Distributed Pairing

Alessandro Monticelli, Jóhannes Helgi Tómasson, Jonas Stahl April 3, 2026

1. Description of the Solution

To solve the Distributed Pairing problem, we implement an iterative, priority-based, greedy algorithm. The basic idea behind the developed algorithm is to create a local hierarchy of nodes inside a group of neighbouring nodes.

Using asyncrhronous messaging through buffered channels, we avoid the use of shared memory.
The possible messages in our system are defined as:

1. **`PROPOSE`:** The sender is asking to pair with the receiver
2. **`ACCEPT`:** The sender is accepting the proposal of the receiver, forming a pair
3. **`MATCHED`:** The sender is communicating it has formed a pair with another node.

Each node has its own state machine, that acts as follows:

1. Each `Node` is initialized as `SINGLE` (i.e. its `pair` variable is initialized to the node's ID.), and with a list of its *neighbors*(Listing 1 - lines 9-13).
2. At each iteration the `Node` evaulates its position relative to its neighbors(Listing 5 - lines 20-24)
   - If it has the highest `ID`, it assumes the role of `proposer`
   - Otherwise, it acts as a `listener`
3. *Proposer:* Sends a `PROPOSE` message to its highest-ID neighbor, and waits for a response (Either an `ACCEPT` or a `MATCHED` response)(Listing 3).
4. *Listener:* Waits for incoming messages, greedily accepting the first incoming proposal, as it is guaranteed to come from a higher-priority node(Listing 4).
5. Once a pair is formed, both the nodes set their `pair` variable to the respective IDs, and broadcast a `MATCHED` message to all their active neighbors and terminate(Listings 3 - line 12; 4 - line 8). If a node receives a `MATCHED` message from a neighbor, it will remove the ID of the sender from its `neighbors` list(Listings 3 - line 15, 4 - line 9).
6. If a node's `neighbors` list is empty, it is marked as `SINGLE` and terminates(Listing 5 - lines 4-7).

2. Proposed Pseudocode

Listing 1: Distributed Pairing - Node initialization

```
func InitNode(id, neighbors[], inbox, network) {
    neighborSet := make(map[int]bool)

    for _, n := range neighbors {
        neighborSet[n] = true
    }

    return &Node{
        ID:        id,
        Inbox:     inbox,
        Network:   network,
        neighbors: neighborSet,
        pair:      id,
    }
}
```

Listing 2: Distributed Pairing - Pair finalization

```
func (n *Node) finalize(partner_id) {
    // Save partner id in pair.
    n.pair = partner_id

    // Notify all the other neighbors of the new pair.
    for nid := range n.neighbors {
        if nid != partner_id {
            n.send(nid, MATCHED)
        }
    }
}
```

Listing 3: Distributed Pairing - Proposer

```
func (n *Node) propose(target_id) {
    n.send(target_id, PROPOSE)
    // Wait for a response to the proposal
    waiting := true
    while waiting {
        msg := <-n.Inbox
        switch msg.Type {
        case ACCEPT:
            if msg.Sender == target_id {
                // Target has accepted our proposal. Yay!
                n.finalize(target_id)
                return
            }
        case MATCHED:
            // Remove the neighbor from our list, it has already matched.
            delete(n.neighbors, msg.Sender)
            if msg.Sender == target_id {
                // Exit waiting loop and re-evaluate who is the local max
                waiting = false
            }
        }
    }
}
```

Listing 4: Distributed Pairing - Listener

```
func (n *Node) listen() {
    msg := <-n.Inbox

    switch msg.Type {
    case PROPOSE:
        // We greedily accept any proposal that comes.
        n.send(msg.Sender, ACCEPT)
        n.finalize(msg.Sender)
    case MATCHED:
        // The sender has been already matched, so we delete it.
        delete(n.neighbors, msg.Sender)
    }
}
```

Listing 5: Distributed Pairing - Make pairs

```
1  func (n *Node) makePairs() {
2
3      for n.pair == n.ID {
4          if len(n.neighbors) == 0 {
5              // SINGLE node.
6              return
7          }
8
9          // We need to find out if we have the highest ID to take priority.
10         maxNeighborID := -1
11
12         for id := range n.neighbors {
13             if id > maxNeighborID {
14                 maxNeighborID = id
15             }
16         }
17
18         haveMaxId := n.ID > maxNeighborID
19
20         if haveMaxId {
21             n.propose(maxNeighborID)
22         } else {
23             n.listen()
24         }
25     }
26 }
```

3. Argument of Correctness

When the algorithm terminates, there are no adjacent nodes marked as `SINGLE`. This partial correctness can be proven by analyzing the precise conditions under which a node definitively terminates in the `SINGLE` state.

Upon initialization, every node u assumes the `SINGLE` state by default, setting `u.pair = u.ID`. The process continues to execute its main loop as long as this state remains unchanged (i.e., `for n.pair == n.ID` (Listing 5 line 3)).

A node u definitively terminates as `SINGLE` only if it satisfies the condition: `len(n.neighbors) == 0` (Listing 5 line 4). When this occurs, the node executes a `return` statement and halts (Listing 5 line 6). Crucially, because it halts immediately without invoking the `finalize` routine, a node terminating as `SINGLE` never broadcasts a `MATCHED` message to the network.

Let's assume that two initially adjacent nodes u and v both terminate as `SINGLE`. For node u to terminate as `SINGLE`, its `neighbors` map must have become empty. Since the graph is undirected, v was present in u's map at initialization. Therefore, v must have been explicitly

removed from u's map during execution.

According to the algorithm's event handling, a neighbor is deleted from the map if and only if a `MATCHED` message is received from that specific neighbor (Listing 4 line 11). Thus, u must have received a `MATCHED` message from v.However, as established, if v terminates as `SINGLE`, it halts without ever sending a `MATCHED` message. For v to have sent a `MATCHED` message to u, v must have successfully paired with some other node w (where $w \neq v$) and executed the `finalize(w)` routine.

If v paired with w, its local state was updated such that `v.pair = w` $\implies$ `v.pair` $\neq$ `v.ID`. Consequently, v did not terminate as `SINGLE` and exited its `makePairs` loop (Listing 5 line 3). This fundamentally contradicts our initial assumption. Therefore, it is impossible for two adjacent nodes to both evaluate empty neighbor maps and terminate as `SINGLE`.

4. Termination

The main loop of the algorithm runs as long as a node remains unpaired (i.e., `for n.pair == n.ID` (Listing 5 line 3)). At each iteration, the `Node` evaluates its position relative to its active neighbors. Because the graph is finite and all nodes have strictly unique IDs, there is guaranteed to be at least one global maximum ID within any active connected component.

If a node has the highest `ID` among its neighbors, it assumes the role of proposer (Listing 5 lines 20-21). This proposer sends a `PROPOSE` message to its highest-ID neighbor, and waits for a response (Listing 3 line 2). Since the receiving neighbor has a lower ID than the proposer, it cannot be a local maximum, meaning it must be into the listener state (Listing 5 line 23). A listener waits for incoming messages, greedily accepting the first incoming proposal (Listing 4 lines 6-7). Once a pair is formed, both the nodes set their `pair` variable to the respective IDs, broadcast a `MATCHED` message to all active neighbors, and terminate (Listings 3 line 12; 4 line 8).

Because at least one global maximum node exists in any non-empty active graph, at least one pair is guaranteed to form in every iteration. This strictly decreases the number of active nodes in the network. Eventually, all nodes will either pair up or find themselves with an empty `neighbors` list, mark themselves as `SINGLE` (Listing 5 lines 4-6), and terminate. Thus, the algorithm strictly avoids infinite loops.

5. Cooperation

For the algorithm to be cooperative, we must guarantee that no node enters a state where it waits indefinitely for messages that are never sent. This can be demonstrated by analyzing the waiting states:

- The proposer waits for either an `ACCEPT` or a `MATCHED` response (Listing 3 lines 4-6). It waits specifically for a response from its `target_id`. Because the proposer is a local maximum, its ID is strictly greater than `target_id`. Consequently, `target_id` cannot be a local maximum and must be executing the `listen()` routine. A listener greedily accepts the first incoming

proposal, as it is guaranteed to come from a higher-priority node (Listing 4 line 6). If the listener had already processed a prior proposal, it would have broadcasted a `MATCHED` message. In either scenario, a message (`ACCEPT` or `MATCHED`) is definitively sent back to the proposer, breaking its waiting loop.

- A listener blocks on `<-n.Inbox` waiting for any message (Listing 4 line 2). Because a listener is not a local maximum, it inherently points towards a neighbor with a higher ID. Following this chain guarantees reaching a local maximum node, which is actively proposing. As established in the termination argument, the active network strictly shrinks as pairs form. The listener will inevitably receive either a `PROPOSE` from a higher-priority node, or `MATCHED` messages from its neighbors until its map is empty.

Finally, note that a proposer only handles `ACCEPT` and `MATCHED` messages (Listing 3 lines 7-22). This is safe and cooperative because a proposer holds a local maximum ID; therefore, no active neighbor can view it as a valid proposal target. Consequently, a proposer will never be sent a `PROPOSE` message while waiting, eliminating the risk of unhandled message deadlocks.