Fast and Scalable Rendezvousing

Yehuda Afek, Michael Hakimi, and Adam Morrison

School of Computer Science
Tel Aviv University

Abstract. In an asymmetric rendezvous system, such as an unfair synchronous queue and an elimination array, threads of two types, consumers and producers, show up and are matched, each with a unique thread of the other type. Here we present a new highly scalable, high throughput asymmetric rendezvous system that outperforms prior synchronous queue and elimination array implementations under both symmetric and asymmetric workloads (more operations of one type than the other). Consequently, we also present a highly scalable elimination-based stack.

1 Introduction

A common abstraction in concurrent programming is that of an *asymmetric rendezvous* mechanism. In this mechanism, there are two types of threads that show up, e.g., producers and consumers. The goal is to match pairs of threads, one of each type, and send them away. Usually the purpose of the pairing is for a producer to hand over a data item (such as a task to perform) to a consumer. The asymmetric rendezvous abstraction encompasses both *unfair synchronous queues* (or *pools*) [12] which are a key building block in Java's thread pool implementation and other message-passing and hand-off designs [2, 12], and the *elimination* technique [13], which is used to scale concurrent stacks and queues [7, 11].

In this paper we present a highly scalable asymmetric rendezvous algorithm that improves the state of the art in both unfair synchronous queue and elimination algorithms. It is based on a distributed scalable ring structure, unlike Java's synchronous queue which relies on a non-scalable centralized structure. It is *nonblocking*, in the following sense: if both producers and consumers keep taking steps, *some* rendezvous operation is guaranteed to complete. (This is similar to the *lock-freedom* property [8], while taking into account the fact that "it takes two to tango", i.e., both types of threads must take steps to successfully rendezvous.) It is also uniform, in that no thread has to perform work on behalf of other threads. This is in contrast to the flat combining (FC) based synchronous queues of Hendler et al. [6], which are blocking and non-uniform.

Our algorithm is based on a simple and remarkably effective idea: the *algorithm itself is asymmetric*. A consumer captures a node in the ring and waits there for a producer, while a producer actively seeks out waiting consumers on the ring. The algorithm utilizes a new *ring adaptivity scheme* that dynamically adjusts the ring size, leaving enough room for all the consumers while avoiding empty nodes that producers will futileness search. Because of the adaptive ring size, we can expect the nodes to be densely populated, and thus a producer that starts to scan the ring and finds a node to be empty, it is likely that a consumer will arrive there shortly. Yet simply waiting at this node, hoping that this will occur, makes the algorithm prone to timeouts and impedes progress. We solve this problem by introducing *peeking* a technique that lets the producer enjoy the best of both worlds: as the producer traverses the ring, it continues to peek at its initial node; if a consumer arrives there, the producer immediately tries to partner with it, thereby minimizing the amount of wasted work.

Our algorithm avoids two problems found in prior elimination algorithms that did not exploit asymmetry. In these works [1, 7, 13], both types of threads would pick a random node in the hope of meeting the right kind of partner. Thus these works suffer from *false matches*, when two threads of the same type meet, and from *timeouts*, when a producer and a consumer both pick distinct nodes and futilely wait for a partner to arrive. Most importantly, our algorithm performs extremely well in practice. On an UltraSPARC T2 Plus multicore machine, it outperforms Java's synchronous queue by up to 60×, the FC synchronous queue by up to 6×, and, when used as the elimination layer of a concurrent stack, yields 3.5× improvement over Hendler et al.'s FC stack [5]. On an Intel Nehalem multicore (supporting less parallelism), our algorithm surpasses the Java pool and the FC pool by 5× and 2×, respectively.

The asymmetric rendezvous problem and our progress property are formally defined in Sect. 2. Section 3 describes related work. The algorithm is presented in Sect. 4 and empirically evaluated in Sect. 5. We conclude in Sect. 6.

2 Preliminaries

Asymmetric rendezvous: In the *asymmetric rendezvous* problem there are threads of two types, producers and consumers. Producers perform `put(`x`)` operations that return `OK`. Consumers perform `get()` operations that return some item x handed off by a producer. Producers and consumers show up and must be matched with a unique thread of the other type, such that a producer invoking `put(`x`)` and a consumer whose `get()` returns x must be active concurrently.

Progress: To reason about progress while taking into account that rendezvous inherently requires waiting, we consider both types of operations' combined behavior. An algorithm **A** that implements asymmetric rendezvous is *nonblocking* if *some* operation completes after enough *concurrent* steps of threads performing `put()` operations *and* of threads performing `get()`. Note that, as with the definition of the lock-freedom property [8], there is no fixed a priori bound on the number of steps after which some operation must complete. Rather, we rule out implementations that make no progress at all, i.e., implementations where in some executions, both types of threads take steps infinitely often and yet no operation completes.

3 Related Work

Synchronous queues: A synchronous queue using three semaphores was described by Hanson [4]. Java 5 includes a coarse-grained locking synchronous queue, which was superseded in Java 6 by Scherer, Lea and Scott's algorithm [12]. Their algorithm is based on a Treiber-style nonblocking stack [16] that at all times contains rendezvous requests by either producers or consumers. A producer finding the stack empty or containing producers pushes itself on the stack and waits, but if it finds the stack holding consumers, it attempts to partner with the consumer at the top of the stack (consumers behave symmetrically). This creates a sequential bottleneck. Motivated by this, Afek, Korland, Natanzon, and Shavit described *elimination-diffracting (ED) trees* [1], a randomized distributed data structure where arriving threads follow a path through a binary tree whose internal nodes are *balancer* objects [14] and the leaves are Java synchronous queues. In each internal node a thread accesses an elimination array in attempt to avoid descending down the tree. Recently, Hendler et al. applied the *flat combining* paradigm [5] to the synchronous queue problem [6], describing single-combiner and parallel versions. In a single combiner FC pool, a thread attempts to become a *combiner* by acquiring a global lock on the queue. Threads that fail to grab the lock instead post their request and wait for it to be fulfilled by the combiner, which matches between the participating threads. In the parallel version there are multiple combiners that each handle a subset of participating threads and then try to satisfy unmatched requests in their subset by entering an *exchange* FC synchronous queue.

Elimination: The elimination technique is due to Touitou and Shavit [13]. Hendler, Shavit and Yerushalmi used elimination with an adaptive scheme inspired by Shavit and Zemach [15] to obtain a scalable linearizable stack [7]. In their scheme threads adapt locally: each thread picks a slot to collide in from sub-range of the collision layer centered around the middle of the array. If no partner arrives, the thread eventually shrinks the range. Alternatively, if the thread sees a waiting partner but fails to collide due to contention, it increases the range. In our adaptivity technique, described in Sect. 4, threads also make local decisions, but with global impact: the ring is resized. Moir et al. used elimination to scale a FIFO queue [11]. In their algorithm an enqueuer picks a random slot in an elimination array and waits there for a dequeuer; a dequeuer picks a random slot, giving up immediately if that slot is empty. It does not seek out waiting enqueuers. Scherer, Lea and Scott applied elimination in their *symmetric* rendezvous system [9], where there is only one type of a thread and so the pairing is between any two threads that show up. Scherer, Lea and Scott also do not discuss adaptivity, though a later version of their symmetric *exchanger channel*, which is part of Java [10], includes a scheme that resizes the elimination array. However, here we are interested in the more difficult *asymmetric* rendezvous problem, where not all pairings are allowed.

4 Algorithm Description

The pseudo code of the algorithm is provided in Fig. 1. The main data structure (Fig. 1a) is a ring of nodes. The ring is accessed through a central array `ring`, where `ring[`i`]` points to the i-th node in the ring.[1] A consumer attempts to capture a node in the ring and waits there for a producer, whereas a producer scans the ring, seeking a waiting consumer. Therefore, to guarantee progress the ring must have room for all the consumers that can be active simultaneously. (We expand on this in Sect. 4.4.) For simplicity, we achieve this by assuming the number of threads in the system, T, is known in advance and pre-allocating a ring of size T. In Sect. 4.3 we sketch a variant in which the maximum number of threads that can show up is not known in advance, i.e., adaptive also to the number of threads.

Conceptually each ring node should only contain an `item` pointer that encodes the node's state:

1. **Free**: `item` points to a global reserved object, `FREE`, that is distinct from any object a producer may enqueue. Initially all nodes are free.
2. **Captured by consumer:** `item` is `NULL`.
3. **Holding data (of a producer):** `item` points to the data.

In practice, ring traversal is more efficient by following a pointer from one node to the next rather than the alternative, traversing the array. Array traversal suffers from two problems. First, in Java reading from the next array cell may result in a cache miss (it is an array of pointers), whereas reading from the (just accessed) current node does not. Second, maintaining a running array index requires an expensive test+branch to handle index boundary conditions or counting modulo the ring size, while reading a pointer is cheap. The pointer field is named `prev`, reflecting that node i points to node $i-1$. This allows the ring to be *resized* with a single atomic `compareAndSet` (CAS) that changes `ring[1]`'s (the head's) `prev` pointer. To support mapping from a node to its ring index, each node holds its ring index in a read-only `index` field.

For the sake of clarity we start in Sect. 4.1 by discussing the algorithm without adaptation of the ring size. The adaptivity code, however, is included and is marked by a ▷ symbol in Fig. 1, and is discussed in Sect. 4.2. Section 4.3 sketches some practically-motivated extensions to the algorithm, e.g. supporting timeouts and adaptivity to the total number of threads. Finally, Sect. 4.4 discusses correctness and progress.

4.1 Nonadaptive Algorithm

Producers (Fig. 1b): A producer searches the ring for a waiting consumer, and attempts to hand its data to it. The search begins at a node, s, obtained by hashing the thread's id (Line 19). The producer passes the ring size to the hash

[1] This reflects Java semantics, where arrays are of *references* to objects and not of objects themselves.

function as a parameter, to ensure the returned node falls within the ring. It then traverses the ring looking for a node captured by a consumer. Here the producer periodically *peeks* at the initial node *s* to see if it has a waiting consumer (Lines 24-26); if not, it checks the current node in the traversal (Lines 27-30). Once a captured node is found, the producer tries to deposit its data using a CAS (Lines 25 and 28). If successful, it returns.

```
1  struct node {
2    item : pointer to object
3    index : node's index in the ring
4    prev : pointer to previous ring node
5  }
6
7  shared vars:
8    ring : array [1,...,T] of pointers to nodes,
9           ring [1]. prev = ring[T],
10          ring[i]. prev = ring[i−1] (i > 1)
11
12 utils:
13 getRingSize() {
14   node tail := ring [1]. prev
15   return tail.index
16 }
```

(a) Global variables

```
17 put(threadId, object item) {
18
19   node s := ring[hash(threadId,
20                        getRingSize())]
21   node v := s.prev
22
23   while (true) {
24     if (s.item == NULL) {
25       if (CAS(s.item, NULL, item))
26         return OK
27     } else if (v.item == NULL) {
28       if (CAS(v.item, NULL, item))
29         return OK
30       v := v.prev
31     }
32   }
33 }
```

(b) Producer code

```
34 get(threadId) {
35   int ringsize , busy_ctr, ctr := 0
36   node s, u
37
38   ringsize := getRingSize()
39   s := ring[hash(threadId, ringsize )]
40   (u,busy_ctr) := findFreeNode(s,ringsize)
41   while (u.item == NULL) {
42 ▷   ringsize := getRingSize()
43 ▷   if (u.index > ringsize and
44 ▷       CAS(u.item, NULL, FREE) {
45 ▷     s := ring[hash(threadId, ringsize )]
46 ▷     (u,busy_ctr) := findFreeNode(s, ringsize)
47 ▷   }
48 ▷   ctr := ctr + 1
49   }
50
51   item := u.item
52   u.item := FREE
53 ▷ if (busy_ctr < $T_d$ and ctr > $T_w$ and ringsize>1)
54 ▷   // Try to decrease ring
55 ▷   CAS(ring[1].prev, ring[ ringsize ], ring[ ringsize −1])
56   return item
57 }
58
59 findFreeNode(node s, int ringsize ) {
60 ▷ int busy_ctr := 0
61   while (true) {
62     if (s.item == FREE and
63         CAS(s.item, FREE, NULL))
64       return (s,busy_ctr)
65     s := s.prev
66 ▷   busy_ctr := busy_ctr + 1
67 ▷   if (busy_ctr > $T_i$·ringsize) {
68 ▷     if ( ringsize < T) // Try to increase ring
69 ▷       CAS(ring[1].prev, ring[ ringsize ], ring[ ringsize +1])
70 ▷     ringsize := getRingSize()
71 ▷     busy_ctr := 0
72 ▷   }
73   }
74 }
```

(c) Consumer code

Fig. 1. Asymmetric rendezvous algorithm. Lines beginning with ▷ handle the adaptivity process.

Consumers (Fig. 1c): A consumer searches the ring for a free node and attempts to capture it by atomically changing its `item` pointer from `FREE` to `NULL` using a CAS. Once a node is captured, the consumer spins, waiting for a producer to arrive and deposit its item. Similarly to the producer, a consumer hashes its id to obtain a starting point for its search, *s* (Line 39). The consumer calls the `findFreeNode` procedure to traverse the ring from *s* until it captures and returns node *u* (Lines 59-74). (Recall that the code responsible for handling adaptivity, which is marked by a ▷, is ignored for the moment.) The consumer then waits until a producer deposits an item in u (Lines 41-49), frees u (Lines 51-52) and returns (Line 56).

4.2 Adding Adaptivity

If the number of active consumers is smaller than the ring size, producers may need to traverse through a large number (linear in the ring size) of empty nodes before finding a match. It is therefore important to *decrease* the ring size if the concurrency level is low. On the other hand, if there are more concurrent threads than the ring size (high contention), it is important to dynamically *increase* the ring. The goal of the adaptivity scheme is to keep the ring size "just right" and allow threads to complete their operations within a constant number of steps.

The logic driving the resizing process is in the consumer's code, which detects when the ring is overcrowded or sparsely populated and changes the size accordingly. If a consumer fails to capture many nodes in `findFreeNode()` (due to not finding a free node or having its CASes fail), then the ring is too crowded and should be increased. The exact threshold is determined by an *increase threshold* parameter, T_i. If `findFreeNode()` fails to capture more than $T_i \cdot$ `ring_size` it attempts to increase the ring size (Lines 67-72). To detect when to decrease the ring, we observe that when the ring is sparsely populated, a consumer usually finds a free node quickly, but then has to wait longer until a producer *finds it*. Thus, we add a *wait threshold*, T_w, and a *decrease threshold*, T_d. If it takes a consumer more than T_w iterations of the loop in Lines 41-49 to successfully complete the rendezvous, but it successfully captured its ring node in up to T_d steps, then it attempts to decrease the ring size (Lines 53-55).

Resizing the ring is made by CASing the `prev` pointer of the ring head (`ring[1]`) from the current tail of the ring to the tail's successor (to increase the ring size) or its predecessor (to decrease the ring size). If the CAS fails, then another thread has resized the ring and the consumer continues. The head's `prev` pointer is not a sequential bottleneck because resizing is a rare event in stable workloads. Even if resizing is frequent, the thresholds ensure that the cost of the CAS is negligible compared to the other work performed by the algorithm, and resizing pays-off in terms of better ring size which leads to improved performance.

Handling consumers left out of the ring: A decrease in the ring size may leave a consumer's captured node outside of the current ring. Therefore, each consumer periodically checks if its node's index is larger than the current ring size (Line 43). If so, it tries to free its node using a CAS (Line 44) and find itself a new node in the ring (Lines 45-46). However, if the CAS fails, then a producer has already deposited its data in this node and so the consumer can take the data and return (this will be detected in the next execution of Line 41).

4.3 Pragmatic Extensions and Considerations

Here we sketch a number of extensions that might be interesting in certain practical scenarios.

Timeout support: Aborting a rendezvous that is taking more than a specified period of time is a useful feature. Unfortunately, our model has no notion of time and so we do not model the semantics of timeouts. We simply describe the details for an implementation that captures the intuitive notion of a timeout. The operations take a parameter specifying the desired timeout, if any. Timeout expiry is then checked in each iteration of the main loop in `put()`, `get()` and `findFreeNode()`. If the timeout expires, a producer or a consumer in findFreeNode() aborts by returning. A consumer that has captured a ring node cannot abort before freeing the node by CASing its `item` from `NULL` back to `FREE`. If the CAS fails, the consumer has found a match and its rendezvous has completed successfully. Otherwise its abort attempt succeeded and it returns.

Avoiding ring pre-allocation: Ring pre-allocation requires in advance knowledge of the maximum number of threads that can simultaneously run, which may not be known a priori. The maximum ring size can be dynamic by allowing threads to add and delete nodes from the ring. To do this we exploit the Java semantics, which forces the `ring` variable to be a *pointer* to the array. This allows a consumer to allocate a new ring (larger or smaller) and CAS `ring` to point to it. Active threads periodically check if `ring` has changed and move themselves to the new ring. We omit the details due to space limitations.

Busy waiting: Both producers and consumers rely on busy waiting, which may not be appropriate for *blocking* applications that wish to let a thread sleep until a partner arrives. Being blocking, such applications may not need our strong progress guarantee. We plan to extend our algorithm to blocking scenarios in future work.

4.4 Correctness

We consider the point at which both `put(`x`)` and the `get()` that returns x take effect as the point where the producer successfully CASes x into some node's `item` pointer (Line 25 or 28). The algorithm thus clearly meets the asymmetric rendezvous semantics. We next sketch a proof that, assuming threads do not request timeouts, the algorithm is nonblocking (as defined in Sect. 2). Assume towards a contradiction that there is an execution in which both producers and consumers take infinitely many steps and yet no rendezvous successfully completes. Since there is a finite number of threads T, there must be a producer/-consumer pair, p and c, each of which runs forever without completing. Suppose c never captures a node. Then eventually the ring size must become T. This is because after c completes a cycle around the ring in `findFreeNode()`, it tries to increase the ring size (say the increase threshold is 1). c's resizing CAS (Line 69) either succeeds or fails due to another CAS that succeeded in increasing, because the ring size decreasing implies a rendezvous completing, which by our assumption does not occur. Once the ring size reaches T, the ring has room for c (since T is the maximum number of threads). Thus c fails to capture a node only by encountering another consumer twice at different nodes, implying this consumer completes a rendezvous, a contradiction. It therefore cannot be that c never captures a node. Instead, c captures some node but p never finds c on the ring, implying that c has been left out of the ring by some decreasing resize. But, since from that point on, the ring size cannot decrease, c eventually executes Lines 43-46 and moves itself into p's range, where either p or another producer rendezvous with it, a contradiction.

5 Evaluation

We evaluate the performance of our algorithm in comparison with prior unfair synchronous queues (Sect. 5.1) and demonstrate the performance gains due to the adaptivity and the peeking techniques). Then we evaluate the resulting stack performance against other concurrent stacks (Sect. 5.2).

Experimental setup: Evaluation is carried out on both a Sun SPARC T5240 and on an Intel Core i7. The Sun has two UltraSPARC T2 Plus (Niagara II) chips. Each is a multithreading (CMT) processor, with 8 1.165 HZ in-order cores with 8 hardware strands per core, for a total of 64 hardware strands per chip. The Intel Core i7 920 (Nehalem) processor has four 2.67GHz cores, each multiplexing 2 hardware threads. Both Java and C++ implementations are tested.[2] Due to space limitations, we focus more on the Sun platform which offers more parallelism and better insight into the scaling behavior of the algorithm. Unless stated otherwise, results are averages of ten 10-second runs of the Java implementation on an idle machine, resulting in very little variance.

Adaptivity parameters used: $T_i = 1$, $T_d = 2$, and $T_w = 64$. A modulo hash function, $\texttt{hash}(t) = t \mod \texttt{ringsize}$, is used since we don't know anything about the thread ids and consider them as uniformly random. If however thread ids are *sequential* the modulo hash achieves perfect coverage of the ring with few collisions and helps in pairing producers/consumers that often match with each other. We evaluate these effects by testing our algorithms with both random and sequential thread ids throughout all experiments.

5.1 Synchronous Queues

Our algorithm (marked **AdaptiveAR** in the figures) is compared against the Java unfair synchronous queue (JDK), ED tree, and FC synchronous queues. The original authors' implementation of each algorithm is used.[3] We tested both JDK

[2] Java benchmarks were ran with HotSpot Server JVM, build `1.7.0-ea-b137`. C++ benchmarks were compiled with Sun C++ 5.9 on the SPARC machine and with `gcc` 4.3.3 (`-O3` optimization setting) on the Intel machine. In the C++ experiments we used the Hoard 3.8 [3] memory allocator.

[3] We remove all statistics counting from the code and use the latest JVM. Thus, the results we report are usually slightly better than those reported in the original papers. On the other hand, we fixed a bug in the benchmark of [6] that miscounted timed-out operations of the Java pool as successful operations; thus the results we report for it are sometimes lower.

and JDK-no-park: a version that always uses busy waiting instead of yielding the CPU (so-called parking), and report its results for the workloads where it improves upon the standard Java pool (as was done in [6]).

Producer/Consumer Throughput

N : N producer/consumer symmetric workload: We measure the throughput at which data is transferred from N producers to N consumers. We focus first on single chip results in Figs. 2a and 2e. The Nehalem results are qualitatively similar to the low thread counts SPARC results. Other than our algorithm, the parallel FC pool is the only algorithm that shows meaningful scalability. Our rendezvous algorithm outperforms the parallel FC queue by $2\times-3\times$ in low concurrency settings, and by up to 6 at high thread counts.

Hardware performance counter analysis (Fig. 2b-2d) shows that our rendezvous completes in less than 170 instructions, of which one is a CAS. In the parallel FC pool, waiting for the combiner to pick up a thread's request, match it, and report back with the result, all add up. While the parallel FC pool hardly

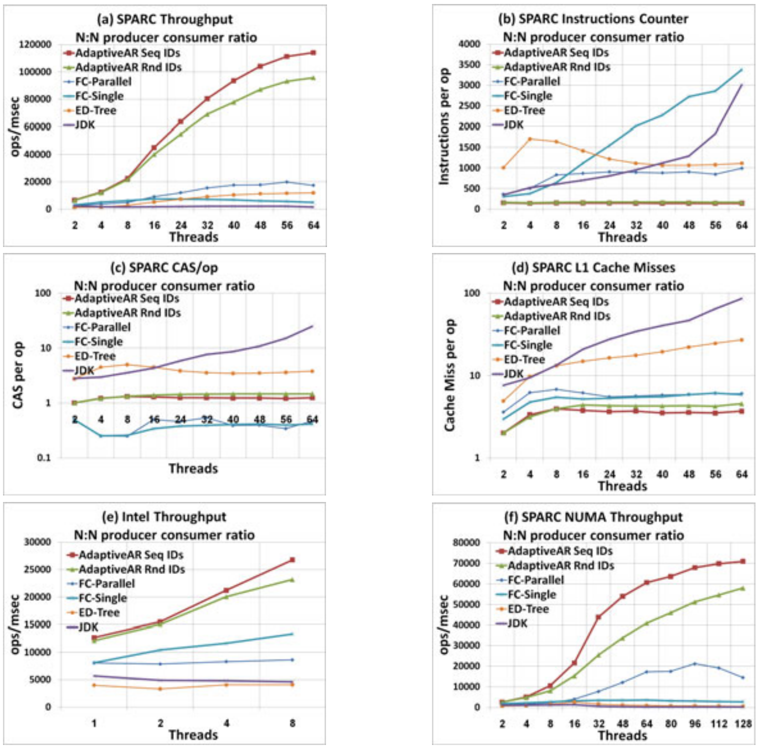


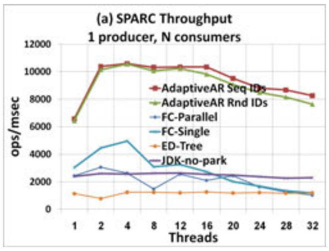
Fig. 2. Rendezvousing between N pairs of producers and consumers. Performance counter plots are logarithmic scale. L2 misses are not shown; all algorithms but JDK had less than one L2 miss/operation on average.

performs CASes, it does require between 3× to 6× more instructions to complete an operation. Similarly, ED tree's rendezvous operations require 1000 instructions to complete and incur more cache misses as concurrency increases. Our algorithm therefore outperforms it by at least 6× and up to 10× at high thread counts. The Java synchronous queue fails to scale in this benchmark. Figures 2b and 2c show its serializing behavior. Due to the number of failed CAS operations on the top of the stack (and consequent retries), it requires more instructions to complete an operation as concurrency grows. Consequently, our algorithm outperforms it by more than 60×.

Adaptivity and peeking impact: From Fig. 2c we deduce that ring resizing operations are a rare event, leading to an average number of one CAS per operation. Thus resizing does not adversely impact performance here. Throughput of the algorithm with and without peeking is compared in Figure 4a. While peeking has little effect at low thread counts, it improves performance by as much as 47% once concurrency increases. Because, due to adaptivity, a producer's initial node being empty usually means that a consumer will arrive there shortly, and peeking increasing the chance of pairing with that consumer. Without it a producer has a higher chance of colliding with another producer and consequently spending more instructions (and CASes) to complete a rendezvous.

NUMA test: When utilizing both processors of the Sun machine the operating system's default scheduling policy is to place threads round-robin on both chips. Thus, the cross-chip overhead is noticeable even at low thread counts, as Fig. 2f shows. Since the threads no longer share a single L2 cache, they experience an increased number of L1 and L2 cache misses; each such miss is expensive, requiring coherency protocol traffic to the remote chip. The effect is catastrophic for serializing algorithms; for example, the Java pool experiences a 10× drop in throughput. The more concurrent algorithms, such as parallel FC and ours, show scaling trends similar to the single chip ones, but achieve lower throughput. In the rest of the section we therefore focus on the more interesting single chip case.

$1 : N$ *asymmetric workloads:* The throughput of one producer rendezvousing with a varying number of consumers is presented in Figs. 3a and 4c. Nehalem results (Fig. 4c) are again similar and not discussed in detail. Since the throughput is bounded by the rate of a single producer, little scaling is expected and observed. However, for all algorithms (but the ED tree) it takes several consumers to keep up with a single producer. This is because the producer hands off an item and completes, whereas the consumer needs to notice the rendezvous has occurred. And while the single consumer is thus busy the producer cannot make progress on a new rendezvous. However, when more consumers are active, the producer can immediately return and find another (different) consumer ready to rendezvous with. Unfortunately, as shown in Figure 3a, most algorithms do not sustain this peak throughput. The FC pools have the worst degradation (3.74× for the single version, and 3× for the parallel version). The Java pool's degradation is minimal (13%), and it along with the ED tree achieves close to



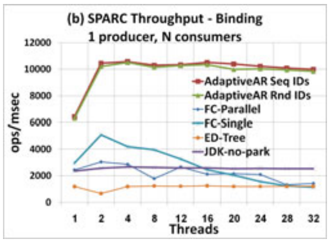


Fig. 3. Single producer and N consumers rendezvousing. **Left:** Default OS scheduling. **Right:** OS constrained to not co-locate producer on same cores as consumers.

peak throughput even at high thread counts. Yet this throughput is low: our algorithm outperforms the Java pool by up to 3× and the ED tree by 12× for low consumer counts and 6× for high consumer counts, despite degrading by 23% – 28% from peak throughput.

Why our algorithm degrades is not obvious. The producer has its pick of consumers in the ring and should be able to complete a hand-off immediately. The reason for this degradation is not algorithmic, but due to *contention on chip resources.* A Niagara II core has two pipelines, each shared by four hardware strands. Thus, beyond 16 threads some threads must share a pipeline — and our algorithm indeed starts to degrade at 16 threads. To prove that this is the problem, we present in Fig. 3b runs where the producer runs on the first core and consumers are scheduled only on the remaining seven cores. While the

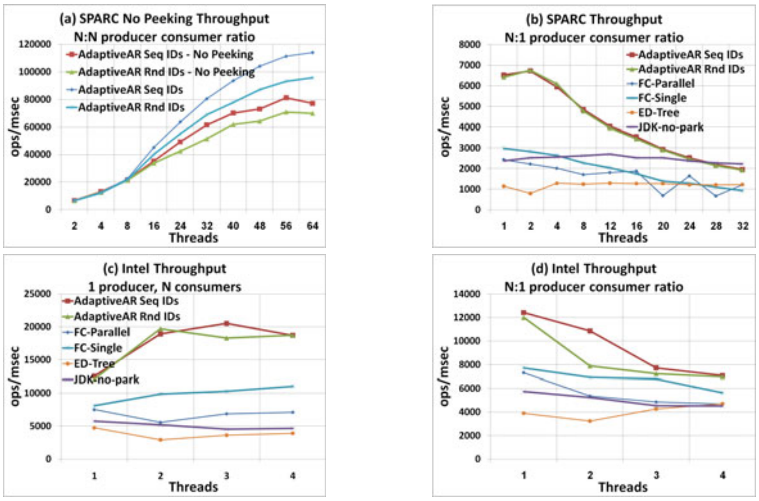


Fig. 4. Top left: Impact of peeking on $N:N$ rendezvous. **Top right:** Rendezvousing between N producers and a single consumer. **Bottom:** Intel $1:N$ and $N:1$ producer/consumer workloads throughput.

trends of the other algorithms are unaffected, our algorithm now maintains peak throughput through all consumer counts.

Results from the opposite workload (which is less interesting in real life scenarios), where multiple producers try to serve a single consumer, are given in Figs. 4b and 4d. Here the producers contend over the single consumer node and as a result the throughput of our algorithm degrades as the number of producers increases (as do the FC pools). Despite this degradation, our algorithm outperforms the Java pool up to 48 threads (falling behind by 15% at 64 threads) and outperforms the FC pools by about 2×.

Bursts: To evaluate the effectiveness of our adaptivity technique, we measure the rendezvous rate in a workload that experiences bursts of activity (on the Sun machine). For ten seconds the workload alternates every second between 31 thread pairs and 8 pairs. The 63rd hardware strand is used to take ring size measurement. Our sampling thread continuously reads the ring size, and records the time whenever the current read differs from the previous read. Figure 5a depicts the result, showing how the algorithm continuously resizes its ring. Consequently, it successfully benefits from the available parallelism in this workload, outperforming the Java pool by at least 40× and the parallel FC pool by 4× to 5×.

Varying arrival rate: In practice, threads do some work between rendezvouses. We measure how the throughput of the producer/consumer pairs workload is affected when the thread arrival rate decreases due to increasingly larger amounts

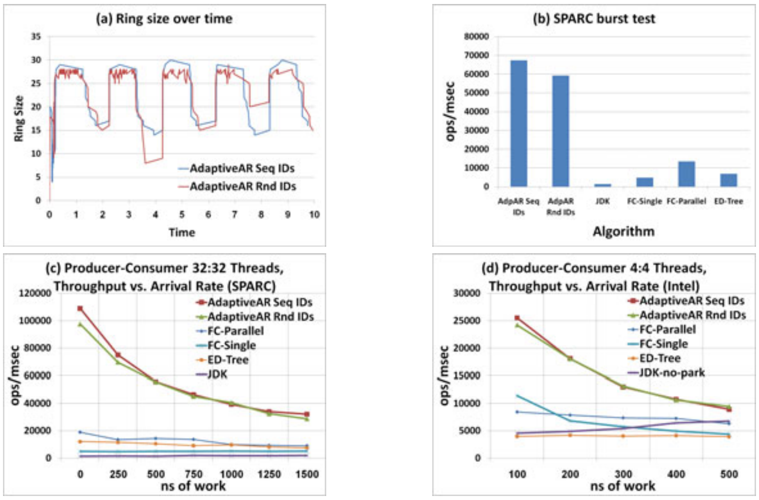


Fig. 5. Top: Synchronous queue bursty workload throughput. Left: Our algorithm's ring size over time (sampled continuously using a thread that does not participate in the rendezvousing). Right: throughput. **Bottom:** $N : N$ rendezvousing with decreasing arrival rate due to increasing amount of work time operations.

of time spent doing "work" before each rendezvous. Figures 5c and 5d show that as the work period grows the throughput of all algorithms that exhibit scaling deteriorates, due to reduced parallelism in the workload. E.g., on the SPARC the parallel FC degrades by 2× when going from no work to 1.5μs of work, and our algorithm degrades by 3.4× (because it starts much higher). Still, on the SPARC there is sufficient parallelism to allow our algorithm to outperform the other implementations by a factor of at least three. (On the Nehalem, by 31%.)

Work uniformity: One clear advantage of our algorithm over FC is work *uniformity*, since the combiner in FC is expected to spend much more time doing work for other threads. We show this by comparing the percent of total operations performed by each thread in a multiple producer/multiple consumer workload. In addition to measuring uniformity, this test is a yardstick for progress *in practice*: if a thread starves, we will see it as performing very little work compared to other threads. We pick the best result from five executions of 16 producer/consumer pairs, and plot the percent of total operations performed by each thread. Figure 6 shows the results. In an ideal setting, each thread would perform 3.125% (1/32) of the work. Our algorithm comes relatively close to this distribution, as does the JDK algorithm. In contrast, the parallel FC pool experiences much larger deviation and best/worst ratio.

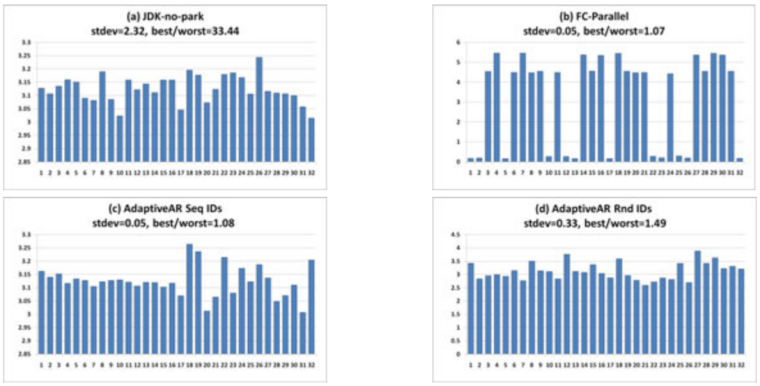


Fig. 6. Percent of total operations performed by each of 32 threads in an $N : N$ test

5.2 Concurrent Stack

Here we use our algorithm as the elimination layer on top of a Treiber-style nonblocking stack [16] and compare it to the stack implementations evaluated in [5]. We implemented two variants. In the first, following [7], rendezvous is used as a *backoff* mechanism that threads turn to upon detecting contention on the main stack, thus providing good performance under low contention and scaling as contention increases. In the second variant a thread first visits the rendezvous structure, accessing the main stack only if it fails to find a partner. If it then

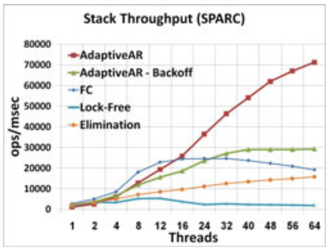


Fig. 7. Comparing stack implementations throughput. Each of N threads performs both `push` and `pop` operations with probability 1/2 for each operation type.

encounters contention on the main stack it goes back to try the rendezvous, and so on.

We compare a C++ implementation of our algorithm to the C++ implementations evaluated in [5]: a lock-free stack (LF-Stack), a lock-free stack with a simple elimination layer (EL-Stack), and an FC based stack. We use the same benchmark as [5], measuring the throughput of an even `push`/`pop` operation mix. Unlike the pool tests, here we want threads to give up if they don't find a partner in a short amount of time, and move to the main stack. We thus need to set T_w to a value smaller than the timeout, to enable a decrease of the ring size. Figure 7 shows the results. Our second (non-backoff) stack scales well, outperforming the FC and elimination stacks by more than 3.5×. The price it pays is poor performance at low concurrency (2.5× slower than the FC stack with a single thread). The backoff variant fails to scale above 32 threads due to contention on the stack head, illustrating the cost incurred by merely trying (and failing) to CAS a central hotspot.

6 Conclusion

We have presented an adaptive, nonblocking, high throughput asymmetric rendezvous system that scales well under symmetric workloads and maintains peak throughput in asymmetric (more consumers than producers) workloads. This is achieved by a careful marriage of new algorithmic ideas and attention to implementation details, to squeeze all available cycles out of the processors.

Several directions remain open towards making the algorithm even more broadly applicable and practical. In ongoing work we are extending the algorithm so that it maintains peak throughput even when there are more producers than consumers, and pursuing making the algorithm's space consumption adaptive. Adapting the algorithm to blocking applications (i.e., avoiding busy waiting) and dynamically choosing the various threshold parameters are also interesting questions.

Our results also raise a question about the flat combining paradigm. Flat combining has a clear advantage in inherently sequential data structures, such as a FIFO or priority queue, whose concurrent implementations have central hot spots. But as we have shown, FC may lose its advantage in problems with inherent potential for parallelism. It is therefore interesting whether the FC technique can be improved to match the performance of our asymmetric rendezvous system.

Availability: Our implementation is available on Tel Aviv University's Multicore Algorithmics group web site at `http://mcg.cs.tau.ac.il/`.

Acknowledgments: We thank Hillel Avni, Nati Linial and Nir Shavit for helpful discussions, and the anonymous reviewers for their comments.

References

[1] Afek, Y., Korland, G., Natanzon, M., Shavit, N.: Scalable producer-consumer pools based on elimination-diffraction trees. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 151–162. Springer, Heidelberg (2010)

[2] Andrews, G.R.: Concurrent programming: principles and practice. Benjamin-Cummings Publishing Co. Inc., Redwood City (1991)

[3] Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: a scalable memory allocator for multithreaded applications. SIGARCH Computer Architecture News 28(5), 117–128 (2000)

[4] Hanson, D.R.: C interfaces and implementations: techniques for creating reusable software. Addison-Wesley Longman Publishing Co., Inc., Boston (1996)

[5] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, pp. 355–364. ACM, New York (2010)

[6] Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Scalable flat-combining based synchronous queues. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 79–93. Springer, Heidelberg (2010)

[7] Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004, pp. 206–215. ACM, New York (2004)

[8] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13, 124–149 (1991)

[9] Scherer III, W.N., Lea, D., Scott, M.L.: A scalable elimination-based exchange channel. In: Workshop on Synchronization and Concurrency in Object-Oriented Languages, SCOOL 2005 (October 2005)

[10] Lea, D., Scherer III, W.N., Scott, M.L.: java.util.concurrent.exchanger source code (2011), `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/Exchanger.java`

[11] Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2005, pp. 253–262. ACM, New York (2005)

[12] Scherer III, W.N., Lea, D., Scott, M.L.: Scalable synchronous queues. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2006, pp. 147–156. ACM, New York (2006)

[13] Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks: preliminary version. In: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1995, pp. 54–63. ACM, New York (1995)

[14] Shavit, N., Zemach, A.: Diffracting trees. ACM Transactions on Computer Systems (TOCS) 14, 385–428 (1996)

[15] Shavit, N., Zemach, A.: Combining funnels: A dynamic approach to software combining. Journal of Parallel and Distributed Computing 60(11), 1355–1387 (2000)

[16] Treiber, R.K.: Systems programming: Coping with parallelism. Tech. Rep. RJ5118, IBM Almaden Research Center (2006)