is assigned to uncolored vertex v with the highest saturation degree at each iteration. In our implementation, each vertex v builds a temporary bit vector that marks the distinct colors adjacent to v. The cost of building the bit vector entries requires traversing over all the edges of the uncolored vertices, which is $O(E)$. Selecting the uncolored vertex v with the highest saturation degree in G entails finding the max value of the bit vector entries which is $O(V)$. Combining these steps yields a worst-case running time of computing of the lower bound to be $O(E) + O(V)$.

The runtime performance of the graph coloring solver is tested with a set of randomly generated graphs. $G_{n,p}$ is a random graph with vertices $\{1, \ldots, n\}$ such that the probability of an edge between any two vertices is $0 < p < 1$, and the edge probability is independent of all other edges. These random graphs are commonly used as benchmarks in graph coloring publications [34, 35, 36]. In our experiments, n was selected from the set $\{10 \cdot i : i \in \mathbb{Z}, 5 \leq i \leq 15\}$ and p was selected from the set $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. Ten random graphs were generated for each possible combination of values for n and p, yielding 550 test cases. Performance results of the graph coloring solver on a Sun Fire T1000 are shown in Figure 4. Graph instances with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'C' and 'D' in Figure 4a. Graph instance 'C' was generated using $n = 70$ and $p = 0.3$. It has a chromatic number of 7 with average of 1.1×10^5 partial solutions generated in the search space. Graph instance 'D' was generated using $n = 90$ and $p = 0.3$. It has a chromatic number of 9 with an average of 3.2×10^7 partial solutions generated in the

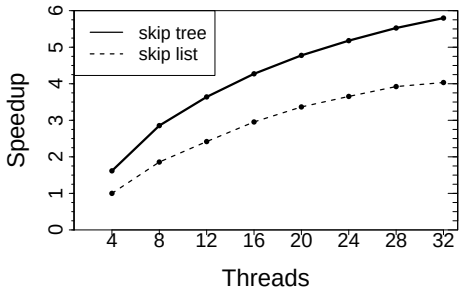


(a) relative speedup of lock-free skip tree

(b) absolute speedup of lock-free skip tree



(c) problem 'C' instance



(d) problem 'D' instance

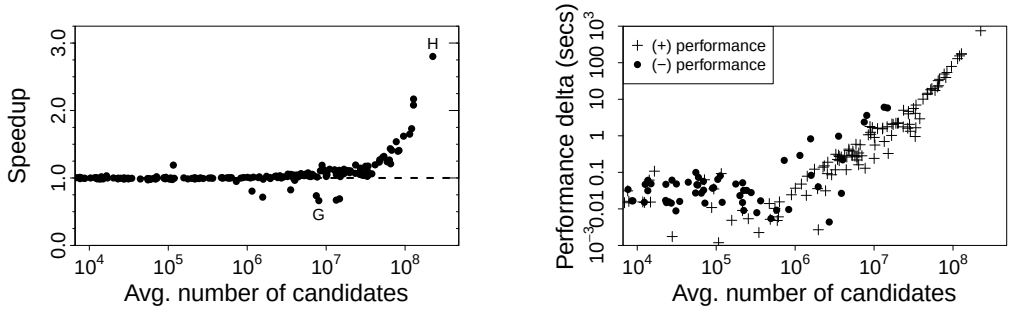
Figure 4: Graph coloring on Sun Fire T1000

search space. There is no speedup of the skip tree relative to the skip list in graph instance 'C' when running on a single core (4 threads). As the thread count increases, the application speedup is smaller using the skip tree as compared to using the skip list. Graph instance 'D' on a single core is solved using the skip tree with a speedup of x1.6 as compared to the skip list. Running on all eight cores, the speedup of the skip list relative to a single core is x4.0 and the speedup of the skip tree relative to the skip list on a single core is x5.8.

Performance results of the graph coloring solver on a quad core Intel Xeon are shown in Figure 5. There are more graph instances in these results as compared to the Sun Fire T1000, as this Intel Xeon had more memory available, implying that configurations with a larger working set size could be solved. The relative speedup of the skip tree to the skip list follows a similar trend on the Intel Xeon benchmarks as seen earlier on the Sun Fire benchmarks. The initial board configurations with the largest performance penalty and largest performance improvement of the skip tree have been assigned the labels 'G' and 'H'. There is no speedup of the skip tree relative to the skip list in board configuration 'G' when running on a single thread. Initial board 'H' on a single core is solved using the skip tree with a speedup of x3.0 as compared to the skip list. Running with eight threads, the speedup of the skip list relative to a single thread is x6.6 and the speedup of the skip tree relative to the skip list on a single thread is x20.3.

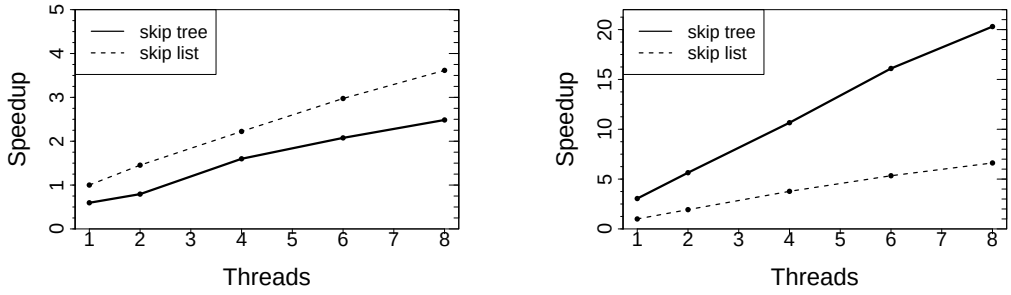
8 Asymmetric Traveling Salesman Problem

The asymmetric Traveling Salesman Problem (ATSP) begins with a directed graph G of vertices V and edges E. Each edge is assigned a weight, w_{ij}, such that the weight from v_i to v_j is not necessarily equal to the weight from v_j to v_i. The objective of ATSP is to find a shortest Hamiltonian path, defined as a path that visits each vertex exactly once and has the smallest sum of all traversed edge weights [37, 38]. Let the cost of the shortest Hamiltonian path for graph G be designated as $ATSP(G)$. In all instances,



(a) relative speedup of lock-free skip tree

(b) absolute speedup of lock-free skip tree



(c) problem 'G' instance

(d) problem 'H' instance

Figure 5: Graph coloring on quad core Intel Xeon

there was no runtime improvement of the solver when using the skip tree versus the skip list.

When a new branch-and-bound algorithm for solving ATSP is published, the Held-Karp lower bound is the most common lower bound that is used as a basis of comparison [39]. The Held-Karp lower bound on optimal tour length is constructed from a relaxation of the constraints imposed by ATSP [40, 41]. The ATSP imposes a constraint on the optimal tour such that each vertex must have an indegree of one, and the constraint that each vertex must have an outdegree of one. If the outdegree constraint is eliminated and some arbitrary node is designated as a root node, then an optimal minimum spanning tree (MST) can be generated. The optimum minimum spanning tree for graph G is designated as $MST(G)$. The directed MST problem can be solved by the Chu-Liu/Edmonds algorithm [42, 43, 44] which is a polynomial time algorithm. Recall that the directed MST was generated by a relaxation of the constraints for ATSP. Therefore the cost of the directed MST is less than or equal to the cost of the optimal Hamiltonian path:

$$MST(G) \leq ATSP(G) \tag{1}$$

If each vertex in the directed minimum spanning tree has an outdegree of one, then the shortest Hamiltonian path has been discovered. Otherwise, an iterative procedure is used to transform the weights of the graph in order to encourage the directed MST to become a Hamiltonian path. Let $\pi(v)$ be any function that maps a vertex v to some real number. Let the transformed weights be defined as $w_{ij}^{\pi} = w_{ij} - \pi(i)$. In the Hamiltonian path each vertex must have an outdegree of 1. This implies that the Hamiltonian path is affected by the weight transformation exactly once per vertex:

$$ATSP(G^{\pi}) = ATSP(G) - \sum(\pi(v) \ : \ v \in V) \tag{2}$$

Combining equations 1 and 2 yields the observation that the transformed directed MST is always a lower bound on the optimal Hamiltonian path:

$$max_\pi \left(MST(G^\pi) + \sum (\pi(v) \ : \ v \in V) \right) \leq ATSP(G) \tag{3}$$

In order to derive a good lower bound on the optimal Hamiltonian path, $\pi(v)$ is assigned some negative value when the outdegree of v is greater than one and a positive value when the outdegree is less than one. If $deg^+(v)$ is the outdegree of v, then $\pi(v) = k\,(1 - deg^+(v))$ where k is an arbitrary constant. In practice, the Held-Karp process is not run long enough to converge on the optimal Hamiltonian path. However, after only a few iterations of the Held-Karp process, a sufficiently accurate lower bound estimate can be calculated.

The TSPLIB library was used for input graphs to test the performance of the ATSP solver. TSPLIB is a collection of graphs along with their solutions for the Traveling Salesman Problem and related problems [45]. The collection of graphs for the asymmetric traveling salesman problem come from a variety of real-world scheduling and resource management applications [39]. The instances of the TSPLIB that could be tested by our branch-and-bound solver contained between 17 and 171 cities. In all instances, there was no runtime improvement of the solver when using the skip tree versus the skip list. In Section 10 we will show that the ATSP solver is a compute-bound problem, which is to say that over 99% of the total runtime is spent computing the lower bounds of the partial solutions.

9 0-1 Knapsack

The 0-1 knapsack problem [46], hereafter referred to as the knapsack problem, is defined using a set of n items, such that each item j is assigned a profit p_j and a weight w_j. The knapsack has a capacity c. A set of binary decision variables $\{x_1, \ldots, x_n\}$ are defined such that item j is placed into the knapsack if $x_j = 1$, and item j is not placed in the knapsack if $x_j = 0$. The objective of the knapsack problem is to maximize $\sum_{j=1}^{n} p_j x_j$ subject to the constraint $\sum_{j=1}^{n} w_j x_j \leq c$. Let the efficiency of item j be defined as the ratio of profit to weight. Assume the items are sorted by their efficiency in decreasing order such that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \ldots \geq \frac{p_n}{w_n}$. A greedy solution to the knapsack problem is to select the largest consecutive sequence of most-efficient items that will fit in the knapsack. The first item s that cannot fit into the knapsack is referred to as the split item. The solution vector $\hat{x}$ with $\hat{x}_j = 1$ for $j = 1, \ldots, s-1$ and $\hat{x}_j = 0$ for $j = s, \ldots, n$ is known as the split solution. There was no runtime improvement of the solver when using the skip tree versus the skip list for all test cases.

A branch-and-bound algorithm commonly used as a basis for comparison on the knapsack problem is the primal-dual algorithm [47]. This algorithm is based on the observation that for many instances of the knapsack problem only a relatively small subset of items is crucial for the determination of the optimal solution. Items with very high efficiencies will almost certainly be included in the optimal solution. Items with very low efficiencies will almost certainly be excluded from the optimal solution. What remains is the set of items of "medium efficiency", referred to as the core set of items. The core set begins with the split item and expands outwards in both directions. That is, the core set of items expands to the "left" to include more efficient items, and expands to the "right" to include less efficient items. In the primal-dual algorithm, the binary decision variables $x_{a+1}, \ldots, x_{b-1}$ are fixed at some value and it is assumed that $x_j = 1$ for $j \leq a$ and $x_k = 0$ for $k \geq b$. The profit and weight sums of the current configuration are denoted $\bar{p}$ and $\bar{w}$, respectively. If $\bar{w} \leq c$ an item is inserted into the knapsack and otherwise an item is removed from the knapsack.

The primal-dual algorithm has a branching factor of 2. Given a configuration of binary decision variables, assume that $\bar{w} \leq c$, then one child configuration is generated by assigning $x_b = 0$ and another configuration is generated by assigning $x_b = 1$. In either case, the right boundary b is incremented, $b' = b + 1$. In a similar fashion, if $\bar{w} > c$ then one configuration is generated by assigning $x_a = 0$ and another is generated by assigning $x_a = 1$. The left boundary of these children is decremented, $a' = a - 1$. Suboptimal solutions are eliminated using an upper bound estimate that is based on either the most efficient non-fixed item in the case of an underfilled knapsack: $\frac{(c - \bar{w})\, p_b}{w_b}$, or the least

efficient item in the knapsack in the case of an overfilled knapsack: $\frac{(c - \bar{w})\, p_y}{w_y}$ such that y is the max where $x_y = 1$.

There are three types of random distributions that are commonly used to generate test instances of the knapsack problem [47]. Each distribution assumes a data range R for the profit and weight values and a problem size n for the total number of items. The uncorrelated distribution assumes that p_j and w_j are randomly distributed in $[1, R]$. The weakly correlated distribution assumes that w_j is randomly distributed in $[1, R]$ and p_j is randomly distributed in $\left[w_j - \frac{1}{10}R, w_j + \frac{1}{10}R\right]$. The term "weakly correlated" is somewhat misleading, as weakly correlated instances have a very high correlation between the profit and weight of an item. The strongly correlated distribution assumes that w_j is randomly distributed in $[1, R]$ and $p_j = w_j + 10$. In all instances, there was no runtime improvement of the solver when using the skip tree versus the skip list. In the next section we show that the knapsack solver is a communication-bound problem. The cost of extracting the minimum element and inserting new elements into the priority queue dominates over the cost of computing the next child configuration.

10 Performance Analysis

The purpose of this section is to help understand why the lock-free skip tree shows an improvement of up to x2.4 and x3.1 on the 15 puzzle and graph coloring applications, but did not yield any improvements on the asymmetric traveling salesman problem and the 0-1 knapsack problem. Determining the reasons behind the performance differences will help us construct of series of properties that are necessary for a branch-and-bound application to prefer a cache-conscious priority queue (Section 12). The first step is to identify the computational phases that are common to all four applications and then study the application differences in behavior across these common phases. The parallel branch-and-bound algorithm consists of three primary phases. These are: (1) extract the minimum element from the shared priority queue, (2) generate several children candidate solutions based on the current candidate solution, and (3) if the bounding criterion has not been reached, then insert each new child candidate solution into the shared priority queue. Extracting the minimum element is an $O(1)$ operation in the absence of thread contention. Inserting a new candidate solution into the shared priority queue is an $O(\log S)$ operation in the absence of thread contention, where S is the number of candidate solutions currently stored in the priority queue. In general, it is difficult to estimate a tight upper bound on S for a specific input instance to a parallel branch-and-bound solver.

To generate a child candidate solution, the full state of the parent solution must be regenerated based on the partial state information stored in the parent solution (explained in Section 5), and then the lower and upper bounds for the child candidate solution must be computed. For the 15 puzzle, generating the upper bound estimate is $O(1)$. For the graph coloring problem, computing the lower bound of the chromatic number is $O(E) + O(V)$, where E is the number of edges and V is the number of vertices in the graph. For the asymmetric traveling salesman problem, our directed minimum spanning tree solver uses the Bock adjacency matrix representation [44] rather than the Chu–Liu/Edmonds graph representation [42, 43] to avoid recreation of the graph at every partial solution. Calculating the upper bound for the ATSP using the Bock representation has a worst case running time of $O(EV)$. The 0-1 knapsack problem calculates the upper bound of the profit on a candidate solution in $O(1)$ time for underfilled knapsacks and $O(N)$ time for overfilled knapsacks, where N is the number of items from which selection is performed.

Tables 2 and 3 show the runtime percentage of each phase of computation for each of the four applications on the Niagara and quad core Xeon processors. Each row shows a percentage that is relative to the total runtime of that problem instance using the skip list data structure. The proportions in each skip list row sum to one hundred, and the results in each skip tree row are some fraction relative to the appropriate skip list row. The fourth column in the table records the total number of candidate solutions explored in the search space. This column is used as a surrogate for the maximum value of S, the number of candidate solutions currently stored in the priority queue. The total number of candidates is used as a proxy measure for the size of the queue. Estimating the number of elements

stored in a lock-free data structure at any point in time is an impractical exercise. The input instances for the 15 puzzle and graph coloring problems are specified in Figures 2a and 4a.

Based on these results, the computational phase that is responsible for the majority of the performance differences between the skip list and the skip tree is the insertion of candidate solutions into the shared priority queue. For input problems with a large total number of candidate solutions explored in the search space, the proportion of time spent on inserting into the queue is approximately halved. On the Niagara architecture, this proportion for instance 'B' of the 15 puzzle problem moves from 89% for the skip list to 38% for the skip tree and from 62% to 28% for instance 'D' of the graph coloring problem. For the instances of the 15 puzzle and graph coloring problems that showed a relative loss in performance using the skip tree versus the skip list, the insertion of candidate solutions is the culprit for this performance penalty. A verification that the insertion of candidate solutions is the bottleneck for small working set sizes is shown below. The distribution of retries for insertion and extraction operations on the queue are measured. For the small problem instances, insertion operations on the skip tree require more attempts in order to succeed as compared to extraction operations.

The next issue to address is the absence of a performance improvement when using the skip tree over the skip list on the asymmetric traveling salesman problem and the 0-1 knapsack problem. The evidence suggests that the working set size of these algorithms exceeds the cache size for the specific problem instances that are evaluated in Table 2. The total number of candidate solutions explored in the search space for the ATSP and 0-1 knapsack problems are $1.0 \cdot 10^6$ and $2.3 \cdot 10^7$, respectively. The total number of candidate solutions explored in the search space for the large instances of the 15 puzzle and graph coloring problems are $2.8 \cdot 10^7$ and $3.2 \cdot 10^7$. The equal runtime of the asymmetric traveling salesman problem solver is explained by the computational phase that calculates the lower bound estimate for each partial solution. Over 99% of the total runtime is spent in this phase for the ATSP solver. This leads to the first lesson in our characterization of parallel branch-and-bound applications in order to show benefit from a cache-conscious data structure: compute bound applications will not show speedup as a consequence of Amdahl's Law.

Table 4 shows the elapsed time per operation on the Sun Fire T1000. In general, the extract minimum operation is the fastest operation. The elapsed time of element insertion is proportional to the total number of candidate solutions explored in the search space. Calculating the upper bound for instances of the traveling salesman problem is two orders of magnitude longer than the bound calculations for the other three applications. The 0-1 knapsack problem exhibits a relatively long elapsed time for the extract minimum operation as compared to the other three applications. For both the skip list and the skip tree, the average time to completion is 21.6 and 31.6 microseconds. The geometric means for the skip list and the skip tree of the other five problem instances are 4.64 and 2.98 microseconds. Why should the extract minimum operation take longer on the 0-1 knapsack problem as compared to the other five problem instances? This is especially puzzling as we have established that the extract minimum operation has $O(1)$ cost in the absence of thread contention. We have established that these delays are due to thread contention. To demonstrate this hypothesis, we conducted a series of tests on the number of retries for the concurrent priority queue in all problem instances, except for the asymmetric traveling salesman problem as we have explained the behavior of that system. The next series of experiments measured the distributions of retries for the extract minimum and insert element operations on the priority queue.

In both the skip list and the skip tree, the failure of a compare-and-swap on an extract minimum operation mandates restarting the extract min operation from the head of the data structure. To measure the number of retries on an insert element operation, we measured the attempts only for the leaf levels of the skip list and the skip tree. When a compare-and-swap is unsuccessful on element insertion in the skip tree, a localized search can proceed to find the appropriate node for insertion. The skip list performs several consistency checks before an element is inserted at the leaf level of the list. If one of the consistency checks is unsuccessful, then the data structure is re-traversed from the root node to the leaves. In addition, if a compare-and-swap operation is unsuccessful then another compare-and-swap attempt will be made. We have separated the measurements for the number of re-traversals and the number of compare-and-swap attempts on a skip list insert operation.

In the measurements for the distribution of operation retries, the insert element operation is not a performance bottleneck in the 0-1 knapsack application (see Table 5). Almost 100% of the element

Application (input)	Extract min	Insert queue	Computation	Number of candidates
	— skip list —			
	— skip tree —			
15 puzzle (A)	4.8 ± 0.34	$44. \pm 2.4$	$51. \pm 1.5$	$5.9 \cdot 10^4 \pm 1.7 \cdot 10^3$
	4.3 ± 0.27	$76. \pm 6.2$	$52. \pm 3.6$	$5.7 \cdot 10^4 \pm 4.3 \cdot 10^3$
15 puzzle (B)	1.1 ± 0.23	$89. \pm 4.8$	9.7 ± 0.36	$2.8 \cdot 10^7 \pm 2.2 \cdot 10^3$
	0.49 ± 0.075	$38. \pm 8.3$	7.2 ± 0.18	$2.8 \cdot 10^7 \pm 1.8 \cdot 10^3$
Graph color (C)	2.7 ± 0.65	8.5 ± 1.6	$89. \pm 11.$	$1.2 \cdot 10^5 \pm 2.2 \cdot 10^4$
	5.8 ± 3.3	$75. \pm 21.$	$83. \pm 9.4$	$1.1 \cdot 10^5 \pm 2.4 \cdot 10^4$
Graph color (D)	0.95 ± 0.10	$62. \pm 2.4$	$37. \pm 0.37$	$3.2 \cdot 10^7 \pm 6.4 \cdot 10^2$
	0.54 ± 0.15	$28. \pm 2.4$	$42. \pm 0.085$	$3.2 \cdot 10^7 \pm 5.6 \cdot 10^2$
Asymmetric TSP	$0.0087 \pm 8.4 \cdot 10^{-5}$	0.11 ± 0.002	$99. \pm 0.11$	$1.0 \cdot 10^6 \pm 0$
	$0.0081 \pm 2.9 \cdot 10^{-4}$	0.16 ± 0.004	$99. \pm 0.06$	$1.0 \cdot 10^6 \pm 0$
0-1 Knapsack	$17. \pm 2.4$	$65. \pm 9.5$	$17. \pm 3.7$	$2.3 \cdot 10^7 \pm 0$
	$26. \pm 3.7$	$49. \pm 4.6$	$20. \pm 1.9$	$2.3 \cdot 10^7 \pm 0$

Table 2: Relative performance on Sun Fire T1000 (as percentage)

Application (input)	Extract min	Insert queue	Computation	Number of candidates
	— skip list —			
	— skip tree —			
15 puzzle (E)	$12. \pm 0.95$	$45. \pm 3.7$	$42. \pm 3.3$	$5.6 \cdot 10^4 \pm 4.1 \cdot 10^3$
	$16. \pm 2.1$	$55. \pm 5.2$	$45. \pm 3.7$	$5.4 \cdot 10^4 \pm 4.5 \cdot 10^3$
15 puzzle (F)	1.9 ± 0.23	$88. \pm 3.3$	9.7 ± 0.39	$1.3 \cdot 10^8 \pm 1.5 \cdot 10^4$
	2.1 ± 0.11	$36. \pm 2.4$	9.3 ± 0.18	$1.3 \cdot 10^8 \pm 1.6 \cdot 10^4$
Graph color (G)	$15. \pm 3.5$	$20. \pm 4.4$	$65. \pm 12.$	$7.8 \cdot 10^6 \pm 1.6 \cdot 10^6$
	$15. \pm 2.6$	$55. \pm 12.$	$63. \pm 11.$	$8.0 \cdot 10^6 \pm 1.6 \cdot 10^6$
Graph color (H)	0.92 ± 0.25	$79. \pm 2.7$	$20. \pm 1.1$	$2.2 \cdot 10^8 \pm 8.2 \cdot 10^3$
	0.85 ± 0.065	$18. \pm 1.2$	$20. \pm 1.1$	$2.2 \cdot 10^8 \pm 1.3 \cdot 10^4$
Asymmetric TSP	$0.018 \pm 4.5 \cdot 10^{-4}$	0.13 ± 0.002	$99. \pm 0.10$	$1.0 \cdot 10^6 \pm 0$
	$0.014 \pm 1.3 \cdot 10^{-4}$	0.17 ± 0.003	$95. \pm 0.10$	$1.0 \cdot 10^6 \pm 0$
0-1 Knapsack	$24. \pm 5.0$	$55. \pm 14.$	$21. \pm 9.2$	$2.3 \cdot 10^7 \pm 0$
	$19. \pm 3.3$	$43. \pm 6.2$	$19. \pm 3.9$	$2.3 \cdot 10^7 \pm 0$

Table 3: Relative performance on quad core Intel Xeon (as percentage)

Application (input)	Extract min	Insert queue	Computation
	— skip list —		
	— skip tree —		
15 puzzle (A)	4.44 ± 0.37	18.8 ± 0.96	22.4 ± 0.53
	3.29 ± 0.09	37.3 ± 0.91	23.4 ± 0.57
15 puzzle (B)	8.52 ± 1.16	$483. \pm 48.2$	42.3 ± 2.99
	6.44 ± 2.91	$308. \pm 68.9$	35.5 ± 8.24
Graph color (C)	7.65 ± 16.9	10.3 ± 1.08	51.7 ± 7.78
	3.28 ± 0.83	79.4 ± 5.84	54.8 ± 9.32
Graph color (D)	8.08 ± 1.03	$124. \pm 9.61$	51.5 ± 5.21
	4.10 ± 0.28	51.4 ± 0.91	52.1 ± 0.39
Asymmetric TSP	0.92 ± 0.03	8.56 ± 0.39	$9.76 \times 10^4 \pm 102.$
	0.83 ± 0.02	11.7 ± 0.22	$9.76 \times 10^4 \pm 70.2$
0-1 Knapsack	21.6 ± 8.37	82.8 ± 32.4	4.16 ± 2.27
	31.6 ± 5.63	61.5 ± 18.5	3.97 ± 1.40

Table 4: Elapsed time per operation on Sun Fire T1000 (in microseconds)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	62.5	31.7	4.90	0.75	0.15
15 puzzle (B)	83.2	15.9	0.85	0.06	0.01
Graph color (C)	64.5	30.3	4.53	0.58	0.01
Graph color (D)	91.9	7.92	0.17	0.00	0.00
0-1 Knapsack	29.6	40.0	14.7	6.27	9.40

(a) skip list extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	83.6	15.3	1.00	0.00	0.00
15 puzzle (B)	91.4	8.6	0.00	0.06	0.00
Graph color (C)	65.0	32.5	2.28	0.14	0.01
Graph color (D)	98.4	1.58	0.00	0.00	0.00
0-1 Knapsack	99.9	0.01	0.00	0.00	0.00

(b) skip list insertion (CAS)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	73.5	19.0	5.09	1.59	0.07
15 puzzle (B)	87.0	12.5	0.46	0.05	0.01
Graph color (C)	42.3	41.6	11.0	3.44	1.67
Graph color (D)	97.1	2.92	0.01	0.00	0.00
0-1 Knapsack	100.0	0.01	0.00	0.00	0.00

(c) skip list insertion (tree traversal)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	40.0	61.1	2.70	0.19	0.07
15 puzzle (B)	57.2	41.5	1.21	0.07	0.02
Graph color (C)	45.6	52.1	1.90	0.26	0.12
Graph color (D)	77.9	21.9	0.16	0.01	0.01
0-1 Knapsack	10.3	25.5	20.1	13.8	30.2

(d) skip tree extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	59.1	11.3	7.31	4.96	17.3
15 puzzle (B)	72.6	18.5	5.11	1.84	1.95
Graph color (C)	2.41	19.3	15.4	11.2	51.7
Graph color (D)	89.7	9.36	0.89	0.11	0.04
0-1 Knapsack	99.9	0.09	0.01	0.00	0.01

(e) skip tree insertion

Table 5: Number of retries for queue operations on Sun Fire T1000 (as percentage)

Application (input)	Extract min	Insert queue	Computation
	— skip list —		
	— skip tree —		
15 puzzle (E)	1.16 ± 0.04	2.11 ± 0.05	2.10 ± 0.09
	1.53 ± 0.04	2.84 ± 0.07	2.14 ± 0.08
15 puzzle (F)	1.05 ± 0.03	27.3 ± 1.08	3.07 ± 0.15
	1.24 ± 0.08	11.6 ± 0.74	2.88 ± 0.05
Graph color (G)	1.47 ± 0.04	1.82 ± 0.05	2.91 ± 0.15
	1.38 ± 0.10	5.24 ± 0.18	2.72 ± 0.12
Graph color (H)	1.32 ± 0.10	31.0 ± 1.45	5.26 ± 0.19
	1.45 ± 0.35	7.70 ± 0.69	5.82 ± 0.44
Asymmetric TSP	1.31 ± 0.04	1.35 ± 0.02	$3.59 \times 10^3 \pm 9.19$
	1.06 ± 0.06	1.67 ± 0.02	$3.57 \times 10^3 \pm 8.29$
0-1 Knapsack	2.23 ± 0.65	5.11 ± 1.34	0.33 ± 0.12
	1.87 ± 0.41	4.08 ± 0.71	0.28 ± 0.07

Table 6: Elapsed time per operation on quad core Intel Xeon (in microseconds)

insertion operations succeed on the first attempt using either the skip list or the skip tree. The knapsack problem suffers contention from the extraction of elements from the head of the queue. With the skip tree, 14.8% of extract min operations on the knapsack problem require five or more attempts. With the skip list, 13.6% of the extract min operations on the knapsack problem require five or more attempts. The contention at the head of the priority queue raises the total proportion of execution time that is spent on the extract minimum operations. This behavior has been verified in Tables 2 and 3. When a greater proportion of time is spent on the extract min operations, as a consequence a smaller proportion of time must be spent on insert element operations. The performance improvements of the skip tree on the insert element operations are observed, but their effects are diluted by the greater time on extract minimum operations.

In the next section, our goal is to generalize the observations made here to an arbitrary optimization problem that is solved using the parallel branch-and-bound solver, based on the information collected from the four applications. We have created a synthetic branch and bound application to study the effects of application characteristics on the relative performance of the skip tree priority queue. The synthetic application is a simplified model of a real application. As a model it captures only the characteristics of the actual application problems that are of interest, and creates a set of assumptions that simplifies application characteristics that are not of interest. It is our hypothesis that too much elapsed time on the lower bound estimate leads to no improvement in performance, as observed in the asymmetric traveling salesman problem. And too little elapsed time on the lower bound estimate also leads to no improvement in performance, as observed in the 0-1 knapsack problem. Of equal interest are the effects of the branching factor and the distribution of lower bound estimates on the performance of the skip tree priority queue.

11 Synthetic Application

We created a synthetic branch-and-bound application in order to study the characteristics of an application that will yield a speedup when the skip tree is used as the centralized data structure. This synthetic application is a simplification of a real branch-and-bound application. It has been designed to test three hypotheses of the branch-and-bound applications: (1) the distribution of lower bounds of the candidates in the search space affects the performance of the skip tree; (2) the computation time of

	0	1-3	4-6	7-9	10+
15 puzzle (A)	67.4	28.6	3.56	0.39	0.04
15 puzzle (B)	93.8	6.08	0.15	0.00	0.00
Graph color (C)	62.2	32.7	4.54	0.51	0.05
Graph color (D)	97.4	2.61	0.01	0.00	0.00
0-1 Knapsack	45.3	41.2	9.78	2.63	1.17

(a) skip list extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	91.8	8.13	0.09	0.00	0.00
15 puzzle (B)	98.9	1.13	0.00	0.00	0.00
Graph color (C)	75.5	24.1	0.38	0.01	0.00
Graph color (D)	99.9	0.23	0.00	0.00	0.00
0-1 Knapsack	100.0	0.00	0.00	0.00	0.00

(b) skip list insertion (CAS)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	81.3	15.2	2.75	0.59	0.15
15 puzzle (B)	96.8	3.15	0.07	0.00	0.00
Graph color (C)	48.5	37.9	9.67	2.77	1.11
Graph color (D)	99.3	0.73	0.00	0.00	0.00
0-1 Knapsack	100.0	0.00	0.00	0.00	0.00

(c) skip list insertion (tree traversal)

	0	1-3	4-6	7-9	10+
15 puzzle (A)	48.4	43.4	6.01	1.35	0.83
15 puzzle (B)	79.9	19.4	0.59	0.04	0.00
Graph color (C)	56.4	39.0	3.72	0.65	0.22
Graph color (D)	87.6	12.3	0.03	0.00	0.00
0-1 Knapsack	37.2	47.9	10.6	2.8	1.43

(d) skip tree extract minimum

	0	1-3	4-6	7-9	10+
15 puzzle (A)	69.4	21.6	6.11	1.88	0.98
15 puzzle (B)	87.7	11.6	0.67	0.05	0.00
Graph color (C)	30.7	47.8	14.8	4.52	2.08
Graph color (D)	96.5	3.52	0.02	0.00	0.00
0-1 Knapsack	100.0	0.03	0.00	0.00	0.00

(e) skip tree insertion

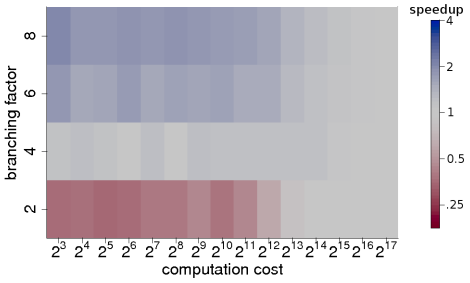
Table 7: Number of retries for queue operations on quad core Intel Xeon (as percentage)

the lower bound affects the performance of the skip tree; and (3) the branching factor of the application affects the performance of the skip tree. In the synthetic application we assume the objective function is a minimization problem. We also assume that there is no upper bound. All generated children instances are valid instances for the priority queue. In a real application, a tight upper bound serves to limit the effective branching factor of the application, and would also add additional computational cost of the upper bound that is not taken into consideration by the synthetic application.

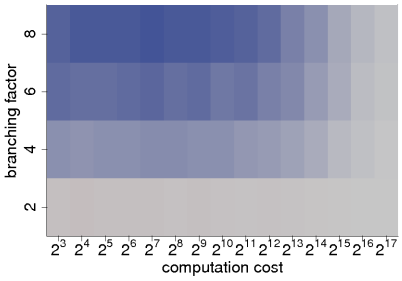
The synthetic application has a unique termination condition in that it does not return a solution that has any semantic meaning. Instead it generates problem instances until a predetermined limit is reached. When this limit has been reached then the algorithm terminates. In our test cases this limit has been fixed at $2.5 \cdot 10^7$ instances on the Niagara server and $1.0 \cdot 10^8$ instances on the Intel Xeon server. The Xeon processor has larger cache sizes and our test machine is configured with more memory as compared to the Niagara server. Each thread generates candidate solutions until a predetermined number of solutions has been generated per worker thread. The state of a problem instance in the synthetic application consists of a single 64-bit integer value that stores the lower bound of the candidate solution. The lower bound of a child instance is derived from the lower bound of the parent instance using one of three random distributions.

The first distribution is uniform: it does not use the parent's lower bound and selects a new lower bound from a uniform probability distribution across the range of 64-bit values. The second distribution is monotonically increasing: take the parent's lower bound and add to this lower bound a positive integer from a uniform distribution. The monotonically increasing distribution places the least amount of contention at the head of the priority queue, as elements tend to be inserted towards the tail of the queue. The 15 puzzle and graph coloring problems are both instances where the lower bound of a child is never less than or equal to the lower bound of the parent. In the 15 puzzle problem, each child adds a new move to the sequence of moves that have already been taken. In the graph coloring problem, each child adds a new coloring to the set of vertices that have already been colored.

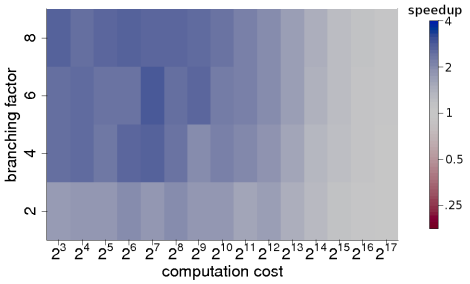
The third distribution is a restricted distribution: take the parent's lower bound and with a probability of 99.98% return the same lower bound, or with a probability of 0.01% increment the lower bound



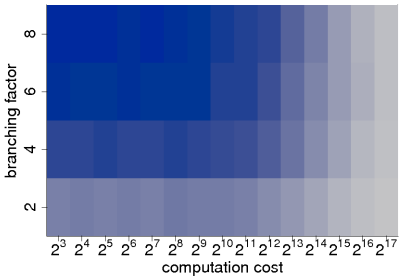
(a) Sun Fire uniform distribution



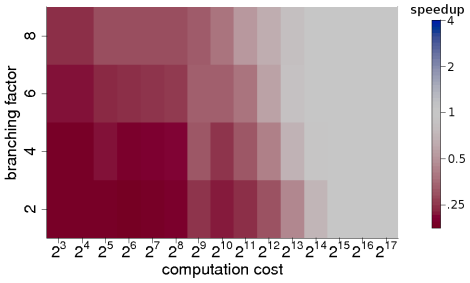
(d) Intel Xeon uniform distribution



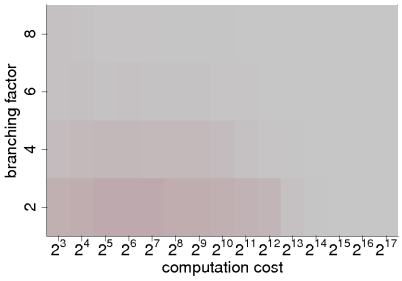
(b) Sun Fire monotonic distribution



(e) Intel Xeon monotonic distribution



(c) Sun Fire restricted distribution



(f) Intel Xeon restricted distribution

Figure 6: Relative speedup on synthetic benchmark

by one, or with a probability of 0.01% decrement the lower bound by one. The primal-dual algorithm we used to solve the 0-1 knapsack problem generates optimal profit estimates that fluctuate within a narrow band of possibilities near the split solution. This characteristic is not specific to the primal-dual method of solving the 0-1 knapsack problem. A restricted distribution of candidate solutions will arise because the optimal solution of the 0-1 knapsack problem tends to be very similar in structure to the greedy solution of the knapsack problem. Fast solvers of the 0-1 knapsack problem that take advantage of the expanding core set of items (see Section 9) will have a restricted distribution of solutions.

A proxy calculation is used to represent the elapsed time of computing the lower bound. The proxy calculation is the sum of the integers from 1 to N, where N can be varied to adjust the duration of computation time. This implementation is tunable, reproducible, and independent of any memory hierarchy effects. In our initial experimental setup, the elapsed time of computing the lower bound was to be determined using an independent variable that would vary the amount of time a worker thread would sleep. The sleep interval would simulate the compute time necessary for estimation of a lower bound. However, sub-millisecond precision for sleep intervals is not implemented on the HotSpot Java Virtual Machine for either x86 or SPARC architectures. Based on the limitations of the virtual machine, the proxy calculation of the computation interval was developed to forgo precision of the estimated computation time in exchange for accuracy.

The relative performance on the synthetic benchmark of the skip tree versus the skip list on the Sun Fire and Intel Xeon is shown in Figure 6. The value N for the computation cost is shown on the horizontal axis. The branching factor is shown on the vertical axis. A heat-map of the relative speedup is plotted on a logarithmic scale. The logarithmic scale yields the same distance between 200% to 400% relative speedup as with 25% to 50% relative speedup. Positive speedup is blue, negative speedup is red, and no change in performance is white. Each probability distribution of lower bound estimates has a direct impact on the relative performance. A monotonic distribution incurs the least contention among concurrent threads, while a restricted distribution incurs the most contention. The branching factor affects the number of candidate solutions that are computed in between extract minimum operations. Therefore the branching factor dictates the rate of the extract minimum operations. In general, a higher branching factor increases the performance of the skip tree over the skip list. A lower branching factor is usually more desirable to minimize the overall size of the search space. The benefit of a low branching factor is that the increase in search space size due to calculating two generations of candidate solutions is relatively small when the branching factor is low. The effect of a longer computation time is to dilute the effects of either the performance advantage or disadvantage of the skip tree.

The distribution of lower bound estimates has a dominant impact on the performance of the skip tree priority queue. On the Niagara architecture with a computation cost between the values of 2^3 and 2^{13}, the mean speedup of the uniform distribution is x1.25, the mean speedup of the monotonic distribution is x2.21, and the mean speedup of the restricted distribution is x0.33. On the Intel Xeon architecture with the same computation costs, the mean speedups are x1.79 for the uniform distribution, x2.67 for the monotonic distribution, and x0.85 for the restricted distribution.

12 Branch-and-Bound Guidelines

In this section, we provide guidelines for when to select the lock-free skip tree to use as a centralized priority queue in a parallel branch-and-bound application versus the lock-free skip list. These guidelines are based on the results of the application benchmarks on the N Puzzle, graph coloring, asymmetric traveling salesman problem, and 0-1 knapsack problem and the results of the synthetic benchmarking experiments just described.

Rule #1: Avoid contention at the head of the queue

A natural bottleneck in a concurrent priority queue is the head of the queue. When new elements are inserted into the middle of the queue or the end of the queue, then fewer operations are placed on the head of the queue. In the synthetic benchmark, the monotonic distribution inserts elements into the end of the queue and the uniform distribution inserts elements into any portion of the queue.

The restricted distribution inserts candidates within a narrow range of lower bound estimates. The lock-free skip tree shows mean speedups of x2.21 and x2.67 on SPARC and x86 platforms under the monotonic distribution and mean speedups of x1.25 and x1.79 under the uniform distribution. The skip tree exhibits worse performance than the skip list under the restricted distribution with mean speedups of x0.33 and x0.85.

In our performance analysis of the knapsack solver, 30% of extract minimum operations using the skip list required four or more retries and 64% of extract minimum operations using the skip tree required four or more retries. The next highest percentage of four or more retries in extract minimum operations is 5% on the graph coloring problem using the skip list and 3.7% on the 15 puzzle problem using the skip tree. Using the performance analysis for the knapsack solver and the synthetic benchmark results, we can show that a relatively small computational cost for the lower bound estimate correlates with higher contention at the head of the queue. Tables 4 and 6 show the elapsed time per operation in microseconds for the Sun Fire and Intel Xeon platforms. The elapsed time per computation for the knapsack solver is 4.06 microseconds on the Sun Fire and 0.31 microseconds on the Intel Xeon. The average elapsed time per computation on the other application benchmarks (excluding the asymmetric traveling salesman solver) is 41.7 microseconds on the Sun Fire and 3.36 microseconds on the Intel Xeon. Figure 6 (c) and (f) show the relative performance of the skip tree on the synthetic application with the restricted distribution to generate lower bound estimates. Smaller computation costs are correlated with weaker performance from the lock-free skip tree.

Rule #2: Seek monotonic heuristic functions

Consistent (or monotone) heuristic functions are general strategies for traversing through a state space that get closer to the solution state without taking any backward steps. Given a node n from the search space and a successor n' of n, the estimated cost of reaching the goal from n is no greater than the cost of getting from n to n' plus the estimated cost of reaching the goal from n': $h(n) \leq c(n, n') + h(n')$. The fifteen puzzle, graph coloring, and traveling salesman problem solvers use monotonic estimates for the lower bound of partial solutions. In the fifteen puzzle, each successor state in the search space represents the addition of a move towards the target configuration of the board. In the graph coloring problem, each successor state assigns a color to an uncolored node until eventually all of the nodes have been assigned colors.The ATSP solver either selects a path along the tour of the graph or eliminates a path from the selection process.

The primal-dual algorithm for solving the knapsack problem uses an inconsistent heuristic function. The knapsack is allowed to overfill so that successor states may remove elements from the knapsack. There are cases when an inconsistent heuristic function is preferable to a consistent heuristic function [48, 49]. In the knapsack problem, the primal-dual algorithm constrains the state space to improve the overall performance of the algorithm. These techniques are most commonly applied to the iterative deepening A* algorithm, which performs iterative rounds of depth-first search. When applied to a breadth-first parallel branch and bound solver, these techniques can add to contention to the head of the priority queue.

Rule #3: Seek large working set sizes

For those applications that satisfy Rules #1 and #2, the greatest improvements in performance of the skip tree versus the skip list are observed for those input problems that have the largest working set sizes. In Figures 2-5, all scenarios have some threshold number of candidates, T, such that all initial states that have T or more candidate solutions exhibit speedup using the skip tree versus the skip list. In general, it is difficult to estimate the number of partial solutions that will be generated for a specific initial state. For example, the N puzzle problem has no known upper bound that is a function of the initial board state. Figure 7 shows two initial states from the iterative deepening A* (IDA) algorithm used to solve one hundred instances of the 15 puzzle in [28]. Both instances have an initial lower bound estimate of 41. The instance on the left generates 24,492,852 partial solutions using the IDA solver. The instance on the right generates 1,369,596,778 partial solutions. We have shown that initial configurations with small working set sizes can perform relatively worse using the skip tree versus the

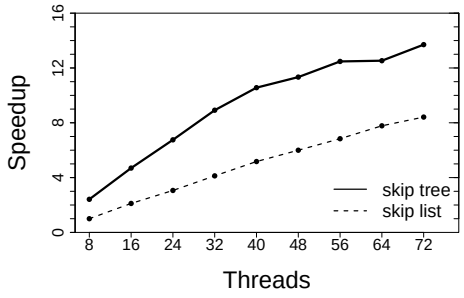
7	15	8	2
13	6	3	12
11		4	10
9	5	1	14

(a)

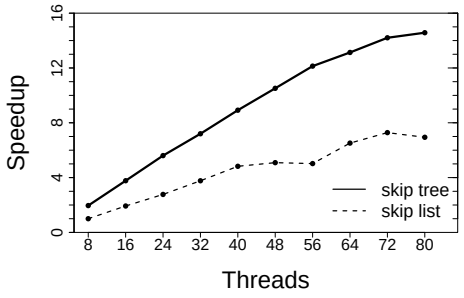
7	6	8	1
11	5	14	10
3	4	9	13
15	2		12

(b)

Figure 7: Two fifteen puzzle instances from [28].



(a) fifteen puzzle problem 'F' instance



(b) graph coloring problem 'H' instance

Figure 8: Azul compute appliance

skip list. A smaller working set size implies a shorter total execution time. The absolute time that is lost on these initial configurations using the skip tree is small as compared to the amount of time that is gained in those configurations with large working set sizes.

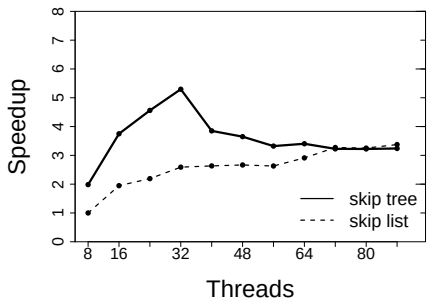
13 Shared Memory Supercomputers

13.1 Azul Appliance

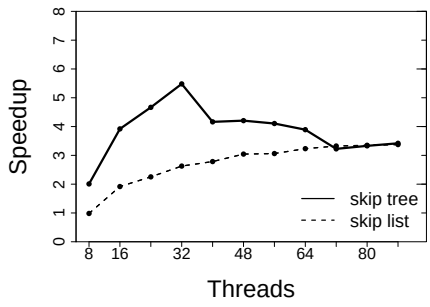
The Azul compute appliance is a custom shared memory supercomputer designed for the Java runtime environment. The processing unit of the compute appliance is a Vega 3 processor, a 54 core 64-bit RISC processor. In addition to the standard RISC instruction set, the Vega processor has a few specialized instructions to aid the Java virtual machine in object allocation and garbage collection. Up to 16 Vega processors can be installed on a compute appliance, for a total of 864 processor cores. On the compute appliance, each processor has three available banks of four memory modules. Each core has a 16 kB L1 instruction cache and 16 kB L1 data cache. Each processor has six 2 MB unified L2 caches. Groups of 9 cores share one L2 cache [50]. The Azul compute appliance runs on top of a minimalist operating system and has its own Java virtual machine implementation that is based on the OpenJDK project.

The Azul Java virtual machine uses a pauseless garbage collection algorithm [51]. At no point in the pauseless garbage collection algorithm is there a Stop-The-World pause, a pause in which all application threads must be simultaneously stopped. Pauseless garbage collection provides soft real-time guarantees with regard to memory management. A comparison of the SpecJBB benchmark with the Azul Java virtual machine versus the HotSpot and IBM Java virtual machine reported that worst-case transaction execution times were over 45 times better and average execution times were comparable on the Azul JVM.

The tradeoff of the pauseless garbage collection is that low latency transactions come at a penalty of increased heap size needed for the copying phase of the garbage collection algorithm. We tested



(a) without NUMA-aware garbage collection



(b) with NUMA-aware garbage collection

Figure 9: Graph coloring on Altix UV 1000

fifteen puzzle problem 'F' instance and graph coloring problem 'H' instance on the Azul compute appliance. These were selected as the problem instances with the largest working set sizes from the two applications that exhibited a performance improvement using the skip tree versus the skip list. In the branch-and-bound applications, the average number of processed candidates is independent of the number of worker threads that are running. The total working set size and the ratio of garbage objects to live objects are constant factors relative to the number of worker threads. Therefore an increase in worker threads yields an increase in the rate of non-live objects produced per unit of time. Using Azul's Real-Time Profiling and Monitoring Tool, the fifteen puzzle and graph coloring applications running at 100 cores were shown to spend 85% of the total runtime on the garbage collection algorithm under the available heap size limitations.

Our resource allocation on an Azul compute appliance comprised 207 cores and 55 GB of heap space. The results of our tests on the Azul compute appliance are shown in Figure 8. The baseline for the two applications we ran is the execution runtime for the skip list using 8 cores. The speedup of the skip tree on the fifteen puzzle is x13.7 using 9 times as many cores when compared to the baseline, and x14.5 on the graph coloring problem when using 10 times as many cores as the baseline. The speedup of the skip list on the fifteen puzzle is x8.4 using 9 times as many cores as the baseline, and x6.9 on the graph coloring problem when using 10 times as many cores.

13.2 SGI Altix UV 1000

The second shared-memory supercomputer that was used to test the application benchmarks was Blacklight, a SGI Altix UV 1000. Blacklight consists of 256 blades connected by a NUMAlink® 5 Interconnect. Each blade holds 2 Intel Xeon X7560 eight-core processors, for a total of 4096 cores on the machine. The 16 cores of each blade share 128 GB of memory; the total capacity of the machine is 32 TB. The processors are connected in an 8 x 8 paired node 2D torus topology. Blacklight is running SUSE Linux Enterprise Server 11 with a modified 2.6.32.12 Linux kernel.

The HotSpot Java virtual machine can run the fifteen puzzle and graph coloring benchmarks on Blacklight up until about 64 cores before no further speedup is observed by the addition of more processors. The same behavior is observed for the lock-free skip list and the lock-free skip tree. The default garbage collection implementations in HotSpot are NUMA-unaware. The parallel collector, also known as the throughput collector, can be optionally enabled with NUMA-aware support. Enabling NUMA support on HotSpot when running on Blacklight crashes the virtual machine during its initialization phase. This behavior has been observed with Java SE 1.6.0_22 and OpenJDK 1.7.0 binary release 114. This crash is caused by an interaction of two features of the Linux kernel that are used to allocate hardware resources on shared-memory NUMA architectures.

libnuma is a user space shared library that provides an API for implementing NUMA policies in applications [52]. It allows an application to determine the underlying memory topology of the hardware at runtime by querying the operating system. libnuma can also be used to bind a thread to a specific processor. NUMA support has been available in the Linux kernel in one form or another for

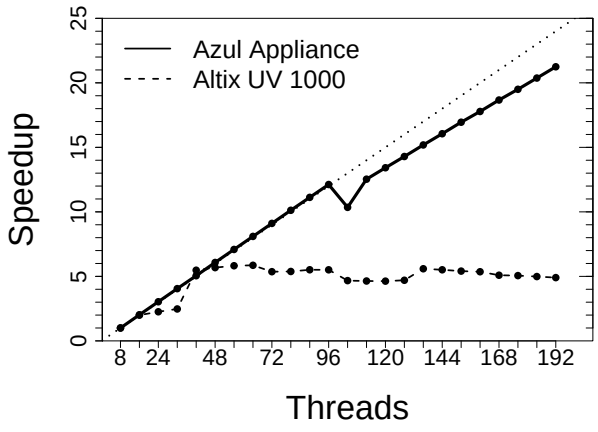


Figure 10: 16 queens on shared-memory supercomputers

all of the 2.6 release series.

Another mechanism for assigning a set of processors and memory nodes to a set of tasks in the Linux kernel is the cpusets interface. A cpuset is composed of a set of processor nodes and a set of memory nodes. The root cpuset contains all of the processor nodes and memory nodes. Given a cpuset, a child cpuset can be defined that contains a subset of the parent resources. Cpusets may be marked exclusive, which ensures that no other cpuset except direct ancestors and descendants may contain overlapping processor or memory resources. Cpusets appeared in version 2.6.7 of the Linux kernel. The Portable Batch System (PBS) is a job scheduler for the allocation of batch jobs in a shared computational environment. PBS can be configured to use cpusets to isolate tasks to a specific set of processors.

The NUMA-aware garbage collector in the HotSpot Java virtual machine is designed to use libnuma for memory allocation, but does not query the cpuset interface. The current implementation queries the machine for the total number of online processors using the sysconf POSIX interface [**?**]. The libnuma API sets a hard limit on the number of processors that can participate in a NUMA memory set. The total number of online processors on the Blacklight supercomputer is much larger than the processor limit set by the libnuma API. When the Java virtual machine attempts to create a NUMA memory set as large as the number of total processors on the system, the result is a crash in the initialization of the virtual machine. We have written a patch for the OpenJDK that detects whether the cpuset interface is present, and if so, then uses the cpuset interface to determine the number of memory processors available.

The libnuma library exports two versions of its API to the user: libnuma 1.1 and libnuma 1.2. The libnuma 1.1 API is not aware of cpuset constraints on a processor, and the libnuma 1.2 API is aware of cpuset constraints. The OpenJDK currently uses the libnuma 1.1 API. We have written a second patch to update the OpenJDK to use the libnuma 1.2 interface. While writing the patch to the OpenJDK for the libnuma 1.2 interface, we encountered a bug in the libnuma library. The documentation for the function numa_get_mems_allowed() states the function returns a bit vector of all the available memory nodes in the cpuset of a process. The function returned an empty bit vector when a cpuset was applied to a process. We submitted a patch to correct the behavior of numa_get_mems_allowed() in the libnuma library. The patch was adopted in the release of libnuma 2.0.6-rc4.

Figures 9a and 9b show the results of the graph coloring problem on the Altix UV 1000. Figure 9a uses the non-NUMA concurrent garbage collection algorithm, and Figure 9b uses the NUMA-aware concurrent garbage collector. Speedup is normalized to the runtime of 8 threads using the skip list priority queue. Each blade of the supercomputer can support 32 hardware threads: 16 cores with hyperthreading enabled. In all cases, the relative speedup does not improve using more than one blade of the machine. The speedup using 88 cores and the skip list priority queue is x1.3 the speedup

using 32 cores. The speedup using 88 cores and the skip tree priority queue is x0.6 the speedup using 32 cores. The NUMA-aware concurrent garbage collector yields a small improvement on the performance of the skip tree priority queue on the graph coloring solver.

To study the behavior of the OpenJDK on the Altix UV 1000 using a standard Java benchmark that does not rely on our lock-free skip tree, we selected the N queens puzzle benchmark from the Java 7 fork/join testing framework. The fork/join testing framework has been written by the members of the JCP JSR-166 Expert Group to measure the performance of the classes in the java.util.concurrent library. The runtime of the N queens problem was measured for the 16 x 16 chess board with 16 queens on the Altix UV 1000 and the Azul Systems compute appliance. The relative speedup on this problem is shown in Figure 10. Speedup is normalized to a runtime of 8 threads on the UV 1000 for measurements on that supercomputer, and a runtime of 8 threads on the Azul Systems compute appliance for measurements on that supercomputer. Speedup is almost linear on the Azul compute appliance. The N queens benchmark exhibits marginal speedup using more than one blade on the Altix UV 1000.

In this section, we have shown that it is possible to use shared-memory supercomputers efficiently to solve parallel branch-and-bound problems. For those parallel branch-and-bound applications that exhibit the properties enumerated in Section 12, the lock-free skip tree shows up to a x2.1 improvement in runtime as compared to the lock-free skip list when running on 88 hardware threads. In order for an application to scale across a shared-memory interconnection communication layer, it is necessary for all the runtime layers underneath the application to scale as well. The Azul compute appliance runs on top of a minimalist operating system and has its own Java virtual machine implementation that is based on the OpenJDK project. The SGI Altix is running a modified 2.6.32.12 Linux kernel and a patched version of the OpenJDK 7 release. We patched the OpenJDK to identify cpuset allocations and to correct the NUMA-aware garbage collector to work in the presence of cpuset policies. the runtime was unable to scale across multiple blades when measured by the Java 7 fork/join testing framework. Given the commercial success of a Java compute appliance that can scale to hundreds of threads using specialized hardware, a specialized operating system, and a specialized Java virtual machine, it is most likely that significant modifications would be necessary to achieve the same scalability on a conventional hardware and software stack.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, University of California at Berkeley, 2006.

[2] Michael Spiegel and Paul F. Reynolds, Jr. Lock-free multiway search trees. In *Proceedings of the 39th International Conference on Parallel Processing (ICPP 2010)*, 2010.

[3] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 50–59, 2004.

[4] Doug Lea. Concurrent skip list map. Approved in JSR 166 for java.util.concurrent in J2SE5.0, 2004.

[5] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[6] Conrado Martinez and Salvador Roura. Randomized binary search trees. *Journal of the ACM*, 45:288–323, 1998.

[7] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35:341–358, 2005.

[8] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *Proceedings of the International Symposium on Distributed Computing*, 2008.

[9] Nikolas Askitis and Ranjan Sinha. HAT-trie: A cache-conscious trie-based data structure for strings. In *Proceedings of the Thirtieth Australasian Computer Science Conference*, 2007.

[10] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Theoretical Informatics and Applications*, 31(3):251–269, 1997.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[12] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44:35–40, 2010.

[13] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley, 1985.

[14] Ailsa H. Land and Alison G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[15] Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors. *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer, 2010.

[16] B. L. Fox, J. K. Lenstra, A. H. G. Rinnooy Kan, and L. E. Schrage. Branching from the largest upper bound: Folklore and facts. *European Journal of Operational Research*, 2:191–194, 1978.

[17] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[18] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37:1657–1665, 1988.

[19] Vipin Kumar, V. Nageshwara Rao, and K. Ramesh. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conf. on Artificial Intelligence (AAAI-88)*, 1988.

[20] Jonathan Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the cm-5. *SIAM Journal on Optimization*, 4:794–814, 1994.

[21] Jonathan Eckstein, Cynthia A. Phillips, and William E. Hart. Pico: An object-oriented framework for parallel branch and bound. In *Studies in Computational Mathematics, Volume 8: Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*. Elsevier, 2001.

[22] Kurt Anstreicher, Nathan Brixius, Jean-Pierre Goux, and Jeff Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91:563–588, 2002.

[23] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multithread systems. Technical report, Chalmers University of Technology and Göteborg University, 2003.

[24] Richard E. Korf. Sliding-tile puzzles and rubik's cube in AI research. *IEEE Intelligent Systems*, 14:8–11, 1999.

[25] G. Kendall, A. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *International Computer Games Association Journal*, 31:13–34, 2008.

[26] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2:397–404, 1879.

[27] P.D.A. Schofield. Complete solution of the eight puzzle. *Machine Intelligence*, 3:125–133, 1967.

[28] Richard E. Korf. Depth-first iterative-deepening: An optimal admissable tree search. *Artificial Intelligence*, 27:97–109, 1985.

[29] Othar Hansson, Andrew Mayer, and Moti Young. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63:207–227, 1992.

[30] Richard E. Korf and Larry A. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[31] Marek Kubale, editor. *Graph Colorings*. American Mathematical Society, 2004.

[32] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

[33] Thomas J. Sager and Shi-Jen Lin. A pruning procedure for exact graph coloring. *ORSA Journal on Computing*, 3:226–230, 1991.

[34] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39:378–406, 1991.

[35] E. C. Sewell. An improved algorithm for exact graph coloring. In *Cliques, coloring and satisfiability: second DIMACS implementation challenge*, pages 359–372, 1996.

[36] D. Costa and A. Hertz. Ants can colour graphs. *The Journal of the Operational Research Society*, 48:295–305, 1997.

[37] Eugène L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and David B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley-Interscience Publications, 1985.

[38] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook, editors. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[39] Jill Cirasella, David S. Johnson, Lyle A. McGeoch, and Weixiong Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. In *Proceedings of the Third International Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2001.

[40] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[41] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[42] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[43] J. Edmonds. Optimum branchings. *Journal Research of the National Bureau of Standards*, 71B:233–240, 1967.

[44] Frederick Bock. An algorithm to construct a minimum spanning tree in a directed network. In *Developments in Operations Research: Proceedings of the Third Annual Israel Conference on Operations Research*, 1969.

[45] Gerhard Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

[46] Hans Kellerer, Ulrich Pferschy, and David Pisinger, editors. *Knapsack Problems*. Springer, 2004.

[47] David Pisinger. An expanding-core algorithm for the exact 0-1 knapsack problem. *European Journal of Operational Research*, 87:175–187, 1995.

[48] Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. In *Proceedings of the 21st national conference on Artificial intelligence*, 2006.

[49] Uzi Zahavi, Ariel Felner, Jonathan Schaeffer, and Nathan Sturtevant. Inconsistent heuristics. In *Proceedings of the 22nd national conference on Artificial intelligence*, 2007.

[50] Cliff Click. Azul's experiences with hardware/software co-design. Keynote presentation at the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), 2009.

[51] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

[52] Andreas Kleen. A NUMA API for LINUX. Whitepaper, April 2005. Prepared by SUSE LINUX Products GmbH, a Novell Business.