Scalable Join Patterns

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Claudio V. Russo
Microsoft Research
crusso@microsoft.com

Abstract

Coordination can destroy scalability in parallel programming. A comprehensive library of scalable synchronization primitives is therefore an essential tool for exploiting parallelism. Unfortunately, such primitives do not easily combine to yield solutions to more complex problems. We demonstrate that a concurrency library based on Fournet and Gonthier's join calculus can provide declarative and scalable coordination. By *declarative*, we mean that the programmer needs only to write down the constraints of a coordination problem, and the library will automatically derive a correct solution. By *scalable*, we mean that the derived solutions deliver robust performance both as the number of processors increases, and as the complexity of the coordination problem grows. We validate our claims empirically on seven coordination problems, comparing our generic solution to specialized algorithms from the literature.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Algorithms, Languages, Performance

Keywords message passing, concurrency, parallelism

1. Introduction

Parallel programming is the art of keeping many processors busy with real work. But except for embarrassingly-parallel cases, solving a problem in parallel requires coordination between threads, which entails waiting. When coordination is unavoidable, it must be carried out in a way that minimizes both waiting time and interprocessor communication. Effective implementation strategies vary widely, depending on the coordination problem. Asking an application programmer to grapple with these concerns—without succumbing to concurrency bugs—is a tall order.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.


```
var j = Join.Create();
Synchronous.Channel[]  hungry;
Asynchronous.Channel[] chopstick;
j.Init(out hungry, n); j.Init(out chopstick, n);
for (int i = 0; i < n; i++) {
  var left = chopstick[i];
  var right = chopstick[(i+1) % n];
  j.When(hungry[i]).And(left).And(right).Do(() => {
    eat(); left(); right(); // replace chopsticks
  });
}
```

Figure 1. Dining Philosophers, declaratively

The proliferation of specialized synchronization primitives is therefore no surprise. For example, `java.util.concurrent` [13] contains a rich collection of carefully engineered classes, including various kinds of locks, barriers, semaphores, count-down latches, condition variables, exchangers and futures, together with nonblocking collections. Several of these classes led to research publications [14, 15, 26, 28]. A JAVA programmer faced with a coordination problem covered by the library is therefore in great shape.

Inevitably, though, programmers are faced with new problems not directly addressed by the primitives. The primitives must be composed into a solution. Doing so correctly and scalably can be as difficult as designing a new primitive.

Take the classic Dining Philosophers problem [3], in which philosophers sitting around a table must coordinate their use of the chopstick sitting between each one; such competition over limited resources appears in many guises in real systems. The problem has been thoroughly studied, and there are solutions using primitives like semaphores that perform reasonably well. There are also many natural "solutions" that do not perform well—or do not perform *at all*. Naive solutions may suffer from deadlock, if for example each philosopher picks up the chopstick to their left, and then finds the one to their right taken. Correct solutions may scale poorly with the number of philosophers (threads). For example, using a single global lock to coordinate philosophers is correct, but will force non-adjacent philosophers to take turns through the lock, adding unnecessarily sequentialization. Avoiding these pitfalls takes experience and care.

In this paper, we demonstrate that Fournet and Gonthier's *join calculus* [4, 5] provides the basis for a declarative and scalable coordination library. By *declarative*, we mean that the programmer needs only to write down the constraints for a coordination problem, and the library will automatically derive a correct solution. By *scalable*, we mean that the derived solutions deliver robust, competitive performance both as the number of processors or cores increase, and as the complexity of the coordination problem grows.

Figure 1 shows a solution to Dining Philosophers using our library, which is a drop-in replacement for Russo's C♯ JOINS library [24]. The library uses the message-passing paradigmn of the join calculus. For Dining Philosophers, we use two arrays of channels (`hungry` and `chopstick`) to carry value-less messages; being empty, these messages represent unadorned events. The declarative aspect of this example is the *join pattern* starting with `j.When`. The declaration says that when events are available on the channels `hungry[i]`, `left`, and `right`, they may be *simultaneously and atomically* consumed. When the pattern fires, the philosopher, having obtained exclusive access to two chopsticks, eats and then returns the chopsticks. In neither the join pattern nor its body is the order of the chopsticks important. The remaining details are explained in §2.[1]

Most implementations of join patterns, including Russo's, use coarse-grained locks to achieve atomicity, resulting in poor scalability (as we show experimentally in §4). *Our contribution is a new implementation of the join calculus that uses ideas from fine-grained concurrency to achieve scalability on par with custom-built synchronization primitives*:

- We first recall how join patterns can be used to solve a wide range of coordination problems (§2), as is well-established in the literature [2, 4, 5]. The examples provide basic implementations of some of the `java.util.concurrent` classes mentioned above.[2] In each case, the JOINS-based solution is as straightforward to write as the one for dining philosophers.
- To maximize scalability, we must allow concurrent, independent processing of messages, avoid centralized contention, and spin or block intelligently. This must be done, of course, while guaranteeing atomicity and progress for join patterns.

  We meet these challenges by (1) using lock-free [9] data structures to store messages, (2) treating messages as resources that threads can race to take possession of, (3) avoiding enqueuing messages when possible ("lazy" queuing and specialized channel representations), and (4) allowing barging ("message stealing").

  We present our algorithm incrementally, both through diagrams and through working C♯ code (§3).
- We validate our scalability claims experimentally on seven different coordination problems (§4). For each coordination problem we evaluate a joins-based implementation running in both Russo's lock-based library and our new fine-grained library. We compare these results to the performance of direct, custom-built solutions.

  In all cases, our new library scales significantly better than Russo's. We scale competitively with—sometimes better than—the custom-built solutions, though we suffer from higher constant-time overheads in some cases.
- We state and sketch proofs for the key safety and liveness properties characterizing our algorithm (§5).

Our goal is not to replace libraries like `java.util.concurrent`, but rather to complement them, by exposing some of their insights in a way easily and safely useable by application programmers. We discuss this point further, along with related work in general, in the final section.

2. The JOINS library

Russo's `Joins` [24] is a concurrency library derived from Fournet and Gonthier's *join calculus* [4, 5], a process algebra similar to Milner's π-calculus but conceived for efficient implementation. In this section, we give an overview of the library API and illustrate it using examples drawn from the join calculus literature [2, 4, 5].

The join calculus takes a message-passing approach to concurrency where messages are sent over channels and channels are themselves first-class values that can be sent as messages. What makes the calculus interesting is the way messages are received. Programs do not actively request to receive messages from a channel. Instead, they employ *join patterns* (also called *chords* [2]) to declaratively specify reactions to message arrivals. The power of join patterns lies in their ability to respond *atomically* to messages arriving simultaneously on several different channels.

Suppose, for example, that we have two channels `Put` and `Get`, used by producers and consumers of data. When a producer and a consumer message are available, we would like to receive both simultaneously, and transfer the produced value to the consumer. Using Russo's `Joins` API, we write:

```
class Buffer<T> {
  public readonly Asynchronous.Channel<T> Put;
  public readonly Synchronous<T>.Channel Get;
  public Buffer() {
    Join j = Join.Create(); // allocate a Join object
    j.Init(out Put);        // bind its channels
    j.Init(out Get);
    j.When(Get).And(Put).Do // register chord
      (t => { return t; });
  }
}
```

[1] By default, our implementation provides only *probabilistic* fairness (and hence probabilistic starvation freedom); see §3.6 for more detail.

[2] Our versions lack some features of the real library, such as timeouts and cancellation, but these should be straightforward to add (§3.6).

This simple example introduces many aspects of the API.

First, we are using two different kinds of channels: `Put` is an asynchronous channel that carries messages of type `T`, while `Get` is a synchronous channel that returns `T` replies but takes no argument. A sender never blocks on an asynchronous channel, even if the message cannot immediately be received through a join pattern. For the `Buffer` class, that means that a single producer may send many `Put` messages, even if none of them are immediately consumed. Because `Get` is a synchronous channel, on the other hand, senders will block until or unless a pattern involving it is enabled. Synchronous channels also return a reply to message senders; the reply is given as the return value of join patterns.

Join patterns are declared using `When`. The single join pattern in `Buffer` stipulates that when one `Get` request and one `Put` message are available, they should both be consumed. After specifying the involved channels through `When` and `And`, the `Do` method is used to give the *body* of the join pattern. The body is a piece of code to be executed whenever the pattern is matched and relevant messages consumed. It is given as a delegate (C♯'s first-class functions), taking as arguments the contents of the messages. In `Buffer`, the two channels `Get` and `Put` yield only one argument, because `Get` messages take no argument. The body of the pattern simply returns the argument `t` (from `Put`), which then becomes the reply to the `Get` message. Altogether, each time the pattern is matched, one `Get` and one `Put` message are consumed, and the argument is transferred from `Put` to the sender of `Get`.

Channels are represented as delegates, so that messages are sent by simply invoking a channel as a function. From a client's point of view, `Put` and `Get` look just like methods of `Buffer`. If `buf` is an an instance of `Buffer`, a producer thread can post a value `t` by calling `buf.Put(t)`, and a consumer thread can request a value by calling `buf.Get()`.

Finally, each channel must be associated with an instance of the `Join` class.[3] Such instances are created using the static factory method `Join.Create`, which optionally takes the maximum number of required channels. Channels are bound using the `Init` method of the `Join` class, which initializes them using an out-parameter. These details are not important for the overall design, and are elided from subsequent examples. The full API—including the determination of types for join pattern bodies—is given in Appendix A.

As we have seen, when a single pattern mentions several channels, it forces synchronization:

```
Asynchronous.Channel<A> Fst;
Asynchronous.Channel<B> Snd;
Synchronous<Pair<A,B>>.Channel Both;
// create j and init channels (elided)
j.When(Both).And(Fst).And(Snd).Do((a,b) =>
    new Pair<A,B>(a,b));
```

[3] This requirement retains compatibility with Russo's original JOINS library; we also use it for the stack allocation optimization described in §3.6.

The pattern will consume messages `Fst(a)`, `Snd(b)` and `Send()` atomically, when all three are available. Only the first two messages carry arguments, so the body of the pattern takes two arguments. Its return value, a pair, becomes the return value of the call to `Both()`.

On the other hand, several patterns may mention the same channel, expressing choice:

```
Asynchronous.Channel<A> Fst;
Asynchronous.Channel<B> Snd;
Synchronous<Sum<A,B>>.Channel Either;
// create j and init channels (elided)
j.When(Either).And(Fst).Do(a => new Left<A,B>(a));
j.When(Either).And(Snd).Do(b => new Right<A,B>(b));
```

Each pattern can fulfill a request on `Either()`, by consuming a message `Fst(a)` or a message `Snd(b)`, and wrapping the value in a variant of a disjoint sum.

Using what we have seen, we can build a simple (non-recursive) `Lock`. As in the dining philosophers example, we use void-argument, void-returning channels as *signals*. The `Release` messages are tokens that indicate that the lock is free to be acquired; it is initially free. Clients must follow the protocol of calling `Acquire()` followed by `Release()` to obtain and relinquish the lock. Protocol violations will not be detected by this simple implementation. However, when clients follow the protocol, the code will maintain the invariant that at most one `Release()` token is pending on the queues and thus at most one client can acquire the lock.

```
class Lock {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Lock() {
    // create j and init channels (elided)
    j.When(Acquire).And(Release).Do(() => { });
    Release();  // initially free
  }
}
```

With a slight generalization, we can obtain a *semaphore*:

```
class Semaphore {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Semaphore(int n) {
    // create j and init channels (elided)
    j.When(Acquire).And(Release).Do(() => { });
    for (; n > 0; n--) Release(); // initially n free
  }
}
```

A semaphore allows at most n clients to `Acquire` the resource and proceed; further acquisitions must wait until another client calls `Release()`. We arrange this by priming the basic `Lock` implementation with n initial `Release()` tokens.

We can also generalize `Buffer` to a synchronous channel that exchanges data between threads:

```
class Exchanger<A, B> {
  readonly Synchronous<Pair<A, B>>.Channel<A> left;
  readonly Synchronous<Pair<A, B>>.Channel<B> right;
  public B Left(A a)  { return left(a).Snd; }
  public A Right(B b) { return right(b).Fst; }
  public Exchanger() {
    // create j and init channels (elided)
    j.When(left).And(right).Do((a,b) =>
      new Pair<A,B>(a,b));
  }
}
```

Dropping message values, we can also implement an n-way *barrier* that causes n threads to wait until all have arrived:

```
class SymmetricBarrier {
  public readonly Synchronous.Channel Arrive;
  public SymmetricBarrier(int n) {
    // create j and init channels (elided)
    var pat = j.When(Arrive);
    for (int i = 1; i < n; i++) pat = pat.And(Arrive);
    pat.Do(() => { });
  }
}
```

This example is unusual in that its sole join pattern mentions a single channel `n` times: the pattern is *nonlinear*. This repetition means that the pattern will not be enabled until the requisite n threads have arrived at the barrier, and our use of a single channel means that the threads need not distinguish themselves by invoking distinct channels (hence "symmetric"). On the other hand, if the coordination problem did call for separating threads into groups (eg. gender is useful in a parallel genetic algorithm [26]), it is easy to do so. We can construct a barrier requiring n threads of one kind and m threads of another, simply by using two channels.

We can also implement a tree-structured variant of an asymmetric barrier, which breaks a single potentially large n-way coordination problem into $O(n)$ two-way problems:

```
class TreeBarrier {
  public readonly Synchronous.Channel[] Arrive;
  private readonly Join j; // create j, init chans ...
  public TreeBarrier(int n) {Wire(0, n-1, () => {});}
  private void Wire(int low, int high, Action Done) {
    if (low == high) j.When(Arrive[low]).Do(Done);
    else if (low + 1 == high)
      j.When(Arrive[low]).And(Arrive[high]).Do(Done);
    else { // low + 1 < high
      Synchronous.Channel Left, Right; // init chans
      j.When(Left).And(Right).Do(Done);
      int mid = (low + high) / 2;
      Wire(low, mid, () => Left());
      Wire(mid + 1, high, () => Right());
    }
  }
}
```

Such tree-structured barriers (or more generally, combiners) have been studied in the literature (see [9] for a survey); the point here is just that adding tree-structured coordination is straightforward using join patterns. As we show in §4, the tree-structured variant performs substantially better than the flat barrier, although both variants easily outperform the .NET `Barrier` class (a standard sense-reversing barrier).

Finally, we can implement a simple reader-writer lock [2], using private asynchronous channels (`idle` and `shared`) to track the internal state of a synchronization primitive:

```
class ReaderWriterLock {
  private readonly Asynchronous.Channel idle;
  private readonly Asynchronous.Channel<int> shared;
  public readonly Synchronous.Channel AcqR, AcqW,
                                      RelR, RelW;
  public ReaderWriterLock() {
    // create j and init channels (elided)
    j.When(AcqR).And(idle).Do(() => shared(1));
    j.When(AcqR).And(shared).Do(n => shared(n+1));
    j.When(RelR).And(shared).Do(n => {
      if (n == 1) idle(); else shared(n-1);
    });
    j.When(AcqW).And(idle).Do(() => { });
    j.When(RelW).Do(() => idle());
    idle(); // initially free
  }
}
```

While we have focused on the simplest synchronization primitives as a way of illustrating JOINS, join patterns can be used to declaratively implement more complex concurrency patterns, from Larus and Parks-style *cohort-scheduling* [2], to Erlang-style *agents* or *active objects* [2], to stencil computations with systolic synchronization [25], as well as classic synchronization puzzles [1].

3. Scalable join patterns

We have seen, through a range of examples, how the join calculus allows programmers to solve coordination problems by merely writing down the relevant constraints. Now we turn to the primary concern of this paper: an implementation that solves these constraints in a scalable way.

The chief challenge in implementing the join calculus is providing atomicity when firing patterns: messages must be noticed and withdrawn from multiple collections simultaneously. A simple way to ensure atomicity, of course, is to use a lock, and this is what most implementations do (§6).[4] For example, Russo's original library associates a single lock with each `Join` object. Each sender must acquire the lock and, while holding it, enqueue their message and determine whether any patterns are thereby enabled. Russo puts significant effort into shortening the critical section: he uses bitmasks summarizing message availability to accelerate pattern matching [12], represents `void` asynchronous

[4] Some implementations use STM [29], which we discuss in §6.

channels as counters, and permits "message stealing" to increase throughput—all the tricks from Benton *et al*. [2].

Unfortunately, even with relatively short critical sections, such coarse-grained locking inevitably limits scalability. The scalability problems with locks are well-documented [19], and they are especially pronounced if we use the JOINS library to *implement* low-level synchronization primitives. At a high level, the problem with coarse-grained locking is that it serializes the process of matching and firing chords: at most one thread can be performing that work at a time. In cases like the exchanger and Dining Philosophers, a much greater degree of concurrency is both possible and desirable. At a lower level, coarse-grained locking can drastically increase interprocessor communication, because all threads processing related messages are reading and writing to a single shared location—the lock status. Since memory bandwidth tends to be quite limited, this extra traffic inhibits scalability, especially for low-level coordination (§4).

In summary, we want an implementation of JOINS that matches and fires chords concurrently while minimizing costly interprocessor communication.

3.1 Overview

We need to permit highly-concurrent access to the collection of messages available on a given channel. Therefore, we use *lock-free* [9] bags to represent channels which allow truly concurrent examination and alteration of data. The result is that, for example, two threads can be simultaneously adding separate messages to the same channel bag, while a third examines a message already stored in the bag—without any of the threads waiting on any other, and in many cases without any memory bus traffic. In choosing a bag rather than, say, a queue, we sacrifice message ordering guarantees to achieve greater concurrency: FIFO ordering imposes a sequential bottleneck on queues. The original join-calculus did not provide any ordering guarantees, and relaxed ordering is typical in implementations [2, 4, 24]. This choice is not fundamental, and ordered channels are easily provided (§3.6). None of our examples rely on ordering.

Lock-free bags allow messages to be added and inspected concurrently, but they do not solve the central problem of *atomically consuming* a pattern's worth of messages. To provide atomic matching, we equip messages with a `Status` field of the following type:

```
enum Stat { PENDING, CLAIMED, CONSUMED };
```

Statuses are determined according to an optimistic protocol:

- Each message is `PENDING` to begin with, meaning that it is available for matching and firing.
- Matching consists of finding sufficiently many `PENDING` messages, then using CAS (see below) to try to change them one by one to from `PENDING` to `CLAIMED`.
- If matching is successful, each message can be marked `CONSUMED` without issuing a memory fence. If it is unsuccessful, each `CLAIMED` message is reverted to `PENDING`, again without a fence.

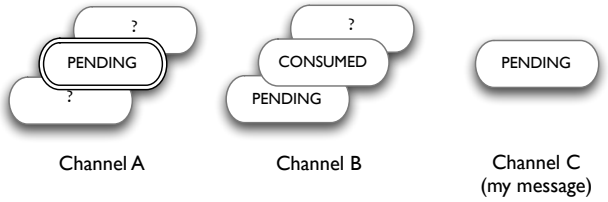
We use compare-and-swap (CAS) to ensure correct mutual exclusion: CAS *atomically* updates a memory location when its current value matches some expected value. Thus, the status field acts as a kind of lock, but one tied to individual messages rather than an entire instance of `Join`. Further, if we fail to "acquire" a message, we do not immediately spin or block. Instead, we can continue looking through the relevant bag of messages for another message to claim—or, more generally, for another join pattern to match (§3.3).

Figure 2 walks through an example run of the protocol.

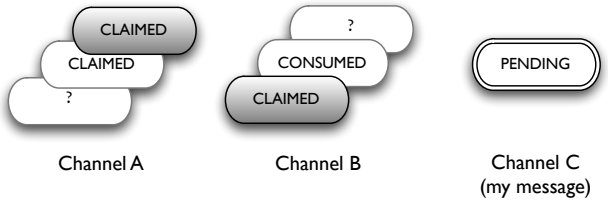
Example `When(A).And(B).And(C).Do(...)`, send on `C`

Legend { Lozenges are messages (with last read status); Double border: about to CAS to `CLAIMED`; Shaded interior: won CAS race; `CLAIMED` by us

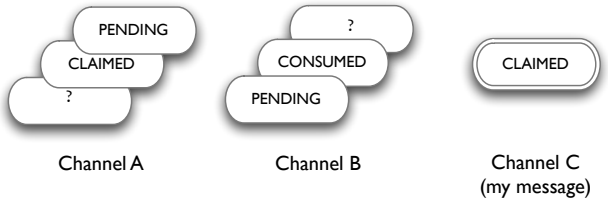
Snapshot 1 Found `PENDING` messages in each relevant bag, including our own message. About to CAS, trying to claim the message from channel A:



Snapshot 2 Failed to claim the message (`CLAIMED` by another thread), but succeeded in claiming another message on A and one on B. About to CAS the final message:



Snapshot 3 Failed to claim the message (`CLAIMED` by another thread). Rollback *our* `CLAIMED` messages to `PENDING`:



Takeaway When running the protocol, the only things we know for sure are: our `CLAIMED` messages (the shaded lozenges) will remain `CLAIMED`, and any `CONSUMED` messages will remain `CONSUMED`. New messages may appear any time.

Figure 2. Example: failing to fire

```
// Msg implements:
  Chan Chan         { get; };
  Stat Status       { get; set; };
  bool TryClaim();  // CAS from PENDING to CLAIMED
  Signal Signal     { get; };
  Match ShouldFire  { get; set; };
  object Result     { get; set; };
```

```
// Chan<A> implements:
  Chord[] Chords              { get; };
  bool IsSync                 { get; };
  Msg AddPending(A a);
  Msg FindPending(out bool sawClaims);
```

```
// Match implements:
  Chord Chord       { get; };
  Msg[] Claims      { get; };
```

Figure 3. The interfaces to our key data structures

There are three reasons the above is just an "overview" and not the full algorithm:

- Knowing when to terminate the above protocol with the result of "no pattern matched" turns out to be subtle: because of the concurrent nature of the message bags, new potential matches can occur at any time. Terminating the protocol is important for returning control to an asynchronous sender, or deciding to block a synchronous sender. But terminating too soon can result in dropped (undetected, but enabled) matches, which can lead to deadlock. The full protocol, including its termination condition, is given in §3.3.
- Patterns involving synchronous channels add further complexity: if an asynchronous message causes such a pattern to be fired, it must alert a synchronous waiter, which must in turn execute the pattern body. Likewise, if there are multiple synchronous senders in a given pattern, they must be coordinated so that only one executes the body and communicates the results to the others. We cover these details in §3.4.
- Two "optimizations" of the protocol turn out to be crucial for achieving scalability: lazy queueing and message stealing. The details of these optimizations are spelled out in §3.5, while their ramifications on scalability are examined empirically in §4.

3.2 Representation

Next we give our implementation using a simplified version of its C♯ code. We begin the interfaces for key data structures, in Figure 3. The `get` and `set` keywords are used to specify getters and setters for properties in .NET.

The `Msg` class, in addition to carrying a message payload, includes a `Chan` field tying it to a particular channel, and the `Status` field discussed above. The `Chan` field is just a convenience for the presentation in this paper. The remaining `Msg` fields (`Signal`, `ShouldFire` and `Result`) are used for blocking on synchronous channels (§3.4).

The `Chan<A>` class implements a lock-free bag of messages of type `A`. The key operations on a channel are:

- `AddPending`, which takes a message payload and atomically adds a `Msg` with `PENDING` status to the bag.
- `FindPending`, which returns (but does not remove) some message in the bag observed to have a `PENDING` status; of course, by the time control is returned to the caller, the status may have been altered by a concurrent thread. If no `PENDING` messages were observed at some linearization point,[5] `null` is returned, and the out-parameter `sawClaims` reflects whether any `CLAIMED` messages were observed at that linearization point.

Notice that `Chan<A>` does not provide any direct means of removing messages; in this respect, it is not a traditional bag. Any message with status `CONSUMED` is considered *logically* removed from the bag, and will be *physically* removed from the data structure when convenient. Our JOINS implementation obeys the invariant that once a message is marked `CONSUMED`, its status is never changed again. We do not detail our bag implementation here; it is similar to a lock-free queue [18], except of course that `FindPending` can traverse the whole bag. A more aggressive bag implementation could increase concurrency further.

`Match` is a simple, immutable class used to communicate data making up a matched pattern: a chord, and an array of `CLAIMED` messages sufficient to fire it.

3.3 The core algorithm: resolving a message

We have already discussed the key safety property for a JOINS implementation: pattern matching should be atomic. In addition, an implementation should ensure at least the following liveness property (assuming a fair scheduler):

If a chord can fire, eventually some *chord is fired.*[6]

Our strategy is to drive the firing of chords by the concurrent arrival of each message: each sender must resolve its own message. We consider a message *resolved* if:

1. It is marked `CLAIMED` by the sending thread, along with sufficiently many other messages to fire a chord; or
2. It is marked `CONSUMED` by another thread, and hence was used to fire a chord; or
3. No pattern can be matched using only the message and messages that arrived *prior* to it.

Ensuring that each message is eventually resolved is tricky, because message bags and statuses are constantly, concur-

[5] A *linearization point* [10] is the moment at which an operation that is *observably* atomic, but not *actually* atomic, is considered to take place.

[6] Notice that this property does not guarantee fairness; see §5.

```
1  Match Resolve(Msg msg) {
2    var backoff = new Backoff();
3    while (true) {
4      bool retry = false;
5      foreach (var chord in msg.Chan.Chords) {
6        Msg[] claims = chord.TryClaim(msg, ref retry);
7        if (claims != null)
8          return new Match(chord, claims);
9      }
10     if (!retry || msg.Status == Stat.CONSUMED)
11       return null;
12     backoff.Once();
13   }
14 }
```

Figure 4. Resolving a message

```
1  partial class Chord {
2    Chan[] Chans;  // the channels making up this chord
3
4    Msg[] TryClaim(Msg msg, ref bool retry) {
5      var msgs = new Msg[Chans.length];  // cached
6
7      // locate enough pending messages to fire chord
8      for (int i = 0; i < Chans.Length; i++) {
9        if (Chans[i] == msg.Chan) {
10         msgs[i] = msg;
11       } else {
12         bool sawClaims;
13         msgs[i] = Chans[i].FindPending(out sawClaims);
14         retry = retry || sawClaims;
15         if (msgs[i] == null) return null;
16       }
17     }
18
19     // try to claim the messages we found
20     for (int i = 0; i < Chans.Length; i++) {
21       if (!msgs[i].TryClaim()) {
22         // another thread won the race; revert
23         for (int j = 0; j < i; j++)
24           msgs[j].Status = Stat.PENDING;
25         retry = true;
26         return null;
27       }
28     }
29
30     return msgs; // success: each message CLAIMED
31   }
32 }
```

Figure 5. Racing to claim a chord involving `msg`

rently in flux. In particular, just as one thread determines that its message `msg` does not enable any chord, another message in another thread may arrive that enables a chord involving `msg`. The key is that each sender need only take responsibility for the messages that came before its own; if a later sender enables a chord, that later sender is responsible for it.

Given the highly concurrent nature of message bags, what does it mean for one message to arrive before another?

There is no need to provide a direct way of asking this question. Instead, we rely on the linearizability [10] of the bag implementation. Linearizability means that we can think of calls to `AddPending` and `FindPending` (along with CASes to `Status`) as being executed atomically, in some global sequential order. The result is that all messages—even those added to distinct bags—are *implicitly* ordered by the time of their insertion. The bag interface does not provide a way to directly observe this ordering, but `FindPending` respects it: if one thread executes

```
Msg m1 = bag1.AddPending(x);
bool sawClaims;
Msg m2 = bag2.FindPending(out sawClaims);
```

then `m2 == null` only if no message in `bag2` prior to `m1` is `PENDING`. This is the only guarantee we need in order to safely stop looking for matches.

Figure 4 gives our implementation of resolution. The `Resolve` method takes a message `msg` that has already been added to the appropriate channel's bag and loops until the message has been resolved. We first attempt to "claim" a chord involving `msg`, successively trying each chord in which `msg`'s channel is involved (lines 5–9). The `Chord` class's `TryClaim` method either returns an array of messages (which includes `msg`) that have all been `CLAIMED` by the current thread, or `null` if claiming failed. In the latter case, the `retry` by-ref parameter is set to true if any of the involved message bags contained a message `CLAIMED` by another thread.

Cumulatively, the `retry` flag records whether an externally-`CLAIMED` message was seen in *any* failing chord. We must track such `CLAIMED` messages because they are unstable, in the sense that they may be reverted to `PENDING`, possibly enabling a chord for which the sender is still responsible.

The first way a message can be resolved—by claiming it and enough other messages to make up a chord—corresponds to the `return` on line 8. The second two ways correspond to the `return` on line 11. If none of the three conditions hold, we must try again. We perform exponential backoff (line 12) in this case, because repeated retrying can only be caused by high contention over messages. Resolution may fail to terminate, but only if the system as a whole is making progress (according to our liveness property above); see §5 for a proof sketch.

Figure 5 gives the code for `TryClaim`, which works in two phases. In the first phase (lines 8–17), we first try to locate sufficiently many `PENDING` messages to fire the chord. We are required to claim `msg` in particular. If we are unable to find enough messages, we exit (line 15) without having written to memory. Otherwise, we enter the second phase (lines 20–28), wherein we race to claim each message. The message-level `TryClaim` method performs a CAS on the `Status` field,

```
1  void AsyncSend<A>(Chan<A> chan, A a){
2    Msg myMsg = chan.AddPending(a);
3    Match m = Resolve(myMsg);
4
5    if (m == null) return; // no chord to fire
6    ConsumeAll(m.Claims);
7
8    if (m.Chord.IsAsync) {
9      // asynchronous chord: fire in a new thread
10     new Thread(m.Fire).Start();
11   } else {
12     // synchronous chord: wake a synchronous caller
13     for (int i = 0; i < m.Chord.Chans.Length; i++) {
14       // pick the first synchronous caller
15       if (m.Chord.Chans[i].IsSync) {
16         m.Claims[i].ShouldFire = m;
17         m.Claims[i].Signal.Set(); // assumed fence
18         return;
19       }
20     }
21   }
22 }
```

```
23 R SyncSend<R, A>(Chan<A> chan, A a) {
24   Msg myMsg = chan.AddPending(a);
25   Match m = Resolve(myMsg);
26
27   if (m == null) { // myMsg CONSUMED, or no match
28     myMsg.Signal.Block();
29     m = myMsg.ShouldFire;
30     if (m == null) return myMsg.Result; // rendezvous
31   } else {
32     ConsumeAll(m.Claims);
33   }
34
35   var r = m.Fire();
36   for (int i = 0; i < m.Chord.Chans.Length; i++) {
37     if (m.Chord.Chans[i].IsSync && // rendezvous
38         m.Claims[i] != myMsg) {
39       m.Claims[i].Result = r;      // transfer result
40       m.Claims[i].Signal.Set();    // assumed fence
41     }
42   }
43   return (R)r;
44 }
```

Figure 6. Sending a message

ensuring that only one thread will succeed in claiming a given message. If at any point we fail to claim a message, we roll back all of the messages claimed so far (lines 23–24). The implementation ensures that the `Chans` arrays for each chord are ordered consistently, so that in any race at least one thread entering the second phase will complete the phase successfully (§5).

Note that the code in Figure 5 is a simplified version of our implementation that does not handle patterns with repeated channels, and does not stack-allocate or recycle message arrays. These differences are discussed in §3.6.

3.4 Firing, blocking and rendezvous

Message resolution does not depend on the (a)synchrony of a channel, but the rest of the message-sending process does.

The code for sending messages is shown in Figure 6, with separate entry points `AsyncSend` and `SyncSend`. The actions taken while sending depend, in part, on the result of message resolution:

Send	We `CLAIMED`	They `CONSUMED`	No match
Async	(AC) Spawn (10) (SC) Wake (13–20)	Return (5)	Return (5)
Sync	Fire (35)	Wait for result (28)	Block (28)

where AC and SC, respectively, stand for asynchronous chord (no synchronous channels) and synchronous chord (at least one synchronous channel).

First we follow the path of an asynchronous message, which begins by adding and resolving the message (lines 2–3). If either the message was `CONSUMED` by another thread (in which case that thread is responsible for firing the chord) or no pattern is matchable, we immediately exit (line 5). In both of these cases, we are assured that any chords involving the message will be (or has been) dealt with elsewhere, and since the message is itself asynchronous, we need not wait for this to occur.

On the other hand, if we resolved the message by claiming it and enough other messages to fire a chord, we proceed by consuming all involved messages (which does not require a memory fence). If the chord's pattern involved only asynchronous channels (line 10) we spawn a new thread to execute the chord body asynchronously. Otherwise we must communicate with a synchronous sender, telling it to execute the body.

A key difference from asynchronous senders is that synchronous sending should not return until a relevant chord has fired. Moreover, synchronous chord bodies should be executed by a synchronous caller, rather than by a newly-spawned thread. Since multiple synchronous callers can be combined in a single chord, exactly one of them should be chosen to execute the chord, and then share the result with (and wake up) all the others (we call this "rendezvous").

Each synchronous message has a `Signal` associated with it. `Signals` provide methods `Block` and `Set`, allowing synchronous senders to block[7] and be woken. Calls to `Set` atomically trigger the signal. If a thread has already called `Block`, it is awoken and the signal is reset. Otherwise, the next call

[7] It spins a bit first; see §3.6

to `Block` will immediately return, again resetting the signal. We ensure that `Block` and `Set` are each called by at most one thread; the implementation ensures that waking only occurs as a result of triggering the signal (no "spurious wakeups").

If an asynchronous sender consumed a synchronous sender's message (line 6), the synchronous sender will be blocked waiting to be informed of this event (line 28). The asynchronous sender chooses *one* such synchronous sender to wake (lines 15–18), telling it which chord and messages to fire (line 16).

Now we consider synchronous senders. Just as before, we first add and resolve the message (lines 24–25). If the message was resolved by claiming enough additional messages to fire a chord, all relevant messages are immediately consumed (line 32). Otherwise, the sender must block (line 28).

There are two ways a blocked, synchronous sender can be woken: by an asynchronous sender or by another synchronous sender (rendezvous). In the former case, the (initially `null`) `ShouldFire` field will contain a match that the synchronous caller is responsible for firing. In the latter case, `ShouldFire` remains `null`, but the `Result` field will contain the result of a chord body as executed by another synchronous sender, which is immediately returned (line 30).

If a synchronous sender does execute a chord body (line 35), it must then wake up any *other* synchronous senders involved in the chord and inform them of the result (lines 36–42). For simplicity, we ignore the possibility that the chord body raises an exception, but proper handling is easy to add and is addressed by the benchmarked implementation.

3.5 Lazy queuing, counter channels, and message stealing

To achieve competitive scalability, we make three enhancements to the above implementation. The importance of these enhancements is shown empirically in §4.

First, we do not *always* need to allocate or enqueue a message when sending. For example, in the `Lock` class when sending an `Acquire` message, we could first check to see whether a corresponding `Release` message is available, and if so, immediately claim and consume it without ever touching the `Acquire` bag. This shortcut saves both on allocation and potentially on interprocessor communication.

More generally, we provide an optimistic fast-path that attempts to immediately match the remaining messages to fire a chord. We call this lazy queuing. Implementing lazy queuing is relatively straightforward, so we do not provide the code here.

The second enhancement applies to void, asynchronous channels (e.g. `Lock.Release`). Sophisticated lock-based implementations of join patterns typically optimize the representation of such channels to a simple counter, neatly avoiding the cost of allocation for messages used just as signals [2]. We have implemented a similar optimization, adapted to suit our protocol.

The main challenge in employing the counter representation is that, in our fine-grained protocol, it must be possible to *tentatively* decrement the counter (the analog of claiming a message), in such a way that other threads do not incorrectly assume the message has actually been consumed. Our approach is to represent void, asynchronous message bags as a word sized pair of half-words, separately counting claimed and pending messages. Implementations of, for example, `Chan.AddPending` and `Msg.TryClaim` are specialized to atomically update the shared-state word by CASing in a classic read-modify-try-update loop. For example, we claim a "message" as follows:

```
bool TryClaim() {
  uint startState = chan.state; // shared state
  uint curState = startState;
  while (true) {
    startState = curState;
    ushort claimed;
    ushort pending = Decode(startState, out claimed);
    if (pending > 0) {
      var nextState = Encode(++claimed, --pending);
      curState =  CAS(ref chan.state,
                      comparand: startState,
                      value: nextState);
      if (curState == startState)
        return true;
    } else return false;
  }
}
```

More importantly, `Chan.FindPending` no longer needs to traverse a data structure but can merely atomically read the bag's encoded state once, setting `sawClaimed` if the `claimed` count is non-zero.

While the counter representation avoids allocation, it does lead to more contention over the same shared state (compared with a proper bag). It also introduces the possibility of overflow, which we ignored here. Nevertheless, we have found it to be beneficial in practice (§4), especially for non-singleton resources like `Semaphore.Release` messages.

The final enhancement is only relevant to synchronous channels. When an asynchronous sender matches a synchronous chord, it consumes all the relevant messages, and then wakes up one of the synchronous senders. If the synchronous caller is actually blocked—so that waking requires a context switch—significant time may lapse before the chord is actually fired.

Since we do not provide a fairness guarantee, we can instead permit "stealing": we can wake up one synchronous caller, but roll back the rest of the messages to `PENDING` status, putting them back up for grabs by currently-active threads—including the thread that just sent the asynchronous message. In low-traffic cases, messages are unlikely to be stolen; in high-traffic cases, stealing is likely to lead to better throughput. This strategy is similar to the one taken in

```
1  bool foundSleeper = false;
2  for (int i = 0; i < m.Chord.Chans.Length; i++) {
3    if (m.Chord.Chans[i].IsSync && !foundSleeper) {
4      foundSleeper = true;
5      m.Claims[i].AsyncWaker = myMsg;  // hand over msg
6      m.Claims[i].Status = Stat.WOKEN;
7      m.Claims[i].Signal.Set();        // assumed fence
8    } else {
9      m.Claims[i].Status = Stat.PENDING; // relinquish
10   }
11 }
```

Figure 7. Waking a synchronous sender while allowing stealing; replaces lines 12–20 of `AsyncSend`

Polyphonic C♯ [2], as well as the "barging" allowed by the `java.util.concurrent` synchronizer framework [15].

Some care must be taken to ensure our key liveness property still holds: when an asynchronous message wakes a synchronous sender, it moves from a safely resolved state (`CLAIMED` as part of a chord) to an unresolved state (`PENDING`). There is no guarantee that the woken synchronous sender will be able to fire a chord involving the original asynchronous message (see [2] for an example). Yet `AsyncSend` simply returns to its caller. We must somehow ensure that the original asynchronous message is successfully resolved. Thus, when a synchronous sender is woken, we record the asynchronous message that woke it, transferring responsibility for the message.

The new code for an asynchronous sender notifying a synchronous waiter is shown in Figure 7; it replaces lines 12–20 of `AsyncSend`. We introduce a new status, `WOKEN`. A synchronous message is marked `WOKEN` if an asynchronous sender is transferring responsibility, and `CONSUMED` if a synchronous sender is going to fire a chord involving it. In both cases, the signal is set after the status is changed; in the latter case, it is set after the chord is actually fired and the return value is available. `Resolve` is revised to return `null` at any point that the message is seen at status `WOKEN` or `CONSUMED`.

The `WOKEN` status ensures that a blocked synchronous caller is woken only once, which is important both for our use of `Signal`, and to ensure that the synchronous sender will only be responsible for *one* waking asynchronous message. The message field `AsyncWaker` replaces `ShouldFire`, and is used to inform the woken thread which asynchronous message woke it up.

Figure 8 gives the revised code for sending a synchronous message in the presence of stealing; it is meant to replace lines 27–33 in the original implementation. A synchronous sender loops until its message is resolved by claiming a chord (exit on line 6), or by another thread consuming it (exit on line 13). In each iteration of the loop, the sender blocks (line 10); even if its message has already been `CONSUMED`, it must wait for the signal to get its return value. In the case

```
1  Msg waker = null;
2  var backoff = new Backoff();
3  while (true) {
4    m = Resolve(myMsg);
5
6    if (m != null) break;      // claimed a chord; exit
7    if (waker != null)
8      RetryAsync(waker);       // retry last async waker
9
10   myMsg.Signal.Block();
11
12   if (myMsg.Status == Stat.CONSUMED) {
13     return myMsg.Result;     // rendezvous
14   } else { // status is Stat.WOKEN
15     waker = myMsg.AsyncWaker; // will retry later
16     myMsg.Status = Stat.PENDING;
17     backoff.Once();
18   }
19 }
20
21 ConsumeAll(claims);
22 // retry last async waker, *after* consuming myMsg
23 if (waker != null) RetryAsync(waker);
```

Figure 8. Sending a synchronous message while coping with stealing; replaces lines 27–23 of `SyncSend`

that the synchronous sender is woken by an asynchronous message (lines 15–17), it records the waking message and ultimately tries once more to resolve its own message. We perform exponential backoff every time this happens, since continually being awoken only to find messages stolen indicates high traffic.

After every resolution of the synchronous sender's message (`myMsg`), the sender retries sending the last asynchronous message that woke it, if any (lines 7–8, 22). Doing so fulfills the liveness requirements outlined above: the synchronous sender has become responsible for the message that woke it. We use `RetryAsync`, which is similar to `AsyncSend` but uses an already-added message rather than adding a new one. We are careful to avoid calling `RetryAsync` while `myMsg` is `CLAIMED` by the calling thread; doing so could result in the retry code forever waiting for us to revert or consume the claim. On the other hand, it is fine to retry the message even if it has already been successfully consumed as part of a chord; `RetryAsync` will simply exit in this case.

3.6 Pragmatics and extensions

There are a few smaller differences between the presented code and the actual implementation:

- We avoid boxing (allocation) and downcasts whenever possible: on .NET, additional polymorphism (beyond what the code showed) can help avoid uses of `object`.
- We do not allocate a fresh message array every time `TryClaim` is called; in fact, we do not heap-allocate mes-

sage arrays at all. Instead, we stack-allocate an array[8] in `SyncSend` and `AsyncSend`, and reuse this array for every call to `TryClaim`.

- We handle nonlinear patterns, in which a single channel appears multiple times. The code is straightforward.

An important pragmatic point is that the `Signal` class first performs some spinwaiting before blocking. Spinning is performed on a memory location associated with the signal, so each spinning thread will wait on a distinct memory location whose value will only be changed when the thread should be woken, an implementation strategy long known to perform well on cache-coherent architectures [17]. The amount of spinning performed is determined adaptively on a per-thread, per-channel basis.

We expect it to be straightforward to add timeouts and nonblocking attempts for synchronous sends to our implementation, because we can always use CAS to consume a message we have added to a bag to abort a send (which will, of course, also discover if the send has already succeeded).

Finally, to add channels with ordering constraints one needs only use a queue or stack rather than a bag. Switching from bags to fair queues and disabling message stealing yields per-channel fairness for joins. In Dining Philosophers, for example, queues will guarantee that requests from waiting philosophers are fulfilled before those of philosophers that have just eaten. Such guarantees come at the cost of significantly decreased parallelism, since they entail sequential matching of join patterns. At an extreme, programmers can enforce a round-robin scheme for matching patterns using an additional internal channel [25].

4. Performance

In this section we provide empirical support for the claims that our implementation is (1) scalable and (2) competitive with purpose-built solutions.

4.1 Methodology

To evaluate our implementation, we constructed a series of microbenchmarks for seven classic coordination problems: dining philosophers, producers/consumers, mutual exclusion, semaphores, barriers, rendezvous, and reader-writer locking.

Our solutions for these problems are fully described in the first two sections of the paper. They cover a spectrum of shapes and sizes of join patterns. In some cases (producer/-consumer, locks, semaphores, rendezvous) the size and number of join patterns stays fixed as we increase the number of processors, while in others a single pattern grows in size (barriers) or there are an increasing number of fixed-size patterns (philosophers).

[8] Stack-allocated arrays are not directly provided by .NET, so we use a custom value type built by polymorphic recursion.

Each benchmark follows standard practice for evaluating synchronization primitives: we repeatedly synchronize, for a total of k synchronizations between n threads [17]. We use $k \geq$ 100,000 and average over three trials for all benchmarks. To test interaction with thread scheduling preemption, we let n range up to 96—twice the 48 cores in our benchmarking machine.

Each benchmark has two variants for measuring different aspects of synchronization:

PARALLEL SPEEDUP In the first variant, we simulate doing a *small* amount of work between synchronization episodes (and during the critical section, when appropriate). By performing some work, we can gauge to what extent a synchronization primitive inhibits or allows parallel speedup. By keeping the amount of work small, we gauge in particular speedup for *fine-grained* parallelism, which presents the most challenging case for scalable coordination.

PURE SYNCHRONIZATION In the second variant, we synchronize in a tight loop, yielding the cost of synchronization in the limiting case where the actual work is negligible. In addition to providing some data on constant-time overheads, this variant serves as a counterpoint to the previous one: it ensures that scalability problems were not hidden by doing too much work between synchronization episodes. Rather than looking for speedup, we are checking for slowdown.

To simulate work, we use .NET's `Thread.SpinWait` method, which spins in a tight loop for a specified number of times (and ensures that the spinning will not be optimized away). To make the workload a bit more realistic—and to avoid "lucky" schedules—we randomize the number of spins between each synchronization, which over 100,000 iterations will yield a normal distribution of total work with very small standard deviation. We ensure that the same random seeds are provided across trials and compared algorithms, so we always compare the same amount of total work. The mean spin counts are determined per-benchmark and given in the next section.

For each problem we compare performance between:

- a join-based solution using our fully-optimized implementation (S-Join, for "scalable joins"),
- a join-based solution using Russo's library (L-Join, for "lock-based joins"),
- at least one purpose-built solution from the literature or .NET libraries (label varies), and
- when relevant, our implementation with some or all optimizations removed to demonstrate the effect of the optimization (*e.g.*, S-J w/o S,C for dropping the Stealing and Counter optimizations; S-J w/o L for dropping the Lazy queuing optimization).

We detail the purpose-built solutions below.

Figure 9. Speedup on simulated *fine-grained* workloads: throughput versus threads

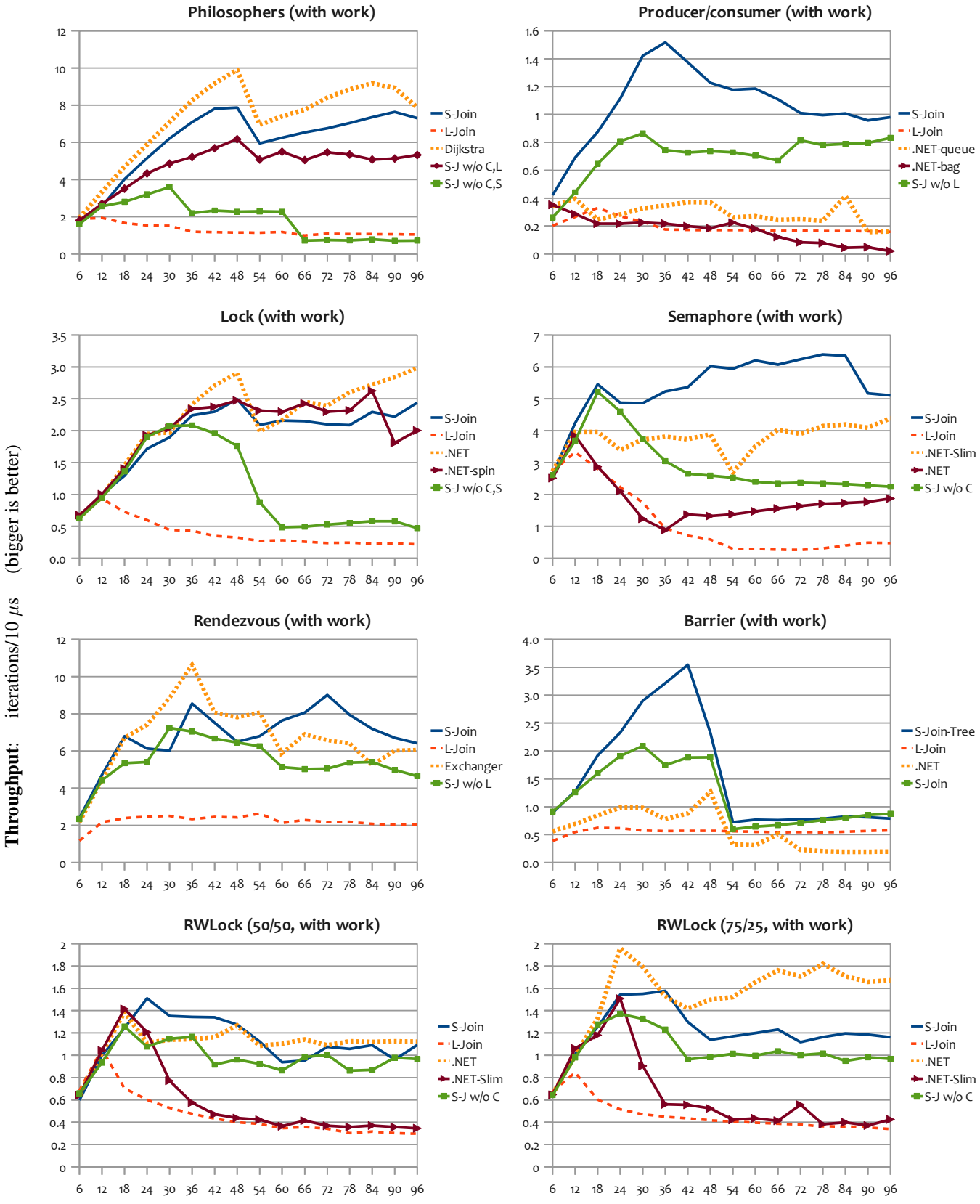
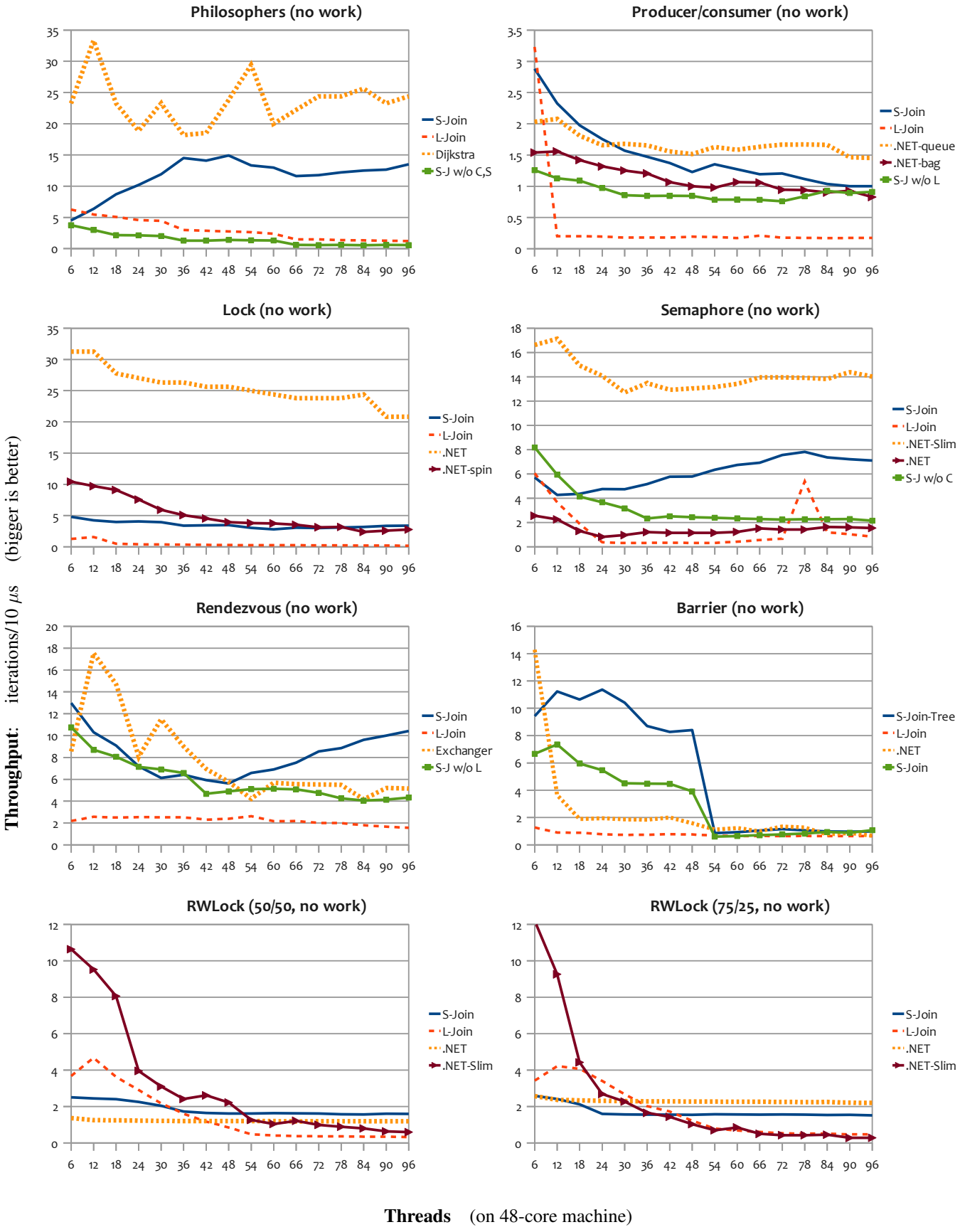


Figure 10. Pure synchronization performance: throughput versus threads



Two benchmarks (rendezvous and barriers) required extending Russo's library to support multiple synchronous channels in a pattern; in these cases, and only in these cases, we use a modified version of the library.

Our benchmarking machine has four AMD "Magny-Cours" Opteron 6100 Series 1.7GHz processors, with 12 cores each (for a total of 48 cores), 32GB RAM, and runs Windows Server 2008 R2 Datacenter. All benchmarks were run under the 64-bit CLR.

4.2 The benchmarks

The results for all benchmarks appear in Figure 9 (for parallel speedup) and Figure 10 (for pure synchronization). The axes are consistent across all graphs: the x-axis measures the number of threads, and the y-axis measures throughput (as iterations performed every 10μs). Larger y values reflect better performance.

For measuring parallel speedup, we used the following mean spin counts for simulated work:

Benchmark	In crit. section	Out of crit. section
Philosophers	25	5,000
Prod/Cons	N/A	producer 5,000
		consumer 500
Lock	50	200
Semaphore	25	100
Rendezvous	N/A	5,000
Barrier	N/A	10,000
RWLock	50	200

With too little simulated work, there is no hope of speedup; with too much, the parallelism becomes coarse-grained and thus insensitive to the performance of synchronization. These counts were chosen to be high enough that at least one implementation showed speedup, and low enough to yield significant performance differences.

The particulars of the benchmarks are as follows, where n is the number of threads and k the *total* number of iterations (so each thread performs k/n iterations):

Philosophers Each of the n threads is a philosopher; the threads are arranged around a table. An iteration consists of acquiring and then releasing the appropriate chopsticks. We compare against Dijkstra's original solution, using a lock per chopstick, acquiring these locks in a fixed order, and releasing them in the reverse order.

Producer/consumer We let $n/2$ threads be producers and $n/2$ be consumers. Producers repeatedly generate trivial output and need not wait for consumers, while consumers repeatedly take and throw away that output. We compare against the .NET 4 `BlockignCollection` class, which transforms a nonblocking collection into one that blocks when attempting to extract an element from an empty collection. We wrap the `BlockingCollection` around the .NET 4 `ConcurrentQueue` class (a variant of Michael and Scott's classic lock-free queue [18]) and `ConcurrentBag`.

Lock An iteration consists of acquiring and then releasing a single, global lock. We compare against both the built-in .NET lock (which is a highly-optimized part of the CLR implementation itself) and `System.Threading.SpinLock` (which is implemented in .NET).

Semaphore We let the initial semaphore count be $n/2$; An iteration consists of acquiring and then releasing the semaphore. We compare to two .NET semaphores: the `Semaphore` class, which wraps kernel semaphores, and `SemaphoreSlim`, a faster, pure .NET implementation of semaphores.

Rendezvous The n threads perform a total of k synchronous exchanges as quickly as possible. Unfortunately, .NET 4 does not provide a built-in library for rendezvous, so we ported Scherer *et al.*'s *exchanger* [26] from JAVA; this is the exchanger included in `java.util.concurrent`.

Barriers An iteration consists of passing through the barrier. We show results for both the tree and the flat versions of the join-based barrier. We compare against the .NET 4 `Barrier` class, a standard sense-reversing barrier.

RWLock An iteration consists of (1) choosing at random whether to be a reader or writer and (2) acquiring, and then releasing, the appropriate lock. We give results for 50-50 and 75-25 splits between reader and writers. We compare against two .NET implementations: the `ReaderWriterLock` class, which wraps the kernel RWLocks, and the `ReaderWriterLockSlim` class, which is a pure .NET implementation.

4.3 Analysis

The results of Figure 9 demonstrate that our scalable join patterns are competitive with—and can often beat—state of the art custom libraries. Application-programmers can solve coordination problems in the simple, declarative style we have presented here, and expect excellent scalability, even for fine-grained parallelism.

In evaluating benchmark performance, we are most interested in the slope of the throughput graph, which measures scalability with the number of cores (up to 48) and then scheduler robustness (from 48 to 96). In the parallel speedup benchmarks, both in terms of scalability (high slope) and absolute throughput, we see the following breakdown:

S-Join clear winner	Producer/consumer, Semaphore, Barrier
S-Join competitive	Philosophers, Lock, Rendezvous, RWLock 50/50
.NET clear winner	RWLock 75/25

The .NET concurrency library could benefit from replacing some of its primitives with ones based on the joins implementation we have shown—the main exception being locks. With further optimization, it may be feasible to build an entire scalable synchronization library around joins.

The performance of our implementation is mostly robust as we oversubscribe the machine. The Barrier benchmark

is a notable exception, but this is due to the structure of the problem: every involved thread must pass through the barrier at every iteration, so at $n > 48$ threads, a context switch is *required* for every iteration. Context switches are very expensive in comparison to the small amount of work we are simulating.

Not all is rosy, of course: the pure synchronization benchmarks show that scalable join patterns suffer from constant-time overheads in some cases, especially for locks. The table below approximates the overhead of pure synchronization in our implementation compared to the best .NET solution, by dividing the scalable join pure synchronization time by the best .NET pure synchronization time:

Overhead compared to best custom .NET solution

n	Phil	Pr/Co	Lock	Sema	Rend	Barr	RWL
6	5.2	0.7	6.5	2.9	0.7	1.5	4.2
12	5.2	0.9	7.4	4.0	1.7	0.3	3.9
24	1.9	0.9	6.6	3.0	1.1	0.2	1.8
48	1.6	1.2	7.4	2.3	1.0	0.2	1.4

(n threads; smaller is better)

Overheads are most pronounced for benchmarks that use .NET's built-in locks (Philosophers, Lock). This is not surprising: .NET locks are mature and highly engineered, and are not themselves implemented as .NET code. Notice, too, that in Figure 10 the overhead of the spinlock (which *is* implemented within .NET) is much closer to that of scalable join patterns. In the philosophers benchmark, we are able to compensate for our higher constant factors by achieving better parallel speedup, even in the pure-synchronization version of the benchmark.

One way to decrease overhead, we conjecture, would be to provide compiler support for join patterns. Our library-based implementation spends some of its time traversing data structures representing user-written patterns. In a compiler-based implementation, these runtime traversals could be unrolled, eliminating a number of memory accesses and conditional control flow. Removing that overhead could put scalable joins within striking distance of the absolute performance of .NET locks. On the other hand, such an implementation would probably not allow the dynamic construction of patterns that we use to implement barriers.

In the end, constant overhead is trumped by scalability: for those benchmarks where the constant overhead is high, our implementation nevertheless shows strong parallel speedup when simulating work. The constant overheads are dwarfed by even the small amount of work we simulate. Finally, even for the pure synchronization benchmarks, our implementation provides competitive scalability, in some cases extracting speedup despite the lack of simulated work.

Effect of optimizations Each of the three optimizations discussed in §3.5 is important for achieving competitive throughput. Stealing tends to be most helpful for those problems where threads compete for limited resources (Philosophers, Locks), because it minimizes the time between resource release and acquisition, favoring threads that are in the right place at the right time.[9] Lazy queuing improves constant factors across the board, in some cases (Producer/Consumer, Rendezvous) also aiding scalability. Finally, the counter representation provides a considerable boost for benchmarks, like Semaphore, in which the relevant channel often has multiple pending messages.

Performance of lock-based joins Russo's lock-based implementation of joins is consistently—often dramatically—the poorest performer for both parallel speedup and pure synchronization. On the one hand, this result is not surprising: the overserialization induced by coarse-grained locking is a well-known problem. On the other hand, Russo's implementation is fairly sophisticated in the effort it makes to shorten critical sections. The implementation includes all the optimizations proposed for POLYPHONIC C♯ [2], including a form of stealing, a counter representation for void-async channels (simpler than ours, since it is lock-protected), and bitmasks approximating the state of messages queues for fast matching. Despite this sophistication, it is clear that lock-based joins do not scale.

We consider STM-based join implementations in §6.2.

5. Correctness

There are many ways to characterize correctness of concurrent algorithms. The most appropriate specification for our algorithm is something like the process-algebraic formulation of the join calculus [4]. In that specification, multiple messages are consumed—and a chord is fired—in a single step. A full proof that our implementation satisfies this specification is too much to present here, and we have not yet carried out such a rigorous proof. We have, however, identified what we believe are the key lemmas for performing such a proof. We take for granted that our bag implementation is linearizable [10] and lock-free [9]; roughly, this means that operations on bags are observably atomic and dead/livelock free even under unfair scheduling. There are a pair of key properties—one safety, one liveness—characterizing the `Resolve` method:

Lemma 1 (Resolution Safety)**.** Assume that `msg` has been inserted into a channel. If a subsequent call to `Resolve(msg)` returns, `msg` is in a resolved state; moreover, the return value correctly reflects how the message was resolved.

Lemma 2 (Resolution Liveness)**.** Assume that threads are scheduled fairly. If a sender is attempting to resolve a message, eventually *some* message is resolved by its sender.

Recall that there are three ways a message can be resolved: it and a pattern's worth of messages can be marked `CLAIMED` by the calling thread; it can be marked `CONSUMED` by another thread; and it can be in an arbitrary status when it is

[9] This is essentially the same observation Doug Lea made about *barging* for abstract synchronizers [15].

determined that there are not enough messages *prior* to it to fire a chord.

Safety for the first two cases is fairly easy to show: we can assume that `CAS` works properly, and can see that

- Once a message is `CLAIMED` by a thread, the next change to its status is by that thread.
- Once a message is `CONSUMED`, its status never changes.

These facts mean that interference cannot "unresolve" a message that has been resolved in those three ways. The other fact we need to show is that the `retry` flag is only `false` if, indeed, no pattern is matched using only the message and messages that arrived before it. Here we use the linearizability assumptions about bags, together with the facts about the status flags just given.

Now we turn to the liveness property. Notice that a call to `Resolve` fails to return only if `retry` is repeatedly `true`. This can only happen as a result of messages being `CLAIMED`. We can prove, using the consistent ordering of channels during the claiming process, that if any thread reaches the claiming process (lines 33–42), *some* thread succeeds in claiming a pattern's worth of messages. The argument goes: claiming by one thread can fail only if claiming/consuming by another thread has succeeded, which means that the other thread has managed to claim a message on a higher-ranked channel. Since there are only finitely-many channels, some thread must have succeeded in claiming the last message it needed to match a pattern.

Using both the safety and liveness property for `Resolve`, we expect the following overall liveness property to hold:

Conjecture 1. Assume that threads are scheduled fairly. If a chord can be fired, eventually *some* chord is fired.

The key point here is that if a chord can be fired, then in particular some message, together with its predecessors, *does* match a pattern, which rules out the possibility that the message is resolved with no pattern matchable.

6. Related work

6.1 Join calculus

Fournet and Gonthier originally proposed the join calculus as an asynchronous process algebra designed for efficient implementation in a distributed setting [4, 5]. It was positioned as a more practical alternative to Milner's π-calculus.

The calculus has been implemented many times, and in many contexts. The earliest implementations include Fournet *et al.*'s JoCaml [7] and Odersky's Funnel [21] (the precursor to Scala), which are both functional languages supporting declarative join patterns. JoCaml's runtime is single-threaded so the constructs were promoted for concurrency control, not parallelism. Though it is possible to run several communicating JoCaml *processes* in parallel, pattern matching will always be sequential. Funnel targeted the Java VM, which can exploit parallelism, but we could find no evaluation of its performance on parallel hardware. Benton, Cardelli and Fournet proposed an object-oriented version of join patterns for C♯ called Polyphonic C♯ [2]; around the same time, von Itzstein and Kearney independently described JoinJava [11], a similar extension of Java. The advent of generics in C♯ 2.0 led Russo to encapsulate join pattern constructs in the Joins library [24], which served as the basis for our library. There are also implementations for Erlang [22], C++ [16], and VB [25].

All of the above implementations use coarse-grained locking to achieve the atomicity present in the join calculus semantics. In some cases (*e.g.* Polyphonic C♯, Russo's library) significant effort is made to minimize the critical section, but as we have shown (§4) coarse-grained locking remains an impediment to scalability.

By implementing joins as a library we forgo some expressivity, not just static checking. JoCaml, for example, supports restricted polymorphic sends: the type of a channel can be generalized in those type variables that do not appear in the types of other, conjoined channels [6]. Since our channels are monomorphic C♯ delegates, we are, unfortunately, unable to capture that polymorphism. Nevertheless, one can still express a wide range of useful generic abstractions (e.g. `Buffer<T>`, `Swap<A,B>`). Another difference is that our rendezvous patterns are more restrictive than JoCaml's. Our implementation only allows us to return a single value to all synchronous channels, instead of returning separately typed values to each synchronous channel. In effect, we strike a compromise between the power of JoCaml and limitations of Polyphonic C♯ (which allowed at most one synchronous channel per pattern). As a consequence, our coding of swap channels is clumsier than JoCaml's, requiring wrapper methods to extract the relevant half of the common return value. JoCaml instead supports (the equivalent of) selective return statements, allowing one to write, e.g. `return b to Left; return a to Right;` within the same chord. The static semantics of selective returns are difficult to capture in a library, so we have avoided them. Note that forcing all channels to wait on a single return statement, as we do, also sacrifices some concurrency.

6.2 STM

We are aware of two join-calculus implementations that do not employ a coarse-grained locking strategy, instead using Haskell's software transactional memory (STM [8]). Singh's implementation builds directly on the STM library, using transacted channels and atomic blocks to provide atomicity [30]; the goal is to provide a simple implementation, and no performance results are given. In unpublished work, Sulzmann and Lam suggest a *hybrid* approach, saying that "an entirely STM-based implementation suffers from poor performance" [31]. Their hybrid approach uses a non-blocking collection to store messages, and then relies on STM for the analog to our message resolution process. In addition to basic join patterns, Sulzmann and Lam allow

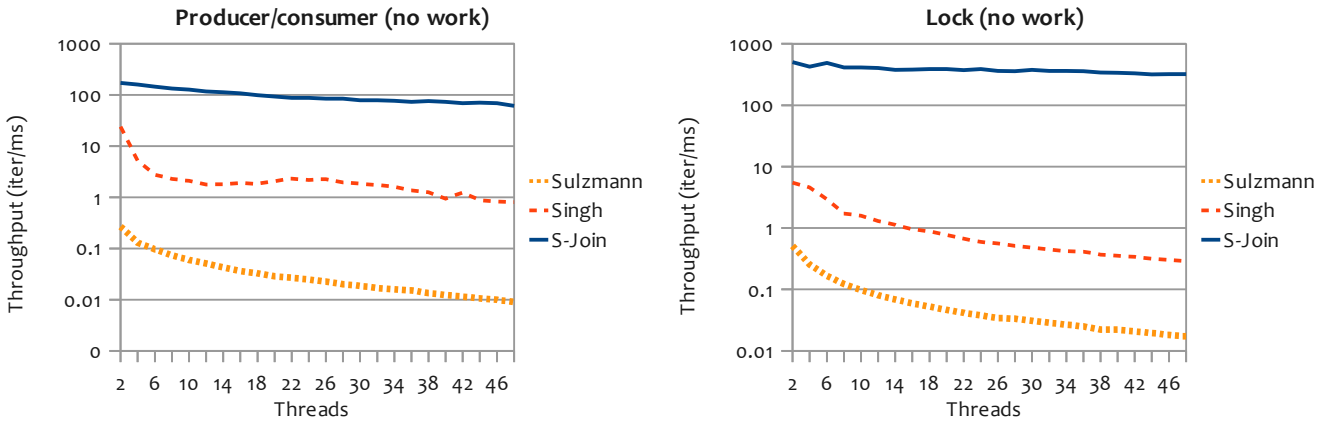


Figure 11. Comparison with Haskell-STM implementations on 48-core machine. **Note log scale**.

guards and *propagated clauses* in patterns, and to handle these features they spawn a thread *per message*; HASKELL threads are lightweight enough to make such an approach viable. The manuscript provides some performance data, but only on a four core machine, and does not provide comparisons against direct solutions to the problems they consider.

The simplest—but perhaps most important—advantage of our implementation over STM-based implementations is that we do not require STM, making our approach more portable. STM is an active area of research, and state-of-the-art implementations require significant effort.

The other advantage over STM is the specificity of our algorithm. An STM implementation must provide a general mechanism for declarative atomicity, conflict-resolution and contention management. Since we are attacking a more constrained problem, we can employ a more specialized (and likely more efficient and scalable) solution. For example, in our implementation one thread can be traversing a bag looking for `PENDING` messages, determine that none are available, and exit message resolution all while another thread is adding a new `PENDING` message to the bag. Or worse: two threads might both add messages to the same bag. It is not clear how to achieve the same degree of concurrency with STM: depending on the implementation, such transactions would probably be considered conflicting, and one aborted and retried. While such spurious retries might be avoidable by making STM aware of the semantics of bags, or by carefully designing the data structures to play well with the STM implementation, the effort involved is likely to exceed that of the relatively simple algorithm we have presented.

To test our suspicions about the STM-based join implementations, we replicated the pure synchronization benchmarks for producer/consumer and locks, on top of both Singh and Sulzmann's implementations. Figure 11 gives the results on the same 48-core benchmarking machine we used for §4. Note that, to avoid losing too much detail, we plot the throughput in these graphs using a log scale. The comparison is, of course, a loose one, as we are comparing across two very different languages and runtime systems. However, it seems clear that the STM implementations suffer from both drastically increased constant overheads, as well as much poorer scalability. Surprisingly, of the two STM implementations, Singh's much simpler implementation was the better performer.

Given these results, and the earlier results for lock-based implementations, our JOINS implementation is the only one we know to scale when used for fine-grained parallelism.

While STM can be used to implement joins, it is also possible to see STM and joins as two disparate points in the spectrum of declarative concurrency: STM allows arbitrary shared-state computation to be declared atomic, while joins only permits highly-structured atomic blocks in the form of join patterns. From this standpoint, our library is a bit like k-compare single swap [20] in attempting to provide scalable atomicity somewhere between CAS and STM. There is a clear tradeoff: by reducing expressiveness relative to a framework like STM, our joins library admits a relatively simple implementation with robust performance and scalability.

6.3 Coordination in `java.util.concurrent`

The `java.util.concurrent` library contains a class called `AbstractQueuedSynchronizer` that provides basic functionality for queue-based, blocking synchronizers [15]. Internally, it represents the state of the synchronizer as a single 32-bit integer, and requires subclasses to implement `tryAcquire` and `tryRelease` methods in terms of atomic operations on that integer. It is used as the base class for at least six synchronizers in the `java.util.concurrent` package, thereby avoiding substantial code duplication. In a sense, our JOINS library is a generalization of the abstract synchronizer framework: we support arbitrary internal state (represented by asynchronous messages), n-way rendezvous, and the exchange of messages at the time of synchronization.

Another interesting aspect of `java.util.concurrent` is its use of *dual data structures* [27], in which blocking calls to a

data structure (such as `Pop` on an empty stack) insert a "reservation" in a nonblocking manner; they can then spinwait to see whether that reservation is quickly fulfilled, and otherwise block. Reservations provide an analog to the *conditions* used in monitors, but apply to nonblocking data structures. JOINS provide another perspective on reservations: for us, `Pop` can be used as a method, but it is really a message-send on a synchronous channel. Reservations, spinwaiting and blocking thus fall out as a natural consequence. With JOINS, it is also easy to allow such synchronous messages to be involved in several different patterns, allowing the requests to be fulfilled in multiple ways, each of which can involve complex synchronization; the correct protocol for handling reservations is then automatically provided.

Our benchmarking results indicate that it may be reasonable to build some parts of a library like `java.util.concurrent` around JOINS; certainly, some of .NET's concurrency library could be profitably replaced with a JOINS-based implementation. While this is a nice result, our goal is not to replace such carefully-engineered libraries. Rather, we want to push forward the frontier, taking the insights that made `java.util.concurrent` successful and putting them in the hands of application programmers, without exposing their intricacies.

6.4 Parallel CML

Our algorithm draws some inspiration from Reppy, Russo and Xiao's PARALLEL CML, a combinator library for first-class synchronous events [23]. The difficulty in implementing CML is, in a sense, dual to that of the join calculus: disjunctions of events (rather than conjunctions of messages) must be atomically resolved. PARALLEL CML implements disjunction (choice) by adding a *single* event to the queue of each involved channel. Events have a "state" similar to our message statuses; event resolution is performed by an optimistic protocol that uses CAS to claim events.

The PARALLEL CML protocol is, however, much simpler than the protocol we have presented: events are resolved while holding a channel lock. In particular, if an event is offering a choice between sending on channel A and receiving on channel B, the resolution code will first lock channel A while looking for partners, then (if no partners are found) unlock A and lock channel B. These channel locks prevent concurrent changes to channel queues, allowing the implementation to avoid subtle questions about when it is safe to stop running the protocol—exactly the questions we address in §3.3. The tradeoff for this simplicity is reduced scalability under high contention due to reduced concurrency, as in our experimental results for lock-based joins.

7. Conclusion

We have demonstrated that it is possible to place many of the insights behind scalable synchronization algorithms directly into the hands of library users—without requiring those users to understand those insights, to use a fixed set of primitives, or to give up scalability. There is still much work to be done, both in evaluation (does declarative coordination help in real applications?) and scope (can we expand beyond the join calculus?), but we believe the algorithms presented in this paper provide a promising foundation for further exploration.

Acknowledgements We are indebted to both Sam Tobin-Hochstadt and Vincent St-Amour for careful reading and many helpful discussions, Jesse Tov, Daniel Brown, and Stephen Chang for feedback on drafts of this paper, Stephen Toub and Microsoft's PCP team for providing access to our test machine, Luc Maranget for motivating our counter optimization, and Samin Ishtiaq and Matthew Parkinson for general support.

References

[1] N. Benton. Jingle bells: Solving the Santa Claus problem in Polyphonic C♯. Unpublished manuscript, Mar. 2003. URL `http://research.microsoft.com/~nick/santa.pdf`.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5), Sept. 2004.

[3] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[4] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1996.

[5] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *APPSEM Summer School, Caminha, Portugal, Sept. 2000*, 2002.

[6] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Implicit typing à la ML for the join-calculus. In *International Conference on Concurrency Theory (CONCUR)*, 1997.

[7] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. JoCaml: a language for concurrent distributed and mobile programming. In *Advanced Functional Programming, 4th International School, Oxford, Aug. 2002*, 2003.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2005.

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

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] G. S. Itzstein and D. Kearney. Join Java: An alternative concurrency semantics for Java. Technical Report ACRC-01-001, University of South Australia, 2001.

[12] F. Le Fessant and L. Maranget. Compiling join-patterns. In *International Workshop on High-Level Concurrent Languages (HLCL)*, Sept. 1998.

[13] D. Lea. URL `http://gee.cs.oswego.edu/dl/concurrency-interest/`.

[14] D. Lea. A java fork/join framework. In *ACM conference on Java*, 2000.

[15] D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293 – 309, 2005.

[16] Y. Liu, 2009. URL `http://channel.sourceforge.net/`.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, February 1991.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.

[19] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.

[20] M. Moir, V. Luchangco, and N. Shavit. Non-blocking k-compare single swap. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2003.

[21] M. Odersky. An overview of functional nets. In *APPSEM Summer School, Caminha, Portugal, Sept. 2000*, 2002.

[22] H. Plociniczak and S. Eisenbach. JErlang: Erlang with Joins. In *Coordination Models and Languages*, 2010.

[23] J. Reppy, C. V. Russo, and Y. Xiao. Parallel concurrent ml. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2009.

[24] C. Russo. The Joins Concurrency Library. In *International Symposium on Practical Aspects of Declarative Languages (PADL)*, 2007.

[25] C. Russo. Join Patterns for Visual Basic. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2008.

[26] W. Scherer, III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[27] W. N. Scherer, III and M. L. Scott. Nonblocking concurrent objects with condition synchronization. In *International Symposium on Distributed Computing (DISC)*, 2004.

[28] W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[29] N. Shavit and D. Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995.

[30] S. Singh. Higher-order combinators for join patterns using STM. ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), June 2006.

[31] M. Sulzmann and E. S. L. Lam. Parallel join patterns with guards and propagation. Unpublished manuscript, 2008.

A. Library Reference

A new `Join` instance j is allocated by calling an overload of factory method `Join.Create`:

$$\texttt{Join}\ j = \texttt{Join.Create}(); \textit{ or}$$
$$\texttt{Join}\ j = \texttt{Join.Create}(\textit{size});$$

The optional integer *size* is used to explicitly bound the number of channels supported by `Join` instance j. An omitted *size* argument defaults to 32; *size* initializes the constant, read-only property j`.Size`.

A `Join` object notionally owns a set *channels*, each obtained by calling an overload of method `Init`, passing the location, *channel*(*s*), of a channel or array of channels using an `out` argument:

$$j\texttt{.Init}(\texttt{out}\ \textit{channel});$$
$$j\texttt{.Init}(\texttt{out}\ \textit{channels}, \textit{length});$$

The second form takes a *length* argument to initialize location *channels* with an *array* of *length* distinct channels.

Channels are instances of delegate types. In all, the library provides six channel flavors:

```
// void-returning asynchronous channels
delegate void Asynchronous.Channel();
delegate void Asynchronous.Channel<A>(A a);
// void-returning synchronous channels
delegate void Synchronous.Channel();
delegate void Synchronous.Channel<A>(A a);
// value-returning synchronous channels
delegate R Synchronous<R>.Channel();
delegate R Synchronous<R>.Channel<A>(A a);
```

The outer class of a channel `Asynchronous`, `Synchronous` or `Synchronous<`R`>` should be read as a modifier that specifies its blocking behaviour and optional return type.

When a synchronous channel is invoked, the caller must wait until the delegate returns (void or some value). When an asynchronous channel is invoked, there is no result and the caller proceeds immediately without waiting. Waiting may, but need not, involve blocking.

Apart from its channels, a `Join` object notionally owns a set of *join patterns*. Each pattern is defined by invoking an overload of the instance method `When` followed by zero or more invocations of instance method `And` followed by a final invocation of instance method `Do`. Thus a pattern definition typically takes the form:

j`.When(`c_1`).And(`c_2`)`$\cdots$`.And(`c_n`).Do(`d`)`

Each argument c to `When`(c) or `And`(c) can be a single channel or an array of channels. All synchronous channels that appear in a pattern must *agree* on their return type.

The argument d to `Do`(d) is a *continuation* delegate that defines the body of the pattern. Although it varies with the pattern, the type of the continuation is always an instance of one of the following delegate types:

$$\texttt{delegate}\ R\ \texttt{Func<}P_1, \ldots, P_m, R\texttt{>}(P_1\ p_1, \ldots, P_m\ p_m);$$
$$\texttt{delegate void Action<}P_1, \ldots, P_m\texttt{>}(P_1\ p_1, \ldots, P_m\ p_m);$$

The precise type of the continuation d, including its number of arguments, is determined by the sequence of channels guarding it. If the first channel, c_1, in the pattern is a synchronous channel with return type R, then the continuation's return type is R; otherwise the return type is `void`.

The continuation receives the arguments of channel invocations as delegate parameters $P_1\ p_1, \ldots, P_m\ p_m$, for $m \leq n$. The presence and types of any additional parameters $P_1\ p_1, \ldots, P_m\ p_m$ is dictated by the type of each channel c_i:

- If c_i is of non-generic type `Channel` or `Channel[]` then `When`(c_i)/`And`(c_i) adds no parameter to delegate d.
- If c_i is of generic type `Channel<`P`>`, for some type P then `When`(c_i)/`And`(c_i) adds one parameter p_j of type $P_j = P$ to delegate d.
- If c_i is an array of type `Channel<`P`>[]` for some type P then `When`(c_i)/`And`(c_i) adds one parameter p_j of array type $P_j = P$`[]` to delegate d.

Parameters are added to d from left to right, in increasing order of i. In the current implementation, a continuation can receive at most $m \leq 16$ parameters.

A join pattern associates a set of channels with a body d. A body can execute only once all the channels guarding it have been invoked. Invoking a channel may enable zero, one or more patterns:

- If no pattern is enabled then the channel invocation is queued up. If the channel is asynchronous, then the argument is added to an internal bag. If the channel is synchronous, then the calling thread is blocked, joining a notional bag of threads waiting on this channel.
- If there is a single enabled join pattern, then the arguments of the invocations involved in the match are consumed, any blocked thread involved in the match is awakened, and the body of the pattern is executed in that thread. Its result - some value or exception - is broadcast to all other waiting threads, awakening them. If the pattern contains no synchronous channels, then its body runs in a new thread.
- If there are several enabled patterns, then an unspecified one is chosen to run.
- Similarly, if there are multiple invocations of a particular channel pending, which invocation will be consumed when there is a match is unspecified.

The current number of channels initialized on j is available as read-only property j`.Count`; its value is bounded by j`.Size`. Any invocation of j`.Init` that would cause j`.Count` to exceed j`.Size` throws `JoinException`.

Join patterns must be well-formed, both individually and collectively. Executing `Do`(d) to complete a join pattern will throw `JoinException` if d is `null`, the pattern repeats a channel (and the implementation requires linear patterns), a channel is `null` or *foreign* to this pattern's `Join` instance, or the join pattern is *empty*. A channel is foreign to a `Join` instance j if it was not allocated by some call to j`.Init`. A pattern is empty when its set of channels is empty (this can only arise through array arguments).

Though not discussed in the body of this paper, array patterns are useful for defining dynamically sized joins, e.g. an n-way exchanger:

```
class NWayExchanger<T> {
  public Synchronous<T[]>.Channel<T>[] Values;
  public NWayExchanger(int n) {
    var j = Join.Create(n); j.Init(out Values, n);
    j.When(Values).Do(vs => vs);
  }
}
```