Minimum-Blocking Parallel Bidirectional Search in Java, C++11 and Tesla-Fermi Cuda

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

Abstract

The present work investigates using a minimum-blocking dataflow software architecture as the basis for improving performance of parallel bidirectional search on multiple-instruction multiple-data (MIMD) and single-instruction multiple-thread (SIMT) multiprocessors. The approach represents individual states as minimum-size, immutable objects. It uses work queues to distribute states-for-expansion among worker threads, and it uses sets to keep track of states previously explored in each direction. The Java design uses a non-blocking queue class and a minimum-blocking set class from the Java library, while the C++11 design uses custom non-blocking classes built atop atomic operations recently added to the language. The SIMT, C++ / Cuda design takes a heterogeneous map / reduce approach that expands states in parallel on a Tesla-Fermi graphical processing unit (GPU), and then eliminates dead states, cycles, and detects solutions on a multicore MIMD host processor. Rather than step worker threads through state transitions using blocking synchronization, these designs flow states to be expanded to worker threads in the order required by bidirectional search. Topics include engineering steps taken in migrating search algorithm design from Java through C++11 to Cuda, along with examination of the impact of each step on performance.

Keywords

bidirectional search, C++11, Cuda, Java, multiprocessing, parallel programming

1. Introduction and related work

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 [1]. 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, but it is not required to do so.

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 bidirectional 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 in some applications) within the states of its 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 that is keyed on location in the space + search direction, or two sets that are 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 [2-4]. 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.

The present work is an outgrowth of curriculum development for a senior and graduate level course in MIMD parallel programming with Java [5, 6]. This work extends that by exploring the software engineering steps taken in migrating parallel bidirectional search from Java to C++11 and Cuda and by examining the performance results.

2. Minimum-Blocking Approach in Java

The initial, Java-based solution to parallel bidirectional search uses the algorithm of Listing 1, which implements a dataflow architecture that routes states-to-be-expanded to worker threads. Each immutable state object contains its internal state fields and an immutable reference to its predecessor in its search path.

Insertion of a state into the work queue and retrieval 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. Those threads resort to a polling loop, trying the queue repeatedly until they receive a state to expand. Idle polling consumes processors only until the work queue begins to grow at an exponential rate. The Java 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.” [7, 8]

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 [9]. When two writers do not collide on the same stripe, they do not impede each other via blocking access to a shared lock.

Application programmers can adjust the number of stripes, trading increased parallelism against the memory cost of maintaining additional lock stripes.

A change of the frontier direction in the algorithm of Listing 1 does not block until other threads have completed expansion of the current direction, forward or backward. An earlier design used the CyclicBarrier class from java.util.concurrent to restrict worker threads to expanding states in one direction at a time, but this state machine-oriented restriction caused unnecessary coarse-grain synchronization delays between threads [6]. The dataflow algorithm of Listing 1 interleaves worker thread expansion of forward and backward states. 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 stores only the set of minimum-length solution paths in the set of solutions. 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.

The algorithm of Listing 1 may make it possible for some advanced states-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.

3. Shortest path performance for Java[1]

3.1 The basic Java implementation

Performance measurement uses a representative puzzle-solving application of bidirectional search, that of finding the solution of the so-called "Penny-Dime puzzle," 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 is typical for bidirectional search. Utilizing application-specific patterns of coin movement can reduce the time complexity of the search to a polynomial-time problem, but the benchmark results reported here do not take this pattern-directed approach. Each state expansion consists of one of up to four legal moves, while avoiding retrograde moves consisting of movement of a penny P to the left or a dime D to the right. The implementation measured here actually expands a state pair at each expansion step, where a state pair consists of a forward state expansion and its mirror-symmetric backward state expansion. Expanding a mirror-symmetric state pair in one step cuts down on redundant computational cost. Bidirectional search problems that exhibit mirror symmetries in forward and backward expansion paths can utilize this improvement, while others cannot.

The total number of possible states for a puzzle of an odd number **p** positions, where one position is blank, **(p-1)/2** positions hold pennies and the same number of positions hold dimes, is

$$\mathbf{p * ((p - 1)! / (((p - 1) / 2)! * ((p - 1) / 2)!)) = p * ((p - 1)! / (((p - 1) / 2)!^2))}$$

The sub-equation **((p – 1)! / (((p – 1) / 2)! * ((p – 1) / 2)!))** is simply the equation for the combination of **n** things taken **r** at a time = **n! / ((n – r)! * r!)** [10], with **n = p – 1** (the number of non-blank locations) and **r = (p – 1) / 2** (the number of locations occupied by a coin of one type). Since **n – r = r** for this puzzle – there is an equal number of pennies and dimes – this sub-equation reduces to $\mathbf{((p - 1)! / (((p - 1) / 2)!^2))}$. The leading "**p** *" multiplier accounts for all possible locations of the blank. For example, a 3-position puzzle has 6 states, a 5-position puzzle has 30 states, a 7-position puzzle has 140 states, and 9-position puzzle has 630 states. Benchmarks considered in this report extend to 49-position puzzles with 1,580,132,580,471,900 possible states and 51-position puzzle with 6,446,940,928,325,352 possible states.

Bidirectional search is involved in exploring only the states within the intersecting *search cones* of Figure 1 in order to find solution paths. In principle the branching factor **b** of the search time and space complexity $\mathbf{O(b^{d/2})}$ as previously discussed is **b = 4**, which is the largest number of legal expansions of a given state using the rules of the puzzle. Eliminating "undo moves" that cancel out the most recent state expansion leading to the current state being expanded reduces the branching factor for the puzzle to **b = 3** for all but the first moves. Eliminating all retrograde moves including "undo moves" and eliminating all cycles for a given direction reduces the effective branching factor further. Examination of empirical growth in the number of states for this puzzle uncovers a data-fitting growth formula of $\mathbf{2^{(p-5)/2+6}}$ for **p** positions, with solution path length $\mathbf{d = ((p + 1) / 2)^2 - 1}$, where **d+1** gives the total number of states along a solution path that include the start and end states. Thus, while the solution path length grows with the square of the puzzle size **p**, pruned search yields a branching factor **b = 2** with an exponent that grows as a linear function of puzzle size **p**.

The machine used for benchmarks is a 3.6 GHz, 16-threaded Intel PC running a 64-bit Linux kernel and Java Version 1.6.0_22. It houses two Intel Xeon x5687 processor packages with four dual-threaded cores in each, yielding 8 cores x 2 threads = 16 threads of execution. Each of the two

[1] All benchmark code reported in this paper is available under the *Intel Academic Community Educational Exchange*, http://software.intel.com/en-us/courseware/, credited to the author, entitled "Parallel Programming in Java," beginning in autumn 2012.

packages has 12 Mbyte of L2 cache. The individual states of the Penny-Dime puzzle are small, although references to a large number of these small state objects reside in the work queue(s) and sets of the algorithm, with roughly half in the queue(s) and half in the sets for **b = 2**. All benchmarks use variations of the minimal-blocking algorithm of Listing 1, without the bottleneck created by the blocking CyclicBarrier of Listing 1 given in an earlier publication [6]. Each implementation of the benchmark searches using 16 software threads unless otherwise noted.

Graph 1 shows execution time in real seconds as a function of the number of worker threads employed, seen on the logarithmic X axis, for four different container class combinations on a puzzle with 29 positions (14 pennies, 14 dimes and 1 blank yielding 262,144 states). The fastest, labeled *libsetq*, uses Java's ConcurrentLinkedQueue and ConcurrentHashSet as previously outlined. A second combination, labeled *libsetmultiq*, uses multiple ConcurrentLinkedQueue objects, one per worker thread. Each worker thread reads only its own work queue of states-to-expand. All threads use a single, shared atomic counter to determine the next queue to write in circular, round-robin order. This multiple queue approach eliminates thread contention for reading a work queue and it minimizes the probability of write contention by spacing consecutive writes to a given queue as far apart as possible in relation to the other queues. Using multiple queues yields substantial performance improvement for some applications, e.g., a 2x improvement as reported in [6]. However, Graph 1 shows that the Penny-Dime problem does not benefit from this approach. The *libsetmultiq* performance curve is marginally worse than the basic *libsetq* approach, converging with the latter at 16 threads. This lack of improvement illustrates the fact that contention for access to the queue is not a bottleneck in this application, while the cost of accessing multiple queues and using an atomic, round-robin write index has a small detrimental effect.

The two remaining curves of Graph 1, labeled *mysetq* and *mysetmultiq*, show results for using the corresponding single-queue and multiple-queue approaches with custom, lock-free queue and set container classes, named MultiCircularQueue and EmbeddedHashSet, built atop Java atomic reference variables [5]. Building and using these custom classes is primarily a step in prototyping their implementation and application for later C++11 and Cuda rewrites of bidirectional search, because these C++ platforms do not have counterparts of Java's ConcurrentLinkedQueue and ConcurrentHashMap classes.

MultiCircularQueue takes the basic *circular buffer* approach to implementing a FIFO [11] of references to state objects while using atomic operations available in Java as well as C++11 and Cuda GPU 2.0 devices to serialize thread access separately at each end of the a queue. Figure 2 illustrates the approach. To enqueue a reference to a state object, a thread atomically compares-and-sets an object reference in place of a null reference at the rear of the queue, where *rear*, *front* and *size* are atomic integer indices. In the case where the preceding atomic reference was in fact null, the thread essentially locks the queue for insertion until it updates the *size* variable, and then the *rear* variable to the next null location in the queue, unlocking the queue. In the case where a thread unsuccessfully attempts to atomically compare-and-set a non-null reference because the queue is currently locked for enqueuing, it either spins until the writing thread releases the spin lock for the single-queue case, or it goes onto the next queue for the multiple-queue case. Dequeuing is similar, using compare-and-set to conditionally retrieve a non-null value, and replace it with a null value, thereby obtaining the *front* spin lock. After completion the reading thread updates *size* and then *front* to index a non-null location, thereby releasing the spin lock. If the queue is empty, the reading thread returns a null reference, similar to using ConcurrentLinkedQueue's *poll* method.

Serialization of thread access within EmbeddedHashSet is simpler. Instead of taking the now-common approach of using a *chaining hash table*, where each bucket in the hash table stores a pointer to a list of elements that happen to collide on the hash function, the present work uses an *open addressing hash table* that follows a hashing collision by rehashing to a different location in the table, repeatedly rehashing if necessary until finding either the search element or a null reference [12]. EmbeddedHashSet stores either a null value or a reference to a valid state object as an atomic value at each location in the table. Key-based retrieval, where the key in this search application is a search state, uses atomic retrieval until finding either the desired state or a null reference, signifying end of search. Key-based insertion uses an atomic compare-and-set operation to replace a null value in the table with a reference to the inserted state value. When a hash bucket's reference is non-null, either it is the searched-for key, in which case the search has terminated by finding a previously stored state object reference, or it is not, in which case search proceeds via rehashing. Listing 2 gives the concise Java code for the core of the table search function, where variable **e** of generic reference class **E** is the subject of search, **table** is the open address hash table of reference type E, and **result** is a two-element integer array used to return the bucket number in element 0 and the search status in element 1 to the calling *add*, *contains* or *get* method.

Well-known problems with efficiency of deletion of elements from an open addressing hash table [13] do not affect the present approach to bidirectional search because the algorithm of Listing 1 never deletes a state reference from a hash table. EmbeddedHashSet supports only insertion and lookup. Open addressing avoids dynamic memory management overhead and synchronization bottlenecks when the size of the table can be constrained in advance, as well as avoiding multiple memory accesses entailed by traversal of table + list data structures. These properties make it a reasonable fit for embedded processors with poor dynamic memory management behavior such as network processors [14, 15] and graphical processing units [16]. The rehashing approach of Listing 2 is to add an odd prime number **prm** to a colliding bucket number, where hashing computes an index into an array of odd prime numbers using a second hash function that differs from the primary hash function, and a prime number **prm** retrieved from the table permutes an N-sized table in at most N steps when N is a power of two. Using two distinct hash functions for primary hashing versus rehashing along with the pseudo-random sequencing of key-selected primes for permuting the table after collisions avoids common clustering problems in the table associated with colliding states / keys [11-13], as long as the hash functions are good. Ensuring a good hash function is a matter for the next section.

Open addressing for bidirectional search is preferable to cuckoo hashing adapted to graphical processors because of design simplicity and fewer memory accesses when using a good hash function [17]. Alcantara's 2011 Ph.D. dissertation [18] investigates an atomic approach to open addressing similar to the one used here [15], listing three problems with open addressing as compared with the adapted cuckoo approach. First, "Performance drops significantly for compact tables," when the *density* (ratio of occupied buckets to total buckets) exceeds 67%. The present work sizes tables so that they never exceed 50% density, trading memory size and spatial locality in order to minimize the number of memory accesses for a good hash function. Second, "High variability in probe sequence lengths" can lead to performance problems. Analysis of the Penny-Dime hashing function summarized below has essentially eliminated this variability. Finally, "Removing items from the table is not straightforward." As noted above, bidirectional search as given in Listing 1 does not remove state object references from the forward-path and backward-path hash sets after insertion. Bidirectional search as given in Listing 1 is not specific to the Penny-Dime puzzle. It works for any application of bidirectional search. Element removal is unnecessary.

To complete the discussion of Graph 1, *mysetq* and *mysetmultiq* that use custom classes MultiCircularQueue and EmbeddedHashSet, both based on non-blocking atomic operations, perform marginally worse here than *libsetq* and *libsetmultiq* based on Java's ConcurrentLinkedQueue and ConcurrentHashmap. Graph 2 measures the same four programs by varying the number of coin positions **p** from 25 through 33 while keeping the number of threads fixed at 16. The relative effectiveness of the approaches is similar, with the cost of multiple queues outweighing the benefits, and with time growing at somewhat more than the doubling rate in the number of states per coin-slot-pair given by $\mathbf{2^{(p-5)/2+6}}$. The fixed terms in execution time as well as inter-thread contention for the memory manager and for hash table access account for some of the steeper growth rate in time. A benchmark examined below uncovers an additional source of overhead in this growth rate.

The marginal differences in performance of configurations in Graphs 1 and 2 is less important than the fact that MultiCircularQueue and EmbeddedHashSet represent demonstrably viable designs for C++11, which offers no thread-safe queue and hash set library classes, as well as for Cuda, which offers support for atomic operations in newer devices but not for lock-based thread-safe queues and hash sets.

3.2 The effects of a better application hash function

Given the promising look of performance in Graphs 1 and 2, the next step in this process is to attempt to improve the hash function for state objects. Listing 3 gives the initial, naïve hash function for a state object stored in the hash table, where *initOpen* is the position of the blank coin slot. The goal when coding this initial version was to get something working. This hash function computes a 32-bit integer hashcode value and stores it in an immutable field in each application state object. The library and custom hash table classes fold the upper bits that are outside the range of their table indices into the lower bits using the exclusive-or operator, and then use the resulting integer as a bucket number.

The function of Listing 3 uses the integer encoding of each penny, dime, and the blank space stored in the byte-valued *initVector* array as a multiplier that is weighted by its position in this array. The State class stores coins and the blank as integers in an array, one position per element. The intent was to use the distinct integer value of each coin type and blank, coupled with the unique weight of its position, to create a hash function capable of distinguishing among state configurations. There are two obvious problems with this hash function at the outset. First, no account is made for the likelihood that some bits contributed by the positioned coins will overflow the 32 bits of the hash code. Second, the encodings for a penny and a dime do not permute the space of potential encodings, making this function likely to result in clustering of hash values.

Instrumenting the code for class EmbeddedHashSet reveals the glaring deficiency of this hash function. A test run of the *mysetq* configuration for 33 coin positions and 16 threads, corresponding to the rightmost measurement of Graph 2, shows that 1,463,731 initial probes into the 474,204 element forward-path hash table led to 821,859,095 subsequent rehashes on collisions, and that 960,974 initial probes into the 474,204 element backward-path hash table led to 541,068,155 subsequent rehashes on collisions, giving the ratio of rehash probes to initial hash probes of about 562 to 1.

Listing 4 shows the final hashing function used after analysis and some experimentation. This function exploits the fact that the positions of the dimes and the blank space, or equivalently the pennies and the blank space, fully determine the state. That is the reason that this hash function inspects only dimes and the blank position. Its use of the "<< 6" operation ensures that it distributes cumulative coin bits across the hashcode, while its simultaneous use of the unshifted cumulative coin bits ensures that it loses no bits of information. It encodes the blank, open position differently in order to differentiate it from a coin. This hash function is loosely based on polynomial arithmetic as implemented for Ethernet CRC formation – "Thus we see that CRC arithmetic is primarily about XORing particular values at various shifting offsets." [19] – with exclusive OR providing the no-carry add operation.

A test run of the *mysetq* configuration for 33 coin positions and 16 threads using this final hash function shows that 1,468,117 initial probes into the forward-path hash table led to 574,405 subsequent rehashes on collisions, and that 963,981 initial probes into the backward-path hash table led to 381,319 subsequent rehashes on collisions, giving the ratio of rehash probes to initial hash probes of about 0.39 to 1, an improvement of 1441x. Tests with various coin position sizes show that this ratio of rehashes to initial hashes never exceeds 0.5. At most ½ of all probes into a table result in one rehash on average.

Graph 3 is the counterpart to Graph 2 using this final hash function. Note that Graph 3 starts where Graph 2 ends at 33 coin positions. The elapsed time values for the original hash function for 33 coins in Graph 2 are *libsetq* : 6.745, *libsetmultiq* : 6.095, *mysetq* : 9.701, and *mysetmultiq* : 11.779. The elapsed time values for the final hash function for 33 coins in Graph 3 are *libsetq* : 1.067, *libsetmultiq* : 0.644, *mysetq* : 0.693, and *mysetmultiq* : 0.67. Bidirectional search in Graph 3 does not reach the execution times of Graph 2 until searching 39 or 41 coin positions, depending on the hash class used, after three or four doublings of the search space size. Furthermore, configuration *mysetmultiq* is the winner for large state spaces at the right end of Graph 3. The nonblocking open addressing table in EmbeddedHashSet works better than the write-locking chaining table in ConcurrentHashMap, and giving each worker thread its own MultiCircularQueue works better than sharing a single queue due to eliminated thread contention for the queue.

However, Graphs 4 and 5 reveal that improvements in the final hash function eliminate most benefits of hardware multithreading in this application of bidirectional search. In retrospect, the performance benefits of multithreading appearing in Graph 1 come about because multiple hardware threads are performing independent but mostly useless work in parallel, thanks to the inefficient hash function. The final, effective hash function eliminates fat from parallel execution. The threads in Graphs 4 and 5 have less work to do, and so a more substantial percentage of their time is spent contending for serial resources, primarily for access to dynamic storage allocation for state objects and for access to hash table buckets. The multiple queue configurations perform better than their single queue counterparts for large coin configurations because they eliminate thread contention for access to queues. Library class ConcurrentHashMap in configuration *libsetmultiq* of Graph 5 surpasses atomic-based EmbeddedHashSet of *mysetmultiq* at the 8-thread mark in part because it uses locks for contending writes. In cases where there is large amount of thread contention for shared resources such as the hash tables of Graph 5, locking is better than atomic synchronization because it reduces the amount of noisy, ineffective polling across memory access data paths [5]. Indeed, *libsetmultiq* is the only configuration in Graph 5 that shows monotonic improvement with the number of threads after going past its initial spike in inefficiency at 2 threads.

4. Porting Java bidirectional search to C++11

The next step in this investigation entails porting the Java implementation of *mysetmultiq* to C++11. C++11 does not offer counterparts to the library classes of java.util.concurrent such as ConcurrentLinkedQueue and ConcurrentHashMap. It does offer C++ utility class wrappers for POSIX threads, but more importantly, it adds an explicit memory model for concurrency to the language, along with atomic operation classes in the library [20]. The availability of counterparts to Java's and Tesla-Fermi's atomic operations is the primary benefit of C++11 for this project.

Listing 5 gives the essentially line-for-line translation of Listing 2's open addressing hash table traversal function from Java into C++11 using atomic operations. Graphs 6 and 7 illustrate performance curves for the Java implementation of *mysetmultiq* in relation to its C++11 counterpart. The latter is as exact a translation from Java to C++11 as is possible. It continues to use dynamic storage allocation to allocate state objects. It sizes and allocates arrays in classes MultiCircularQueue and EmbeddedHashSet at startup time. Of course, C++11 does not have Java's garbage collector, and the only significant use of the *delete* operator is in explicit recovery of redundant or ill-formed State objects pruned by worker threads, where an ill-formed State is a State in which the location of the blank space would move outside of the coin position vector, or retrograde motion of a coin would occur. The compiler used is g++ version 4.6.2 with the –O3 optimization level.

Graph 6 shows essentially the same performance for Java and C++11 for a 33-position puzzle, with C++11 monotonic performance improvement through 8 threads. Graph 7 shows that 16-threaded C++11 outperforms Java for larger puzzle configurations.

Graphs 8 and 9 show performance curves for the C++11 *mysetmultiq* implementation of Graphs 6 and 7 compared to the new C++11 *static* configuration. C++11 *static* represents a major step in migrating bidirectional search to Cuda, comprising two substantial changes to C++11 *mysetmultiq*.

First, C++11 *static* uses a Python script that takes the puzzle size and number of threads as input and then generates a configuration header file with the various table sizes and state encodings defined as C++ constant values. The goal is to provide the compiler with as much static information as possible to assist with optimization. Second, the Python script generates a custom definition for *class State* with a different approach to encoding state from the previous Java and C++ configurations. This encoding replaces using a byte array of coin information that stores one byte per coin with a highly compressed array of 32-bit integers that uses one bit per coin and that stores the blank coin location in a separate integer outside the **p**-bit vector. Though bit vector encodings can lead to longer execution times than integer or byte array encodings for some applications, due to the time needed to extract application data from the bit vectors, this is not the case for bidirectional search because the hash table operations that use State objects do not need to extract application interpretations of the integers storing the bit vectors. Once a State object's constructor builds its bit vector and hashcode, hashing can check for bucket collisions using far fewer comparison operations than the byte array approach. A 43 position puzzle, for example, requires up to 43 byte equality comparisons in the *mysetmultiq* configuration, while in static configuration it requires only 3 integer comparisons consisting of two for the 32-bit integers housing the coin vectors and one for the 16-bit open position. Also, the reduction in memory space consumed by State objects is essential for Cuda, both because of device memory limits and because of the high overhead of global memory access. The static configuration continues to use the C++ *new* operator to allocate object space and the *delete* operator to recover redundant and ill-formed State objects. Note that operating system paging is not a concern in the performance graphs examined in this paper because all benchmarks are constructed with puzzle sizes that avoid paging on the lightly loaded benchmark machine. The reduction in time resulting from the reduction in size is not a matter of reduction in paging overhead; it is a matter of a reduction in the number of storage locations that are written during state construction and compared during hash table manipulation.

Graph 8 shows almost monotonic improvement with thread count in the C++11 static configuration, with improvement decreasing after 4 threads and execution time rising slightly after 8. Reduction in inter-thread contention within hash table lookups and within the memory allocator are the primary causes for improved leverage for multithreading. Graph 9 shows continuing improvement in the growth in execution time as a function of the puzzle size. In fact, Graph 9 is the first graph that shows an approximate doubling of time with each step increase in puzzle size, with no additional overhead, corresponding to the $\mathbf{2^{(p - 5) / 2 + 6}}$ growth in number of states discussed earlier. The main reason that execution time now achieves this doubling rate is because earlier algorithms increase time twice as a function of puzzle size, once for the doubling in the number of states manipulated, and again for the increase in time needed to manipulate growing byte arrays housing the coin data. The move to heavily compressed bit vectors in C++11 *static* changes State object initialization and comparison from an **O(p)** operation to an **O(1)** operation. All State classes in the 33 through 43-coin positions **p** of Graph 9 use two 32-bit integers to encode coin state and one 16-bit integer to record the open position.

5. Porting C++11 to Cuda

5.1 A homogeneous Cuda solution

The details of NVIDIA's Tesla-Fermi GPU architecture and the accompanying Cuda tools appear in several primary textbooks [16, 21-22], manuals, and numerous conference and journal papers. This section repeats only the essentials for this study of bidirectional search. Figure 3 gives a block diagram for the 1.15 GHz NVIDIA C2070 GPU card used in the study [21]. The C2070 is a so-called Compute 2.0 Tesla-Fermi device. Each Streaming Multiprocessor (SM) consists of 32 hardware threads, comprising a thread *warp* in NVIDIA parlance. A higher level, software construct for thread scheduling is the thread block that can contain more than the 32-threaded warp size, with an application running more than 14 thread blocks if so configured. Effective thread block scheduling typically requires the configuration of more thread blocks than actual hardware SMs, often with more threads-per-block than threads-per-SM. Configuring a high number of thread blocks makes it possible for the run-time scheduler to execute one ready block while others are waiting for memory transfers. Configuring more than 32 threads per block is useful when multiple threads within a block communicate via shared memory. The primary bottleneck in Cuda programs tends to be memory access. The first set of GPU benchmark results presented here has the number of thread blocks equal to the number of SMs (14) and the number of threads per block equal to the number of hardware threads per SM (32). Increasing these parameters did not improve performance for this memory-bound application.

Threads execute within a warp as a Single Instruction Multiple Data (SIMD) stream, accessing parallel data locations, and stalling during the time that conditional execution paths diverge. Distinct SMs can execute different portions of a program in parallel, leading to the characterization of the GPU as a Single Instruction Multiple Thread (SIMT) device. The SIMT C2070 is a collection of 14 SIMD devices.

This study has examined numerous configurations of the C2070 device and the Cuda code. Graph 10 gives results for the two most notable configurations, using version 4.1 of the Cuda software tools. The limit on device memory size limits the number of states that can be allocated, hence the limits of 23 and 33 coin positions in Graph 10. The slower *heap* version uses the C++ *new* and *delete* operators for State object allocation and recovery as used in the C++11 benchmarks. The faster *no heap* version allocates the total space for State objects before beginning the search. Individual State object allocation obtains storage from a given thread's pool of State object storage, and the search algorithm never deletes a redundant or ill-formed State object. A searching thread reuses the storage from any such object in its allocation of the next expanded State object. Allocation of fixed-size State objects from a statically allocated pool is faster because it is a simple, monotonic storage allocator that moves forward in a pool at each request, it avoids any garbage collection or object recovery overhead, it avoids thread contention for a shared memory pool, and it conserves space by avoiding maintaining storage management data in regions adjacent to allocated memory objects. The benefits are the substantial performance enhancements in speed and available puzzle size illustrated in Graph 10 in going from the *heap* to *no heap* approach.

The homogeneous GPU implementation of Graph 10 does not compare well with the Java and C++11 implementations on the 16-threaded Intel processor. Execution times for 33 coins are Cuda *no heap* : 4.92 seconds, C++11 *static* : 0.35 second, C++ *dynamic* : 0.68 second, and Java *mysetmultiq* : 0.67 second. So-called GPU occupancy, the ratio of time spent computing within GPU threads to available time, is a low 16.7% for test runs of various puzzle sizes. The 33 coin test run shows a ratio of thread-divergent program branches to total branches of 5,938,364 / 40,565,579 = 0.15, which is a fairly good number. Most state expansions are not redundant or ill formed, and most hash table lookups complete with a single probe of the table.

Spatial and temporal locality of MultiCircularQueue manipulation in the homogeneous Cuda implementation is good, making somewhat effective use of the L1 and L2 caches. As Figure 2 illustrates, pointers to State objects that are adjacent in a circular buffer FIFO reside in adjacent memory locations. The Cuda program makes its only use of *shared memory* among multiple threads in a thread block by synchronizing reading and writing of the work queues so that all threads in a block read their respective states to expand and write their expanded states in parallel. More aggressive use of shared memory to stage states-to-expand in shared memory arrays resulted in small performance degradation because of the overhead of copying data from L2 cache to shared memory and thence to thread registers. Configuring the available 64K of high-speed intra-SM memory as 48K L1 cache and 16K shared memory works better than the alternative of explicitly staging data in shared memory. The only worthwhile uses of shared memory in this program are for synchronization of parallel initialization of data arrays at startup, and for synchronization of reading and writing adjacent entries in a MultiCircularQueue's array by multiple threads in that queue's thread block.

Hash table access is the primary cause of poor GPU performance when compared with the 16-threaded CPU. Good, non-colliding, non-clustering hash table access exhibits poor spatial locality, substantially reducing the benefits of caching. This study avoided using cuckoo hashing that might increase spatial locality slightly because of the costs of additional hash table and related memory accesses [18]. An investigation in the Java program concerning caching hash table entries for this application reveals that this puzzle application of bidirectional search exhibits poor temporal locality. Temporally proximate interactions with the hash table show poor utilization of a simulated table entry cache in Java. In addition to poor spatial and temporal locality of hash table accesses, there are simply a high number of accesses to main memory that do not result in multiple uses of the data once it arrives at GPU registers. GPU devices work best with coalesced reads and writes of adjacent memory locations and multiple computations using data after those data arrive in registers. Interactions with the state queue approach this ideal, but interactions with the hash tables do not. The non-local, non-coalesced, high volume nature of hash table-resident data, and the relatively low amount of reuse of data in registers, are the primary culprits in the poor performance of Cuda relative to multiprocessor C++11 as seen in Graph 10.

An additional source of overhead is the fact that use of atomic memory access by highly synchronized threads introduces substantial additional stalls in memory interaction because these threads collide on SIMD atomic operations [23], choking the memory access data paths. MultiCircularQueue avoids this problem by using one thread per thread block that uses atomics to stage coalesced reads and writes and by using a distinct queue for each thread block. Unfortunately, a hash table has exactly the wrong dynamic, applying parallel atomic accesses by all synchronized threads during interaction with the hash table. This observation repeats that made at the end of Section 3 concerning the benefits of *libsetmultiq*'s blocking approach over *mysetmultiq*'s atomic approach for high contentious threads at the right side of Graph 5. Locking performs better than atomic access for highly contentious threads.

5.2 A heterogeneous Cuda / multicore CPU solution

The final architectural approach of this study uses heterogeneous processing, with a state expansion phase taking place on the GPU and with all other work occurring on the 16-threaded Intel CPU. This approach attempts to leverage the relative strengths of the two processing architectures at the cost of copying state data back and forth between the two processors.

Figure 1 provides the basis for understanding the substantially altered approach. Once the search algorithm constructs a non-redundant, well-formed State object, there is never a need to delete that object and recover its storage. Given the fact that every State object must pass through a queue at least one time, it becomes straightforward to convert from a queue of pointers to State objects, to a queue of State objects where the circular queue's array is the State storage arena. In the heterogeneous approach there is one MultiCircularQueue object whose array serves as both the storage arena and the queue. In terms of Figure 1, this algorithm starts out by constructing a single state-pair object containing the initial State and final State objects at location 0 in the queue's array. The multicore CPU (a.k.a. "host") passes a copy of all unexpanded queue State objects to the GPU. The GPU creates an expansion of each State object into four new State objects without checking for redundant States, ill-formed States, or solutions, placing the new State objects in the "next ring" of Figure 1, i.e., by building State objects at the frontier without conventional memory allocation costs. Listing 6 gives GPU kernel function *generateStates* along with the signature for helper function *makeState* that initializes the storage of a new State object at the frontier.

Upon completion of GPU State expansion the program copies the newly constructed frontier back to CPU memory where it uses the multiple CPU threads (16 in this study) to execute parallel functions *workerTaskReduceCompact*, *workerTaskReduceRecord* and *workerTaskReduceDetect*.

Each function is multithreaded, and a custom C++ cyclic barrier class constrains all CPU worker threads to perform the work of one of these functions in tandem with all other threads.

Function *workerTaskReduceCompact* compacts the frontier returned by the GPU to eliminate ill-formed States and redundant States whose references were previously placed in a hash table, performing its work in parallel. The total frontier is divided into sub-regions, evenly among threads, and then each thread compacts its own sub-region by using an algorithm based on the partitioning algorithm of quick sort. Moving one pointer up from the start of its sub-region and another down from the end, a worker thread copies a valid State object down and over an invalid State object whenever the up-moving pointer encounters an invalid State and the down-moving pointer encounters a valid State. The thread iterates until the pointers pass each other, and then waits in a cyclic barrier until all threads complete this phase of *workerTaskReduceCompact*. Each thread records the final boundary between its valid State objects and the start of undefined, tail-end storage in a per-sub-region data structure. Once this initial sub-region compaction is complete, *workerTaskReduceCompact* employs up to half of the active threads to merge valid States from the highest regions into the lowest regions in which space is available. This phase of *workerTaskReduceCompact* is inspired by merge sort, since it is basically a merge of "sorted" data, where the sort values consist of valid State objects followed by invalid State object storage. It uses the highly efficient *memcpy* library function to copy all appropriate, valid State objects in one call per thread. After each phase of coarse-grain merger, *workerTaskReduceCompact* sets the number of threads employed to the number of sub-regions still requiring State objects from above. Upon completion *workerTaskReduceCompact* will have merged all valid State objects into the bottom of the frontier; subsequent queue array space is available for new State construction. Like quick sort and merge sort, the compaction algorithm is O(n log(n)) on the size of the frontier being compacted, with hash table lookup required only during the quick sort-inspired phase.

After completion of *workerTaskReduceCompact*, all worker threads enter function *workerTaskReduceRecord* to record all states in the hash tables. Listing 7 gives the C++ code for this function. It is important to note that, since Cuda 4.1 supports g++ 4.4 but not g++ 4.6, and that the former does not support C++ atomics, it was necessary to recode class *EmbeddedHashSet* to use POSIX mutex locks instead of atomics [24]. Reversion to locks is beneficial because *workerTaskReduceCompact* consists almost entirely of writing pointers into the two hash tables via method EmbeddedHashSet::add of Listing 7. Locks cut down on memory data path contention when many threads are accessing shared data paths heavily. In addition to using lock striping to encourage parallelism in hash table updates, the heterogeneous algorithm makes no use of hash table locks or other table synchronization anywhere outside of *workerTaskReduceRecord*. Given the fact that all hash table updates occur from within *workerTaskReduceRecord* – the other host algorithms never mutate the table – it is possible for *workerTaskReduceCompact* and *workerTaskReduceDetect* to read the effectively immutable hash tables while making no use of thread synchronization.

Finally, all threads enter function *workerTaskReduceDetect* of Listing 8 to check the frontier for solutions to the puzzle by inspecting the hash tables. For this coin puzzle it is possible to compute the length of a solution path in advance using the length expression $\mathbf{((p + 1) / 2)^2 - 1}$ given earlier. The heterogeneous host program uses this information to defer invoking *workerTaskReduceDetect* until the paths to the frontier satisfy this length constraint. Detection of solutions in the frontier terminates the algorithm.

Graphs 11 through 13 give the performance curves for various related configurations. C++11 *static* is the earlier, pre-Cuda C++ configuration of Graphs 8 and 9, *hybrid initcpu* is the heterogeneous algorithm just described with parallel initialization of the hash table arrays occurring on the host machine, and *hybrid initgpu* is the heterogeneous algorithm with parallel initialization of the hash table arrays occurring on the GPU. *CPU g++ with locks* is the current algorithm implemented entirely on the CPU using the state expansion algorithm from the GPU. Finally, *hybrid interleaved* is a modification of *hybrid initgpu* that interleaves execution of the CPU and GPU. It expands half of the frontier on the GPU while compacting and recording the previous half on the host CPU; otherwise the heterogeneous algorithm remains the same.

The thread counts refer to CPU threads. Experimentation determined that using 112 thread blocks containing 128 threads each yields the best performance for the GPU on this configuration. GPU occupancy was a consistent value of 66.7% for both array initialization and State generation, a much better value than the 16.7% value for the homogeneous Cuda configuration. Arguably, given the fact that the simple multithreaded, highly parallel hash table initialization loop achieves 66.7% occupancy, that value is probably a valid limit for what state expansion on the GPU can achieve, given the limits on re-use of data in GPU registers.

Graph 11 shows the *CPU-only g++* configuration just beating CPU+GPU variations for 43 coin positions starting at the 4-threaded point. All of the new benchmarks near the bottom of Graph 11 show monotonic improvement with the addition of threads, as do those benchmarks for 49 coin positions in Graph 12. Graphs 11 and 12 also show that for a large search space, when there are only 1 or 2 threads allocated to perform the host CPU's portion of the work, using the GPU for state expansion results in improved performance over the *CPU-only g++* configuration. Graph 13 shows the *CPU-only g++* curve to be the marginal winner when using 16 multiprocessor host threads, with *hybrid initcpu* coming in second on average. The CPU-only curve is the only one capable of solving the 51-position puzzle due to memory limits of the GPU.

The results in Graphs 11 and 12 are of interest for comparing single-threaded or dual-threaded performance on a 3.6 GHz conventional Intel processor using a large cache size to a 1.15 GHz Tesla-Fermi GPU. Using this hybrid configuration of the program to solve the 49-position puzzle, which searches through about 268,435,456 states in the 1,580,132,580,471,900-state space, takes the following number of seconds in the following single-host-threaded configurations: *CPU-only g++*: 57.8; *hybrid initcpu*: 35.6; *hybrid initgpu*: 40.8; and *hybrid interleaved*: 41.5. The numbers for the dual-host-threaded configurations are: *CPU-only g++*: 34.1; *hybrid initcpu*: 27.6; *hybrid initgpu*: 28.7; and *hybrid interleaved*: 26.6. For the single-threaded host the improvement in using heterogeneous CPU+GPU processing over CPU-only processing is (57.8-35.6) / 57.8 = 38.4%. For the dual-threaded host the improvement is (34.1-26.6) / 34.1 = 22.0%. A dual-threaded host is a common host configuration at the time of this study. Furthermore, the retail price of the Intel-based host machine used in these benchmarks was about $7500 when purchased in 2011, while the retail price of the NVIDIA 2070 was about $2500. The improvement in going from 2 to 16 threads on the *CPU-only g++* host configuration is (34.1-12.6) / 34.1 = 63.0%, giving an improvement ratio of about 2.9x in using MIMD CPU acceleration instead of SIMT GPU acceleration for a dual-threaded host, while incurring a roughly 2.14x increase in monetary cost of $7500 for the 16-threaded processor versus $1000 + $2500 for a conventional dual-threaded host + C2070.

This study also investigated using page-locked host memory shared between the CPU and the GPU in the non-interleaved, *hybrid initcpu* solution. Compared to 17.1 seconds for 49 positions using 16 host threads for *hybrid initcpu*, the page-locked implementation requires 190.6 seconds. Clearly, the overhead of fetching and storing individual memory locations of host memory via the host's PCI bus is an order of magnitude worse than the block copy overhead entailed when the GPU uses its own memory.

Making valid comparisons between CPUs and GPUs is difficult. It is easy enough to find benchmark programs that run faster on GPUs than on single-threaded CPUs, especially CPUs with relatively slow clocks. In the interest of comparing GPU performance on this problem to a range of MIMD processor architectures, Graph 14 gives performance for the Cuda *no heap* solution of Graph 10 compared to the Intel *CPU-only g++* curve of Graph 13 (CPU g++ w locks) along with that same CPU-only algorithm implemented on two other multicore machines. The AMD architecture is a Sun x6400 server with a 2.7 GHz, 8 core AMD Opteron 885 processor (16 hardware threads) and 32 GByte of memory with 128 Kbyte of L1 cache and 1 MByte of L2 cache per dual-threaded core. Like the Intel machine, the AMD is rich in cache. The Sparc architecture is a Sun T5120 server with a 1.2 GHz, 8 core Sparc T2 processor (x 8 threads per core = 64 threads) with 16 Kbytes of L1 instruction cache and 8 Kbytes of L1 data cache per 8-threaded core, and 4 MByte of L2 cache distributed among all cores [25]. The Sparc machine is low in L1 and L2 data cache compared to the AMD server. Graph 14 gives the Sparc performance at 16 threads (the fastest Sparc curve), and increasingly slower execution when using 32 and 64 threads. The problem with using more than 16 hardware threads for this Sparc configuration is inter-thread cache contention and memory path contention, making it an excellent platform for comparison with the GPU. At 16 threads there are 2 hardware threads per core contending for limited L1 caches and memory access data paths. There are benchmarks for which the Sparc machine outperforms the AMD machine, and there are benchmarks for which the Sparc machine can utilize all 64 hardware thread effectively. This bidirectional search algorithm is not one of them because the poor memory access locality of the hash tables leads to poor cache utilization and cache contention. Homogeneous Cuda beats the Sparc g++ 32 configuration up to the 27 position puzzle and the Sparc g++ 64 configuration up to the 29 position puzzle.

6. Conclusions and future work

Despite the fact that the GPU offered little to no performance advantage over the available MIMD processors for bidirectional search, due to the substantial number of irregular memory accesses, the steps of refactoring the original Java algorithm to make it possible to run bidirectional search on a GPU has led to a number of important discoveries that help to accelerate any application of bidirectional search.

First, a designer should prototype an application of bidirectional search in a language with adequate library support in order to discover empirical curves in the growth of state space and processing time, as well as solution path length. The discovery of these values makes it possible to configure constant-size data structures that eliminate the expense of dynamic storage allocation and recovery. The prototype need not be multithreaded.

Second, referring to Figure 1 and Listings 6 through 8, the most efficient approach in time and space utilizes the work queue through which every state must flow as the actual storage arena. Figure 1 illustrates monotonically advancing addresses in the queue's linear address space as concentric search frontiers. Bidirectional search allocates copies of the initial and final state objects at the beginning of this linear queue (i.e., at the initial *frontier*), and then performs the following steps cyclically. The best performing test runs in the current study perform the follow steps sequentially, although it is possible to interleave execution of the *generateStates* step with the remaining steps, operating on each half of the frontier concurrently, as outlined for the *hybrid interleaved* configuration above. Internally, each of the following steps admits to a highly parallel implementation.

1. The *generateStates* step takes each of the immutable state objects at the highest current frontier **f** and generates **b** states at the new frontier **f+1** where **b** is the worst-case branching factor. The storage location for a new, expanded object lies at a linear offset within the queue's array. There is no dynamic storage allocation. Expansion takes the form of initialization of state object fields using storage in the pre-allocated queue array. Multiple threads can expand multiple states, indexed using the unique thread number, without synchronization. This step corresponds to Listing 6.
2. The *reduceCompact* step eliminates ill-formed states, which are states that violate state-specific constraints and heuristics, as well as redundant states representing cycles in search paths, using a parallel compaction algorithm that copies well-formed states occurring later in the queue array down to the locations of the ill-formed states. This step first divides frontier **f+1** evenly among the worker threads. Each worker thread compacts its own sub-region using an algorithm similar to quick sort's partitioning algorithm, where all well-formed, non-redundant states must precede all ill-formed or redundant states in the sort of the sub-region. Detection of redundant states requires concurrent reading of the hash sets of previously encountered states in each

direction, but because these hash sets are immutable during this phase, they have no need for synchronized access. Once all worker threads have partitioned their sub-regions, *reduceCompact* cyclically merges well-formed states from higher sub-regions into lower sub-regions by copying well-formed states over ill-formed states via the efficient *memcpy* library function. Each merge phase can utilize as many worker threads as there are pairs of sub-regions to be merged. At the point that there are no more pairs of sub-regions to merge, frontier **f+1** has been compacted.

3. The *reduceRecord* step records references for all states in frontier **f+1** into the appropriate forward or backward hash set. Multiple threads can record all references from frontier **f+1** in parallel, advancing through the frontier using address sequencing that maximizes cache utilization for reading the queue array. Given the fact that all threads are writing to the hash sets in a tight loop, there is a somewhat high probability of thread contention, leading to a preference for hashing using lock stripes over hashing using atomic stripes. This is the only step requiring synchronized access to hash table buckets. This step corresponds to Listing 7.
4. The *reduceDetect* step inspects all states in frontier **f+1** for membership in the hash set coming from the opposite side of the search space. For each such member detected it recovers, records and reports a solution path. As in *reduceCompact*, because the hash sets are immutable during this phase, they have no need for synchronized access. At the end of *reduceDetect*, if it has detected any solutions, then this algorithm terminates. In applications where it is possible to determine an a priori solution path length as a function of search problem parameters, as it is for the Penny-Dime problem, it is possible to avoid the overhead of executing *reduceDetect* until the search path length **f+1 * 2** reaches the required value.

These guidelines apply to any application of bidirectional search, and constitute the primary outcome of this study. They represent a *map / partial reduce* approach in the functional programming sense [26], where *generateStates* maps frontier **f** to frontier **f+1** and the other steps partially reduce **f+1**.

Replacing the hash table with another data structure that implements a set interface such as a sorted tree or a skip list [27] has little promise for GPUs, since traversing linked structures entails multiple, non-coalesced memory accesses. Designing a hash function that increases spatial locality of application-proximate keys in the search space may be possible, but the magnitude of the search problem makes any effective solution on a current GPU architecture doubtful. For a problem with branching factor **b** = 2, half of the explored states reside in the frontier being checked for redundant and solution states.

For bidirectional search problems with substantially more computational overhead in the *generateStates* step than that of Penny-Dime, the *hybrid interleaved* approach still holds promise of leveraging the GPU. One problem with this approach for Penny-Dime is that the *generateStates* step is too cheap, and so the cost of transporting state to the GPU outweighs the benefit. For problems with state expansion that grows in complexity without a commensurate growth in memory size, and where states expansion uses a GPU-friendly model of memory interaction, the *hybrid interleaved* approach may be worth investigating.

The next step in the current study is to investigate the viability of porting search algorithms without the hash set requirements of bidirectional search to heterogeneous MIMD and GPU parallel processing architectures.

7. 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 an initial round of designing benchmarks and course materials. A second curriculum development grant from Intel's *Parallelism in the Classroom* program has made ongoing extension of such materials possible. Finally, an equipment grant from NVIDIA made the exploration of a Tesla GPU implementation of this work possible.

8. References

[1] Pohl I., “Bi-Directional Search,” *Machine Intelligence* (6), 1971, pp. 127-140.
[2] Toptsis A., Chaturvedi R. A., and Feroze A., “Kohonen-guided Parallel Bidirectional Voronoi-assisted Heuristic Search,” *International Journal of Advanced Science and Technology* (5), April, 2009.
[3] Nelson P. C., “Parallel Bidirectional Search Using Multi-Dimensional Heuristics”, Ph.D. Dissertation, Northwestern University, Evanston, Illinois, June 1998.
[4] Nelson P. C., and Toptsis A., “Unidirectional and Bidirectional Search Algorithms”, *IEEE Software* 9(2), March 1992, pp. 77-83
[5] Goetz B, et. al., *Java Concurrency in Practice*, Addison-Wesley, 2006.
[6] Parson D. and Schwesinger D., “Minimum-Blocking Parallel Bidirectional Search,” *Proceedings of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA Technology Press, Las Vegas, NV, July 2012.
[7] Oracle Corporation, documentation for classes in package javat.util.concurrent, http://docs.oracle.com/javase/6/docs/api/index.html, August 2012.
[8] Michael M. M. and Scott M. L., “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.
[9] Herlihy and Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.
[10] Johnsonbaugh R., *Discrete Mathematics*, Second Edition, New York, NY: Macmillan, 1990.
[11] Cormen T., et. al., *Introduction to Algorithms*, Third Edition, Cambridge, MA: MIT Press, 2009.
[12] Knuth, D., *The Art of Computer Programming, Volume 3, Search and Sorting*, Reading, MA: Addison-Wesley, 1973.
[13] Brass P., *Advanced Data Structures*, Cambridge University Press, 2008.
[14] Comer D., *Network System Design with Network Processors, Agere Version*, Upper Saddle River, NJ: Prentice Hall, 2005.
[15] Parson D., *Incremental Reorganization for Hash Tables*, U.S. Patent 6,915,296 B2, July 2005.
[16] Kirk D. and Hwu W., *Programming Massively Parallel Processors*. Burlington, MA: Morgan Kaufmann, 2010.
[17] Alcantara, D., et. al., “Real-Time Parallel Hashing on the GPU,” *ACM Transactions on Graphics* (Proceedings of ACM SIGGRAPH Asia 2009), Yokohama, Japan.
[18] Alcantara, D., *Efficient Hash Tables on the GPU*, Ph.D. dissertation, University of California, Davis, 2011.
[19] Williams R., “A Painless Guide to CRC Error Detection Algorithms,” August 1993, http://www.ross.net/crc/download/crc_v3.txt, link checked August 2012.
[20] Williams A., *C++ Concurrency in Action*. Shelter Island, NY: Manning Publications, 2012.
[21] Farber R., *Cuda Application Design and Development*. Waltham, MA: Morgan Kaufmann, 2011.
[22] Sanders J. and Kandrot E., *Cuda by Example*. Upper Saddle River, NJ: Addison-Wesley, 2011.
[23] Stuart J. and Owens J., “Efficient Synchronization Primitives for GPUs,” Cornell University Library, arXiv:1110.4623v1 [cs.OS], http://arxiv.org/abs/1110.4623v1, link checked August 2012.
[24] Lewis B. and Berg D., *Multithreaded Programming with Pthreads*. Mountain View, CA: Sun Microsystems / Prentice Hall, 1998.

[25] 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.

[26] Ullman J., *Elements of ML Programming*. Englewood Cliffs, NJ: Prentice Hall, 1994.

[27] Pugh W., "Skip Lists: A Probabilistic Alternative to balanced Trees," *Communications of the ACM* 33(6) (June 1990), p. 668-676.

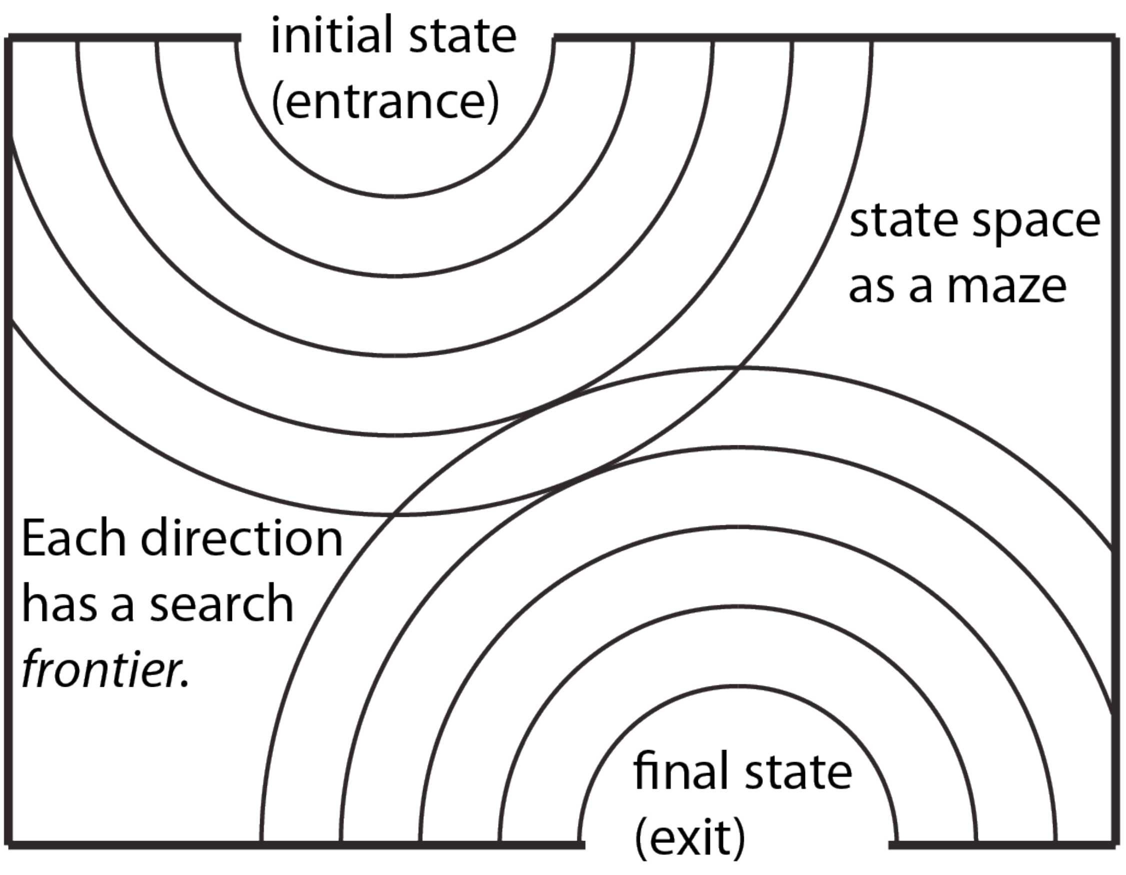
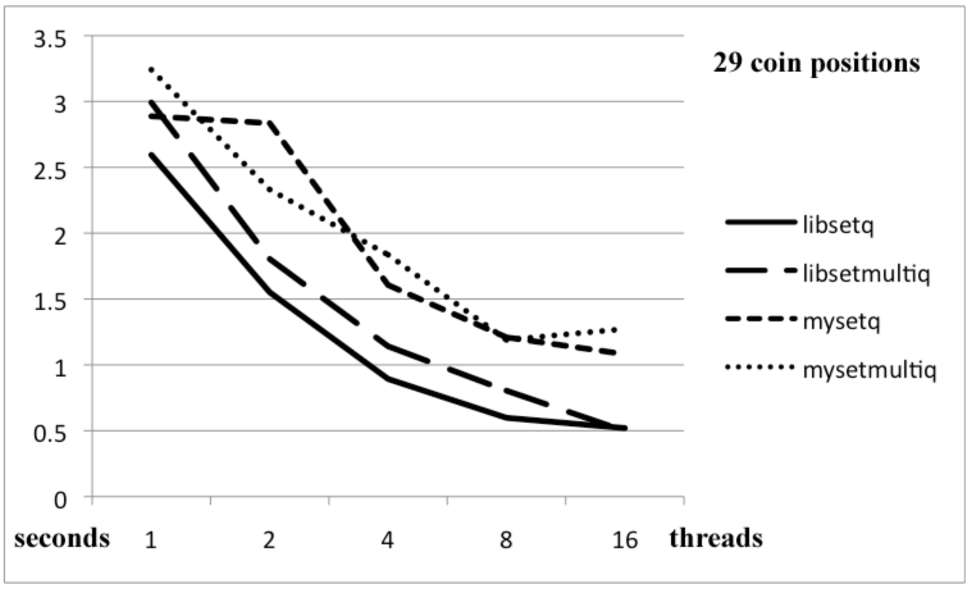


Figure 1: Bidirectional Search as a Maze

```
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
start parallel threads executing the following code:
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 1: Parallel, minimum-blocking dataflow algorithm



Graph 1: Execution time for 4 classes of queues and sets

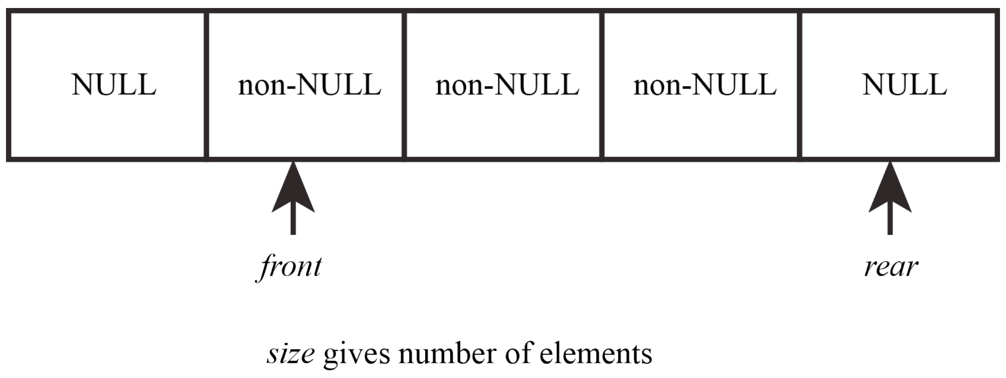
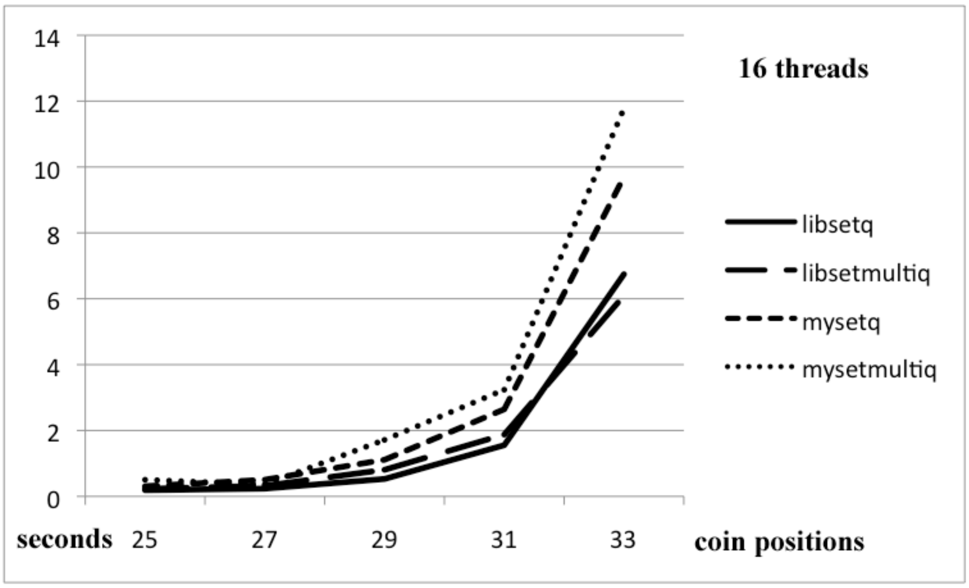


Figure 2: Circular buffer implementation of a FIFO using atomic data

```
for (int tries = 0 ; tries < limit ; tries++) {
        E v ;
        if (isinsert && table.compareAndSet(bucket, null, e)) {
            v = e ;
            result[1] = 1 ;
            numElements.incrementAndGet();
        } else {
            v = table.get(bucket);
        }
        if (v == null || v.equals(e)) {
            result[0] = bucket ;
            return result ;
        } else {
            bucket = (bucket + prm) & bitmask ;
        }
    }
```

Listing 2: The core of the thread-safe, non-blocking hash table search function



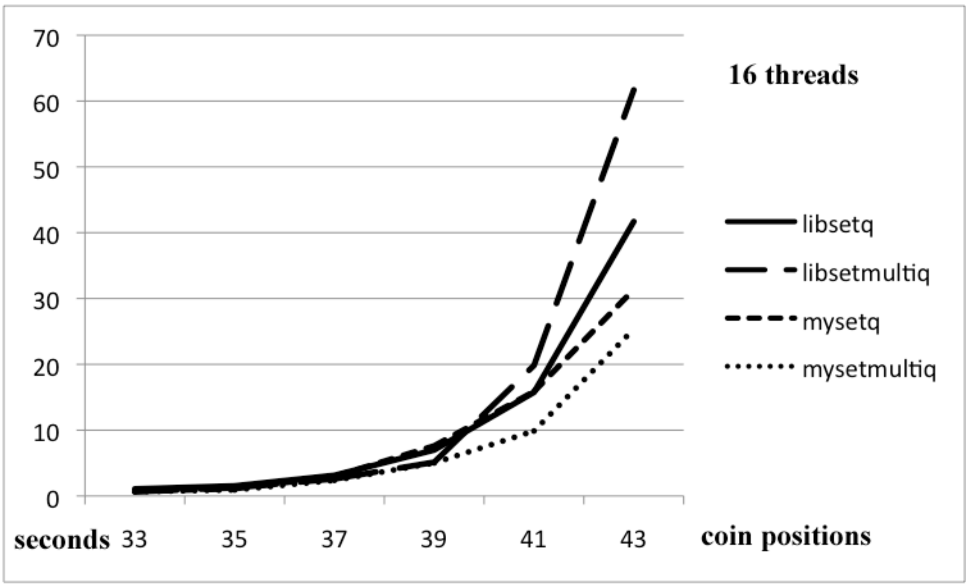
Graph 2: Execution time for 4 classes of queues and sets

```
int tmphash = initOpen ^ initVector.length ;
for (int i = 0 ; i < initVector.length ; i++) {
   // Since all states will have the same number of pennies,
   // dimes and a single space, add some weight by location
   tmphash = tmphash ^ (initVector[i] * i);
}
hashcode = tmphash ;
```

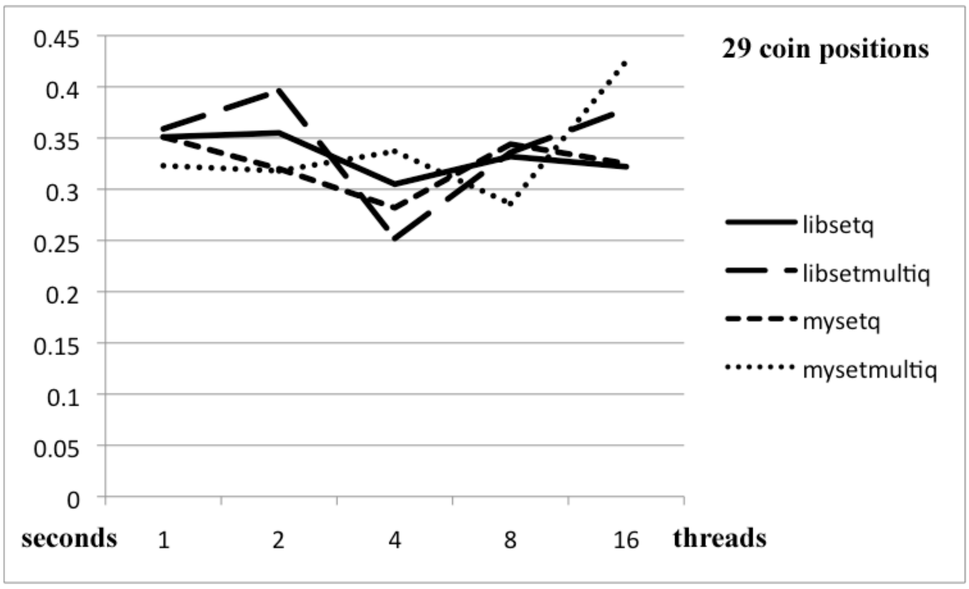
Listing 3: The initial, naive hash function for a state object

```
int tmphash = initOpen ;
for (int i = 0 ; i < initVector.length ; i++) {
   if (initVector[i] == DIME) {
      tmphash = tmphash ^ (tmphash << 6) ^ i ;
   } else if (initVector[i] == EMPTY) {
      tmphash = tmphash ^ (tmphash << 6) ^ (i * i);
   }
}
hashcode = tmphash ;
```

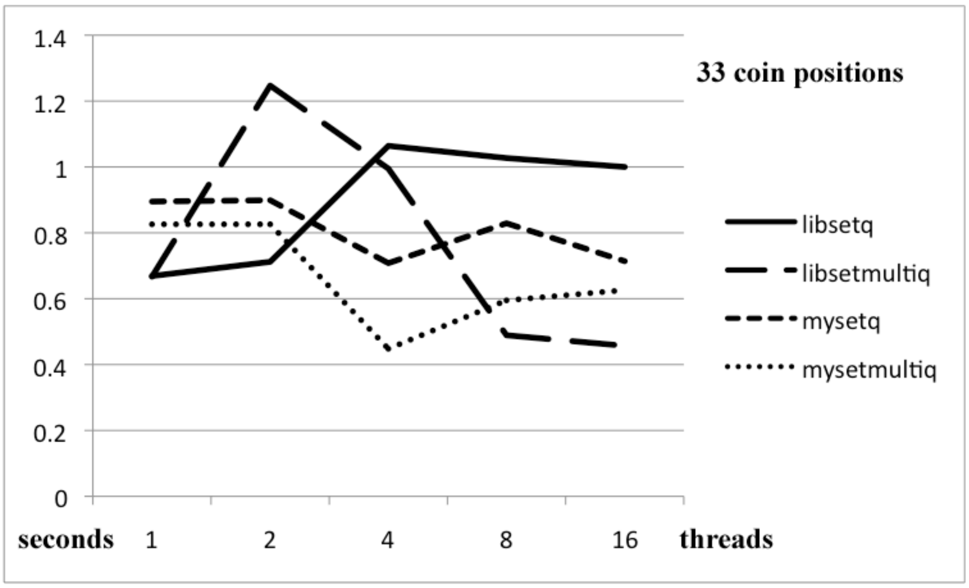
Listing 4: The final, effective hash function for a state object



Graph 3: Execution time using the final hash function



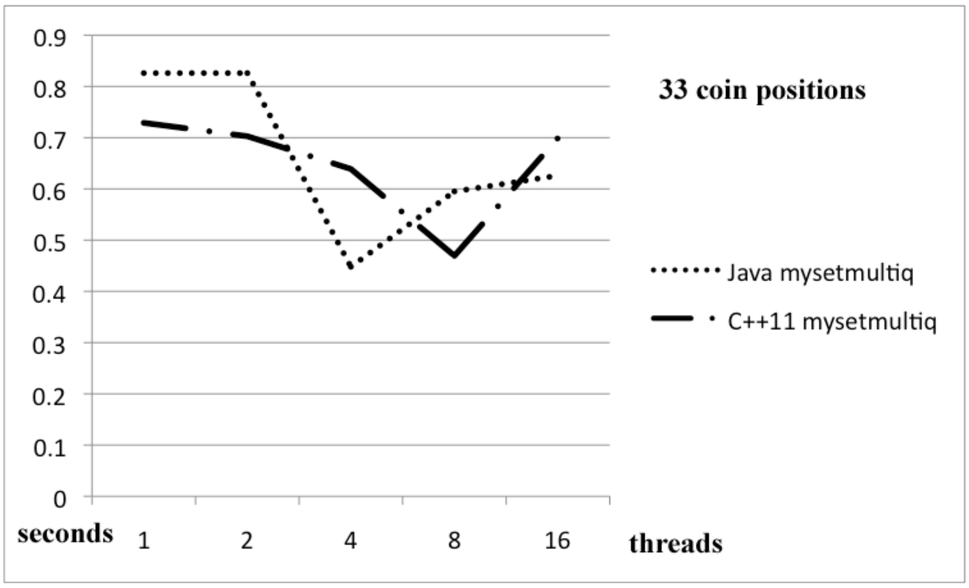
Graph 4: Execution time for 29 positions using the final hash function



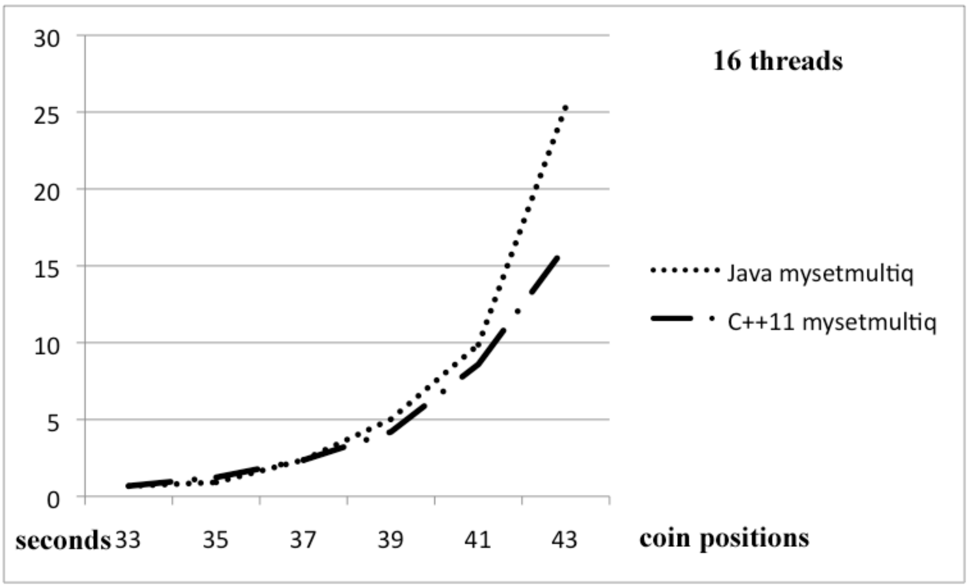
Graph 5: Execution time for 33 positions using the final hash function

```
for (int tries = 0 ; tries < limit ; tries++) {
   E * v ;
   E * null = NULL ;
   // compare_exchange_strong can overwrite null
   if (isinsert && table[bucket].compare_exchange_strong(null, e)) {
      v = e ;
      result[1] = 1 ;
      numElements++ ;
   } else {
      v = table[bucket].load();
   }
   if (v == NULL || v->equals(e)) {
      result[0] = bucket ;
      return ;
   } else {
      bucket = (bucket + prm) & bitmask ;
   }
}
```

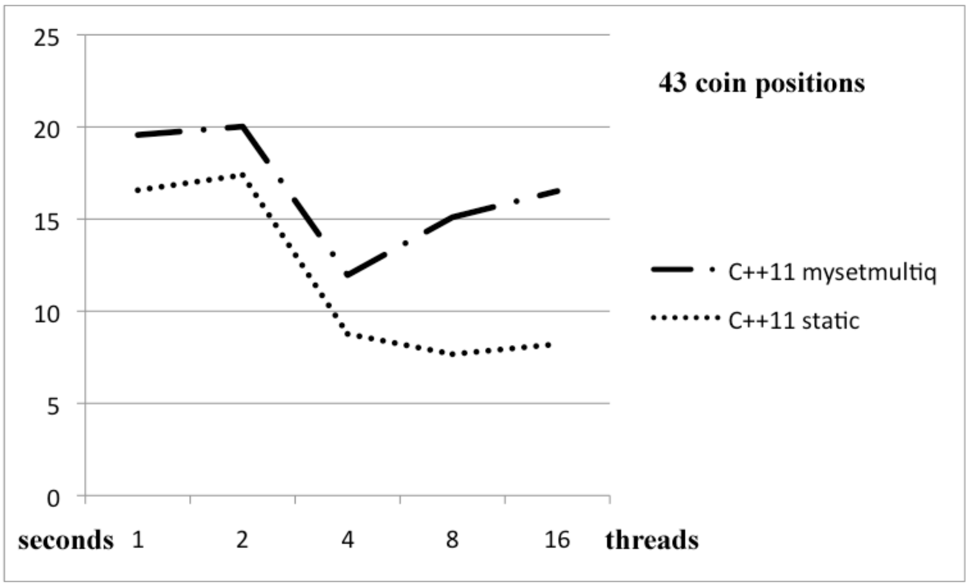
Listing 5: The thread-safe, non-blocking hash table search function in C++11



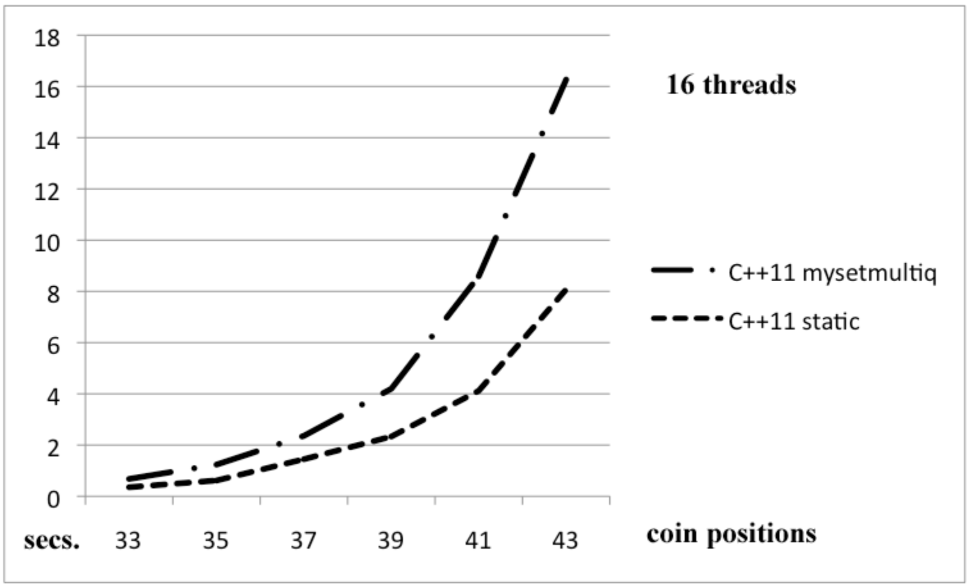
Graph 6: Execution time for 33 positions in Java and C++11



Graph 7: Execution time for 16 threads in Java and C++11



Graph 8: Execution time for 43 positions using two C++11 configurations



Graph 9: Execution time for 16 threads using two C++11 configurations

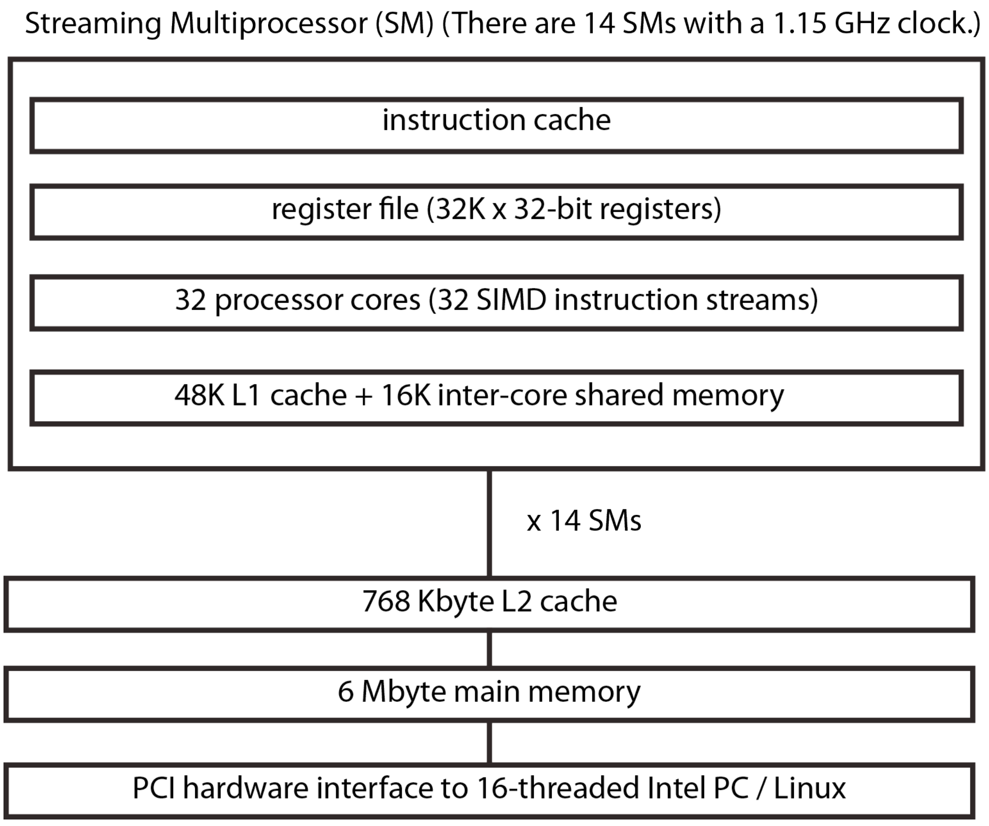
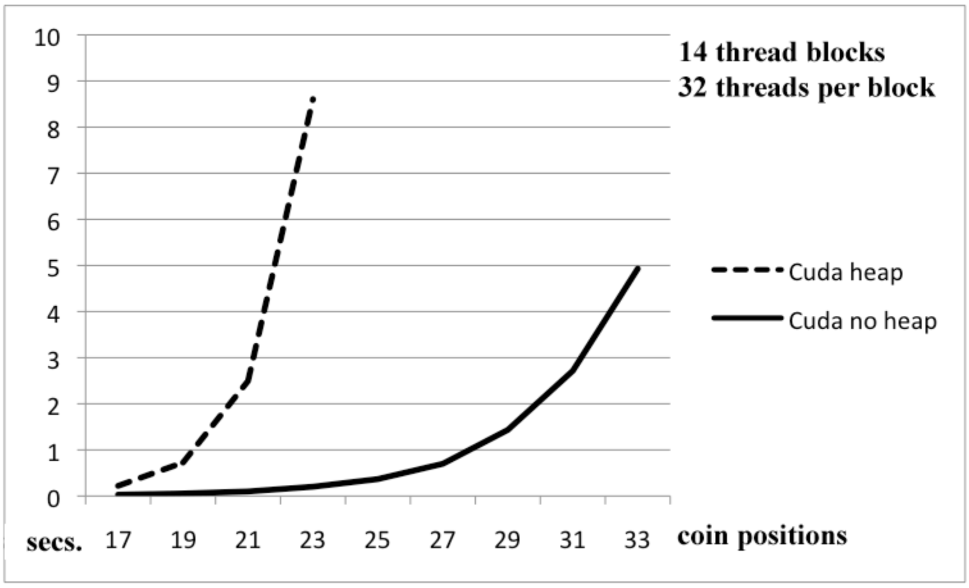

Streaming Multiprocessor (SM) (There are 14 SMs with a 1.15 GHz clock.)
instruction cache
register file (32K x 32-bit registers)
32 processor cores (32 SIMD instruction streams)
48K L1 cache + 16K inter-core shared memory
x 14 SMs
768 Kbyte L2 cache
6 Mbyte main memory
PCI hardware interface to 16-threaded Intel PC / Linux

Figure 3: Tesla-Fermi Compute 2.0 in the NVIDIA C2070 GPU



Graph 10: Execution time for two Tesla-Fermi Cuda configurations

```
// This is the CUDA kernel that performs State expansion.
// The host places the pairsToRead at the beginning of queueArray, and
// the pairsToWrite are just beyond the pairs to read. Each CUDA thread
// expands 1 pair-to-read into up to 4 pairs-to-write.
__global__ void generateStates(unsigned long pairsToRead,
        unsigned long startingStateOffset) {
    uint32_t myix = blockIdx.x * THREADSPERBLOCK + threadIdx.x ;
    while (myix < pairsToRead) {
        uint32_t stateix = ((&(queueArray[myix].fwdState))
            - ((State *) queueArray)) + startingStateOffset ;
        StatePair *myout = queueArray + pairsToRead
            + (myix * STATE_EXPANSION_PER_STEP);
        for (int transform = 0 ; transform < STATE_EXPANSION_PER_STEP
                ; transform++) {
            makeState(&(queueArray[myix].fwdState), stateix, transform,
                &(myout->fwdState));
            makeState(&(queueArray[myix].bkdState), stateix+1, transform,
                &(myout->bkdState));
            myout++ ;
        }
        myix += THREADSTOTAL ;
    }
}

__device__ void makeState(State *predecessor, uint32_t predecessorIndex,
        int transform, State *storage) /* This function initializes the State object. */
```

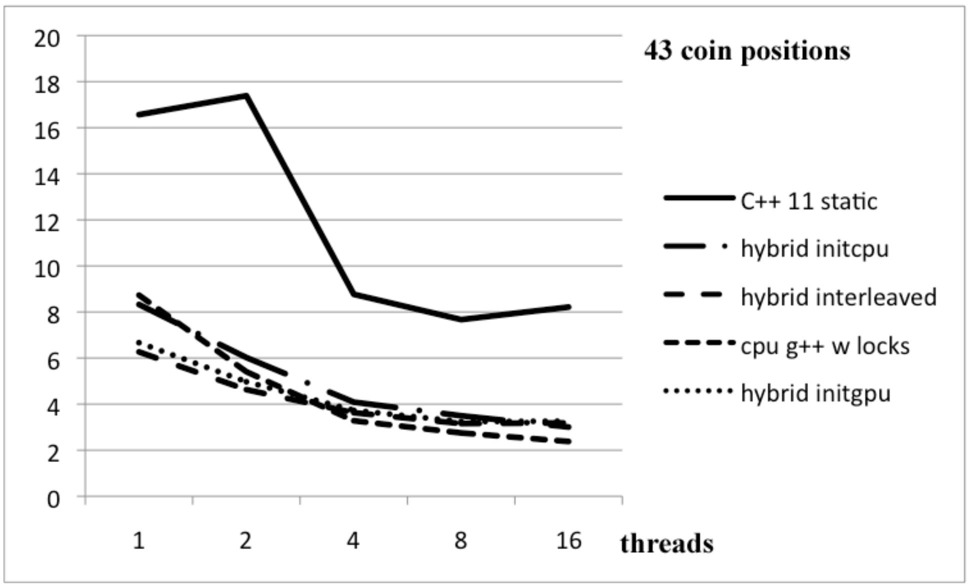
Listing 6: Cuda state expansion for the heterogeneous CPU / GPU approach

```
static void workerTaskReduceRecord(int threadnum, int threadcount) {
   // Record by stripes to maximize cache utility.
   // This is the only phase in which the hash table uses locks.
   if (threadnum < sizeOfReduction) {
      int32_t riser = startOfReduction + threadnum ;
      while (riser < topOfStatePairs){
         fwdStatesGlobal->add(&(hoststate[riser].fwdState));
         bkdStatesGlobal->add(&(hoststate[riser].bkdState));
         riser += threadcount ;
      }
   }
   workerCyclicBarrier(threadnum, threadcount, NULL);
}
```

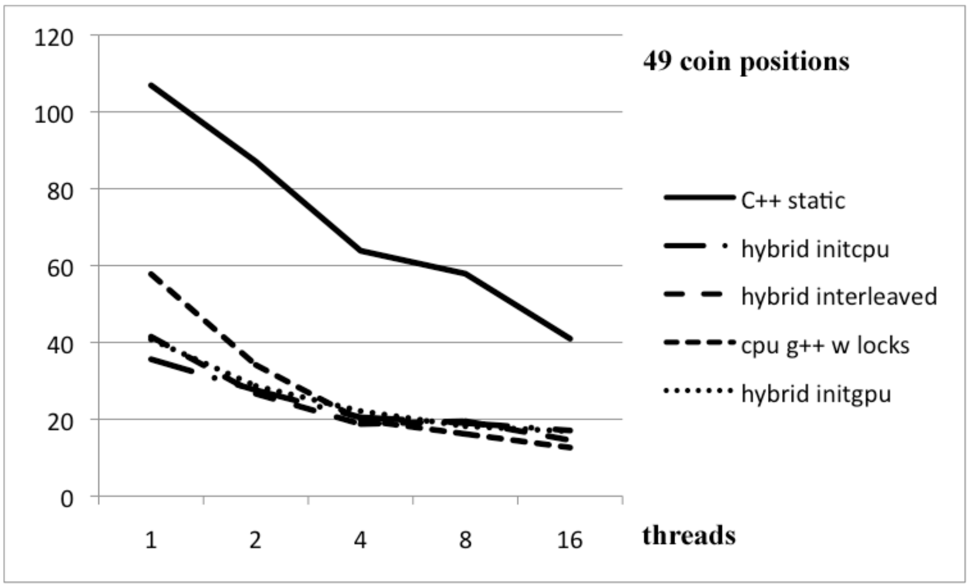
Listing 7: Heterogeneous hash table updates on the CPU using GPU-like code

```
static __host__ bool isSolution(State *fwds, State *bkds) ;
static void workerTaskReduceDetect(int threadnum, int threadcount) {
   // Inspect by stripes to maximize cache utility.
   if (threadnum < sizeOfReduction) {
      int32_t riser = startOfReduction + threadnum ;
      while (riser < topOfStatePairs) {
         if (hoststate[riser].fwdState.getOpen() != -1) {
            isSolution(&(hoststate[riser].fwdState),
               &(hoststate[riser].bkdState)) ;
         }
         riser += threadcount ;
      }
   }
   workerCyclicBarrier(threadnum, threadcount, NULL);
}
```

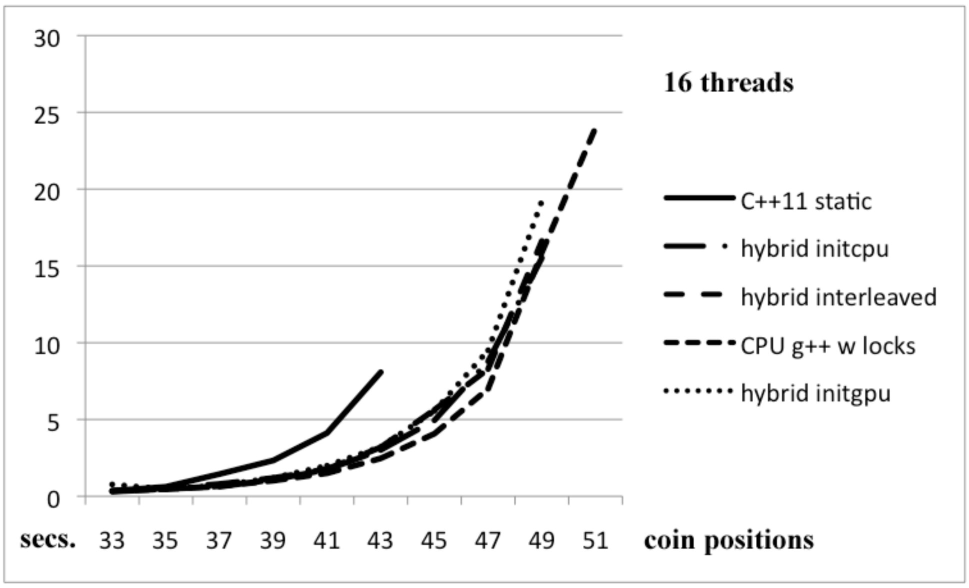
Listing 8: Heterogeneous solution detection on the CPU using GPU-like code



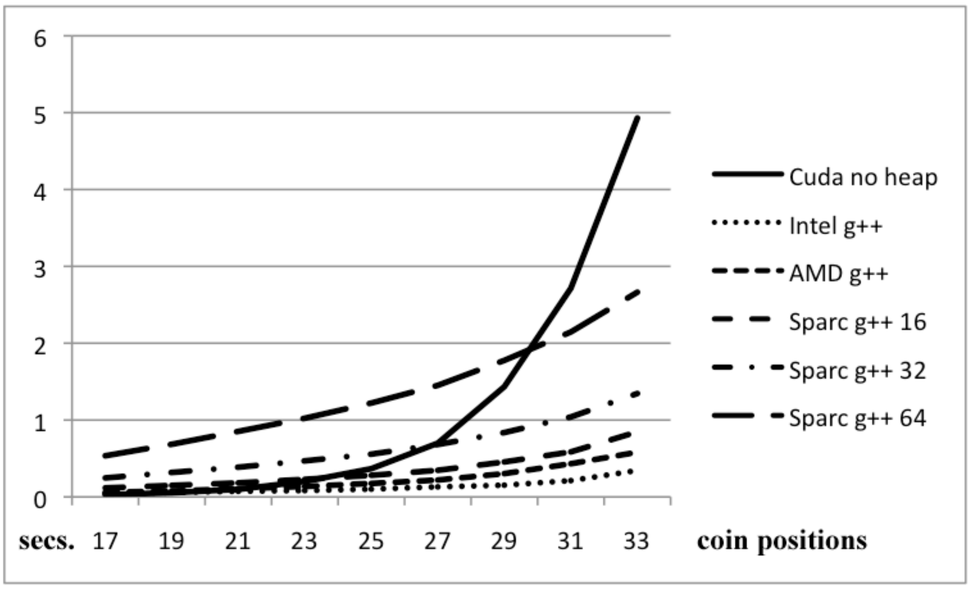
Graph 11: Execution time for 43 positions using CPU / GPU configurations



Graph 12: Execution time for 49 positions using CPU / GPU configurations



Graph 13: Execution time for 16 CPU threads using CPU / GPU configurations



Graph 14: Execution time for Cuda and CPUs on various architectures