A Scalable Elimination-based Exchange Channel[*]

William N. Scherer III
Dept. of Computer Science
University of Rochester
Rochester, NY 14627
scherer@cs.rochester.edu

Doug Lea
Computer Science Dept.
SUNY Oswego
Oswego, NY 13126
dl@cs.oswego.edu

Michael L. Scott
Dept. of Computer Science
University of Rochester
Rochester, NY 14627
scott@cs.rochester.edu

ABSTRACT

We present a new nonblocking implementation of the *exchange channel*, a concurrent data structure in which $2N$ participants form N pairs and each participant exchanges data with its partner. Our new implementation combines techniques from our previous work in dual stacks and from the elimination-based stack of Hendler et al. to yield very high concurrency.

We assess the performance of our exchange channel using experimental results from a 16-processor SunFire 6800. We compare our implementation to that of the Java SE 5.0 class *java.util.concurrent.Exchanger* using both a synthetic microbenchmark and a real-world application that finds an approximate solution to the traveling salesman problem using genetic recombination. Our algorithm outperforms the Java SE 5.0 *Exchanger* in the microbenchmark by a factor of two at two threads up to a factor of 50 at 10; similarly, it outperforms the Java SE 5.0 *Exchanger* by a factor of five in the traveling salesman problem at ten threads. Our exchanger has been adopted for inclusion in Java 6.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

algorithms, performance, experimentation

Keywords

nonblocking synchronization, dual data structures, elimination, exchanger, lock-freedom, parallel genetic algorithms

1. INTRODUCTION

The problem of *exchange channels* (sometimes known as *rendezvous channels*) arises in a variety of concurrent programs. In it, a thread t_a with datum d_a that enters the channel pairs up with another thread t_b (with datum d_b) and exchanges data such that t_a returns with d_b and t_b returns with d_a. More generally, $2N$ threads form N pairs $\langle t_{a_1}, t_{b_1}\rangle, \langle t_{a_2}, t_{b_2}\rangle, \langle t_{a_3}, t_{b_3}\rangle, ..., \langle t_{a_N}, t_{b_N}\rangle$ and exchange data pairwise.

In the basic exchange problem, if no partner is available immediately, thread t_a waits until one becomes available. In the *abortable* exchange problem, however, t_a specifies a patience p_a that represents a maximum length of time it is willing to wait for a partner to appear; if no partner appears within p_a μseconds, t_a returns empty-handed. Caution must be applied in implementations to ensure that a thread t_b that sees t_a just as it "gives up" and returns failure must not return d_a: Exchange must be bilateral.

Exchange channels are frequently used in parallel simulations. For example, the Promela modeling language for the SPIN model checker [8] uses rendezvous channels to simulate interprocess communication channels. Another typical use is in operating systems and server software. In a system with one producer and one consumer, the producer might work to fill a buffer with data, then exchange it with the buffer-draining consumer. This simultaneously bounds memory allocation for buffers and throttles the producer to generate data no faster than the consumer can process it.

2. BACKGROUND

2.1 Nonblocking Synchronization

Linearizability [5] has become the standard technique for demonstrating that a concurrent implementation of an object is correct. Informally, it "provides the illusion that each operation... takes effect instantaneously at some point between its invocation and its response" [5, abstract]. Linearizability is *nonblocking* in that it never requires a call to a total method (one whose precondition is simply `true`) to wait for the execution of any other method. The fact that it is nonblocking makes linearizability particularly attractive for reasoning about nonblocking *implementations* of concurrent objects, which provide guarantees of various strength regarding the progress of method calls in practice. In a *wait-free* implementation, every contending thread is guaranteed to complete its method call within a bounded number of its own time steps [6]. In a *lock-free* implementation, *some* contending thread is guaranteed to complete its method call within a bounded number of steps (from any thread's point of view) [6]. In an *obstruction-free* implementation, a thread is guaranteed to complete its method call within a bounded number of steps in the absence of contention, i.e. if no other threads execute competing methods concurrently [4].

2.2 Dual Data Structures

In traditional nonblocking implementations of concurrent objects, every method is total: It has no preconditions that must be satisfied before it can complete. Operations that might normally block before completing, such as dequeuing from an empty queue, are generally *totalized* to simply return a failure code in the case that their preconditions are not met. Then, calling the totalized method in a loop until it succeeds allows one to simulate the partial operation.

[*]This work was supported in part by NSF grants numbers EIA-0080124, CCR-0204344, and CNS-0411127, and by financial and equipment grants from Sun Microsystems Laboratories.

But this doesn't necessarily respect our intuition for the semantics of an object! For example, consider the following sequence of events for threads A, B, C, and D:

A calls dequeue
B calls dequeue
C enqueues a 1
D enqueues a 2
B's call returns the 1
A's call returns the 2

If thread A's call to dequeue is known to have started before thread B's call, then intuitively, we would think that A should get the first result out of the queue. Yet, with the call-in-a-loop idiom, ordering is simply a function of which thread happens to try a totalized dequeue operation first once data becomes available. Further, each invocation of the totalized method introduces performance-sapping contention for memory–interconnect bandwidth on the data structure. Finally, note that the mutual-swap semantics of an exchange channel do not readily admit a totalized implementation: If one simply fails when a partner is not available at the exact moment one enters the channel, the rate of successful rendezvous connections will be very low.

As an alternative, suppose we could register a request for a partner in the channel. Inserting this *reservation* could be done in a nonblocking manner, and checking to see whether someone has come along to *fulfill* our reservation could consist of checking a boolean flag in the data structure representing the request. Even if the overall exchange operation requires blocking, it can be divided into two logical halves: the pre-blocking reservation and the post-blocking fulfillment.

In our earlier work [10], we define a *dual data structure* to be one that may hold *reservations* (registered requests) instead of, or in addition to, data. A nonblocking dual data structure is one in which (a) every operation either completes or registers a request in nonblocking fashion; (b) fulfilled requests complete in non–blocking fashion; and (c) threads that are waiting for their requests to be fulfilled do not interfere with the progress of other threads. In a lock-free dual data structure, then, every operation either completes or registers a request in a lock-free manner and fulfilled requests complete in a lock-free manner.

For a more formal treatment of linearizability for dual data structures, and for practical examples of dual stacks and queues, we refer the reader to our earlier work [10].

2.3 Elimination

Elimination is a technique introduced by Shavit and Touitou [11] that improves the concurrency of data structures. It exploits the observation, for example, that one Push and one Pop, when applied with no intermediate operations to a stack data structure, yield a state identical to that from before the operations. Intuitively, then, if one could pair up Push and Pop operations, there would be no need to reference the stack data structure; they could "cancel each other out". Elimination thus reduces contention on the main data structure and allows parallel completion of operations that would otherwise require accessing a common central memory location.

More formally, one may define linearization points for mutually-canceling elimination operations in a manner such that no other linearization points intervene between them; since the operations effect (collectively) no change to the base data structure state, the history of operations – and its correctness – is equivalent to one in which the two operations never happened.

Although the original eliminating stack of Shavit and Touitou [11] is not linearizable [5], follow-up work by Hendler et al. [3] details one that is. Elimination has also been used for shared counters [1] and even for FIFO queues [9].

3. ALGORITHM DESCRIPTION

Our exchanger uses a novel combination of nonblocking dual data structures and elimination arrays to achieve high levels of concurrency. The implementation is originally based on a combination of our dual stack [10] and the eliminating stack of Hendler et al. [3], though peculiarities of the exchange channel problem limit the visibility of this ancestry. Source code for our exchanger may be found in the Appendix.

To simplify understanding, we present our exchanger algorithm in two parts. Section 3.1 first illustrates a simple exchanger that satisfies the requirements for being a lock-free dual data structure as defined earlier in Section 2.2. We then describe in Section 3.2 the manner in which we incorporate elimination to produce a scalable lock-free exchanger.

3.1 A Simple Nonblocking Exchanger

The main data structure we use for the simplified exchanger is a modified dual stack [10]. Additionally, we use an inner node class that consists of a reference to an `Object` offered for exchange and an `AtomicReference` representing the hole for an object. We associate one node with each thread attempting an exchange. Exchange is accomplished by successfully executing a `compareAndSet`, updating the hole from its initial null value to the partner's node. In the event that a thread has limited patience for how long to wait before abandoning an exchange, signaling that it is no longer interested consists of executing a `compareAndSet` on its *own* hole, updating the value from null to a `FAIL` sentinel. If this `compareAndSet` succeeds, no other thread can successfully match the node; conversely, the `compareAndSet` can only fail if some other thread has already matched it.

From this description, one can see how to construct a simple nonblocking exchanger. Referencing the implementation in Listing 1: Upon arrival, if the top-of-stack is null (line 07), we `compareAndSet` our thread's node into it (08) and wait until either its patience expires (10-12) or another thread matches its node to us (09, 17). Alternatively, if the top-of-stack is non-null (19), we attempt to `compareAndSet` our node into the existing node's hole (20); if successful, we then `compareAndSet` the top-of-stack back to null (22). Otherwise, we help remove the matched node from the top of the stack; hence, the `compareAndSet` is unconditional.

In this simple exchanger, the initial linearization point for an in-progress swap is when the `compareAndSet` on line 18 succeeds; this inserts a reservation into the channel for the next data item to arrive. The linearization point for a fulfilling operation is when the `compareAndSet` on line 20 succeeds; this breaks the waiting thread's spin (lines 9-16). (Alternatively, a successful `compareAndSet` on line 11 is the linearization point for an aborted exchange.) As it is clear that the waiter's spin accesses no remote memory locations and that both inserting and fulfilling reservations are lock-free (a `compareAndSet` in this case can only fail if another has succeeded), the simple exchanger constitutes a lock-free implementation of the exchanger dual data structure as defined in Section 2.2.

3.2 Adding Elimination

Although the simple exchanger from the previous section is nonblocking, it will not scale very well: The top-of-stack pointer in particular is a hotspot for contention. This scalability problem can be resolved by adding an elimination step to the simple exchanger from Listing 1.

```
Object exchange(Object x, boolean timed,
  long patience) throws TimeoutException {
  boolean success = false;
  long start = System.nanotime();
  Node mine = new Node(x);
  for (;;) {
    Node top = stack.getTop();
    if (top == null) {
      if (stack.casTop(null, mine)) {
        while (null == mine.hole) {
          if (timedOut(start, timed, patience) {
            if (mine.casHole(null, FAIL))
              throw new TimeoutException();
            break;
          }
          /* else spin */
        }
        return mine.hole.item;
      }
    } else {
      success = top.casHole(null, mine);
      stack.casTop(top, null);
      if (success)
        return top.item;
    }
  }
}
```

Listing 1: A simple lock-free exchanger

In order to support elimination, we replace the single top-of-stack pointer with an `arena` (array) of $(P+1)/2$ Java SE 5.0 `AtomicReferences`, where P is the number of processors in the runtime environment. Logically, the reference in position 0 is the top-of-stack; the other references are simply locations at which elimination can occur.

Following the lead of Hendler et al., we incorporate elimination with backoff when encountering contention at top-of-stack. As in their work, by only attempting elimination under conditions of high contention, we incur no additional overhead for elimination.

Logically, in each iteration of a main loop, we attempt an exchange in the 0th arena position exactly as in the simple exchanger. If we successfully insert or fulfill a reservation, we proceed exactly as before. The difference, however, comes when a `compareAndSet` fails. Now, instead of simply retrying immediately at the top-of-stack, we back off to attempt an exchange at another (randomized) arena location. In contrast to exchanges at `arena[0]`, we limit the length of time we wait with a reservation in the remainder of the arena to a value significantly smaller than our overall patience. After canceling the reservation, we return to `arena[0]` for another iteration of the loop.

In iteration i of the main loop, the arena location at which we attempt a secondary exchange is selected randomly from the range $1..b$, where b is the lesser of i and the arena size. Hence, the first secondary exchange is always at `arena[1]`, but with each iteration of the main loop, we increase the range of potential backoff locations until we are randomly selecting a backoff location from the entire arena. Similarly, the length of time we wait on a reservation at a backoff location is randomly selected from the range $0..2^{(b+k)}-1$, where k is a base for the exponential backoff.

From a correctness perspective, the same linearization points as in the simple exchanger are again the linearization points for the eliminating exchanger; however, they can occur at any of the arena slots, not just at a single top-of-stack. Although the eliminating stack can be shown to support LIFO ordering semantics, we have no particular ordering semantics to respect in the case of an exchange channel: Any thread in the channel is free to match to any other thread, regardless of when it entered the channel.

The use of timed backoff accentuates the probabilistic nature of limited-patience `exchange`. Two threads that attempt an exchange with patience zero will only discover each other if they both happen to probe the top of the stack at almost exactly the same time. However, with increasing patience levels, the probability decreases exponentially that they will fail to match after temporally proximate arrivals. Other parameters that influence this rapid fall include the number of processors and threads, hardware instruction timings, and the accuracy and responsiveness of timed waits. In modern environments, the chance of backoff arena use causing two threads to miss each other is far less than the probability of thread scheduling or garbage collection delaying a blocked thread's wakeup for long enough to miss a potential match.

3.3 Pragmatics

Our implementation of this algorithm (shown in the Appendix) reflects a few additional pragmatic considerations to maintain good performance:

First, we use an array of `AtomicReferences` rather than a single `AtomicReferenceArray`. Using a distinct reference object per slot helps avoid some false sharing and cache contention, and places responsibility for their placement on the Java runtime system rather than on this class.

Second, the time constants used for exponential backoff can have a significant effect on overall throughput. We empirically chose a base value to be just faster than the minimum observed round-trip overhead, across a set of platforms, for timed `parkNanos()` calls on already-signaled threads. By so doing, we have selected the smallest value that does not greatly underestimate the actual wait time. Over time, future versions of this class might be expected to use smaller base constants.

4. EXPERIMENTAL RESULTS

4.1 Benchmarks

We present experimental results for two benchmarks. The first is a microbenchmark in which threads swap data in an exchange channel as fast as they can.

Our second benchmark exemplifies the way that an exchange channel might be used in a real-world application. It consists of a parallel implementation of a solver for the traveling salesman problem, implemented via genetic algorithms. It accepts as parameters a number of cities C, a population size P, and a number of generations G, in each of which B breeders mate and create B children to replace B individuals that die (the remaining $P-B$ individuals carry forward to the next generation). Breeders enter a central exchange channel one or more times to find a partner with which to exchange genes (a subset of the circuit); the number of partners ranges from four down to one in a simulated annealing fashion. Between randomization of the order of breeders and the (semi-) nondeterministic manner in which pairings happen in the exchange channel, we achieve a diverse set of matings with high probability.

4.2 Methodology

All results were obtained on a SunFire 6800, a cache-coherent multiprocessor with 16 1.2GHz UltraSPARC III processors. We tested each benchmark with both our new exchanger and the Java SE 5.0 *java.util.concurrent.Exchanger*[1] in Sun's Java SE 5.0

[1] Actually, the Java SE 5.0 *Exchanger* contains a flaw in which a wake-up signal is sometimes delivered to the wrong thread, forcing a would-be exchanger to time out before noticing it has been matched. For our tests, we compare instead to a modified version (*Exchanger15a*) that corrects this issue.

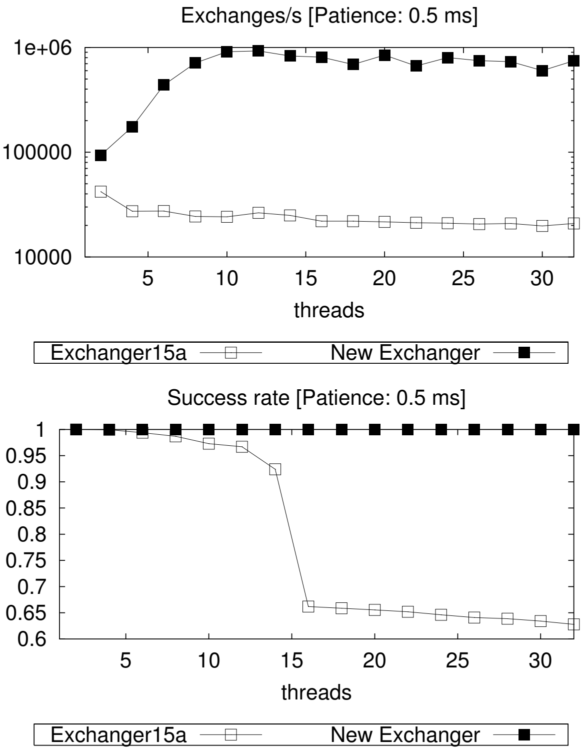

Figure 1: Microbenchmark: Throughput (top) and Success rate (bottom). Note that the throughput graph is in log scale.

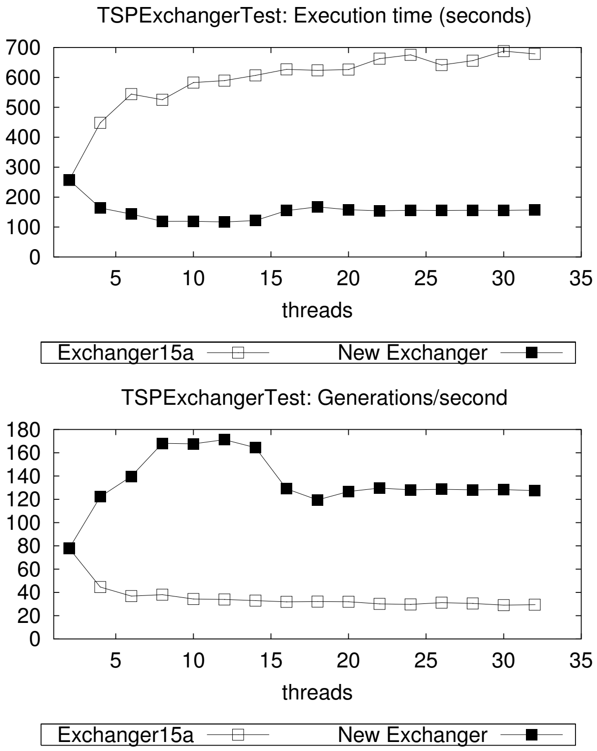

Figure 2: Parallel GA-Traveling Salesman: Total Execution Time (top) and Generations per second throughput (bottom).

HotSpot VM, ranging from 2 to 32 threads. For both tests, we compute the average of three test runs.

For the traveling salesman application, we used 100 cities, a population of 1000 chromosomes, and 200 breeders. We measure the total wall-clock time required to complete 20000 generations and calculate the total number of generations per second achieved at each thread level.

Figure 1 displays throughput and the rate at which exchanges are successful for our microbenchmark analysis of exchangers. Figure 2 presents total running time, as a function of the number of threads, and the generation completion throughput for our parallel genetic algorithm-based traveling salesman solver.

4.3 Discussion

As can be seen from the top half of Figure 1, our exchanger outperforms the Java SE 5.0 *Exchanger* by a factor of two at two threads up to a factor of 50 by 10 threads. The performance of the Java SE 5.0 *Exchanger* degrades as the number of threads participating in swaps increases. This lack of scalability may be attributed to the coarse-grained locking strategy that it uses. Another part of the reason for this difference may be observed from the bottom half of Figure 1. While our nonblocking exchanger is able to maintain nearly 100% success in exchange operations, the Java SE 5.0 *Exchanger* gets dramatically worse as the number of active threads increases, particularly in the presence of preemption (beyond 16 threads).

For the traveling salesman application, we see no difference in the total running time (Figure 2 top) at two threads, but by 10 threads the difference in running time is nearly a factor of five. When we look at the throughput rate of computation (generations per second) in the bottom half of Figure 2, we see that again, our exchanger scales more-or-less linearly up to 8 threads, and continues to gain parallel speedup through 12, but the Java SE 5.0 *Exchanger* degrades in performance as the number of threads increases. The drop-off in throughput for our exchanger we see beginning at 16 threads reflects the impact of preemption in slowing down exchanges and inter-generation barriers

4.4 Field Notes: Multi-party Exchange

Consider the exchanger used with more than two threads. In our parallel traveling salesman implementation (pseudocode for a piece of which appears in Listing 2), each thread is assigned a certain number of individuals to breed, and the breedings are conducted in parallel. As mentioned earlier, each breeding consists of swapping genes with a partner found from a central exchange channel.

With two threads, no complications arise, but beginning at four, a problem appears wherein all but one thread can meet up enough times to finish breeding, leaving one thread high and dry. Consider the following example, in which four threads $(T_1..T_4)$ each have three elements (a, b, c) to swap. Suppose that swaps occur according to the following schedule:

$$\begin{array}{l}\langle T_1(a), T_2(a)\rangle \\ \langle T_1(b), T_3(a)\rangle \\ \langle T_2(b), T_3(b)\rangle \\ \langle T_1(c), T_2(c)\rangle \\ \langle T_3(c), T_4(a)\rangle\end{array}$$

Now, T_4 still needs to swap b and c, but has no one left to swap with. Although one could use a barrier between trips to the exchanger to keep threads synchronized, this would be an exceedingly high-overhead solution; it would hurt scalability and performance. Further, it would result in more deterministic pairings be-

```
     for (int 1 = 0; i < numToBreed; i++) {
       Chromosome parent = individuals[breeders[first+i]];
       try {
         Chromosome child = new Chromosome(parent);
*1*      Chromosome peer = x.exchange(child, 100,
           TimeUnit.MICROSECONDS);
         children[i] = child.reproduceWith(peer, random);
       } catch (TimeoutException e) {
*2*      if (1 == barrier.getPartiesRemaining()) {
           // No peers left, so we mate with ourselves
*3*        mateWithSelf(i, numToBreed, children);
           break;
         }
*4*      // Spurious failure; retry the breeding
         --i;
       }
     }
     barrier.await();
```

Listing 2: Multi-party exchange

```
  for (int 1 = 0; i < numToBreed; i++) {
    Chromosome parent = individuals[breeders[first+i]];
    Chromosome child = new Chromosome(parent);
    Exchanger.Color clr = ((1 == tid & 1) ? RED : BLUE;
    Chromosome peer = x.exchange(child, clr);
    children[i] = child.reproduceWith(peer, random);
  }
```

Listing 3: Multi-party exchange with a red-blue exchanger

tween threads in our traveling salesman application, which is undesirable in genetic algorithms.

Instead, we detect this case by limiting how long would-be breeders wait in the exchange channel. If they time out from the breeding (*1*), we enter a catch block where we check a barrier to see if we're the only ones left (*2*). If not, we retry the breeding (*4*); otherwise, we know that we're the only thread left and we simply recombine within the breeders we have left (*3*).

It is unfortunate that an external checking idiom is required to make use of the exchange channel with multiple threads. Our traveling salesman benchmark already needs a barrier to ensure that one generation completes before the next begins, so this adds little overhead in our specific case. However, the same cannot be readily guaranteed across all potential multi-party swap applications.

On the other hand, suppose we had a channel that allows an exchange party to be one of two colors (red or blue) and that constrains exchanges to being between parties of dissimilar color. Such an exchanger could be built, for example, as a shared-memory implementation of generalized input-output guards [2] for Hoare's Communicating Sequential Processes (CSP) [7]. Then, by assigning `RED` to threads with odd ID and `BLUE` to threads with even ID, we could simplify the code as shown in Listing 3.

Note that we no longer need timeout: Assuming the number of threads and breeders are both even, an equal number will swap with red as with blue; stranded exchanges are no longer possible. (The slightly reduced non-determinism in breeding seems a small price to pay for simpler code and for eliminating the determinism that occurs when a thread must breed with itself.) A red-blue exchanger also generalizes producer-consumer buffer exchange to multiple producers and multiple consumers. By simply marking producers blue and consumers red, swaps will always consist of a producer receiving an empty buffer and a consumer receiving a full one.

Implementing a red-blue exchanger would be a relatively straightforward extension to our new exchanger. In particular, rather than assuming that any pair of threads match in `arena[0]`, we could switch to a full implementation of a dual stack, using `Push()` for red threads and `Pop()` for blue. We would similarly need to update elimination in the nonzero arena slots. No other changes to our algorithm would be needed, however, to support a bi-chromatic exchange channel.

5. CONCLUSIONS

In this paper, we have demonstrated a novel lock-free exchange channel that achieves very high scalability through the use of elimination. To our knowledge, this is the first combination of dual data structures and elimination arrays.

In a head-to-head microbenchmark comparison, our algorithm outperforms the Java SE 5.0 *Exchanger* by a factor of two at two threads and a factor of 50 at 10 threads. We have further shown that this performance differential spells the difference between performance degradation and linear parallel speedup in a real-world genetic algorithm-based traveling salesman application. Our new exchange channel has been adopted for inclusion in Java 6.

For future work, we suspect that other combinations of elimination with dual data structures might see similar benefits; in particular, our dual queue seems to be an ideal candidate for such experimentation. Additionally, we look forward to implementing and experimenting with our red-blue exchanger.

6. ACKNOWLEDGMENTS

We are grateful to Christopher Brown for useful suggestions in designing the genetic algorithm-based traveling salesman solver.

7. REFERENCES

[1] W. Aiello, C. Busch, M. Herlihy, M. Mavronicolas, N. Shavit, and D. Touitou. Supporting Increment and Decrement Operations in Balancing Networks. In *Chicago Journal of Theoretical Computer Science*, Dec. 2000. Originally presented at the 16th Intl. Symp. on Theoretical Aspects of Computer Science, Trier, Germany, March 1999, and published in *Lecture Notes in Computer Science*, Vol. 1563, pp. 393-403.

[2] A. J. Bernstein. Output Guards and Nondeterminism in Communicating Sequential Processes. *ACM Trans. on Programming Languages and Systems*, 2(2):234–238, Apr. 1980.

[3] D. Hendler, N. Shavit, and L. Yerushalmi. A Scalable Lock-Free Stack Algorithm. In *Proc. of the 16th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, pages 206–215, Barcelona, Spain, June 2004.

[4] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May, 2003.

[5] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[6] M. Herlihy. Wait-Free Synchronization. *ACM Trans. on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[7] C. A. R. Hoare. Communicating Sequential Processes. *Comm. of the ACM*, 21(8):666–677, Aug. 1978.

[8] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, 23(5), May 1997.

[9] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *Proc. of the 17th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, pages 253–262, Las Vegas, NV, July 2005.

[10] W. N. Scherer III and M. L. Scott. Nonblocking Concurrent Objects with Condition Synchronization. In *Proc. of the 18th Intl. Symp. on Distributed Computing*, Amsterdam, The Netherlands, Oct. 2004.

[11] N. Shavit and D. Touitou. Elimination Trees and the Construction of Pools and Stacks. In *Proc. of the 7th Annual ACM Symp. on Parallel Algorithms and Architectures*, Santa Barbara, CA, July 1995.

APPENDIX

Exchanger Source Code

The following code may be found online at `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/Exchanger.java`

```
public class Exchanger<V> {
  private static final int SIZE =
    (Runtime.getRuntime().availableProcessors() + 1) / 2;
  private static final long BACKOFF_BASE = 128L;
  static final Object FAIL = new Object();
  private final AtomicReference[] arena;
  public Exchanger() {
    arena = new AtomicReference[SIZE + 1];
    for (int i = 0; i < arena.length; ++i)
      arena[i] = new AtomicReference();
  }

  public V exchange(V x) throws InterruptedException {
    try {
      return (V)doExchange(x, false, 0);
    } catch (TimeoutException cannotHappen) {
      throw new Error(cannotHappen);
    }
  }
  public V exchange(V x, long timeout, TimeUnit unit)
    throws InterruptedException, TimeoutException {
    return (V)doExchange(
      x, true, unit.toNanos(timeout));
  }

  private Object doExchange(
    Object item, boolean timed, long nanos)
    throws InterruptedException, TimeoutException {
    Node me = new Node(item);
    long lastTime = (timed)? System.nanoTime() : 0;
    int idx = 0;
    int backoff = 0;

    for (;;) {
      AtomicReference<Node> slot =
        (AtomicReference<Node>)arena[idx];

      // If this slot is occupied, an item is waiting...
      Node you = slot.get();
      if (you != null) {
        Object v = you.fillHole(item);
        slot.compareAndSet(you, null);
        if (v != FAIL)     // ... unless it's cancelled
          return v;
      }

      // Try to occupy this slot
      if (slot.compareAndSet(null, me)) {
        Object v = ((idx == 0)?
          me.waitForHole(timed, nanos) :
          me.waitForHole(true, randomDelay(backoff)));
        slot.compareAndSet(me, null);
        if (v != FAIL)
          return v;
        if (Thread.interrupted())
          throw new InterruptedException();
        if (timed) {
          long now = System.nanoTime();
          nanos -= now - lastTime;
          lastTime = now;
          if (nanos <= 0)
            throw new TimeoutException();
        }
        me = new Node(item);
        if (backoff < SIZE - 1)
          ++backoff;
        idx = 0;   // Restart at top
      }

      else // Retry with a random non-top slot <= backoff
        idx = 1 + random.nextInt(backoff + 1);
    }
  }

  private long randomDelay(int backoff) {
    return ((BACKOFF_BASE << backoff) - 1) &
      random.nextInt();
  }

  static final class Node
    extends AtomicReference<Object> {
    final Object item;
    final Thread waiter;
    Node(Object item) {
      this.item = item;
      waiter = Thread.currentThread();
    }

    Object fillHole(Object val) {
      if (compareAndSet(null, val)) {
        LockSupport.unpark(waiter);
        return item;
      }
      return FAIL;
    }

    Object waitForHole(
      boolean timed, long nanos) {
      long lastTime = (timed)?
        System.nanoTime() : 0;
      Object h;
      while ((h = get()) == null) {
        // If interrupted or timed out, try to
        // cancel by CASing FAIL as hole value.
        if (Thread.currentThread().isInterrupted() ||
          (timed && nanos <= 0)) {
          compareAndSet(null, FAIL);
        } else if (!timed) {
          LockSupport.park();
        } else {
          LockSupport.parkNanos(nanos);
          long now = System.nanoTime();
          nanos -= now - lastTime;
          lastTime = now;
        }
      }
      return h;
    }
  }
}
```