A Lock-Free Interaction Combinator Evaluator

Victor Taelin

Abstract

We present a lock-free evaluator for interaction combinators, a simple yet powerful model of distributed computation. The proposed algorithm is based on the concept of implicit ownership regions and redirection wires, allowing for efficient parallel evaluation without race conditions or deadlocks. We provide detailed pseudocode for the core procedures and discuss their properties. Our implementation demonstrates promising performance and near-ideal speedup on massively parallel architectures.

1 Introduction

Interaction nets are a graphical model of computation designed as a generalization of the proof structures of linear logic [**?**]. They consist of graph-like structures composed of agents and edges, where each agent has a type and an arity connected to other agents via its ports. Interaction nets are inherently distributed, allowing computations to take place simultaneously in many parts of a net without synchronization. This makes them suitable for modeling massive parallelism.

A specific type of interaction nets is the system called *interaction combinators*, introduced by Lafont [**?**]. These are built from three types of nodes: constructor (CON), duplicator (DUP), and eraser (ERA). Each node has a principal port and two auxiliary ports. The nodes interact according to six rules, divided into two categories: commutation and annihilation. Commutation rules apply when two different types of nodes are connected through their principal ports, while annihilation rules apply when two nodes of the same type are connected through their principal ports.

Interaction combinators possess several key properties such as determinism, locality, strong confluence (diamond property), and universality. These properties make them particularly suitable for modeling distributed and parallel computations [**?**].

Despite the theoretical appeal of interaction combinators as a model of distributed computation, there is still no real-world implementation that performs well in practice 26 years after their introduction. In this paper, we propose a lock-free evaluator for interaction combinators that achieves massive parallelism on GPUs with promising performance and near-ideal speedup. The heart of our implementation is a lock-free `link()` procedure which we describe in detail.

The rest of the paper is organized as follows: Section 2 provides background information on interaction nets and interaction combinators; Section 3 presents our lock-free evaluator proposal along with its core procedures; Section 4 provides detailed pseudocode for all functions; Finally, Section 5 concludes the paper.

2 Background

In this section, we provide an overview of interaction nets and interaction combinators to establish the necessary background for understanding our proposed algorithm.

2.1 Interaction Nets

Interaction nets were introduced by Lafont as a generalization of Girard's proof structures for linear logic [**?**]. They are graph-like structures composed of agents (nodes) connected by edges (wires). Each agent has a type and an arity determined by its number of ports. Ports can be connected to other agents via edges or left unconnected (free ports). The free ports form the interface of an interaction net.

The computation in an interaction net is performed by applying a set of *interaction rules* that rewrite pairs of connected agents. These rules are local, meaning they only affect a small, constant-size region of the net. This locality property ensures that computations can take place simultaneously in many parts of an interaction net without synchronization, making them suitable for modeling parallelism.

2.2 Interaction Combinators

Interaction combinators are a specific type of interaction nets introduced by Lafont [**?**]. They consist of three types of nodes: constructor (CON), duplicator (DUP), and eraser (ERA). Each node has a principal port and two auxiliary ports. The nodes interact according to six rules, divided into two categories: commutation and annihilation.

Commutation rules apply when two different types of nodes are connected through their principal ports:

- CON-CON: The constructors exchange their auxiliary ports.
- CON-DUP: The constructor's auxiliary ports are connected to the duplicator's auxiliary ports.
- CON-ERA: The constructor is erased.

Annihilation rules apply when two nodes of the same type are connected through their principal ports:

- ERA-ERA: Both erasers disappear.
- DUP-DUP: Both duplicators disappear.
- CON-DUP: Both constructor and duplicator duplicate each other.

3 Lock-Free Evaluator

In this section, we present our lock-free evaluator for interaction combinators. We first recap the basics of interaction combinators and discuss the problem with naive parallel reduction. Then, we briefly describe a lock-based solution and its limitations before presenting our lock-free proposal.

3.1 Problem with Naive Parallel Reduction

The naive approach to reducing interaction combinators in parallel is to maintain a set of redexes (active pairs) and process them concurrently. However, this can lead to race conditions due to overlapping regions.

Consider the following example:

```
(CON x a a)
(CON x b c)
(CON y b c)
(CON y d d)
```

There are two redexes: one formed by the first two nodes and another formed by the last two nodes. Each redex can affect two ports on the other redex (the 'b' and 'c' ports), resulting in an overlap of 2 ports among 12 total ports. Attempting to reduce these nodes in parallel naively could result in race conditions.

3.2 Lock-Based Solution

A possible solution is to use locks on regions that need to be rewritten. For example, we could lock "surrounding" nodes before performing an interaction since a full rewrite can involve up to 6 nodes. Locking these nodes would make it thread-safe.

However, safely locking these 6 nodes presents challenges such as avoiding deadlocks or ensuring correct surrounding node sets after central node modifications by other threads.

One approach is first locking central nodes, which fixes surrounding node sets, then attempting surrounding node locks afterward. If successful, perform rewrite; otherwise release all locks and back-off for later retry attempts.

This solution works but may result in temporary deadlocks where threads repeatedly lock/unlock their respective regions until one obtains entire region control—performance heavily relies on the scheduler. On CPU, this algorithm performs satisfactorily; on GPU, where warp threads are in lockstep, performance may suffer.

3.3 Lock-Free Proposal

We propose an efficient lock-free reduction algorithm based on implicit ownership regions and redirection wires. The idea is to process a set of redexes in parallel but perform rewrites locally without immediately affecting surrounding nodes. Redirection wires allow us to replace central node ports with redirection pointers, enabling nodes pointing to them now to semantically point to their post-reduction locations.

When a redirection pointer points to a main port, we also implicitly own that pointer and start the `link()` procedure (detailed pseudocode provided in Section 4). This procedure attempts connecting owned node (`src_ref`) towards target location (`dir_ptr`), resolving all encountered redirection nodes using atomic operations if necessary.

As long as the `link()` operation remains atomic, other parts of the algorithm can be performed in parallel. Moreover, we can break redexes into four segments for concurrent processing by separate threads:

- Thread A1: moves A.aux1 towards B.aux1
- Thread A2: moves A.aux2 towards B.aux2
- Thread B1: moves B.aux1 towards A.aux1
- Thread B2: moves B.aux2 towards A.aux2

This optimization increases granularity further and allows 4x more cores than active redex count usage.

4 Pseudocode

In this section, we provide detailed pseudocode for our proposed lock-free evaluator's core functions:

5 Conclusion

We have presented a lock-free evaluator for interaction combinators based on implicit ownership regions and redirection wires. Our proposed algorithm allows efficient parallel evaluation without race conditions or deadlocks. The detailed pseudocode provided demonstrates the core procedures of our implementation.

Our lock-free evaluator shows promising performance and near-ideal speedup on massively parallel architectures such as GPUs. This work contributes to the practical realization of interaction combinators as a powerful model of distributed computation.

References

Algorithm 1 The link() procedure

```
1:  procedure LINK(src_ref : &Pointer, dir_ptr : Pointer)
2:      while True do
3:          trg_ref ← dir_ptr.target()
4:          trg_ptr ← trg_ref.read()
5:          if trg_ptr.is_red() then
6:              trg_ref.write(0)
7:              dir_ptr ← trg_ptr
8:              continue
9:          else if trg_ptr.is_aux() then
10:             src_ptr ← src_ref.read()
11:             if trg_ref.compare_exchange(trg_ptr, src_ptr) = trg_ptr then
12:                 trg_ref ← dir_ptr.target()
13:                 trg_ptr ← trg_ref.read()
14:                 while trg_ptr.is_red() do
15:                     trg_ref.write(0)
16:                     trg_ref ← trg_ptr.target()
17:                     trg_ptr ← trg_ref.read()
18:                 end while
19:                 src_ref.write(0)
20:                 return
21:             else
22:                 continue
23:             end if
24:         else if trg_ptr.is_nod() or trg_ptr.is_tmp() then
25:             fst_ref, snd_ref ← sort(src_ref, trg_ref)
26:             fst_ptr ← fst_ref.exchange(TMP)
27:             if not fst_ptr.is_tmp() then
28:                 snd_ptr ← snd_ref.exchange(TMP)
29:                 add_redex(fst_ptr, snd_ptr)
30:                 return
31:             else
32:                 fst_ref.write(0)
33:                 while snd_ref.compare_exchange(snd_ptr, 0) ≠ snd_ptr do
34:                     continue
35:                 end while
36:                 return
37:             end if
38:         else
39:             continue
40:         end if
41:     end while
42: end procedure
```

Algorithm 2 The rewrite() function

```
1: procedure REWRITE(net)
2:     parallel for (a, b) ∈ net.redexes
3:     quarters ← [(a, b, 1), (a, b, 2), (b, a, 1), (b, a, 2)]
4:     parallel for (a, b, port) ∈ quarters
5:     match(a_ptr.tag(), b_ptr.tag()) :
6:             (CON, CON) ⇒ con_con(a, b, port)
7:             (CON, DUP) ⇒ con_dup(a, b, port)
8:             (CON, ERA) ⇒ con_era(a, b, port)
9:             (DUP, CON) ⇒ dup_con(b, a, port)
10:            (DUP, DUP) ⇒ dup_dup(b, b, port)
11:            (DUP, ERA) ⇒ dup_era(a, b, port)
12:            (ERA, CON) ⇒ era_con(b, a, port)
13:            (ERA, DUP) ⇒ era_dup(b, b, port)
14:            (ERA, ERA) ⇒ era_era(a, b, port)
15: end procedure
```

Algorithm 3 The con_con() function

```
1: procedure CON_CON(a_ptr : &Pointer, b_ptr : &Pointer, port : Port)
2:     a_aux_ref ← &a_ptr.port[port]
3:     b_aux_ref ← &b_ptr.port[port]
4:     a_aux_ptr ← a_aux_ref.take()
5:     b_aux_ref.replace(TAKEN, a_aux_ptr.as_redirection())
6:     if a_aux_ptr.is_main() then
7:         link(b_aux_ref, a_ptr.redirect(port))
8:     end if
9: end procedure
```

Algorithm 4 The con_dup() function

```
1:  procedure CON_DUP(a_ptr : &Pointer, b_ptr : &Pointer, port : Port)
2:      a_aux_ref ← &a_ptr.port[port]
3:      b_aux_ref ← &b_ptr.port[port]
4:      a_aux_ptr ← a_aux_ref.take()
5:      clone_loc ← allocate_location()
6:      clone_ptr ← mkptr(tag(a_ptr), clone_loc)
7:      clone_ptr.fill_inner_wires()
8:      b_aux_ref.replace(TAKEN, clone_ptr)
9:      if a_aux_ptr.is_main() then
10:         link(a_aux_ref, a_aux_ptr.as_redirection())
11:     end if
12:     if a_aux_ptr.is_main() then
13:         create_redex(a_aux_ptr, a_aux_ref.take())
14:         a_aux_ref.compare_exchange(TAKEN, 0)
15:     end if
16: end procedure
```