Scalable Join Patterns

Claudio V. Russo
Microsoft Research
crusso@microsoft.com

Aaron Turon
Northeastern University
turon@ccs.neu.edu

Abstract

Coordination can destroy scalability in parallel programming, so a comprehensive library of scalable synchronization primitives is an essential tool. Unfortunately, such primitives do not easily combine to yield scalable solutions to complex coordination requirements. We demonstrate that a C♯ 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 for a coordination problem, and the library will derive a correct solution. By *scalable*, we mean that the derived solutions deliver robust performance both as the number of processors increase, and as the complexity of the coordination problem grows. We validate our scalability claims empirically on six coordination problems, comparing our generic solution to specialized algorithms.

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.

The proliferation of specialized synchronization primitives is therefore no surprise. For example, `java.util.concurrent` [11] 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 [12, 13, 24, 26]. A JAVA programmer faced with a coordination problem covered by the library is in great shape.

Inevitably, though, programmers are faced with new coordination problems not directly addressed by the primitives. The primitives must be composed into a solution. Doing so in an efficient and correct way 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; this kind of competition over limited resources appears in many guises in real systems. Dining philosophers 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*. Most commonly, naive solutions suffer from deadlock, when each philosopher picks up the chopstick to their left, and then finds the one to their right taken. Even a correct solution may scale poorly with the number of philosophers (threads). For example, if philosophers sitting far apart must interact by acquiring a global lock, time spent on unnecessary communication will, in the limit, overwhelm time spent on useful work. Avoiding these pitfalls takes experience and care.

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

Our library is a drop-in replacement for Russo's C♯ JOINS library [22], and inherits its declarative nature. We can use it to solve dining philosophers by merely stating the problem:[1]

```
var j = Join.Create();
Synchronous.Channel[]  hungry;     j.Init(out hungry, n);
Asynchronous.Channel[] chopstick;  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
  });
}
```

The join calculus is a message-passing process algebra. Here we are using two arrays of channels (`hungry` and `chopstick`) to carry valueless messages; being empty, these messages represent unadorned events. The declarative aspect of this example is the *join pattern* starting with `j.When`. Intuitively, 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.

Most implementations of the join calculus, 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 we recite provide basic implementations of many of the `java.util.concurrent` classes

[1] Note that our implementation only provides probabilistic fairness, so our solution provides mutual exclusion but not starvation freedom.

mentioned above.[2] In each case, the JOINS-based solution is no harder to write than the one for dining philosophers.

- We introduce our new algorithm through lightly-simplified excerpts from the C♯ library code (§3). The key challenge is scalably providing the atomicity required for join patterns. We meet this challenge by (1) using lock-free [8] data structures to store messages and (2) treating messages as resources that threads can race to take possession of. In this way we lift word-level atomicity (provided by compare-and-swap) to pattern-level atomicity. The algorithm we give ensures dead/livelock-freedom, avoids lost wakeups, and attempts to maximize concurrency while processing messages.
- We validate our scalability claims experimentally on six 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, and compare these results to the performance of direct, custom solutions. In all cases, our new library scales significantly better than Russo's. In addition, we scale competitively with 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 the correctness of our algorithm (§5).

Note to reviewers: The complete code and benchmarks will be made available on the web after the blind reviewing phase.

2. The JOINS library

Russo's `Joins` [22] 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. Designed for message-passing concurrency, the library provides typed channels together with the ability to write custom synchronization patterns over those channels. 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]. Some examples require extensions to the library, which we explain at the end of the section.

As in π-calculus, channels are first-class values. Each channel is associated with an instance of the `Join` class. Channels may be synchronous or asynchronous and are represented as delegates (C♯'s first-class functions); messages are sent by simply invoking the channel as a function. What makes JOINS interesting is that there is no direct way to *receive* a message on a channel. Instead, *join patterns* (also called *chords* [2]) declaratively specify reactions to message arrivals. The power of join patterns lies in their ability to respond, atomically, to message arrival on any multiset of channels. We begin with a simple example, the unordered buffer:

```
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); j.Init(out Get);   // bind its channels
    j.When(Get).And(Put).Do( t => {return t;} ); // register chord
  }}
```

Here, `Put` is an asynchronous channel that carries messages of type `T`. `Get` is a synchronous channel that returns `T` replies but takes no argument. From a client's point of view, `Put` and `Get` look just like methods. Given an instance `buf` of class `Buffer<T>`, a producer thread can post a value `t` to `buf` by calling `buf.Put(t)`; a consumer thread can request a value by calling `buf.Get()`. Because `Get` is a synchronous channel, the call will block until or unless a pattern involving it is enabled. The single join pattern stipulates that when one `Get` and one `Put` message are available, they should both be consumed. The body of the pattern, given as a delegate argument to the `Do` method, says that the single argument `t` (from the `Put` channel) should be returned as the result to `Get`.

In all, the library provides six channel flavors:

```
// void-returning asynchronous channels
delegate void Asnchronous.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);
```

In general, every channel takes at most one argument and, when synchronous, can return at most one result. If required, multiple values can be passed or returned by tupling. If the channel is `Synchronous` or `Synchronous<`R`>`, the sender must wait until a pattern is fired, yielding an R-result like an ordinary method call. If the channel is `Asynchronous`, the send completes at once and returns control, potentially queueing the message.

Given a join object j and some channels c_i, we construct patterns using chained method calls:

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

The continuation d is a delegate, provided as an implicitly typed λ-statement `(`$\overline{a}$`) =>{`$stmt$`}` or appropriately-typed method `o.M`. Unary channels, (type `Channel<`A`>`) contribute one parameter of type A to the continuation delegate d. Nullary channels (type `Channel`) do not extend the parameters of d but do affect the pattern's synchronization behavior. If any channel c_i is synchronous, then the delegate d and all other synchronous channels in the pattern must have the same return type (if any). This agreement on return type ensures that any synchronous waiter can execute the continuation and share its value with all other synchronous waiters. If all channels are asynchronous, then d must return `void`. All type constraints are checked statically.

Patterns can only be registered prior to sending any messages on the `Join` instance's channels.

When a message enables a pattern, the pattern's body can run, atomically consuming the relevant messages. If there is no enabled pattern, the message is queued and, for synchronous channels, the sender is blocked. If there are several enabled patterns, an unspecified pattern is selected to run. An enabled pattern involving only asynchronous channels spawns a new thread on which to execute the continuation. An enabled pattern involving a least one synchronous channel selects a synchronous waiter to execute the continuation. The continuation's return value (if any) is then broadcast to the remaining synchronous waiters involved in the pattern.

In the remainder of this section, we illustrate the range of JOINS by encoding several coordination problems using join patterns. The simplest is the humble (non-recursive) `Lock` class:

```
class Lock {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Lock() {
    var j = Join.Create(); j.Init(out Acquire); j.Init(out Release);
    j.When(Acquire).And(Release).Do(() => { });
    Release();  // initially free
  }}
```

[2] Our versions lack some of the features of the real library, such as timeouts and cancellation, but they could easily be added to the JOINS library (§3.4).

As in the dining philosophers example, we use void-argument, void-returning channels as *signals*. The `Release` messages are tokens that indicate whether 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.

With a slight generalization, we obtain *semaphores*:

```
class Semaphore {
  public readonly Synchronous.Channel Acquire;
  public readonly Asynchronous.Channel Release;
  public Semaphore(int n) {
    var j = Join.Create(); j.Init(out Acquire); j.Init(out Release);
    j.When(Acquire).And(Release).Do(() => { });
    for (; n > 0; n--) Release(); // initially free for n clients
  }}
```

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 lock with n initial `Release()` tokens.

With little effort, we can also generalize `Buffer` to a synchronous swap channel that exchanges data between threads:

```
class Exchanger<A, B> {
  readonly Synchronous<Pair<A, B>>.Channel<A> sendL;
  readonly Synchronous<Pair<A, B>>.Channel<B> sendR;
  public B SendL(A a) { return sendL(a).Snd; } // project right
  public A SendR(B b) { return sendR(b).Fst; } // project left
  public Exchanger() {
    var j = Join.Create(); j.Init(out sendL); j.Init(out sendR);
    j.When(sendL).And(sendR).Do((a,b) => Pair.Create(a,b));
  }}
```

Another generalization of binary rendezvous leads to a *barrier* that causes n threads to wait until all have arrived:

```
class SymmetricBarrier {
  public readonly Synchronous.Channel SignalAndWait;
  public SymmetricBarrier(int n) {
    var j = Join.Create(); j.Init(out SignalAndWait);

    var pat = j.When(SignalAndWait);
    for (int i = 1; i < n; i++) pat = pat.And(SignalAndWait);
    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 be 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 [24]), it is easy to do so. We can construct a barrier that requires n threads of one kind and m threads of another, simply by using two channels rather than one.

To support examples like `Exchanger` and `SymmetricBarrier` we allow patterns to mention multiple synchronous channels, and to mention a channel more than once; neither were allowed in Russo's original JOINS library.

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 [23], 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. In this section, 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 the appearance of 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).[3] 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 keeping the critical section short: he uses bitmasks summarizing message availability to accelerate pattern matching; represents `void` asynchronous channels as counters; and permits "message stealing" to hide the latency of context switches—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 elsewhere [17], but 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 rendezvous and dining philosophers, a much greater degree of concurrency is both possible and desirable. At a lower level, coarse-grained locking may require more explicit interprocessor synchronization (*e.g.*, calls to CAS) than is strictly necessary. When implementing low-level coordination with joins, such overheads are measurably significant (§4).

In summary, we want an implementation of JOINS that matches and fires chords concurrently while minimizing costly interprocessor communication. Our basic strategy is to use nonblocking collections to store messages, and to equip messages with a `Status` field of the following type:

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

Message statuses provide the basis for atomically matching and firing chords:

- 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 to change them one by one 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.

Thus, the status field acts as a kind of lock, but one tied to individual messages rather than an entire instance of `Join`. Further, when we fail to "acquire" a message using the `TryClaim` method, we have recourse beyond spinning or blocking, as we will see in §3.2. We use compare-and-swap to ensure correct mutual exclusion: CAS *atomically* updates a memory location when its current value matches some expected value. `WOKEN` is discussed later (§3.3).

Below, we describe our implementation *via* a slightly simplified variant of its C$^\sharp$ code, using "`partial`" to break up class definitions.

[3] Some implementations use STM [27], which we discuss in §6.

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

```
// Chan<A> implements:
  Chord[] Chords              { get; };
  bool IsSync                 { get; };
  ThreadLocal<int> SpinCount { get; };
  Msg AddPendingMsg(A a);
  Msg FindPendingMsg(out bool sawClaimed);
```

Figure 1. The interfaces to our key data structures

3.1 Representation

We begin with the representation of messages and channels (Figure 1). 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 `WOKEN` status and the remaining `Msg` fields (`Signal`, `AsyncWaker` and `Result`) are used for blocking on synchronous channels, which is discussed in §3.3.

We need to permit highly-concurrent access to the collection of messages available on a given channel. Thus, the `Chan<A>` class implements a nonblocking—in particular, lock-free [8]—bag of messages of type `A`.[4] 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. Others do the same [2, 4, 22]. This choice is not fundamental, and ordered channels are easily provided (§3.4). Our examples do not require ordering.

The key operations on a channel are:

- `AddPendingMsg`, which takes a message payload and atomically adds a `Msg` with `PENDING` status to the bag.
- `FindPendingMsg`, 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 `sawClaimed` 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.

3.2 The core algorithm: resolving a message

A basic goal of our algorithm is to ensure the liveness property under a fair scheduler:

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

[4] We do not detail our bag implementation here, but it is a straightforward adaptation of a lock-free queue [16]. A more aggressive implementation could take advantage of the semantics of bags to permit greater concurrency.

[5] A *linearization point* [9] 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 provide a fairness guarantee; see §5.

```
 1 partial class Join {
 2   // resolves myMsg, which was already added to its channel's bag
 3   // returns null: if no chord is firable or myMsg is consumed
 4   //          o/w : a firable chord and sufficient claimed messages
 5   Pair<Chord, Msg[]> Resolve(Msg myMsg) {
 6     var backoff = new Backoff();
 7     while (true) {
 8       bool retry = false, sawClaimed;
 9       foreach (var chord in myMsg.chan.Chords) {
10         Msg[] claims = chord.TryClaim(myMsg, out sawClaimed);
11         if (claims != null) return Pair.Create(chord, claims);
12         retry |= sawClaimed;
13       }
14
15       if (!retry || myMsg.Status == Stat.CONSUMED
16                  || myMsg.Status == Stat.WOKEN) return null;
17
18       backoff.Once();
19     }
20   }
21 }
```

Figure 2. Resolving a message

Our strategy is to drive the firing of chords by the concurrent arrival of each message: each sender must resolve its 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;
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, concurrently in flux. In particular, just as one thread determines that its message `m` does not enable any chord, another message in another thread may arrive that enables a chord involving `m`. The key is that each sender need only take responsibility for the messages that came before it; if a later sender enables a chord, that later sender is responsible for it. The linearizability of the underlying message bags ensures the existence of a coherent, global ordering of message arrivals.

The `Resolve` method takes a message `myMsg` that has already been added to the appropriate channel's bag and loops until the message has been resolved (lines 7–19). We first attempt to "claim" a chord involving `myMsg`, successively trying each chord in which `myMsg`'s channel is involved (lines 9–13). The `Chord` class's `TryClaim` method either returns an array of messages (which includes `myMsg`) that have all been `CLAIMED` by the current thread, or `null` if claiming failed. In the latter case, the `sawClaimed` out-parameter reflects whether any messages in any of the involved message bags was `CLAIMED` by another thread. Likewise, the `retry` flag records whether any externally-`CLAIMED` messages were seen in *any* chord (line 12). We need this flag because any externally-`CLAIMED` messages may later 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 11. The second two ways correspond to the `return` on line 16 (we discuss `WOKEN` in §3.3). Notice that the third form of resolution—determining that no chord can be fired—only occurs when `retry` is `false`. If none of the three conditions hold, we must try again. We perform exponential backoff (line 18) in this case,

```
22 partial class Chord {
23   Chan[] Chans;  // the channels making up this chord
24
25   // try to claim enough messages -- including myMsg -- to fire
26   Msg[] TryClaim(Msg myMsg, out bool sawClaimed) {
27     var msgs = new Msg[Chans.length];  // cached in the real code
28     sawClaimed = false;
29
30     // first locate enough pending messages to fire the chord
31     for (int i = 0; i < Chans.Length; i++) {
32       if (Chans[i] == myMsg.Chan) {
33         msgs[i] = myMsg;
34       } else {
35         msgs[i] = Chans[i].FindPendingMsg(out sawClaimed);
36         if (msgs[i] == null) return null;
37       }
38     }
39
40     // now try to claim the messages we found
41     for (int i = 0; i < Chans.Length; i++) {
42       if (!msgs[i].TryClaim()) {
43         // another thread got a message before we did; revert
44         for (int j = 0; j < i; j++) msgs[j].Status = Stat.PENDING;
45         sawClaimed = true; // there may be CLAIMED messages
46         return null;
47       }
48     }
49
50     return msgs; // success: each msgs[i].Status == Stat.CLAIMED
51   }
52 }
```

Figure 3. Racing to claim a chord involving a given message

```
53 partial class Join {
54   void AsyncSend<A>(Chan<A> chan, A a) {
55     HandleAsyncMsg(chan.AddPendingMsg(a));
56   }
57   // resolve myMsg (which is asynchronous) and respond accordingly
58   void HandleAsyncMsg(Msg myMsg){
59     Pair<Chord, Msg[]> res = Resolve(myMsg);
60     if (res == null) return;    // no chord to fire; nothing to do
61
62     var chord = res.Fst;
63     var claims = res.Snd;
64
65     if (chord.IsAsync) {
66       // completely asynchronous chord: fire in a new thread
67       ConsumeAll(claims);
68       new Thread(_ => chord.Fire(claims)).Start();
69     } else {
70       // synchronous chord: wake a synchronous waiter
71       bool foundSleeper = false;
72       for (int i = 0; i < chord.Chans.Length; i++) {
73         if (chord.Chans[i].IsSync && !foundSleeper) {
74           foundSleeper = true;
75           claims[i].AsyncWaker = myMsg; // transfer responsibility
76           claims[i].Status = Stat.WOKEN;
77           claims[i].Signal.Set();
78         } else {
79           claims[i].Status = Stat.PENDING; // relinquish message
80         }
81       }
82     }
83 }}
```

Figure 4. Sending an asynchronous message

because repeated retrying can only be caused by high contention over messages.

The code for `TryClaim` is fairly straightforward. We present 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 §3.4.

`TryClaim` works in two phases. In the first phase (lines 30–38), we first try to locate sufficiently many `PENDING` messages to fire the chord. We are required to claim `myMsg` in particular. If we are unable to find enough messages, we exit (line 36) without having written to memory. Otherwise, we enter the second phase (lines 40–48), wherein we race to claim each message. The message-level `TryClaim` method performs a CAS on the `Status` field, 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 (line 44). 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).

3.3 Firing, stealing and rendezvous

Resolving a message is only half the battle: we also need to fire chords when appropriate. Once more, the process is driven by the arrival of messages (`AsyncSend` and `SyncSend`).

First we follow the path of an asynchronous message (line 54). We first resolve the message (line 59). 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 60). In both of these cases, we are assured that any chords involving our message will be 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. The easy case is when the chord's pattern involved only asynchronous channels (line 65). Then, we simply consume all involved messages (which does not require a memory fence), and spawn a new thread to execute the chord body asynchronously.

The remaining code in the listing deals, in one way or another, with synchronous channels. A key difference from asynchronous channels is that synchronous senders should *block* until a chord is firable. Moreover, the synchronous chord bodies should be executed by a synchronous caller, rather than by a newly-spawned thread. Finally, since multiple synchronous callers can be combined in a single chord, one of them should be chosen to execute the chord, and then share the result with (and wake up) all the others. In the end, we must deal with these situations on a case-by-case basis, as follows:

Send	We `CLAIMED`	They `CONSUMED`	No match	`WOKEN`
A	(A) Spawn (67–68) (S) Wake (71–81)	Return (60)	Return (60)	–
S	Fire (112–123)	Get result (102)	Block (99)	Retry (92–107)

We have already covered most of the first row; the remaining case is when an asynchronous message is resolved by claiming messages for a chord that involves synchronous channels (lines 71-81). One option at this juncture would be to consume all of the relevant messages, and inform a synchronous caller that it should fire a chord. If, however, the synchronous caller has already blocked—so its thread is no longer active—significant time may lapse before the chord is actually fired.

Since we do not provide a fairness guarantee, we can instead permit "stealing": we wake up one synchronous caller, but return the rest of the messages to `PENDING` status, putting them back up for grabs by currently-active threads—including the thread that just

```
 84 partial class Join {
 85   // create a message with payload a, and block until chord fired
 86   R SyncSend<R, A>(Chan<A> chan, A a) {
 87     var backoff = new Backoff();
 88     Msg waker = null;
 89     Pair<Chord, Msg[]> res;
 90
 91     Msg myMsg = chan.AddPendingMsg(a);
 92     while (true) {
 93       res = Resolve(myMsg);
 94       if (res != null) break;   // claimed a chord; break to fire
 95
 96       if (waker != null)        // resend our last async waker
 97         RetryAsync(waker);
 98
 99       myMsg.Signal.Block(chan.SpinCount);
100
101       if (myMsg.Status == Stat.CONSUMED)
102         return (R)myMsg.Result; // rendezvous: chord already fired
103
104       waker = myMsg.AsyncWaker; // record waker to retry later
105       myMsg.Status = Stat.PENDING;
106       backoff.Once();
107     }
108
109     var chord = res.Fst;
110     var claims = res.Snd;
111
112     ConsumeAll(claims);
113     if (waker != null)          // resend our last async waker,
114       RetryAsync(waker);        // *after* consuming our message
115
116     var r = chord.Fire(claims); // ignore exceptions for simplicity
117     for (int i = 0; i < chord.Chans.Length; i++) {
118       if (chord.Chans[i].IsSync) {      // synchronous rendezvous
119         claims[i].Result = r;           // transfer computed result
120         claims[i].Signal.Set();
121       }
122     }
123     return (R)r;
124   }
125   void RetryAsync(Msg myMsg) {
126     if (myMsg.Status != Stat.CONSUMED) HandleAsyncMsg(myMsg);
127   }
128 }
```

Figure 5. Sending a synchronous message

sent the asynchronous message. In low-traffic cases, the 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 Polyphonic C$^\sharp$ [2], as well as the "barging" allowed by the `java.util.concurrent` synchronizer framework [13]; we find it works well in practice (eg. on dining philosophers for small n).

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 (all the messages in a chord are `CLAIMED`) to an unresolved state (`PENDING`). There is no guarantee that the sender will be able to fire a chord involving the original asynchronous message (see [2] for an example). Thus, when a synchronous sender is woken in this way, we also record the asynchronous message that woke it (line 75), transferring responsibility for the message.

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 to `Block` will immediately return, again resetting the signal. We ensure that `Block` and `Set` are each called by at most one thread.

`Signals` are not quite sufficient, however, because a synchronous message can be resolved by another thread before the synchronous sender has blocked. In this case, we rely on the status of the message for communication. This is why we change the synchronous message status from `CLAIMED` to `WOKEN` on line 76 before setting the signal on line 77. Note that a `WOKEN` status causes the synchronous sender to return from `Resolve` (line 16); we consider this to be a fourth way that messages can be resolved.

In general, 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 (lines 119–120).

Finally, we have the code for sending a synchronous message (lines 86–124). The primary difference from asynchronous sending is that we loop (lines 92–107) until the message is resolved by our own thread claiming it (exit on line 94), or another thread consuming it (exit on line 102). In each iteration of the loop, we block (line 99); even if our message has already been `CONSUMED`, we must wait for the signal to get the return value. In the case that our message is woken by an asynchronous message (lines 104–106), we record the waking message and try once more to resolve our own message (line 93). We perform exponential backoff every time this happens, since continually being awoken only to find messages stolen indicates high traffic.

To ensure liveness, we retry sending the asynchronous message that awoke us (lines 96–97, and 113–114). It is important that retrying does not take place when we have claimed our own synchronous message: that 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`, as well as `HandleAsyncMsg`, will simply exit in this case.

3.4 Pragmatics and extensions

As well as stronger typing to avoid boxing, the actual implementation contains two very important optimizations not shown here.

First, it does not allocate a fresh message array every time `TryClaim` is called; in fact, it does not heap-allocate message arrays at all. Instead, we stack-allocate a custom array in `SyncSend` and `AsyncSend`, and reuse this array for every call to `TryClaim`.

Second, the implementation does not *always* allocate a message when performing a synchronous send: there is a "fast path" in which we attempt to claim enough existing messages that, together with the message we are trying to send, suffice to fire a chord. If no chord can be claimed, the slow path (Figure 5) is taken. The stealing protocol for asynchronous messages prevents us from employing a similar optimization in `AsyncSend`. Interestingly, this fast path/slow path distinction closely mirrors Scherer and Scott's notion of *dual data structures* [25]; we say more about this in §6.

The presented code does not handle nonlinear patterns, in which a single channel appears multiple times. Extending it to support this feature is straightforward.

An important pragmatic point is that our `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 ar-

[7] It spins a bit first; see §3.4

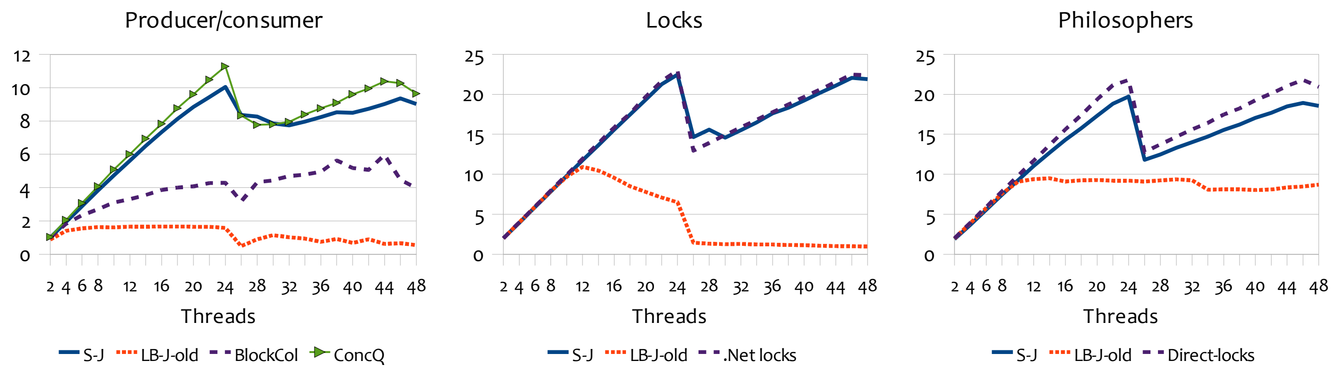

Figure 6. Benchmark results: with work. Vertical axis measures speedup; **larger values are better**.

chitectures [15]. The amount of spinning performed is determined adaptively on a per-thread, per-channel basis.

We expect it to be easy to add timeouts and asynchronous attempts for synchronous sends to our implementation, along the lines of `java.util.concurrent`. It is also straightforward to add channels with ordering constraints by using a lock-free queue or stack rather than a bag.

4. Empirical results

In this section we provide empirical support for the claim that our implementation is scalable. There are two axes along which we want to scale: the number of processors, and the size of the coordination problem. Depending on the particulars of the problem, we either look for performance to remain robust as more processors are involved (*e.g.*, for locks) or to improve (*e.g.*, for rendezvous).

4.1 Methodology

To evaluate our implementation, we constructed a series of microbenchmarks solving six classic coordination problems:

> Producer/consumer, mutual exclusion, semaphores, barriers, rendezvous, and dining philosophers.

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).

The benchmarks are meant to evaluate the cost of synchronization in the limiting case where the actual work being done is negligible. Each benchmark follows standard practice for evaluating synchronization primitives: we repeatedly synchronize, for a total of k synchronizations between n threads [15]. We use $k = 100000$ and average over three trials for all benchmarks. To test performance under preemption, we let n range up to twice the number of cores in our benchmarking machine.

In addition, for three of the benchmarks (producer/consumer, locks and philosophers) we also provide results under simulated workloads, to gauge scalability under more "realistic" usage.

For each problem we compare performance between (1) a join-based solution using our implementation ("S-Joins", for "scalable joins"), (2) a join-based solution using Russo's library ("LB-Joins-old", for "lock-based joins"), and at least one purpose-built solution from the literature or .NET libraries. We detail the purpose-built solutions below.

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 ("LB-Joins-mod").

Our benchmarking machine is a Dell PowerEdge R900 with four Intel Xeon E7450 2.4GHz processors, each containing 6 cores (for a total of 24 cores) and 32GB RAM, running Windows Server 2008 Enterprise 64-bit. We present results from the 32bit CLR, which were similar to the 64bit version.

4.2 The benchmarks

We describe each benchmark below. The results for all six benchmarks without simulated work appear in Figure 7; here, the vertical axis (time in milliseconds) is log scale and smaller values are better. The results for the three benchmarks with simulated workloads appear in Figure 6; here, the vertical axis measures speedup, and hence larger values are better. Speedup is relative to the best time for the single thread case in locks and philosophers, and the best single-producer, single-consumer time for producer/consumer.

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. In addition to the Joins-based implementations (given by the `Buffer` class), we show results for the .NET 4 `BlockingCollection` 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 [16]; similar results were obtained when we used `ConcurrentStack` and `ConcurrentBag`. For comparison, we also give times when using the `ConcurrentQueue` directly; in that test, we busywait when trying to dequeue from an empty queue.

We also provide results from a variant of the benchmark that simulates work by having producers iterate for an average of 5,000 cycles before producing each value and consumers iterate for an average 500 cycles after consuming each value.

Locks We let the n threads perform a total of k acquisitions and releases as quickly as possible. This means that the locks are maximally-contended, and that contention increases directly with the number of threads. We compare against both the built-in .NET locks and `System.Threading.SpinLock`. In the work-simulating variant of the benchmark, we spin for 100 cycles in the critical section, and 20,000 cycles outside it, in each of the k iterations.

Semaphores We let the initial semaphore count be $n/2$, and let the n threads perform a total of k acquisitions and releases as quickly as possible. We compare to two .NET semaphores: the

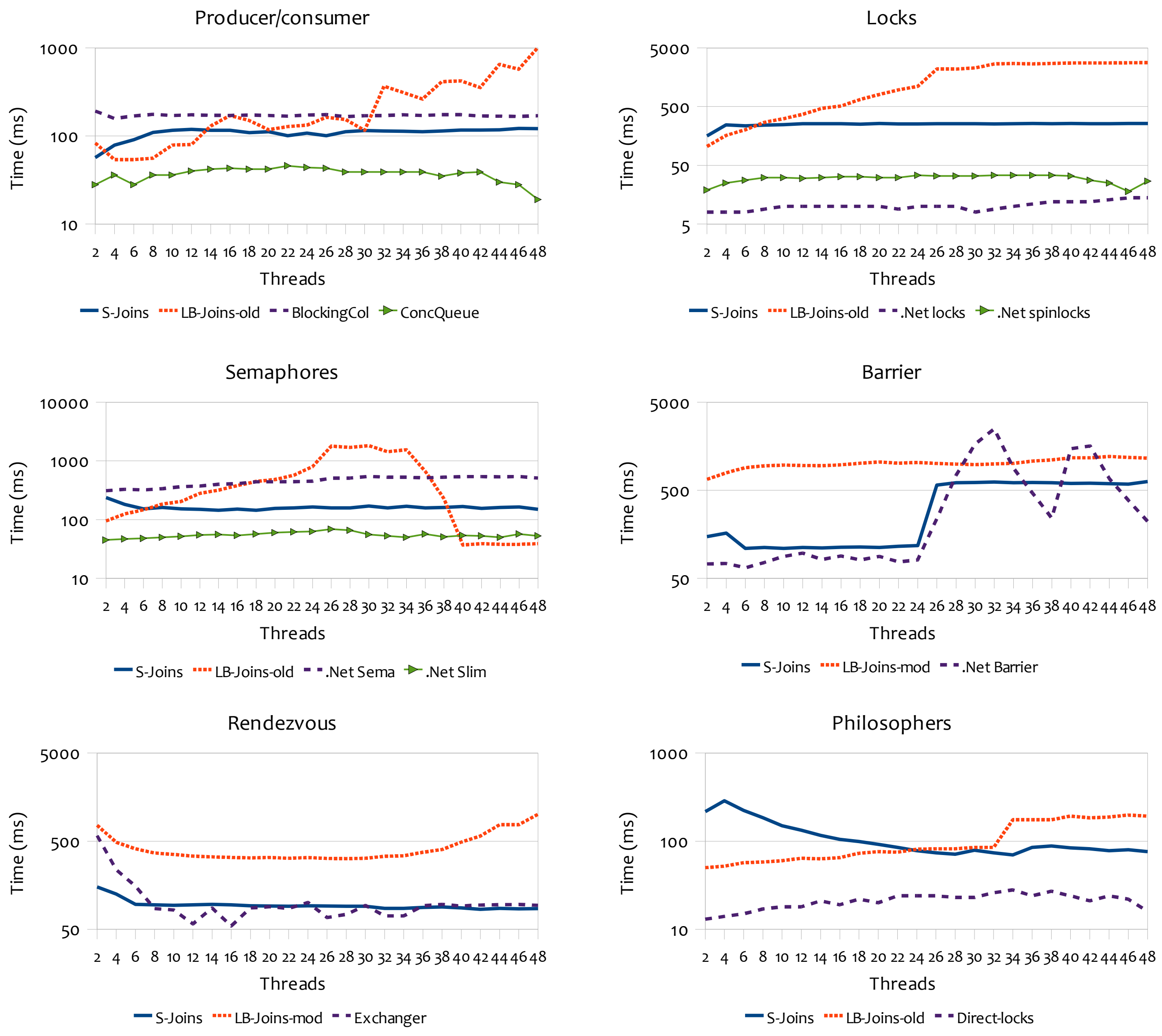

Figure 7. Benchmark results: without work. Vertical axis measures time (ms) in log scale; **smaller values are better**.

`Semaphore` class, which uses kernel semaphores, and `SemaphoreSlim`, a faster, a user-mode implementation of semaphores.

Barriers The n threads undergo a series of k/n barrier episodes as quickly as possible; thus, the size of the coordination problem grows with the number of threads, but the number of coordination episodes decreases proportionally. Since we need multiple synchronous channels in join patterns, we test against our modified version of Russo's library ("LB-Joins-mod"). We also compare against the .NET 4 `Barrier` class, a standard sense-reversing barrier.

Rendezvous The n threads perform a total of k synchronous exchanges as quickly as possible. Since we need multiple synchronous channels in join patterns, we test against our modified version of Russo's library ("LB-Joins-mod"). Unfortunately, .NET 4 does not provide a built-in library for rendezvous, so we ported Scherer *et al.*'s *exchanger* [24] from JAVA; this is the exchanger included in `java.util.concurrent`.

Philosophers For dining philosophers, we use the Joins-based implementation given at the beginning of the paper. 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. We give results both for philosophers who eat and think as quickly as possible (the "without work" case) and for philosophers who take 50 loop iterations for eating and 10,000 loop iterations for thinking (the "with work" case). We let n, the number of threads, also determine the number of philosophers (and therefore the size of the coordination problem), and require the n philosophers to eat and think a total of k times.

4.3 Analysis

On the whole, the benchmark results indicate that our JOINS library provides competitive scalability across a wide range of problems, both in the limiting synchronization-only case, as well as with simulated workloads. In addition, the library appears to interact well with the thread scheduler, as it does not suffer from the large timing fluctuations seen by the lock-based implementation, .NET barriers or the JAVA exchanger.

In one case—the synchronization-only version of producer-consumer—we initially scale poorly (up to about 8 threads). This is likely due to our underlying bag implementation, which is quite conservative: it does not attempt to parallelize insertions, nor does it make any attempt to vary the traversal order by thread, which would lead to fewer failed message claims.

Our library clearly takes a hit from higher constant-time overheads, as most dramatically seen in the lock benchmark. This is not too surprising: .NET locks are mature and highly engineered, whereas our library code is relatively unoptimized. In addition, the library has *interpretive* overhead, as it crawls through runtime data structures representing the relevant join patterns; a *compiled* version of join patterns is likely to come within striking distance of the absolute performance of .NET's locks. Note also that, 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.

Finally, in the work-simulating benchmarks we see that our higher overhead is not an impediment to achieving parallel speedup; our performance is very close to the bespoke implementations.

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 far too much to present here, and indeed 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 [9] and lock-free [8]; 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 the appropriate channel's bag. 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 four 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; it can be marked `WOKEN` by another thread; and it can be in an arbitrary status when it is determined that there are not enough messages *prior* to it to fire a chord.

Safety for the first three 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, its status is only changed by that thread.
- Once a message is `CONSUMED`, its status is never changed again.
- Once a message is `WOKEN`, its status is only changed by the thread that enqueued it.

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` on line 15 if, indeed, no pattern is matched using only the message and messages that arrived before it. Here we use the correctness assumptions about bags, together with the facts about the status flags just given. A particularly subtle point is `WOKEN` messages: when a message is marked `WOKEN`, we consider it to be unavailable for use in pattern matching; when it is subsequently remarked as `PENDING`, we consider that as a *fresh* insertion into the bag.

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 41–48), *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 [6] and Odersky's FUNNEL [19] (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. FUNNEL targeted the JAVA VM, which can exploit parallelism, but we could find no evaluation of its performance on parallel hardware. Cardelli, Benton and Fournet proposed an object-oriented version of join patterns for C$^\sharp$ called POLYPHONIC C$^\sharp$ [2]; around the same time, von Itzstein and Kearney independently described JOINJAVA [10], a similar extension of JAVA. The advent of generics in C$^\sharp$ 2.0 led Russo to encapsulate join pattern constructs in the JOINS library [22], which served as the basis for our library. There are also implementations for ERLANG [20], C++ [14], and VB [23].

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$^\sharp$, 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.

We are aware of two implementations that do not employ a coarse-grained locking strategy, instead using HASKELL's software transactional memory (STM [7]). Singh's implementation builds directly on the STM library, using transacted channels and atomic blocks to provide atomicity [28]; 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" [29]. Their hybrid approach uses a nonblocking 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 *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 promising 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 probably most important—advantage of our implementation over STM-based implementations is that we do not require STM, making our approach more portable. STM is still an active area of research, and high-performance implementations require significant effort. It also seems likely that even the best STM-based implementations will suffer from spurious retries when message bags change in harmless ways, whereas we can draw on specialized knowledge about our problem domain.

6.2 Coordination and declarative atomicity

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 [18] 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.

The `java.util.concurrent` library contains a class called `AbstractQueuedSynchronizer` that provides basic functionality for queue-based, blocking synchronizers [13]. Internally, it represents the state of the synchronizer as a single 32bit 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* [25], 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. We were surprised to realize that dual data structures are a natural consequence of the way we use synchronous channels. For example, our `Buffer` class essentially amounts to a dual bag.

Our algorithm draws inspiration from Reppy *et al.*’s PARALLEL CML, a combinator library for first-class synchronous events [21]. We too use transactional techniques, but go further in eliminating all channel locks, using lock-free bags for greater scalability.

References

[1] N. Benton. Jingle bells: Solving the Santa Claus problem in Polyphonic C$^\sharp$. 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$^\sharp$. *TOPLAS*, 26, Sept. 2004.

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

[4] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *POPL*, pages 372–385, 1996.

[5] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *APPSEM Summer School, Caminha, Portugal, Sept. 2000*, volume 2395 of *LNCS*. Springer-Verlag, 2002.

[6] 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*, volume 2638 of *LNCS*. Springer-Verlag, 2003.

[7] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60. ACM, 2005.

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

[9] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12:463–492, July 1990.

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

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

[12] D. Lea. A java fork/join framework. In *JAVA*, pages 36–43, 2000.

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

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

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[16] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275. ACM, 1996.

[17] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51:1–26, May 1998.

[18] M. Moir, V. Luchangco, and N. Shavit. Non-blocking k-compare single swap. In *SPAA*, pages 314–323, 2003.

[19] M. Odersky. An overview of functional nets. In *APPSEM Summer School, Caminha, Portugal, Sept. 2000*, volume 2395 of *LNCS*. Springer-Verlag, 2002.

[20] H. Plociniczak and S. Eisenbach. JErlang: Erlang with Joins. In *Coordination Models and Languages*, volume 6116 of *Lecture Notes in Computer Science*, pages 61–75. Springer Berlin, 2010.

[21] J. Reppy, C. V. Russo, and Y. Xiao. Parallel concurrent ml. In *ICFP*, pages 257–268. ACM, 2009.

[22] C. Russo. The Joins Concurrency Library. In *PADL*, pages 260–274. Springer-Verlag, 2007.

[23] C. Russo. Join Patterns for Visual Basic. In *OOPSLA*, 2008.

[24] W. Scherer, III, D. Lea, and M. L. Scott. A scalable elimination-based exchange channel. In *SCOOL*, 2005.

[25] W. N. Scherer, III and M. L. Scott. Nonblocking concurrent objects with condition synchronization. In *DISC*, 2004.

[26] W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *PPoPP*, pages 147–156. ACM, 2006.

[27] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213. ACM, 1995.

[28] S. Singh. Higher-order combinators for join patterns using STM. TRANSACT, June 2006.

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