Minimum-Blocking Parallel Bidirectional Search

Dale E. Parson
Kutztown University of PA
15200 Kutztown Road
Kutztown, PA 19530-0730

Dylan Schwesinger
Lehigh University
Memorial Drive West
Bethlehem, PA 18015

Abstract

The present work investigates using non-blocking and minimum-blocking Java library classes as a basis for improving performance of parallel bidirectional search on a multiple-instruction multiple-data (MIMD) processor. The approach represents individual states as minimum-size, immutable objects. It uses a work queue to distribute states-for-expansion among worker threads, and it uses two sets for keeping track of states previously explored in each direction. The queue class is thread-safe and non-blocking, and the set class is thread-safe and non-blocking for read operations, with parallel locking of subsets for write operations. It is essentially a dataflow approach as opposed to a state machine approach. Rather than step worker threads through state transitions using blocking synchronization, it flows states to be expanded to worker threads in the order required by bidirectional search. This approach has clear, measurable advantages over approaches that use blocking synchronization.

Keywords

bidirectional search, concurrent programming, Java, multiprocessing, parallel programming

1. Introduction and related work

This report is an outcome of curriculum development for a senior and graduate-level course in multiprocessor programming.[1] The course uses the Java™ programming language because of its extensive library of thread-safe container classes and atomic data types, and its explicit memory model that supports aggressive optimization of dynamically compiled code [1]. We have found that for some algorithm benchmarks Java outperforms statically optimized C/C++ using the native compiler. This report focuses on applying Java library classes to the problems of parallel bidirectional search.

Bidirectional search is a classic approach to solving search space problems when both the initial and final states of the search are known in advance [2]. It searches for paths that connect these two states, typically searching for minimum-length paths. In problems with exponential growth of the search space size as a function of search path length, bidirectional search reduces the number of states inspected over unidirectional approaches by integrating the results of two shorter paths that grow simultaneously from the initial and final states.

Bidirectional search is an interesting algorithm for adaptation to parallel programming because it aims at improving run-time performance over simpler search algorithms such as depth-first or breadth-first search, and because it lends itself to parallel implementation. Figure 1 is a schematic view of bidirectional search as exploration of a maze in search of the shortest path, given knowledge of both the entrance and exit locations. Regardless of the concrete problem being solved, bidirectional search always requires knowledge of the starting and ending states of the search. It often utilizes problem-specific heuristics to prune the search space of dead ends, but it is not required to do so.

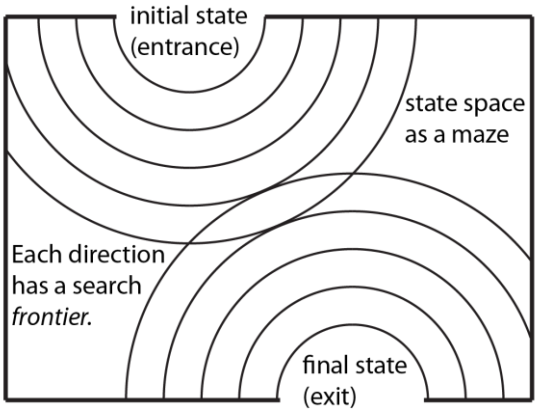


Figure 1: Bidirectional Search as a Maze

The fundamental point of bidirectional search is to limit the exponential growth in number of states explored in a single direction by exploring two shorter paths, one from each direction, and then detecting states in which those opposing paths meet. The outermost set of states currently being explored in either direction constitutes that direction's *frontier*. A single-threaded search based on breadth-first search uses a first-in first-out (FIFO) queue of states to expand as a work queue. The algorithm first enqueues the initial state and final state in the work queue, after which it iteratively removes a state, computes its single-step expansions, checks for cycles and converging DAG paths within the states of its

[1] This work was made possible by equipment grants from Sun Microsystems and the NVIDIA Corporation, and by stipend grants from the Intel Corporation and the PA State System of Higher Education. Please see the Acknowledgements section for details.

originating direction, and checks for collisions with states coming from the opposite direction. Cycle / converging paths and collision checking require storing explored states in a set (keyed on location in the space + search direction) or two sets (keyed on location only). Detection of opposing-path collisions uncovers shortest-path solutions to the problems. In the absence of cycles / converging paths and solutions, the algorithm enqueues one or more single-step expansions and repeats these steps until it locates a solution.

The worst case time, space complexity for unidirectional breadth-first search is $\mathbf{O(b^{d+1})}$, where base **b** is the number of alterative branches (*branching factor*) in the search path that can be taken at any step, and exponent **d** is the *depth* (or equivalently *length*) of the path. When b==3, for example, an un-pruned frontier contains 3 possible states after 1 step, 9 possible states after 2 steps, and so on, generalizing to $\mathbf{b^d}$ states at the frontier, although some may be eliminated through detection of cycles, converging DAG paths, or via application-specific heuristics. The total states explored leading up to the frontier + the frontier itself grows at the rate $\mathbf{O(b^{d+1})}$.

Bidirectional breadth-first search, in contrast, grows at the much lower rate $\mathbf{O(b^{d/2})}$. Each of the two search directions in bidirectional search grows to only half the length of the corresponding unidirectional search, thereby cutting down on the massive exponential growth in explored states that comes with the relative doubling of length in unidirectional search.

Recent work reported on integrating parallel processing with bidirectional search focuses on applying parallel implementation of heuristic strategies to prune the search space [3-5]. Using application-oriented heuristics to radically reduce the number of states explored is the primary means for accelerating the basic bidirectional algorithm. Observing the incremental state expansion of a search domain often uncovers useful heuristics.

2. Minimum Blocking Approach

Our initial solution to parallel bidirectional search used the following algorithm, which implements a two-phase state machine. Each immutable state object contains its internal state fields and an immutable reference to its predecessor in its search path.

```
enqueue initial state into work queue
enqueue final state into work queue
set forwardStatesSet to the set of {initial state}
set backwardStatesSet to the set of {final state}
set setOfSolutions to empty set {}
set direction to Forward
set isdone flag to False
while not isdone
    dequeue a state-to-expand from front of work queue
    if directionOf(state-to-expand) not equal direction
        post a pending-change-of-direction to all threads.
        block until all threads ready to reverse direction.
        set direction to its reverse.
    for each single-step expansion of state-to-expand
        if expansion is in StatesSet from opposing side
            if 1st solution or cost equals solution's cost
                add expansion's path to setOfSolutions
            else (cost is greater)
                set isdone flag to True
        else if expansion is in StatesSet from this side
            // a cycle or converging DAG path detected
            do not use this expansion
        else
            add expansion to StatesSet from this side
            enqueue expansion in work queue
```

Listing 1: Parallel, blocking state machine

Enqueues into the work queue and dequeues from the work queue do not block in this algorithm. The viability of non-blocking retrieval depends on the fact that exponential growth of the search space ensures that most dequeue operations will receive a state-to-expand from the work queue. It is only at the beginning of the search that some threads do not initially find states-to-expand via the non-blocking dequeue operation within a given phase (forward or backward). Those threads resort to a polling loop, trying the queue repeatedly until they receive data or until another thread posts a *pending-change-of-direction* flag. Idle polling consumes processors only until the work queue begins to grow at an exponential rate. Our implementation uses the ConcurrentLinkedQueue from the java.util.concurrent library package as the work queue. The documentation for that class states that, "This implementation employs an efficient "wait-free" algorithm." [6, 7]

The *forwardStatesSet* and *backwardStatesSet* of Listing 1 are objects of class ConcurrentHashMap of java.util.concurrent. There is no comparable Set class per se, but the keys of a Map can serve as elements of a Set. The documentation for this class states, "However, even though all operations are thread-safe, retrieval operations do *not* entail locking." Write locks are distributed across a number of *stripes*, where a stripe is a subset of the buckets in the hash table [8]. When two writers do not collide on the same stripe, they do not impede each other. Application programmers can adjust the number of stripes, trading increased parallelism against the memory cost of maintaining additional lock stripes.

A change of direction in the algorithm of Listing 1 entails waiting until all threads have completed expansion of the current direction, forward or backward. Referring to Figure 1, all threads expand the frontier of only one direction at a time during one phase of this state machine approach. Our thinking was to keep the set of states coming from the opposing side stable for collision testing during the expansion of states in the current side. We implemented blocking using the CyclicBarrier class from java.util.concurrent [6]. This library class blocks all calling worker threads until the last worker thread has entered the barrier. The final thread to enter reverses the direction of the search state variables, and then all threads enter the next phase of the search. CyclicBarrier provides very coarse-grain synchronization. The intent in

using CyclicBarrier was to minimize fine-grain synchronization while supporting stability in testing for state membership in the opposing *StateSet*.

After working with our implementation of Listing 1 we realized that we could eliminate locking altogether. Listing 2 gives the revised algorithm.

```
enqueue initial state into work queue
enqueue final state into work queue
set forwardStatesSet to the set of {initial state}
set backwardStatesSet to the set of {final state}
set setOfSolutions to empty set {}
set isdone flag to False
while not isdone
    dequeue a state-to-expand from front of work queue
    for each single-step expansion of state-to-expand
        if expansion is in StatesSet from opposing side
            if 1st solution or cost equals solutions' cost
                add expansion's path to setOfSolutions
            else (cost is greater)
                set isdone flag to True
        else if expansion is in StatesSet from this side
            // a cycle or converging DAG path detected
            do not use this expansion
        else
            add expansion to StatesSet from this side
            enqueue expansion in work queue
```

Listing 2: Parallel, minimum-blocking dataflow machine

The dataflow algorithm of Listing 2 dispenses with the CyclicBarrier waiting of Listing 1. The dequeue operation remains a non-blocking poll with looping until the work queue begins to fill, tests for *StatesSet* membership are non-blocking, and insertion of new states into *StatesSet* occurs using concurrent lock stripes. With a high lookup-to-insertion ratio for *StatesSet* members (all insertions are preceded by lookups to detect cycles and solutions), locking is minimal and configurable via the *StatesSet's* stripes constructor parameter.

Dispensing with coarse-grain synchronization of the two-phase state machine is possible because states flow through the *work queue* in approximately the correct order. Forward states-to-expand alternate with reverse states-to-expand, partitioned by frontier-being-expanded for the most part.

This temporal sequencing of wave fronts is stochastic, not deterministic. A thread that finds most (but not all) of its state expansions to be dead ends (cycles or converging DAG paths) for a series of dequeue operations places frontier states onto the work queue quickly; some worker threads could be two phases ahead of other threads, expanding a path of length L+1 for a given direction while some threads are expanding paths of length L for that same direction. There is no particular problem in occasionally "getting ahead," as implied by the overlapping frontiers of Figure 1. A thread that has gotten ahead on one turn may find an opposing path one level deeper into the opposing side's search space, but the discovered path is still a solution path. The algorithm retains only the set of minimum-length solution paths. Normally, by the time a thread has reached level N+1 in the search from its dequeued state-to-expand's origin, all other threads have dequeued all level N states from the work queue, and they will complete expansion of those N-level states before checking the *isdone* flag set by the first solution's discovery. Some of those N level expansions may be redundant with the N+1 level solution from the thread that "got ahead." The algorithm discards such redundant solutions.

There is still a potential problem, although we have not seen it in practice. The algorithm of Listing 2 makes it possible for some advanced state-to-expand to be multiple frontier levels ahead of other states being expanded in the same direction. If the thread that is expanding a level N+2 (or higher) state sets the *isdone* flag while other level N states are being expanded in the same direction, then some solutions could be missed in an exhaustive search for all distinct minimum-length paths. The fix is to discard a state-to-expand after a solution has been found, if the state-to-expand has a path length greater than the integer ceiling of ½ of the known solution's length, setting the *isdone* flag at that point. The length of the first known solution helps to prune state expansions. At the point that the work queue becomes empty after the *isdone* flag is set to True, worker threads can terminate their work. Of course, a thread may detect this condition and terminate just before another thread enqueues a state-to-expand, but at that point processing is converging on the last of the solutions, and the thread that enqueued the state-to-expand is guaranteed to be available to dequeue that state-to-expand, if no other thread gets there first. States-to-expand in possible solution paths will not be left in the work queue by all terminating threads, and detection of the *isdone* flag in combination with an empty work queue indicates convergence on the last of the solution paths.

One final problem with the fact that some states-to-expand may get multiple levels ahead of most states being expanded in a direction is the fact that these states may not lie in a solution path. The advanced state has gotten outside of the intersecting cones of the frontiers of Figure 1. This problem is a small efficiency concern, not a bug. There may be some unnecessary searching outside the intersecting frontiers of Figure 1, but the performance impact is insignificant compared to the benefits of the minimum-blocking algorithm. Exploration of such states does not lead to false solutions or premature termination.

3. Shortest path performance

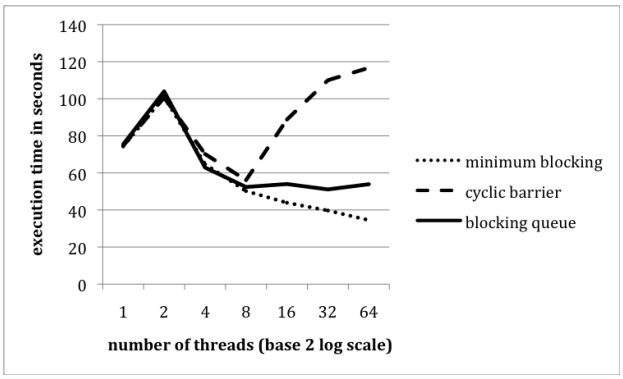
For the performance measurements of this section we ran two representative applications of bidirectional search. The first finds the solution of the so-called "Penny-Dime problem," where there is an arrangement of some number N of pennies P, followed by one blank space, followed the same number N of dimes D. The goal is to find the series of moves that will reverse a sequence such as

PPPPPP_DDDDDD to the sequence DDDDDD_PPPPPP. Legal moves consist of moving a coin one location into the space, or jumping a coin over a single neighbor (as in checkers) to the space. Heuristics such as avoiding retrograde moves can accelerate the search, but the overall form of the algorithm remains unchanged.

The other benchmark, which is the one reported here, is a simplification of a maze construction problem. The original algorithm searches for non-shortest paths that match some minimum-length threshold, in the interest of constructing interesting mazes.

For the benchmark reported here, we use an algorithm that simply searches for the shortest path from the maze entrance to its exit by crossing empty space with no obstacles other than outside walls. When the program starts, there is a pseudo-randomly selected entrance location, exit location, and outer walls, but the inside of the proposed maze is empty at that point. The algorithm simply finds the shortest path between two points, where the search from the entrance does not have knowledge of the location of the exit, and the search from the exit does not have knowledge of the location of the entrance. This is essentially blind search.

The machine used for this benchmark is a 64-threaded Sparc server obtained via a 2009 grant from Sun Microsystems [9]. It houses 8 cores x 8 threads-per-core = 64 hardware threads, 16 Gbytes of main memory, and a 1.2 Ghz clock speed. Each core is limited to a rather small 16 Kbytes of L1 instruction cache and 8 Kbytes of L1 data cache, with 4 Mbytes of L2 cache distributed across the cores. While not the best fit for large data sets with random access patterns due to the limited cache, it performs admirably when running Java programs with good memory locality or modest memory consumption. The individual states of the bidirectional search problems that we have benchmarked are small, although references to a large number of these small state objects can reside in the work queue and sets of the algorithm.

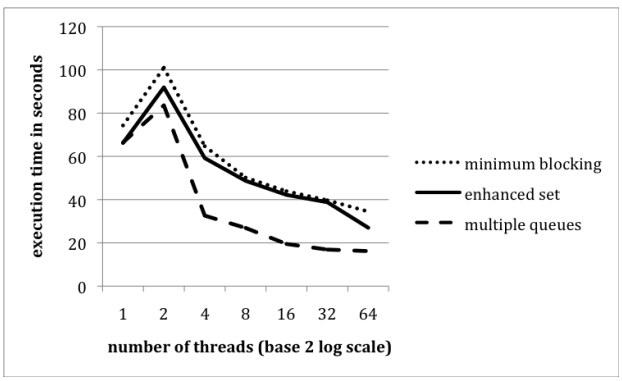


Graph 1: Multithreaded bidirectional maze construction

Graph 1 plots execution time in real seconds on the Y axis as a function of number of threads on the logarithmic X axis for construction of a minimal path of length 2947 in a 2500 x 2500 space using blind bidirectional search. The dashed *cyclic barrier* curve of Listing 1 reaches its peak performance at 8 threads (56.1 seconds compared to 74.7 seconds of its single-threaded case), after which execution time grows with the number of hardware threads employed. Normally adding threads beyond some optimal spot for an algorithm increases start-up and scheduling overhead, although in this case scheduling overhead is minimal because all 64 hardware threads are available for execution. The problem here is that when one or two threads lag behind the others in the exploration of a frontier, the remaining 30-to-31 or 62-to-63 threads block idly in the cyclic barrier until those one or two threads enter the barrier. With only 8 hardware threads employed, each thread has more states to expand for a given frontier, and the cost of exploration averages more evenly across the threads, so there are fewer opportunities for entering this degenerate state repeatedly. More threads wait repeatedly until time-consuming laggards complete their work in the 64-threaded case. In the 8-threaded case the per-thread workload averages out, minimizing the stalling effect of the cyclic barrier.

The dotted *minimum blocking* curve of Listing 2 starts at 74.3 seconds, shows a loss of performance due to two-threaded contention at 101 seconds, after which execution time decreases consistently to 34.5 seconds for 64 threads.

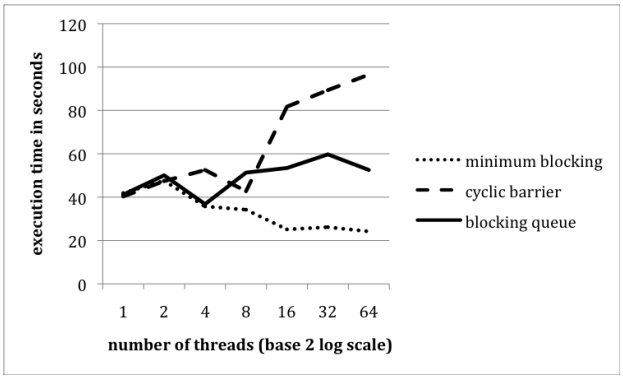
The final, *blocking queue* curve shows the result of replacing the ConcurrentLinkedQueue of *minimum blocking* case with the LinkedBlockingQueue class of the Java library, and using the blocking take() method instead of the non-blocking poll() method for dequeueing states-to-expand. Replacing queue polling with blocked waiting increases processing and scheduling overhead by the Java Virtual Machine and operating system. Polling, even when its finds no work in the queue, is better for this application because it avoids calls into the operating system. Given the exponential growth in number of states to be searched, most dequeue calls for LinkedBlockQueue do not block, yet the cost of calling dequeue methods that must acquire and release locks is clear from looking at Graph 1. After starting at 75.4 seconds for the single-threaded case and then rising to 104 seconds for the double-threaded case, the blocking queue approach drops to 52.4 seconds at 8 threads and then essentially levels off.



Graph 2: Improved nonblocking data structures

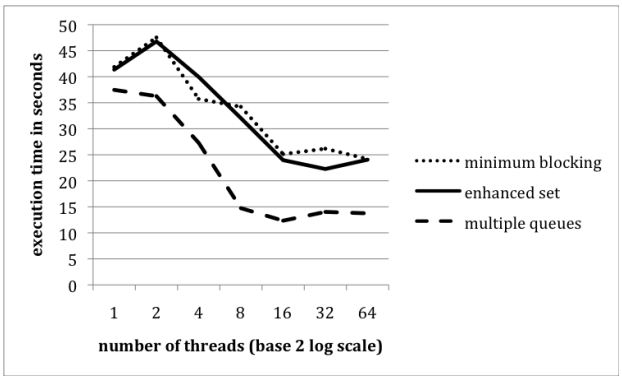
Because the overall halving of execution time in going from 1 thread to 64 threads for the *minimum blocking* approach is disappointing, we decided to attempt to tune the thread-safe data structures to get additional gains. Graph 2 shows the results. The dotted *minimum blocking* curve is the same as in Graph 1. The solid *enhanced set* curve shows the modest gains resulting from initializing the *StatesSet* implemented using ConcurrentHashMap to its known maximum size (4 million elements, determined empirically), reducing its load factor from the default .75 to .5, and increasing its number of lock stripes from the default 16 to 128. Initializing the set size reduces repeated growth overhead. Reducing the load factor reduces hash table collision overhead, and increasing the number of lock stripes reduces contention among concurrent writing threads.

More substantial gains come with the *multiple queues* test case that allocates one work queue per worker thread. Whenever a worker thread is about to enqueue a new state-to-expand, it increments an atomic integer and uses that value as an index to a thread-specific queue. The round-robin nature of enqueuing reduces the probability of thread contention for enqueuing, because a given queue will have its enqueue operation invoked only once for every T enqueue operations, where T is the number of worker threads. A given queue will have dequeue invoked only by its worker thread, eliminating dequeue contention entirely. This *multiple queues* approach bottoms out at an execution time of 16.2 seconds for the 64-threaded case, as compared with 27 seconds of the *enhanced set* approach and the 34.5 seconds of the basic *minimum blocking* approach at 64 threads. The multiple queue approach basically yields a second doubling of performance from its 66.4 second starting point when compared to the other approaches of Graph 2.



Graph 3: Graph 1 benchmarks on a 16-threaded Opteron

Graphs 3 and 4 repeat the benchmarks of Graphs 1 and 2 on a 16-threaded AMD Opteron server also obtained via a grant from Sun Microsystems. The server houses 8 cores x 2 threads-per-core = 16 hardware threads, 32 Gbytes of main memory, and a 2.7 Ghz clock speed. Each core has a substantial 128 Kbyte L1 cache and a 1 Mbyte L2 cache.



Graph 4: Graph 2 benchmarks on a 16-threaded Opteron

Differences in machine architectures such as instruction sets or cache sizes can make substantial differences in performance curves, but in this case the curves of Graphs 3 and 4 repeat the dynamics of Graphs 1 and 2.

4. Conclusions and future work

Bidirectional search is amenable to minimum blocking implementation using immutable state objects, non-blocking queuing of states-to-expand, non-blocking set membership tests in checking for cycles, converging DAG paths and solutions, and parallel lock stripes for updating sets. Java's ConcurrentLinkedQueue and ConcurrentHashMap library classes are excellent matches for the queue and set data structures required by this approach. Adjusting initial set size, load factor and lock striping can contribute modest performance enhancement. Replacing a single non-blocking work queue with multiple work queues that are written by all threads in round-robin order, and that are read respectively by only a single worker thread, leads to a second doubling of performance over the basic minimum blocking approach.

We anticipate porting this work to a NVIDIA Tesla graphical processing unit (GPU) in search of further performance gains. Initial study indicates that a heterogeneous MIMD CPU / GPU approach may be effective. A multithreaded CPU can construct multiple queues for GPU processing elements, one per element, during a CPU phase, along with building the required sets in GPU global, read-only memory. During a GPU phase, the processing elements drain their respective queues, discard cyclic and converging-path states, and send queue and set updates back to the CPU phase. The CPU can update set membership in global, read-only memory incrementally, redistribute the queues, and resume the GPU phase for the next round of state expansion. There are many details remaining to be ironed out in this basic plan.

5. Acknowledgements

This work was made possible by the generous grant of three multiprocessor servers from Sun Microsystems in 2009. A curriculum development grant from the PA State

System of Higher Education supported the initial round of designing benchmarks and course projects such as the one described here. A second curriculum development grant from Intel's *Parallelism in the Classroom* program is making ongoing extension of such materials possible. Finally, an equipment grant from NVIDIA makes the exploration of a Tesla GPU implementation of this work possible.

6. References

[1] Goetz, Brian, et. al., *Java Concurrency in Practice*, Addison-Wesley, 2006.

[2] I. Pohl, "Bi-Directional Search", Machine Intelligence, 1971, pp. 127-140.

[3] A. Toptsis, R. A. Chaturvedi, and A. Feroze, "Kohonen-guided Parallel Bidirectional Voronoi-assisted Heuristic Search," *International Journal of Advanced Science and Technology* Vol. 5, April, 2009.

[4] P.C. Nelson, "Parallel Bidirectional Search Using Multi-Dimensional Heuristics", Ph.D. Dissertation, Northwestern University, Evanston, Illinois, June 1998

[5] P.C. Nelson, and A.A. Toptsis, "Unidirectional and Bidirectional Search Algorithms", *IEEE Software*, Vol. 9, No. 2, March 1992, pp. 77-83.

[6] Oracle Corporation, documentation for classes in package javat.util.concurrent, http://docs.oracle.com/javase/6/docs/api/index.html, March 2012.

[7] M. M. Michael and M. L. Scott, "Simple, Fast and Practical Non-blocking and Blocking Concurrent Queue Algorithms," *Proceedings of Fifteenth ACM Symposium on Principles of Distributed Computing* (PODC '96), Philadelphia, PA, 1996.

[8] Herlihy and Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.

[9] Fujitsu, Sparc Enterprise T5120, T5220, T5140 and T5240 Server Architecture, http://www.fujitsu.com/downloads/SPARCE/whitepapers/T5x20-T5x40-wp-e-200907.pdf, July 2009.