Scalable Join Patterns

Claudio Russo

Microsoft Research

Joint work with Aaron Turon (Northeastern).

Outline

- Join patterns
- Examples
- Implementation
- Optimizations
- Evaluation
- Conclusion

Outline

Join patterns

Examples

Implementation

Optimizations

Evaluation

Conclusion

Background

Parallel programs eventually need to synchronize.

- Writing correct synchronization code is hard.
- Writing efficient synchronization code is harder still.

Neglect efficiency and promised performance gains are easily eroded …
…by the cost of synchronization!

Can we make this easier for programmers?

Goal

Our Goal:

- provide scalable, parallel synchronization
- using declarative, high-level abstractions

Our Recipe:

1. Take our favourite concurrency model:

   Join Calculus [Fournet & Gonthier]

2. Give it a scalable, parallel implementation.

Join Calculus in a nutshell

A basic calculus for message passing concurrency. Like π-calculus, but with a twist....

Definitions can declare synchronous and asynchronous channels.

Threads communicate and synchronize by sending messages:

- a synchronous send waits until the channel returns some result;
- an asynchronous send returns immediately, posting a message.

Chords

Definitions contains collections of chords (a.k.a. join patterns).
A chord pairs a pattern over channels with a continuation.
The continuation may run when these channels are filled.

Each send may enable:

some pattern causing a request to complete or a new thread to run;

no pattern causing the request to block or the message to queue.

All messages in a pattern are consumed in one atomic step.

Example (C#)

```
Asynchronous.Channel Knife, Fork;          // () -> void
Synchronous.Channel  Hungry, SpoonFeed;    // () -> void
```

Calling `Hungry()` waits until the channel returns control (a request).
Calling `Knife();` returns immediately, but posts a resource.

Chords specify alternative reactions to messages.

```
 // Hungry() waits for Knife() and Fork() ...
When(Hungry).And(Knife).And(Fork).Do(
    () => { eat(); Knife(); Fork(); });
// ... or a rendezvous with SpoonFeed()
When(Hungry).And(SpoonFeed).Do(
    () => { eat(); });
```

Here's a program with cutlery and two parallel threads....

```
Knife();Fork();                                  // set the table
spawn(() => { while (true) Hungry();} );         // child
spawn(() => { while (true) SpoonFeed();});       // parent
```

Dining Philosophers

Joins are declarative.

Solving a problem is often little more than stating it!

```
var table = Join.Create();
// declare the table's channels
Asynchronous.Channel F1,F2,F3,F4,F5; // fork resources
Synchronous.Channel H1,H2,H3,H4,H5;  // hungry requests
Asynchronous.Channel<Synchronous.Channel> Spawn; // NB: higher-order!

table.When(H1).And(F1).And(F2).Do(()=>{eat(); F1(); F2(); think();});
table.When(H2).And(F2).And(F3).Do(()=>{eat(); F2(); F3(); think();});
table.When(H3).And(F3).And(F4).Do(()=>{eat(); F3(); F4(); think();});
table.When(H4).And(F4).And(F5).Do(()=>{eat(); F4(); F5(); think();});
table.When(H5).And(F5).And(F1).Do(()=>{eat(); F5(); F1(); think();});
// spawning a single philosopher
table.When(Spawn).Do(H => { while (true) H(); });

F1(); F2(); F3(); F4(); F5(); // set the table
// spawn the philosophers
Spawn(H1); Spawn(H2); Spawn(H3); Spawn(H4); Spawn(H5);
```

Outline

Join patterns

Examples

Implementation

Optimizations

Evaluation

Conclusion

Buffers

```
class Buffer<T> {
  public readonly Asynchronous.Channel<T> Put; // T -> void
  public readonly Synchronous<T>.Channel Get;  // () -> T
  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; });
  }
}
```

- a call to `Get()` must wait for a `Put(t)`.
- argument of `Put` is returned to caller of `Get()`.

Locks

```
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
  }
}
```

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
  }
}
```

Just like `Lock`, but primed with n `Release` tokens.

Synchronous Swap Channels

```
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));
  }
}
```

- A call to `left(a)` must wait for a `right(b)` and vice versa.
- pair `(a,b)` returned to both callers.

Asymmetric Barrier

```
class Barrier {
  private readonly Synchronous.Channel[] Arrivals;
  public void Arrive(int i) {  Arrivals[i](); }

  public Barrier(int n) {
     // create j and init channels (elided)
    j.When(Arrivals[0]). /* ... */ .And(Arrivals[n-1]).Do(() => {});
}}
```

- a call to **`Arrive[i]`** must wait for a call to **`Arrive[j]()`**, for every other j.
- Each participant has a dedicated channel, **`Arrive[i]`** (hence asymmetric)
- A generalized and specialized **`Exchanger`** (n versus 2 channels, no data).

SymmetricBarrier

```
class SymmetricBarrier {
  public readonly Synchronous.Channel Arrive;
  public SymmetricBarrier(int n) {
    // create j and init channels (elided)
    j.When(Arrive). /* ... */ .And(Arrive).Do(() => {});
  }
}
```

- All participants share the same channel, **`Arrive`** (hence symmetric).
- uses a non-linear pattern (several occurrences of the same channel).

TreeBarrier

```
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());
}}}
```

- a better asymmetric barrier
- distributes work of waking participants (cf. combiners)
- improved scalability?

Parallel Performance

Joins are a great abstraction for concurrent synchronization.

Parallel Performance

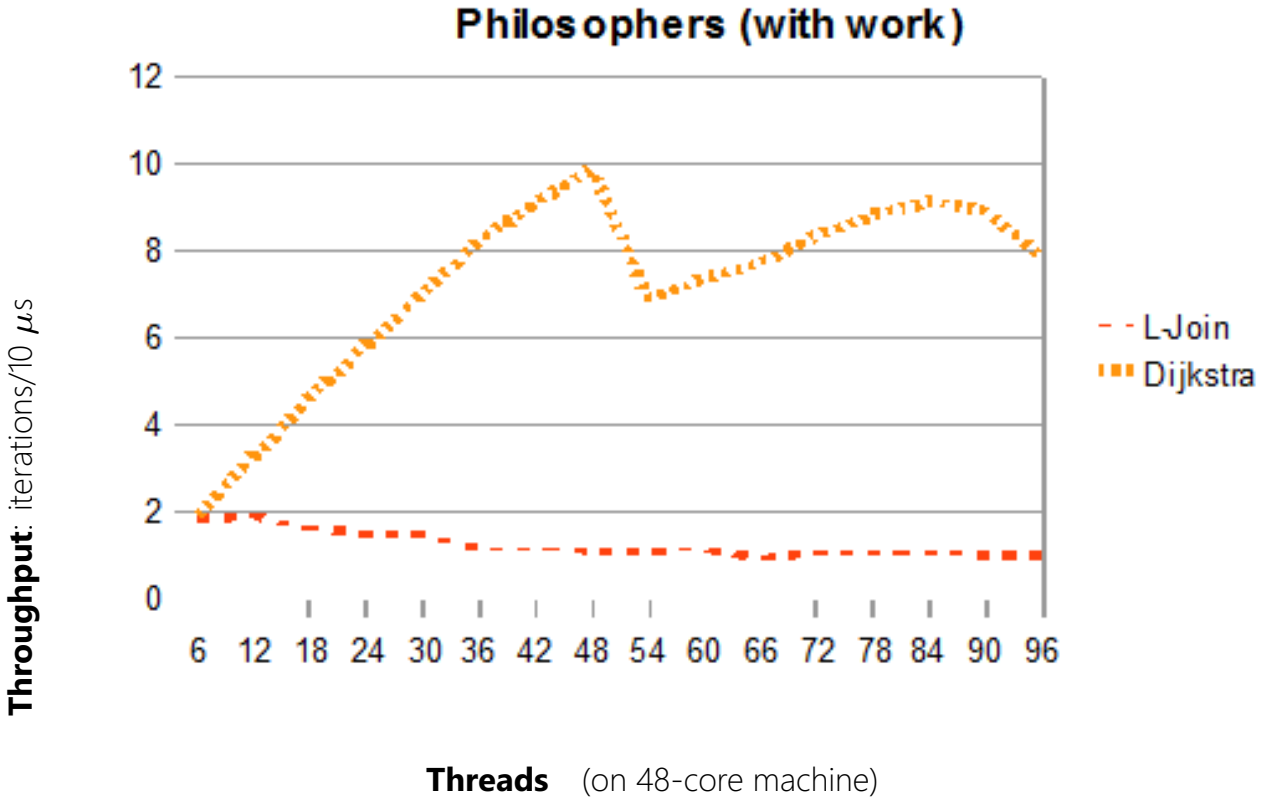
Joins are a great abstraction for concurrent synchronization.

But are they any good for parallel synchronization?

Parallel Performance

Joins are a great abstraction for concurrent synchronization.

But are they any good for parallel synchronization?

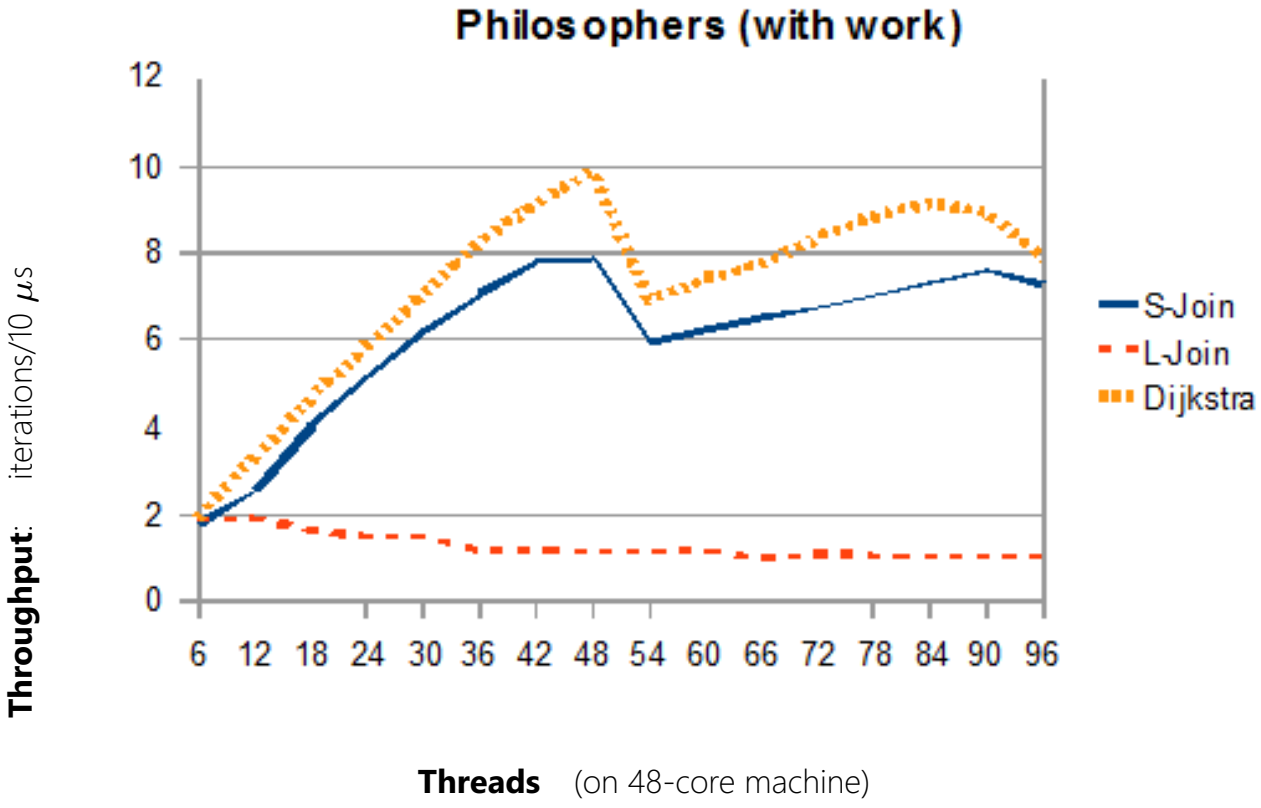


Almost all previous implementations use coarse-grained locks.
Unsurprisingly, locking doesn't scale.

Can we do better?

Can we do better?

Yes, with a parallel implementation.



Outline

Join patterns

Examples

Implementation

Optimizations

Evaluation

Conclusion

Previous Implementations

In lock-based implementations, sending a message amounts to:

```
R Chan.Send(A a) {
     var match = null;
     lock (this.Join) { // LOCK!
        // enqueue a
        var msg = this.Queue.Add(a);
        // find a match, consuming messages
        var match = this.Chords.FindMatch(msg);
     }
     if (match != null)
       return match.Fire(); // process a request or spawn a thread
     else
       return msg.Result(); // block on msg or return from async.
}
```

Clever implementations minimize time within the lock.
But the lock, and the inherent serialization of sends, remains.

Breaking the Bottleneck

Locks don't scale: a lock is a funnel, serializing execution.
We need a new implementation that avoids the lock.

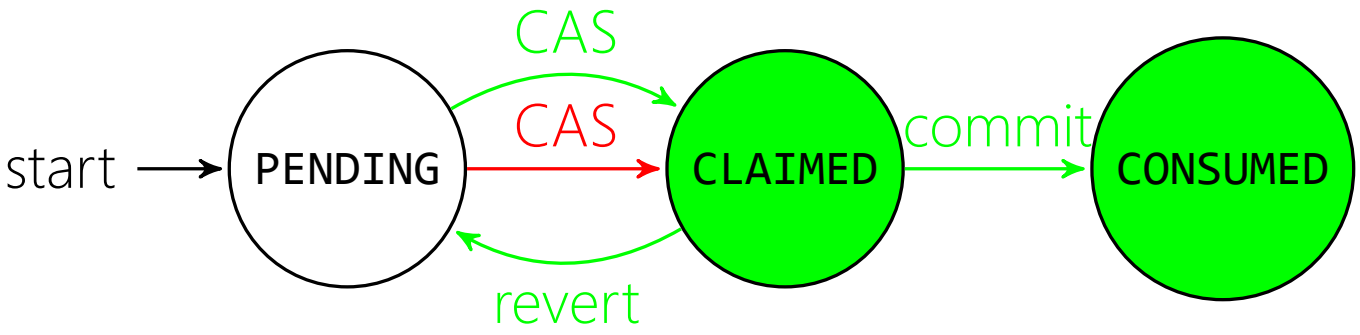
Represent each channel using a lock-free bag of messages:

- "lock-free" allows parallel addition of messages
- "bag semantics" (rather than queue) allows parallel search for available messages:

To atomically consume a pattern-full of messages:

- treat messages as resources, acquired by CASing on per-message status words.
- don't wait for the release of a message, try another instead!

Message State Transitions



- `PENDING` messages are available to any thread.
- Threads race to claim messages.
- Winner may revert/commit a claim.
- `CONSUMED` messages are logically deleted.
- A claim by another thread may become available later.

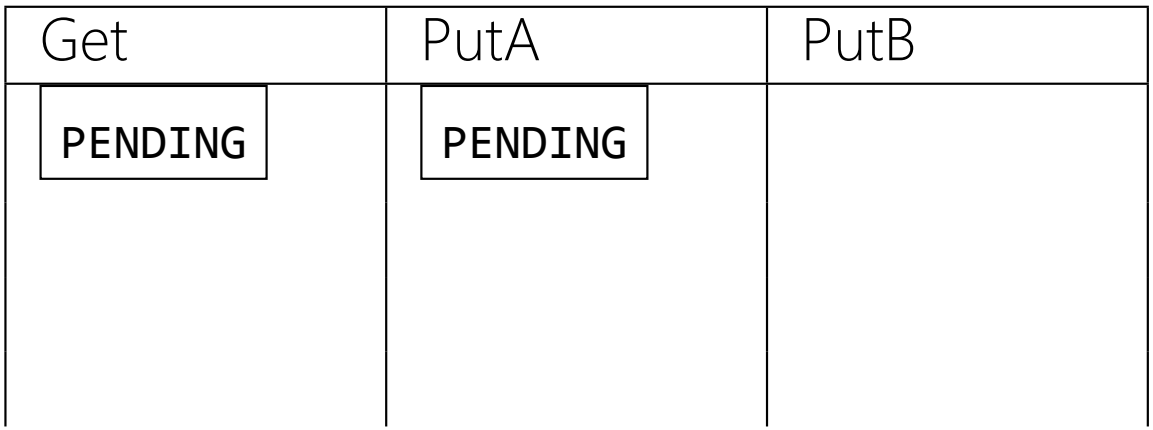
```
When(Get).And(PutA).And(PutB).Do(...)
```

Get	PutA	PutB

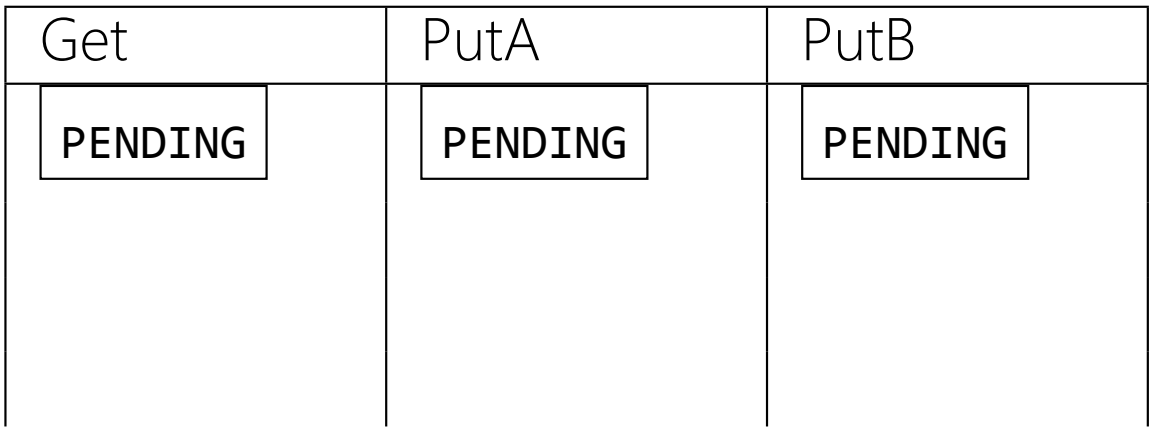
```
When(Get).And(PutA).And(PutB).Do(...)
```

Get	PutA	PutB
PENDING		

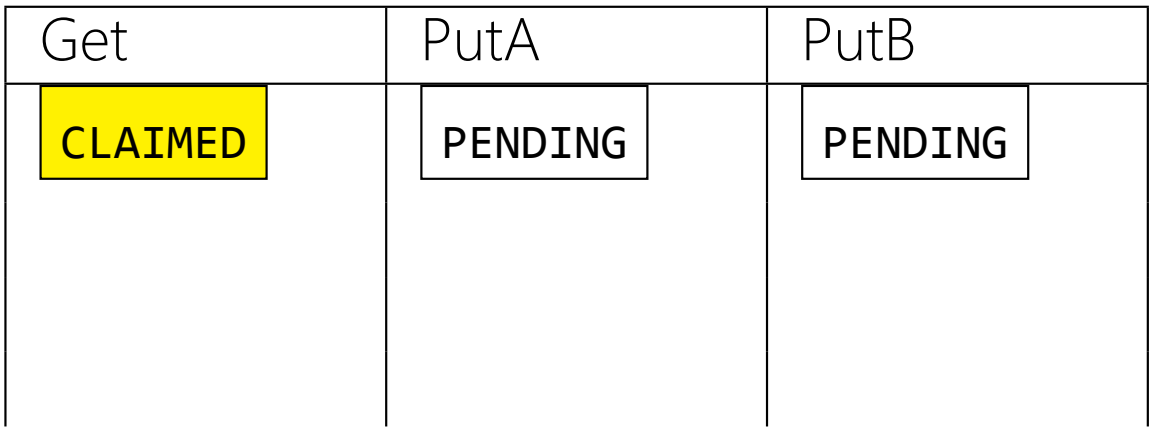
```
When(Get).And(PutA).And(PutB).Do(...)
```



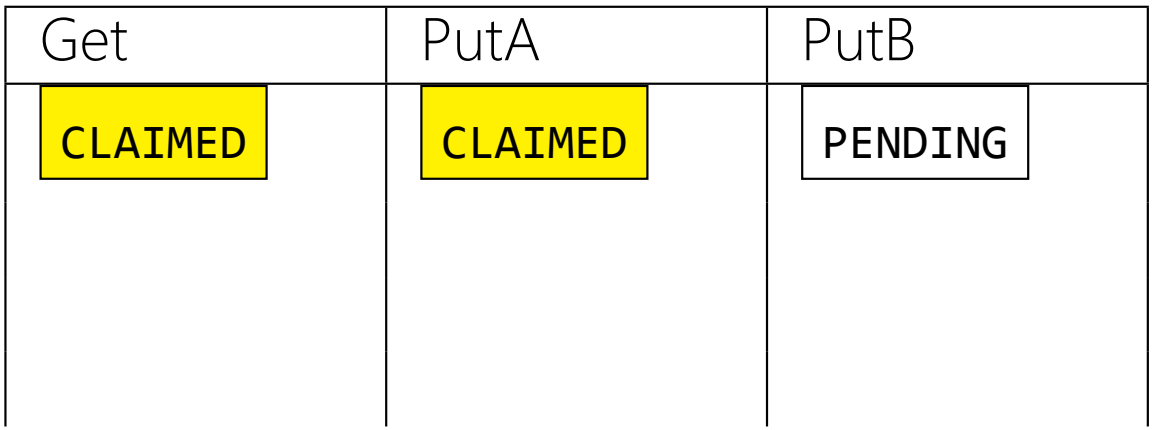
```
When(Get).And(PutA).And(PutB).Do(...)
```



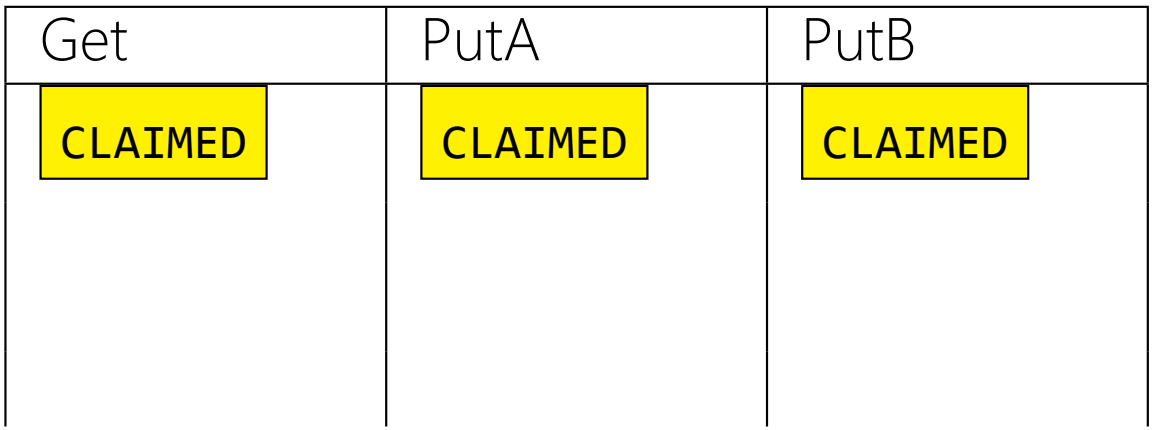
```
When(Get).And(PutA).And(PutB).Do(...)
```



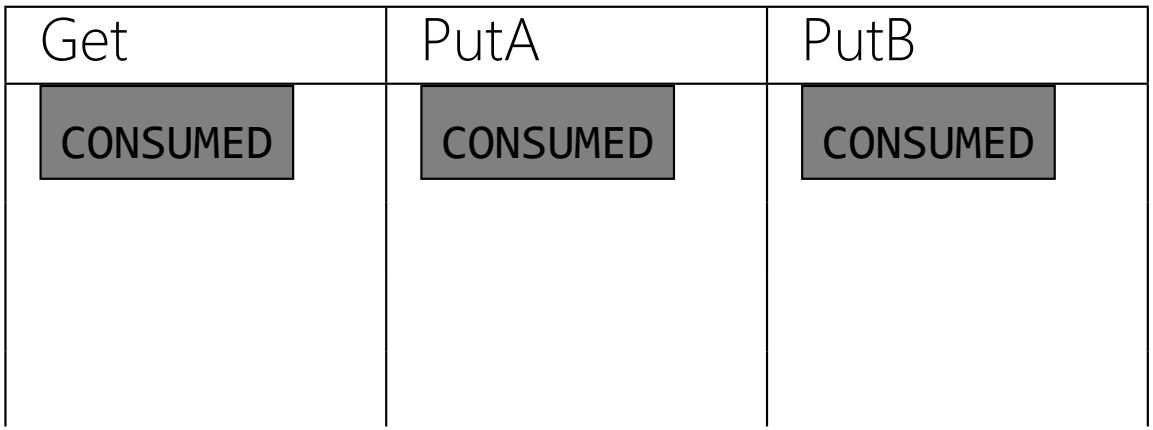
```
When(Get).And(PutA).And(PutB).Do(...)
```



```
When(Get).And(PutA).And(PutB).Do(...)
```



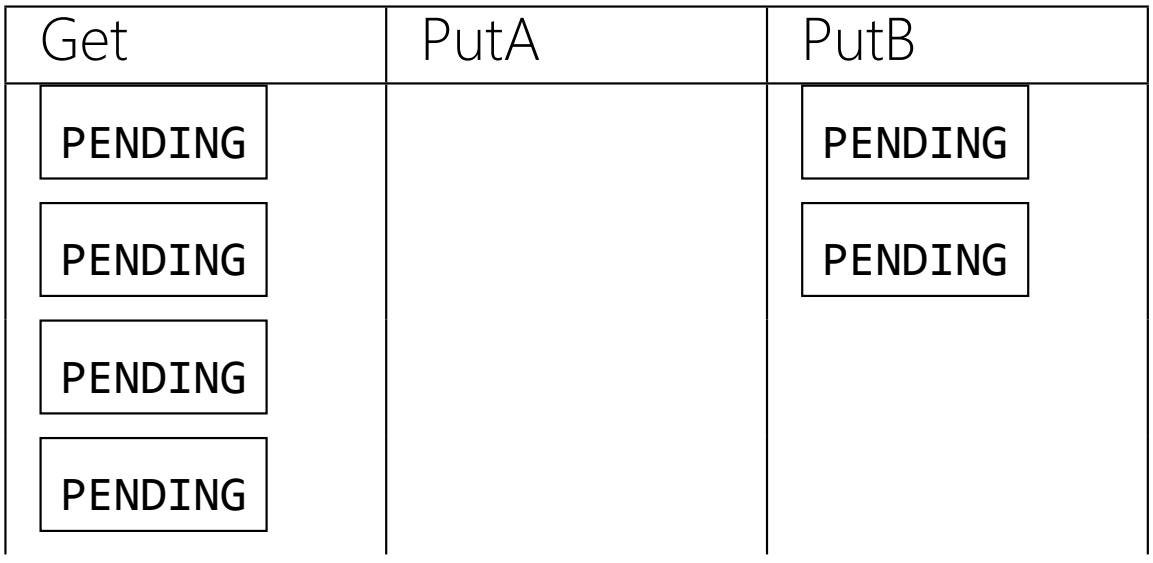
```
When(Get).And(PutA).And(PutB).Do(...)
```



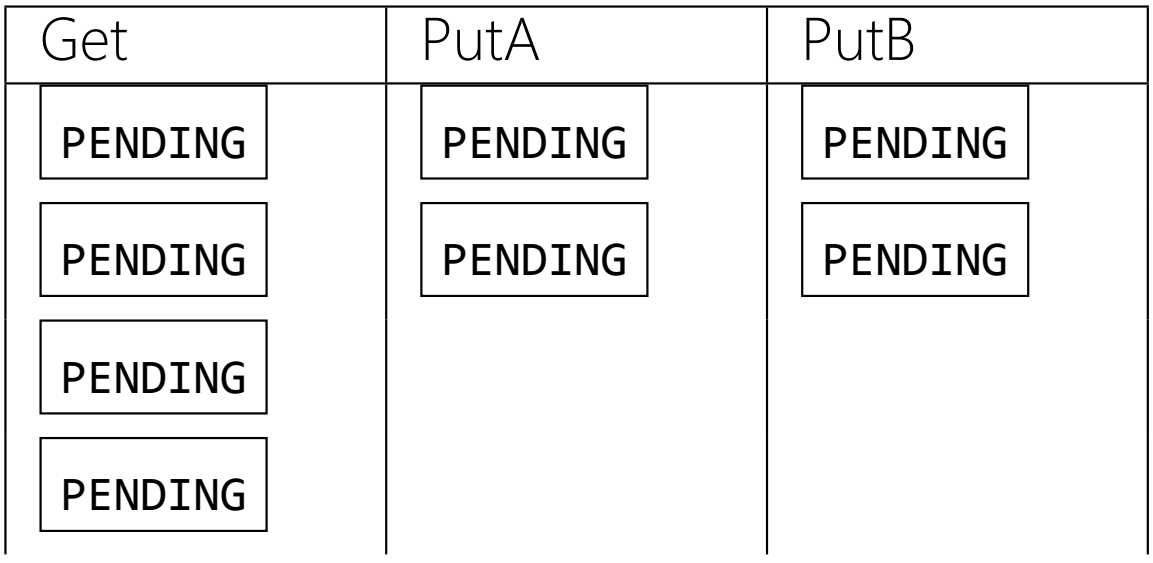
```
When(Get).And(PutA).And(PutB).Do(...)
```

Get	PutA	PutB

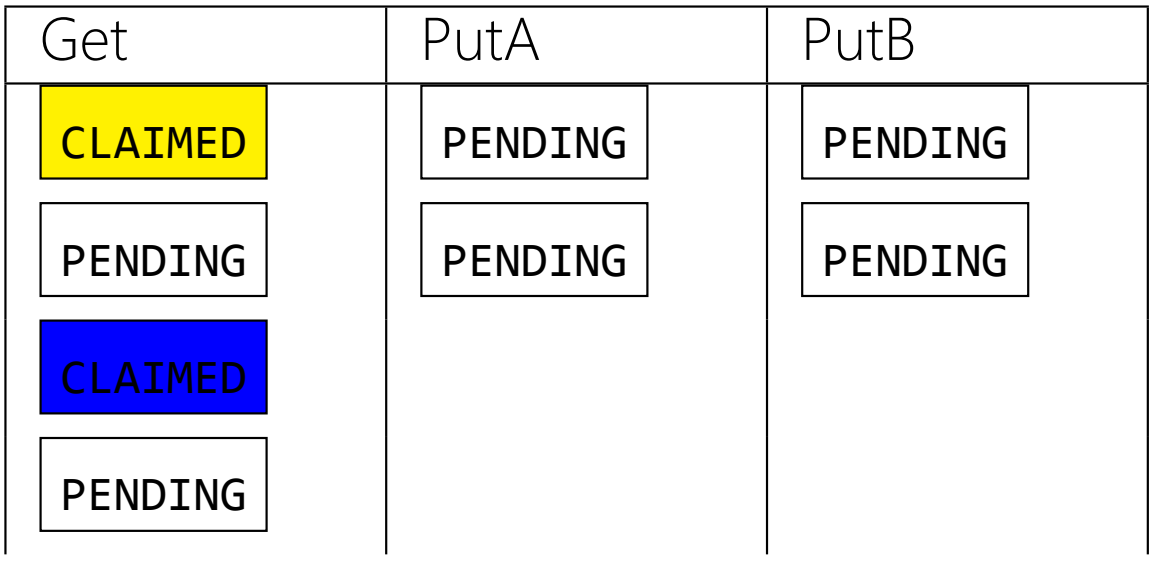
```
When(Get).And(PutA).And(PutB).Do(...)
```



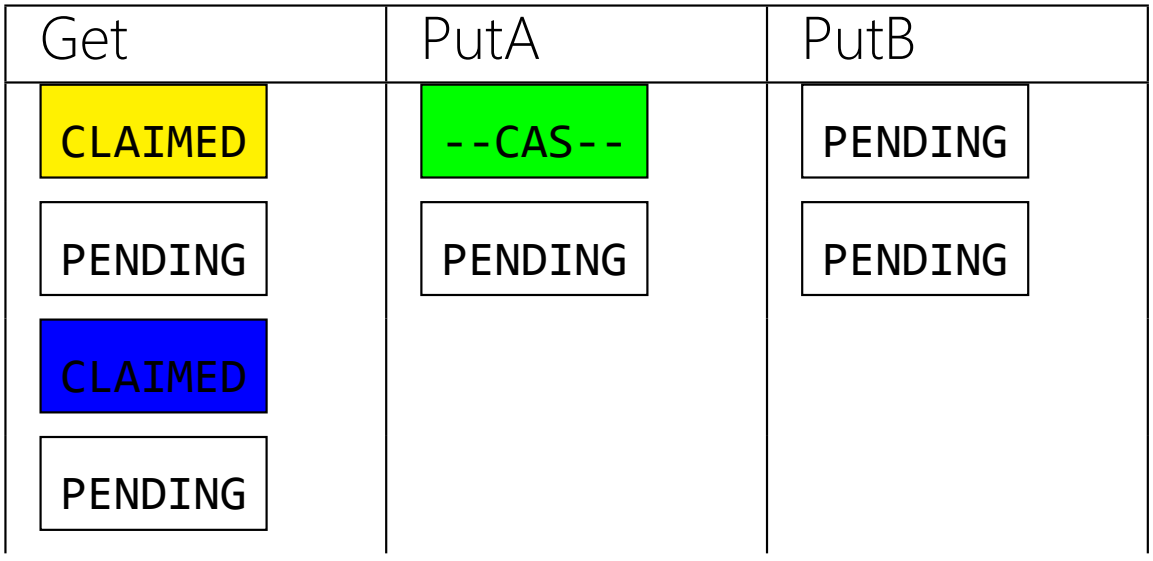
```
When(Get).And(PutA).And(PutB).Do(...)
```



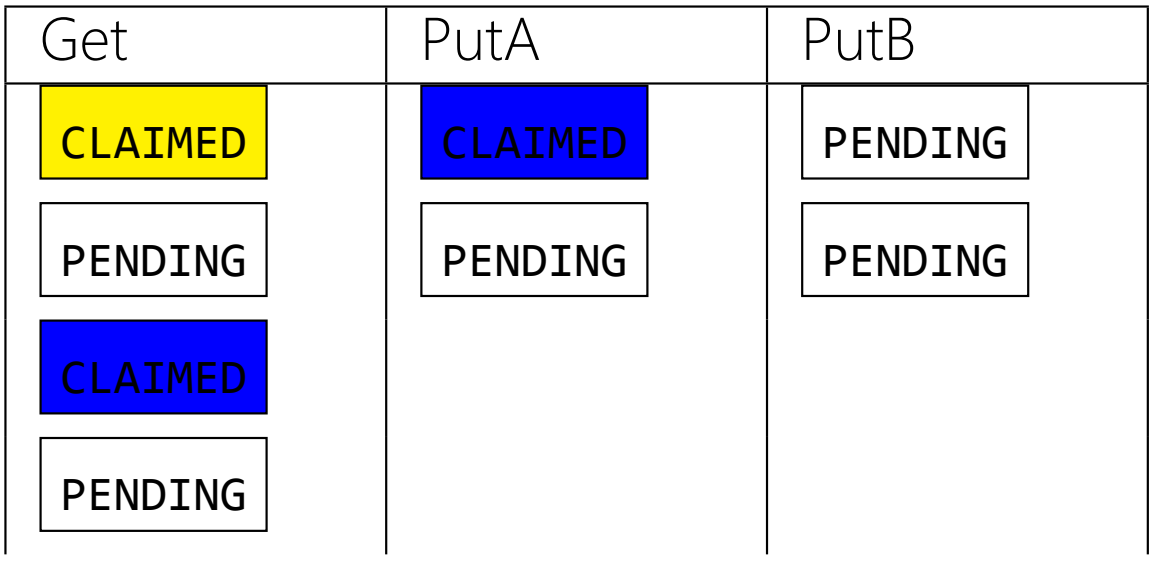
```
When(Get).And(PutA).And(PutB).Do(...)
```



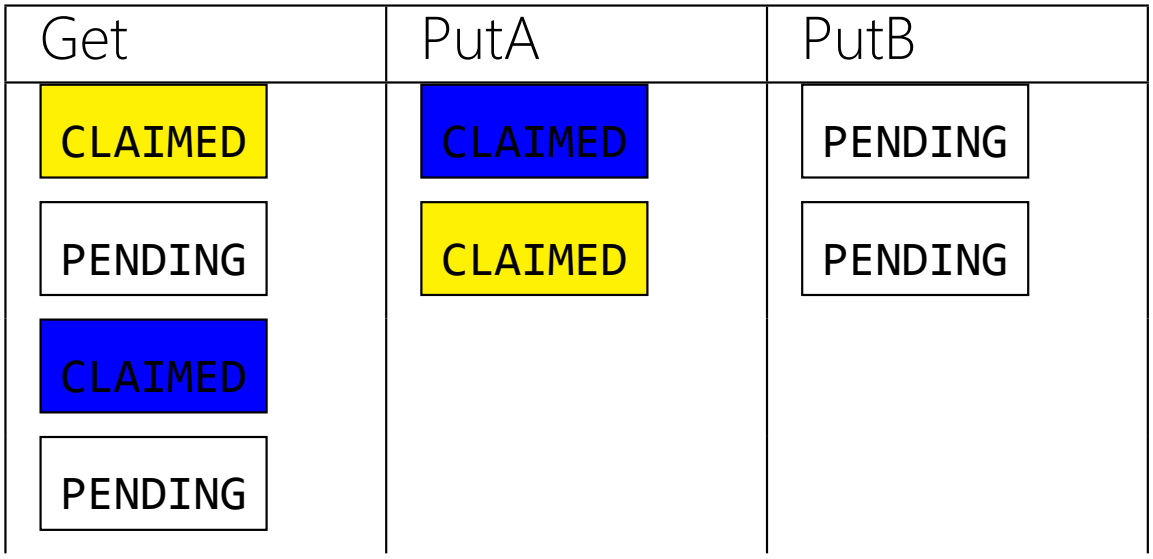
```
When(Get).And(PutA).And(PutB).Do(...)
```



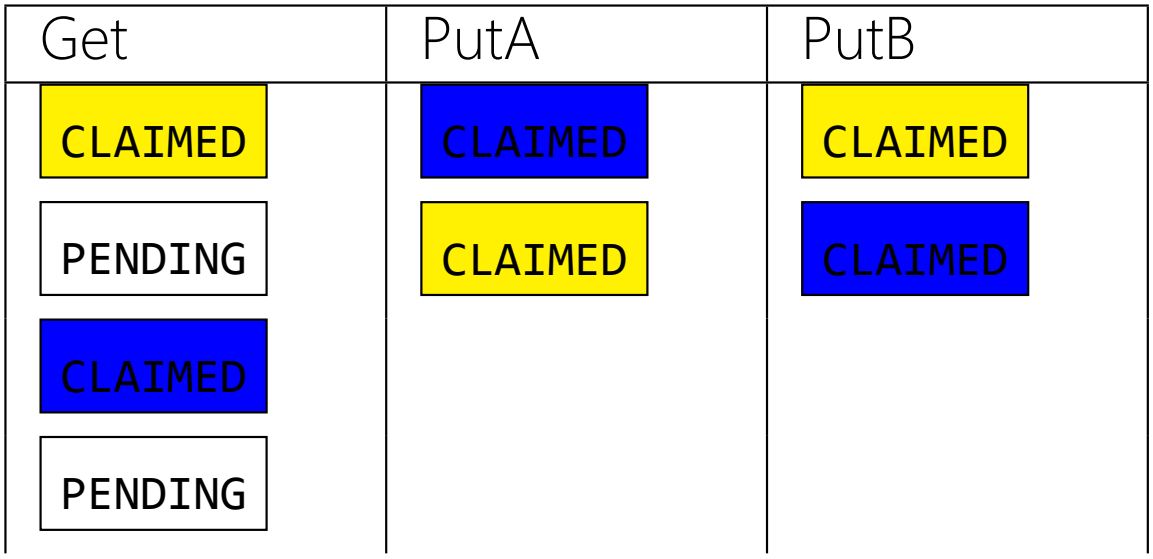
```
When(Get).And(PutA).And(PutB).Do(...)
```



```
When(Get).And(PutA).And(PutB).Do(...)
```



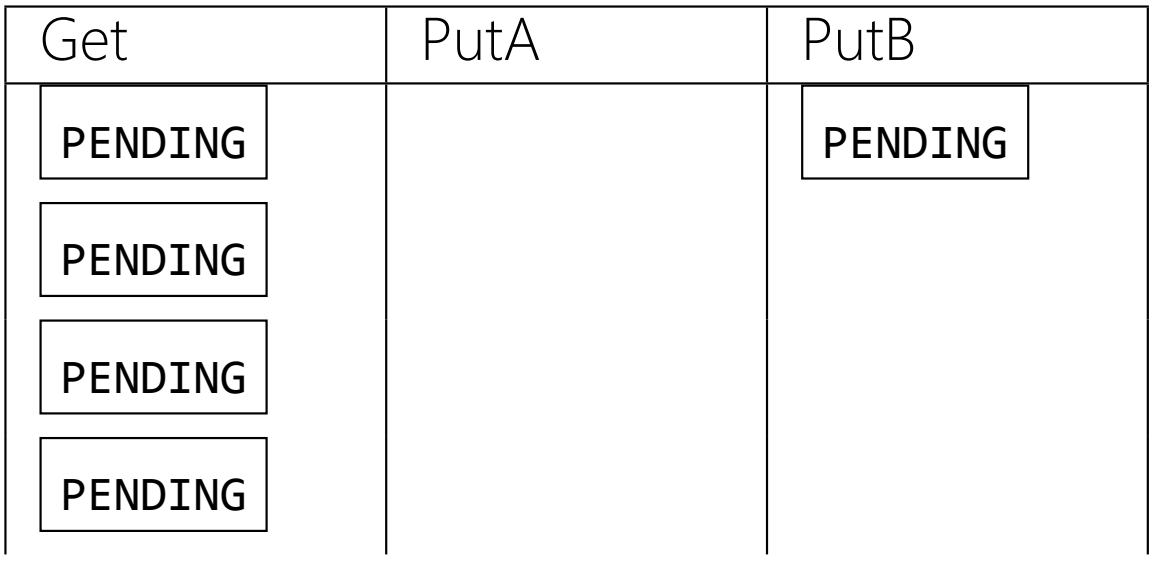
```
When(Get).And(PutA).And(PutB).Do(...)
```



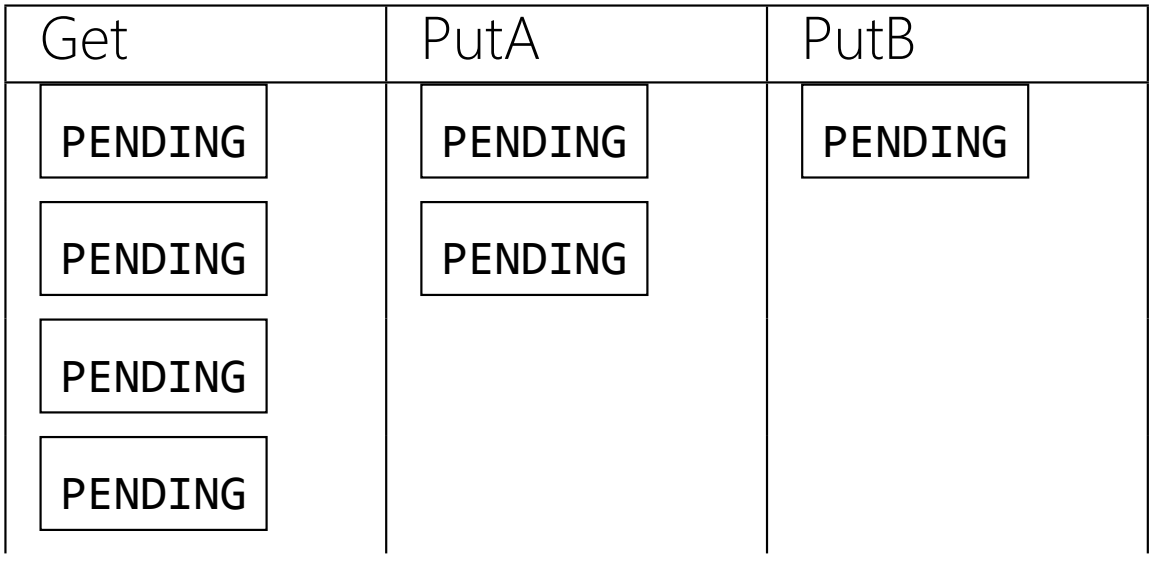
```
When(Get).And(PutA).And(PutB).Do(...)
```

Get	PutA	PutB

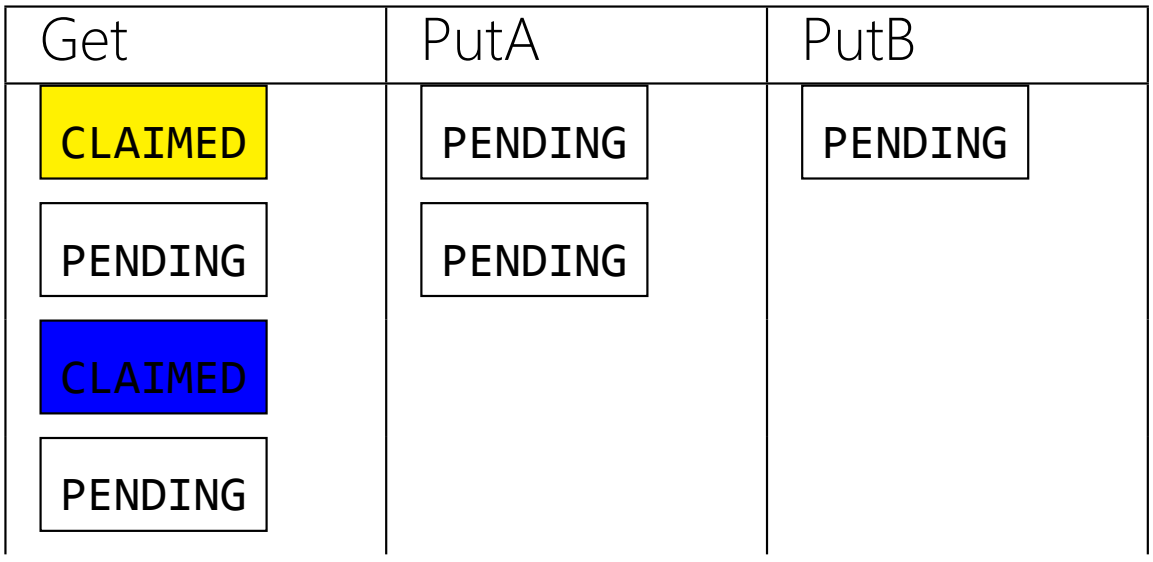
```
When(Get).And(PutA).And(PutB).Do(...)
```



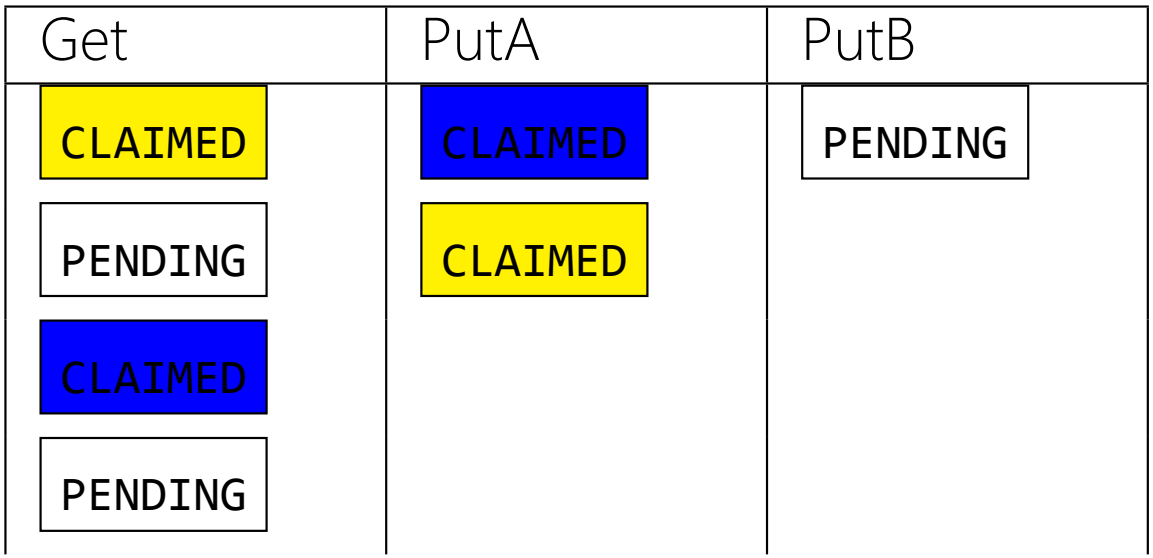
```
When(Get).And(PutA).And(PutB).Do(...)
```



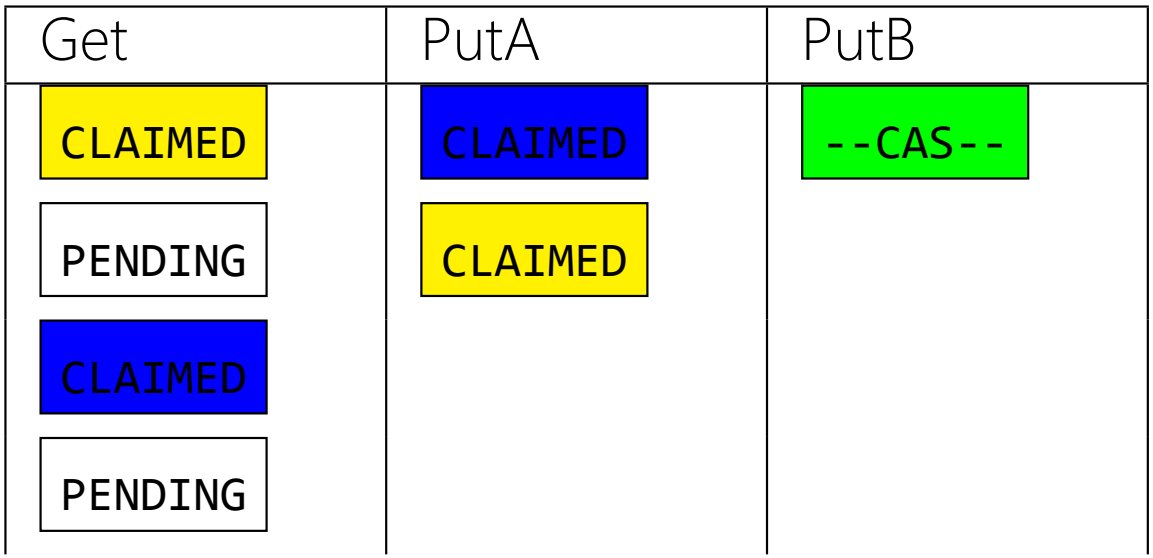
```
When(Get).And(PutA).And(PutB).Do(...)
```



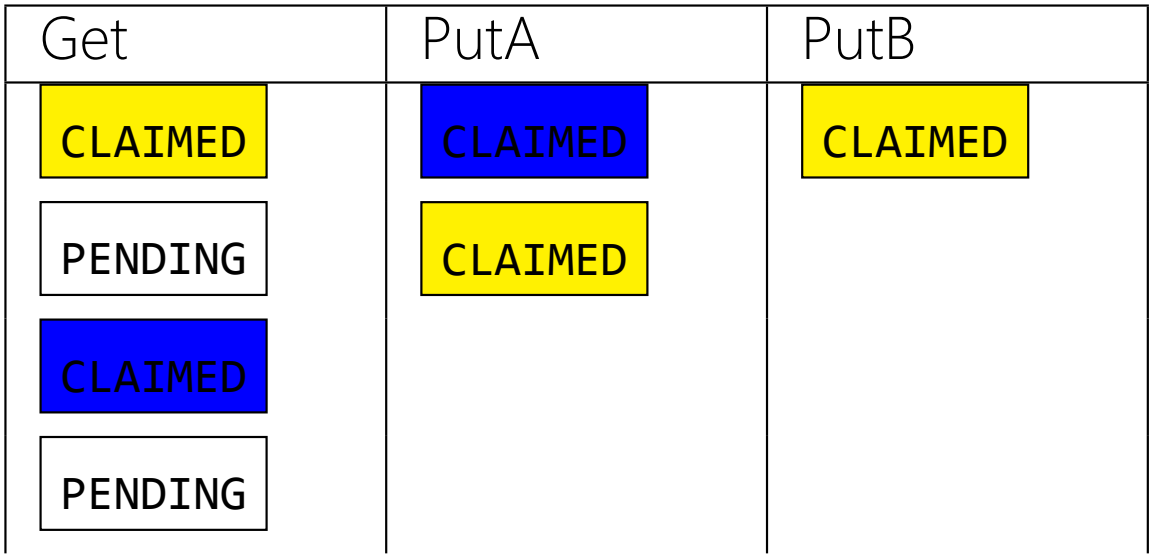
```
When(Get).And(PutA).And(PutB).Do(...)
```



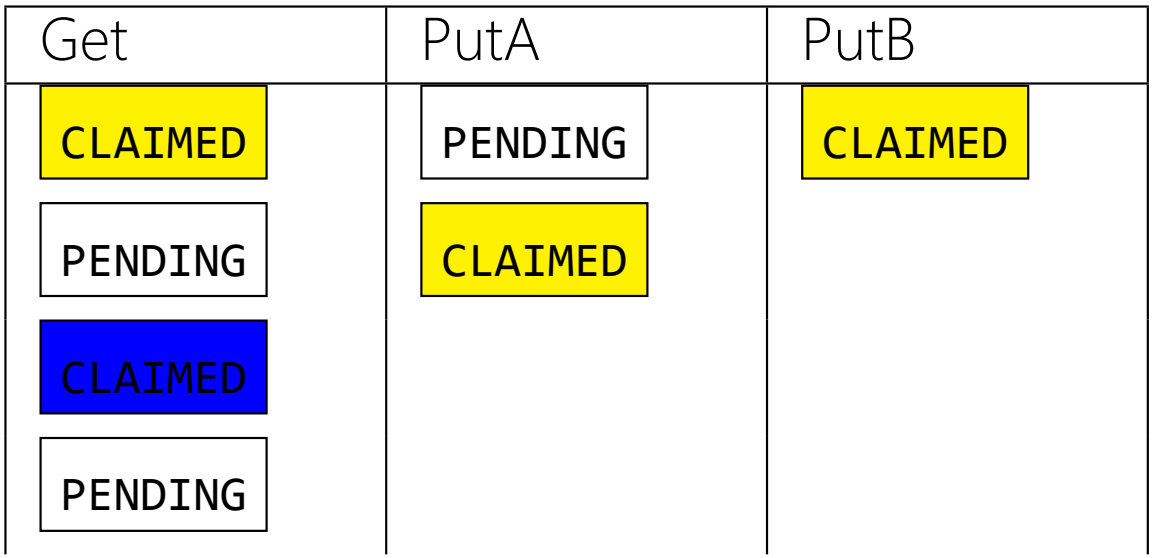
```
When(Get).And(PutA).And(PutB).Do(...)
```



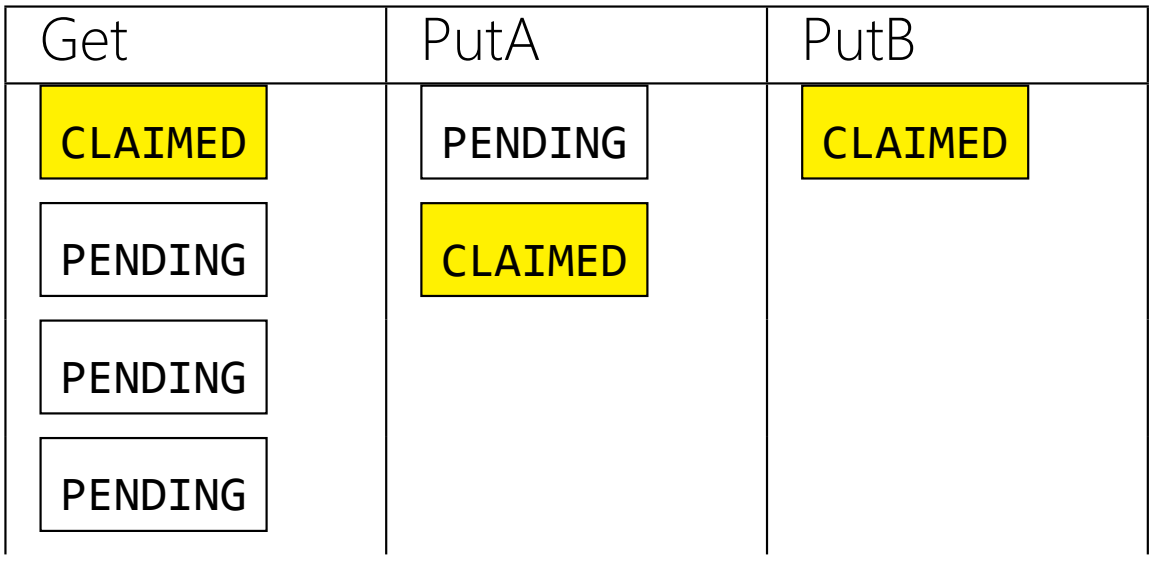
```
When(Get).And(PutA).And(PutB).Do(...)
```



```
When(Get).And(PutA).And(PutB).Do(...)
```



```
When(Get).And(PutA).And(PutB).Do(...)
```



Messages

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

class Msg {
  Chan Chan          { get; };       // enclosing channel
  Stat Status        { get; set; };  // current status
  bool TryClaim();  // CAS from PENDING to CLAIMED

  // synchronous messages
  Signal Signal      { get; };
  Match ShouldFire   { get; set; };
  R Result      { get; set; };
}
```

Messages are payloads paired with status words.

Channels

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

- AddPending(a) atomically adds a new message to the bag with payload `a` and status `PENDING`.
- FindPending(ref sawClaims):
  - traverses the bag (in order of addition) for the first `PENDING` message, skipping `CLAIMED` ones;
  - returns `null` if no `PENDING` message;
  - sets `sawClaims` to `true` if any `CLAIMED` message was skipped.
- Message returned by `FindPending()` need not stay `PENDING`.

Matches

```
// a pair of a chord and enough messages to fire it.
class Match {
  Chord Chord       { get; };
  Msg[] Claims      { get; };
}
```

Message Resolution

Every sender is responsible for resolving his own message:
A message is resolved if:

- `CLAIMED` by this thread (as part of a match).
- `CONSUMED` by another thread.
- Unable to complete any chord using previously sent messages.

Message Resolution

```
Match Resolve(Msg msg) {
  bool retry;
  do { retry = false;
       foreach (var chord in msg.Chan.Chords) {
         Msg[] claims = chord.TryClaim(msg, ref retry);//saw CLAIMED?
         if (claims != null) // got a match!
            return new Match(chord, claims);
       }
       if (msg.Status == Stat.CONSUMED) break;
  } while (retry)
  return null; // message consumed by other or no-match
}}
```

- Returns:
  - some match with chord and enough claims to fire.
  - **null** (no match), leaving **msg** in bag (possibly **CONSUMED**)
- Must retry while **CLAIMED** messages seen (may become **PENDING**)

Chord.TryClaim

```
Msg[] Chord.TryClaim(Msg msg, ref bool retry) {
  var msgs = new Msg[Chans.length];
  // locate enough pending messages to fire chord
  for (int i = 0; i < Chans.Length; i++) {
    if (Chans[i] == msg.Chan) msgs[i] = msg;
    else { bool sawClaims;
           msgs[i] = Chans[i].FindPending(out sawClaims);
           retry = retry || sawClaims;
           if (msgs[i] == null) return null; } }
  // try to claim the messages we found
  for (int i = 0; i < Chans.Length; i++)
    if (!msgs[i].TryClaim()) {// another thread won; revert!
       for (;--i >= 0;) msgs[i].Status = Stat.PENDING;
       retry = true;
       return null; };
  return msgs; // success: each message CLAIMED
}
```

Either returns successfully claimed messages needed to fire this chord; or **`null`** and the instruction whether to **`retry`**.

Asynchronous Send

```
void AsyncSend<A>(Chan<A> chan, A a){
 Msg myMsg = chan.AddPending(a); // add message
 Match m = Resolve(myMsg);       // find some match
 if (m == null) return;          // nothing to do: exit
 // otherwise, some chord found: schedule it!
 ConsumeAll(m.Claims);  // commit claims
 if (m.Chord.IsAsync) { // async chord: fire in new thread
   new Thread(m.Fire).Start();
 } else {               // sync chord: wake a waiter
   for (int i = 0; i < m.Chord.Chans.Length; i++) {
     // pick the first synchronous caller
     if (m.Chord.Chans[i].IsSync) {
       m.Claims[i].ShouldFire = m;  // transfer match to waiter
       m.Claims[i].Signal.Set();    // signal waiter
       return;
 }}}}
```

- asynchronous chord? Spawn a thread to fire the chord.
- synchronous chord? Elect one waiter to fire the chord & wake him up.

Synchronous Send

```
R SyncSend<R, A>(Chan<A> chan, A a) {
  Msg myMsg = chan.AddPending(a);
  Match m = Resolve(myMsg);
  if (m == null) {           // myMsg CONSUMED, or no match
    myMsg.Signal.Block();    // wait until woken
    m = myMsg.ShouldFire;
    if (m == null) return myMsg.Result; // chord returned Result
  } else ConsumeAll(m.Claims);    // found a match without blocking

  var r = m.Fire(); // execute the chord (on this thread)
  for (int i = 0; i < m.Chord.Chans.Length; i++) { // rendezvous
    if (m.Chord.Chans[i].IsSync && m.Claims[i] != myMsg) {
      m.Claims[i].Result = r;    // share my result
      m.Claims[i].Signal.Set(); }}  // wake up other waiter
  return r; // return the result
}
```

- Blocks (on signal) when resolved to no match possible.
- Executes match for result now or later (when awoken).
- If not selected, returns result computed by selected waiter.

Outline

Join patterns

Examples

Implementation

Optimizations

Evaluation

Conclusion

Stealing Optimization

When an asynchronous message enables a synchronous chord, our algorithm eagerly consumes messages, then wakes up a waiter.

- Waking a waiter has high latency.
- To increase throughput, it's better to signal a waiter to try again, then revert claimed messages to `PENDING` to allow others to steal them.
- Requires one additional state for synchronous messages (`WOKEN`). Details in the paper.

Exercise: spot the bug in the paper.

Lazy Enqueue

Adding a message to a channel is expensive:

- entails heap allocation;
- requires (costly) **CAS**.

It's faster to look for a match first, and only enqueue and retry if necessary.
(easy to implement)

Counter Optimization

The `Release()` channel of a lock/semaphore has no payload - it's a pure signal.

Other join implementations optimize the representation of signals to a simple integer that

- counts the available messages (think `unit list` $\simeq$ `nat`);
- avoids expensive heap allocation.

For our implementation, a simple counter won't suffice.

Instead, we can use a pair of half-words, counting pending and claimed messages, encoded in a single, atomic word…

Counter Optimization (code)

```
bool Msg.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) { // try to claim
      var nextState = Encode(++claimed, --pending);
      curState =  CAS(ref chan.state,startState,nextState);
      if (curState == startState) return true; // success!
    } else return false; // fail!
  } // retry
}
```

- Use classic read-modify-CAS in a loop to claim one pending message (return **`true`**)
- If none pending, return **`false`**.

Micro Optimizations

- Avoid all heap allocation:
  - stack allocate `Msg[]`
  - flatten tuples
  - use arrays (not linked lists).
- Avoid casts (use generics but know what to avoid).
- Our signals spin-wait before blocking.
- Do "exponential back-off" between retries to avoid contention.

Outline

Join patterns

Examples

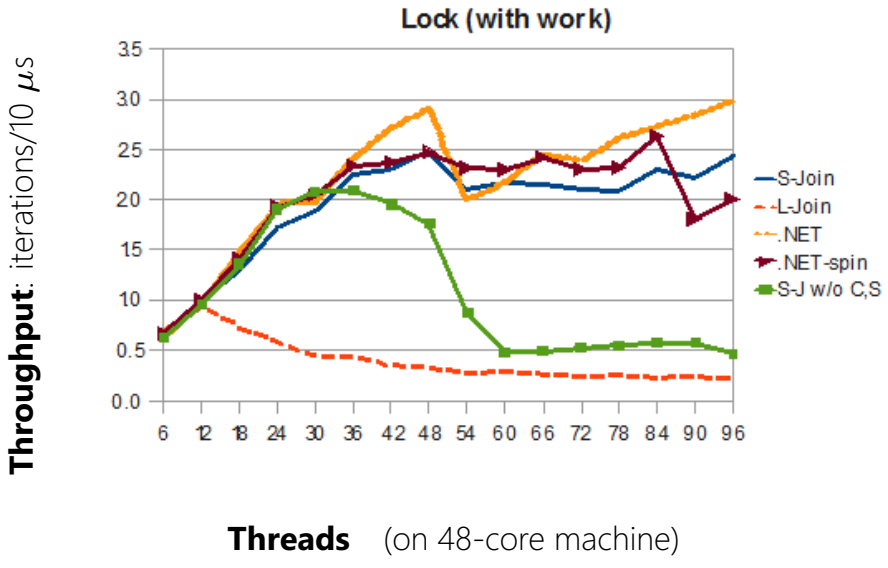
Implementation

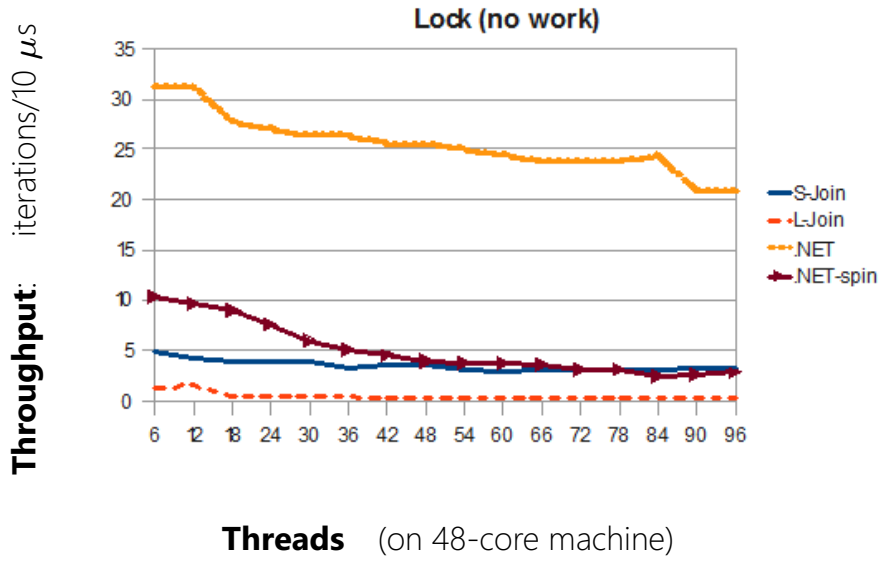
Optimizations

Evaluation

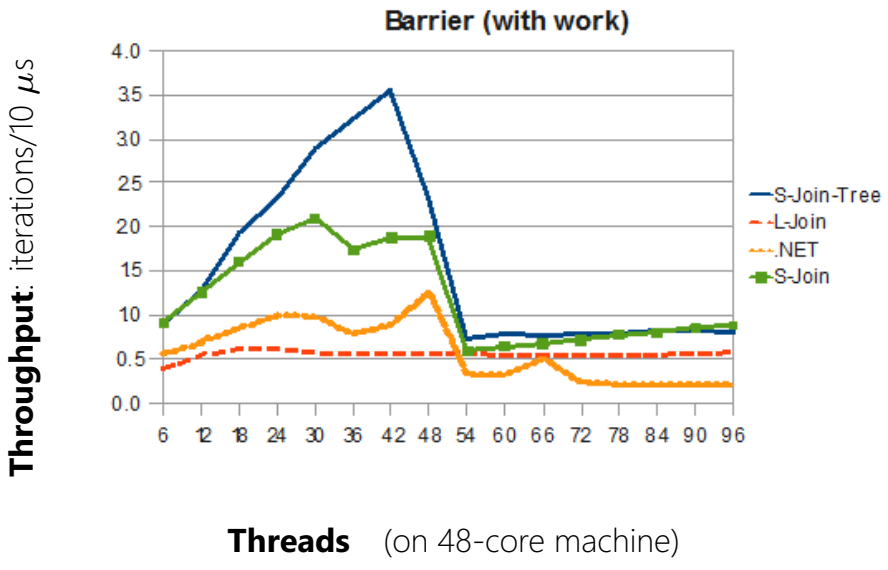
Conclusion

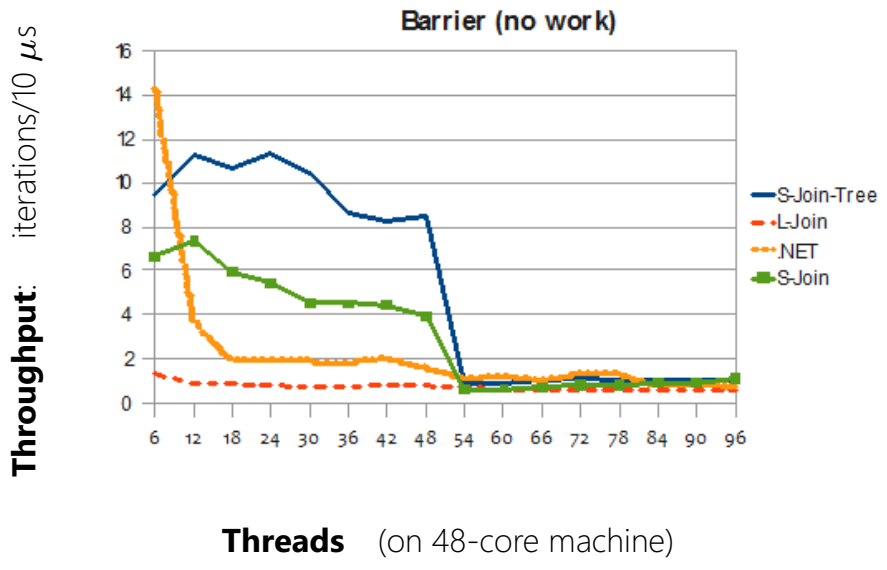
Lock Performance



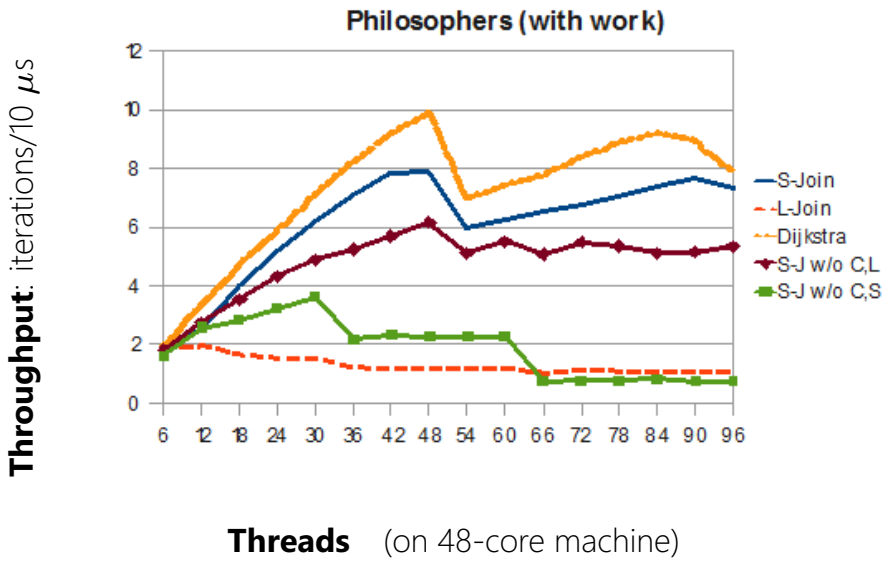


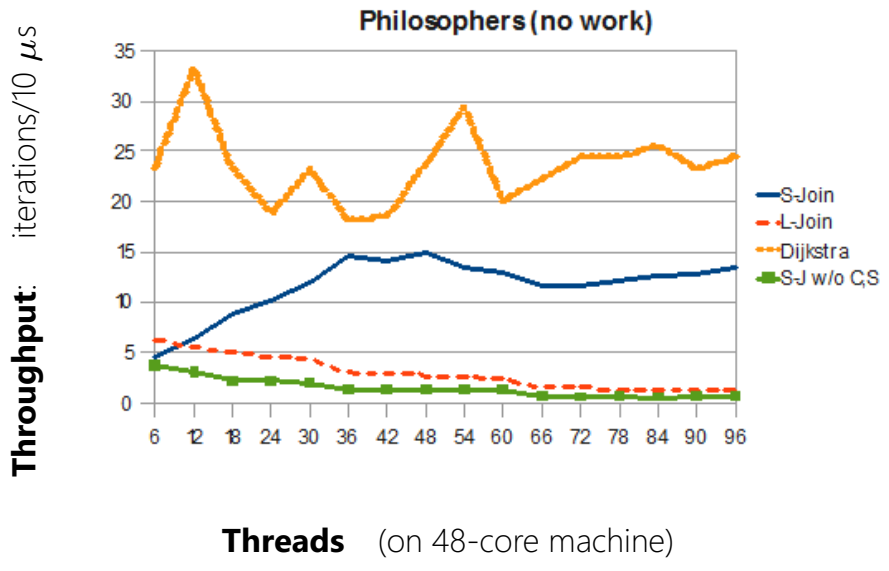
Barrier Performance



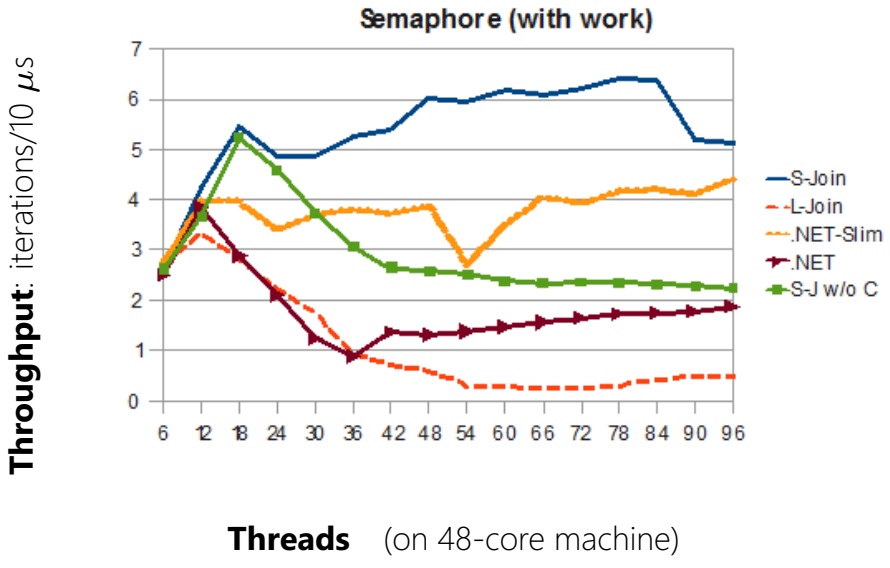


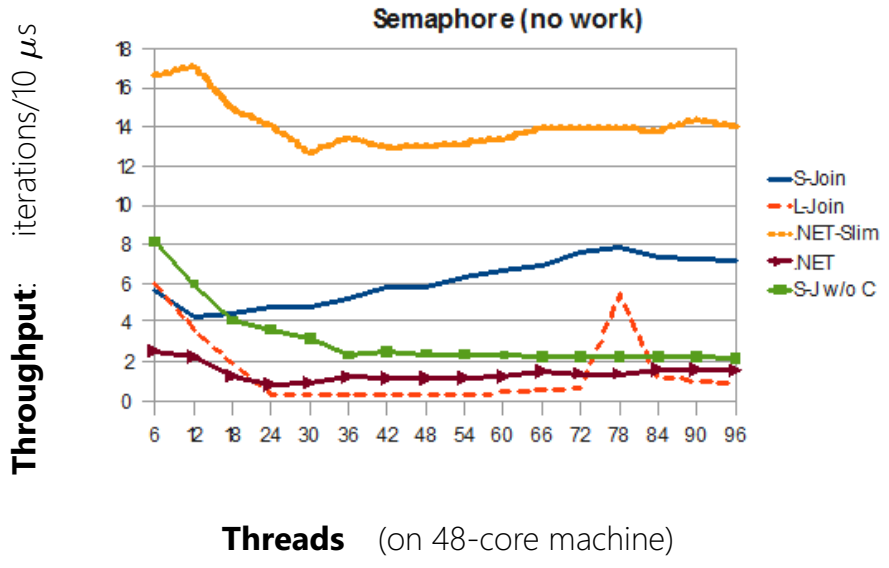
Philosophers Performance



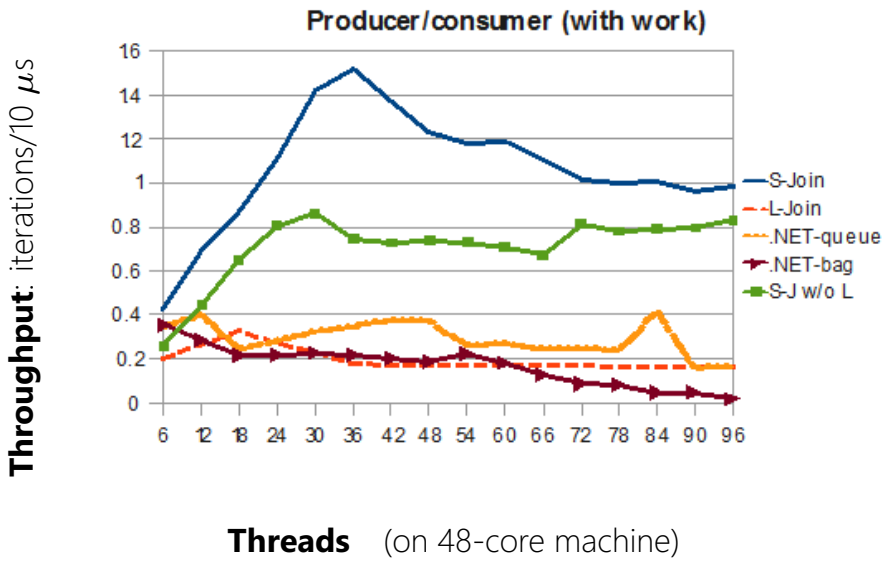


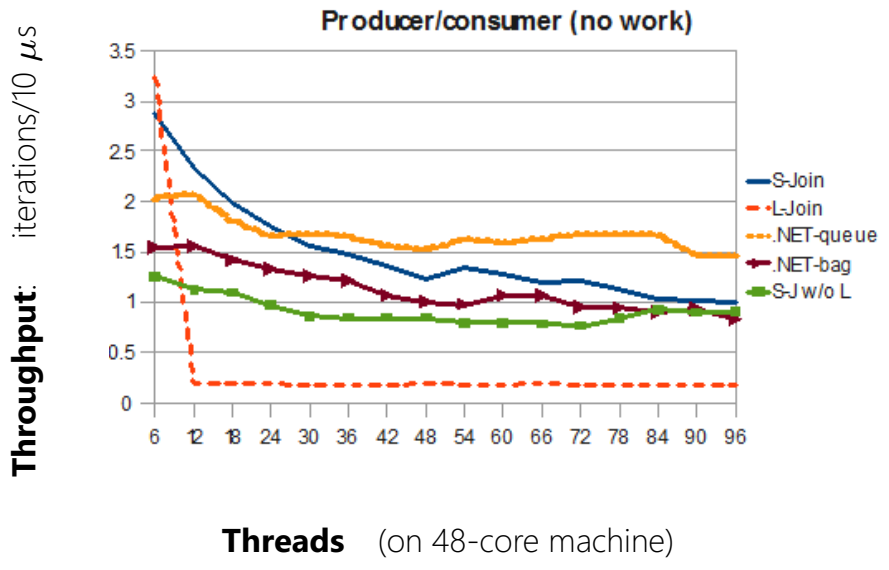
Semaphore Performance



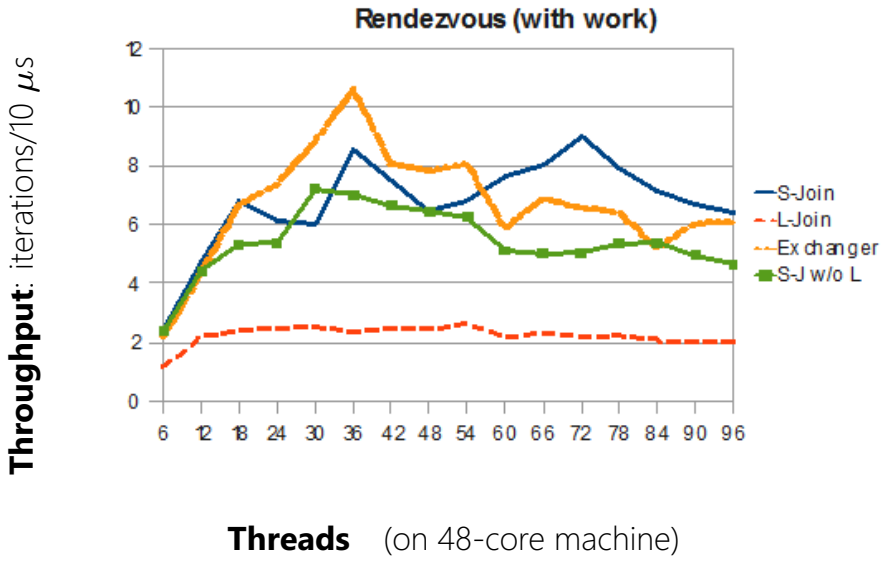


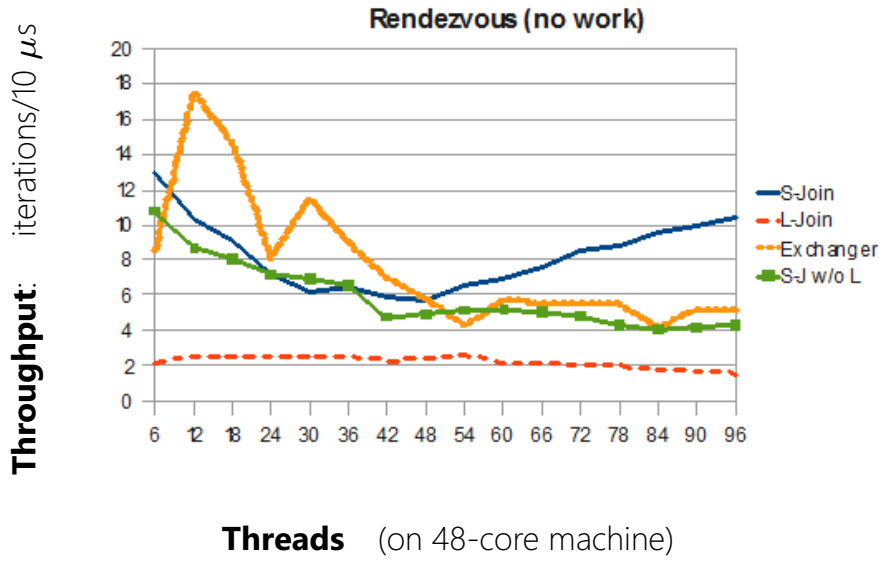
Producer/Consumer Performance





Rendezvous Performance





Correctness

It's fast,

Correctness

It's fast, but is it correct?

Correctness

It's fast, but is it correct?

Informal safety and liveness argument in the paper; Aaron is working on techniques for a formal proof.

Key points:

1. We rely on linearizability of our bags to establish a global message order.
2. A sender is responsible for detecting any matches involving messages sent before this one.
   (That's why we need to retry until certain it's possible to proceed or safe to give up.)
3. The status of a resolved message cannot change due to previous messages (only new ones).
4. Channels are sorted to avoid deadlock.

Outline

Join patterns

Examples

Implementation

Optimizations

Evaluation

Conclusion

Summary

We use join patterns [Fournet & Gonthier] for parallel synchronization:

- Expressive: make old and new synchronization primitives easy to write.
- Scalable: often competitive with bespoke implementations (provided by platforms).

Most platforms just provide a small library of coordination primitives.

Could (something like) joins provide one abstraction for ″the next 700 synchronization primitives″?

Related Work

- Join Calculus [Fournet & Gonthier]
- JoCaml [Fournet et. al.] single-threaded extension of OCaml (no shared memory parallelism).
- Funnel [Odersky et. al.] functional join patterns on the JVM (lock-based).
- Polyphonic C# [Benton et. al.], Cω [MSRC], Concurrent Basic [Russo] all implement join patterns, support parallelism, but are lock-based.
- Joins [Russo] is C# is lock-based library for join patterns. This work is a drop-in replacement (with some extensions).
- Parallel Concurrent ML [Reppy et. al.]. Similar in flavour and our original inspiration; each channel still protected by a dedicated lock.
- Reagents [Turon]. Composable abtractions for concurrent data structures (not just synchronization).
- STM Joins [Sulzman,Singh] Haskell implementations using STM - don't appear to scale - too many conflicts?

Links

Paper:
Scalable Join Patterns
A. Turon (Northeastern U.) and C. Russo (MSR)
OOPSLA 2011
`http://dl.acm.org/citation.cfm?id=2048111`

The (original) lock-based Joins library project page.
`http://research.microsoft.com/en-us/um/people/crusso/joins/`
Download (for Visual Studio 2008 with automatic upgrading to VS 2010/2012)
Includes binary for the library and sources for samples.
`http://research.microsoft.com/en-us/downloads/f9d6994e-45f6-49b8-b3c9-2a44bb2a4c50/`

Resources:
Visual Studio Asynchronous Programming
`http://msdn.microsoft.com/en-us/vstudio/async.aspx`