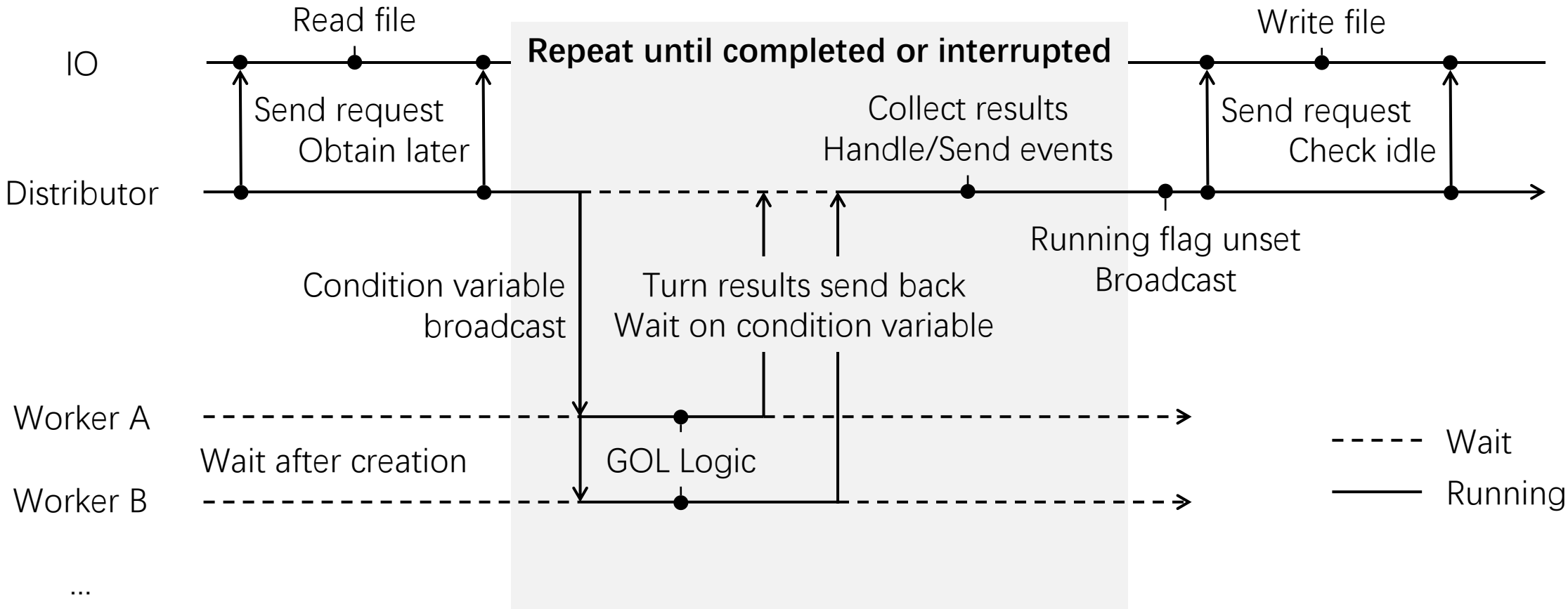
Game of Life VIVA

Group Members: Zik Zhao, Zixuan Zhu, Chenxing Liu

Parallel: Evaluation Steps



Parallel: Divide Matrix Into Blocks

We allocate works for each worker by dividing matrix into evenly sized blocks instead of strips.

This is implemented as the `divideIntoBlocks` function in distributor.go. This function returns a slice of an anonymous structure, which contains two positions as fields, one for the top-left corner of block and one for bottom-right corner (not inclusive) of block.

This function has a limitation that it does not accept prime number of threads, prime number will be floored to nearest composite number.

Parallel: Dispatch Tasks

All worker goroutines are created before the evaluation loop:

1. Status variables and synchronisation primitives are created.
2. Relevant information for each worker is stored in structure called `WorkerParams`.
3. Workers are created with said structures passed in.
4. Workers immediately acquires locker of condition variable, send an empty result to channel to notify the distributor that this worker is ready, and then wait on condition variable.
5. Distributor read the empty result from channel.

Parallel: Matrix Structure and Counts of Surrounding Alive Cells

We have a `Matrix` structure storing pixel values of the image, its width and height, and an extra 2D slice storing the counts of surrounding alive cells for every cell in the matrix.

The `Matrix` structure provides three methods:

- `getSurrounding` is used to get the positions of eight surrounding cells for a specific cells.
- `checkAndFlip` and `checkAndFlipUnsafe` are two functions transforming status of cells in different ways.

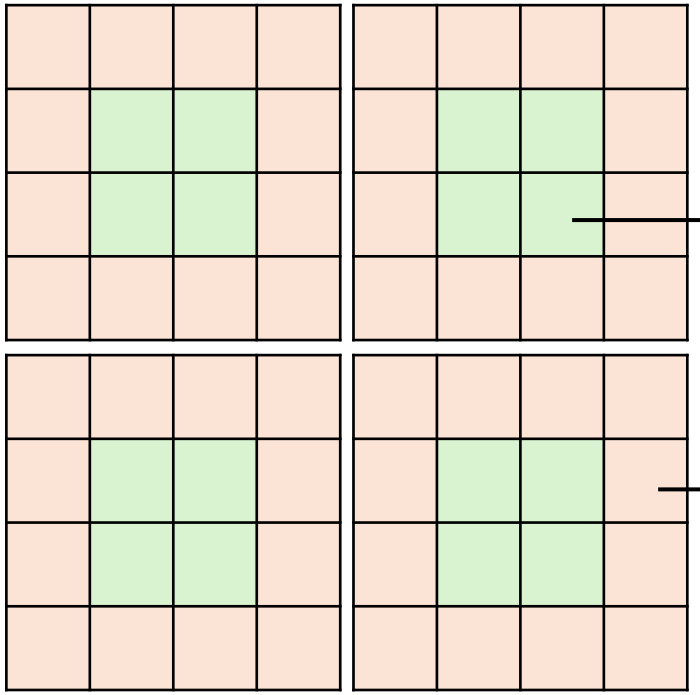
This surrounding count 2D slice is maintained during evaluation. Any flipping in cells causes the surrounding counts of its surrounding cells to change.

The introduction of surrounding counts reduces workload but produces data races.

Parallel: Solving Data Races

Data races are produced at the boundaries of blocks since the flipping of cells at boundaries need to update surrounding counts for surrounding cells that are living in other blocks.

Thus, we call different methods for different part of blocks:



Example: 8x8 grid, 4 threads

Green parts are safe regions → **`checkAndFlip`**
Because all its surrounding cells are in the same block
Will update surrounding counts immediately

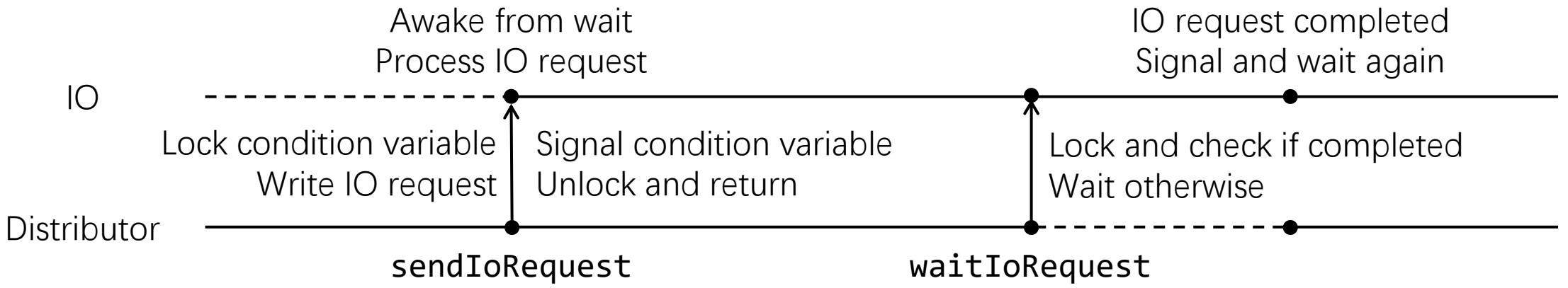
Red parts are unsafe boundaries → **`checkAndFlipUnsafe`**
Because at least one of its surrounding cells is not in the same block
Will not update but record these cells

Parallel: Refactored IO

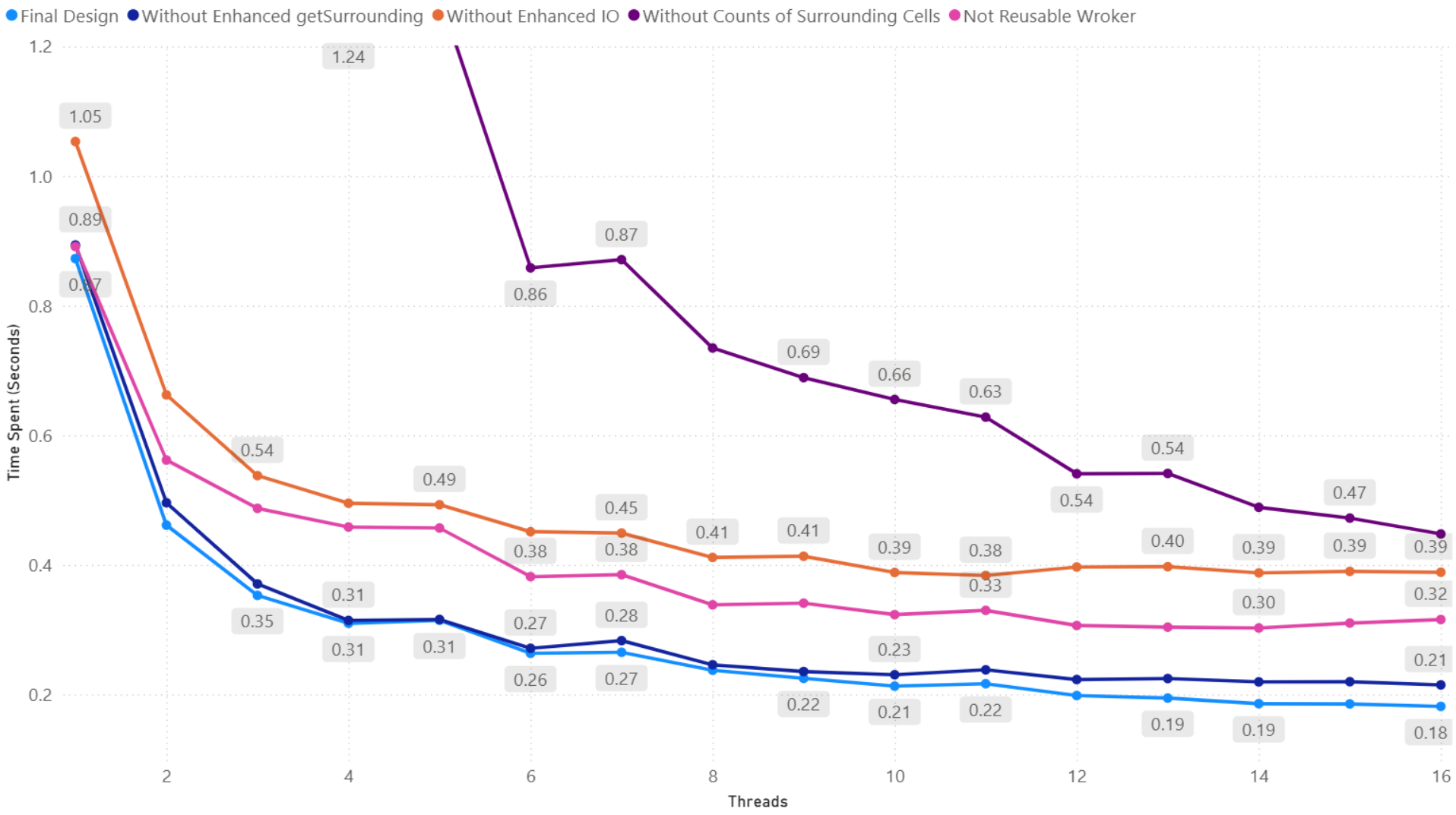
The refactored IO component provides two more functions for distributor: `sendIoRequest` and `waitIoRequest`.

The distributor can initiate an IO request by calling the former one and the IO component will complete that request in background. The distributor can then gather data or check idle later by calling `waitIoRequest` which blocks until last request is completed.

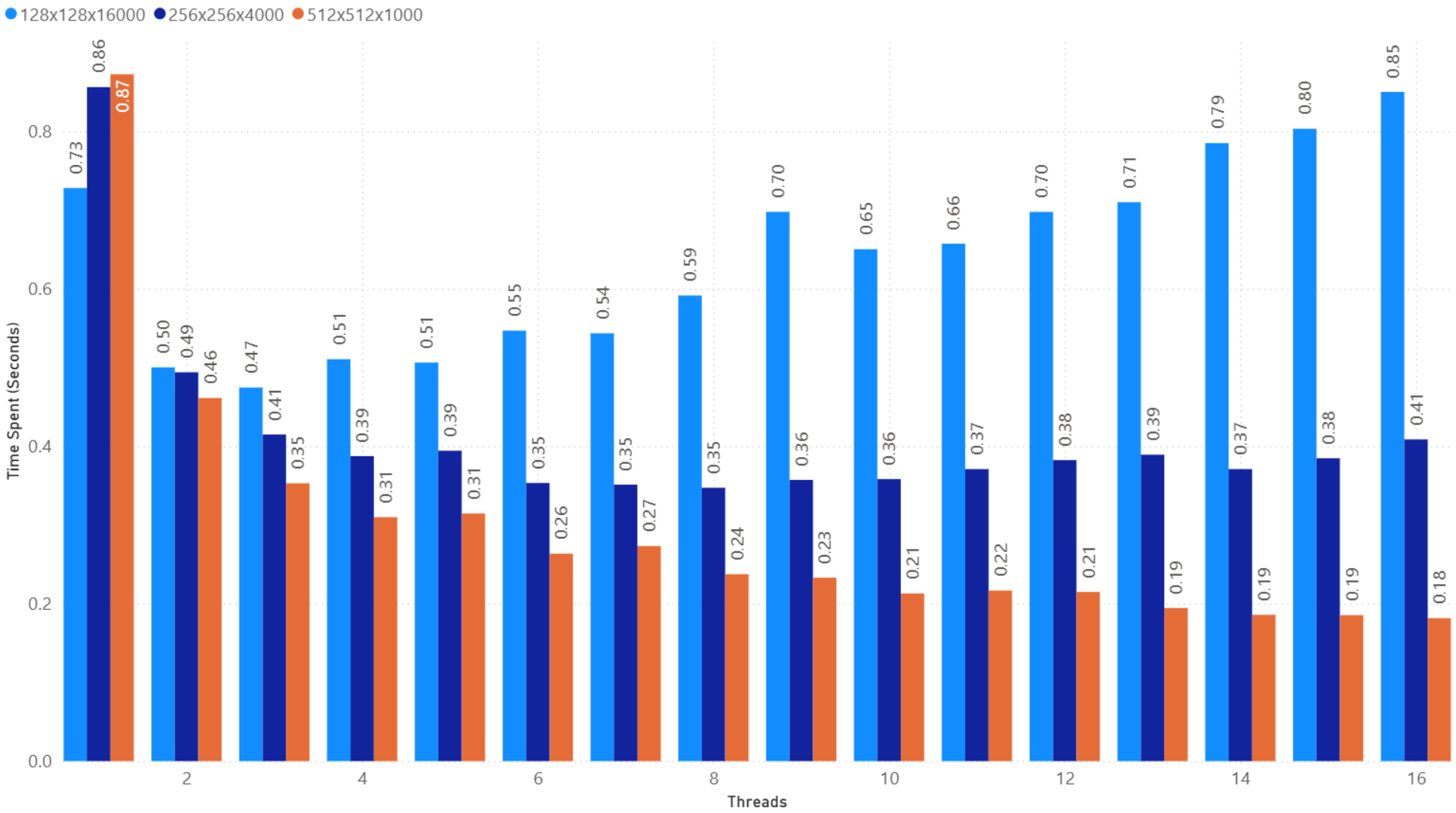
An IO request is represented by the `ioOperation` structure. It becomes a field of `ioState`. The condition variable in `ioState` is used to synchronise the access to this field.



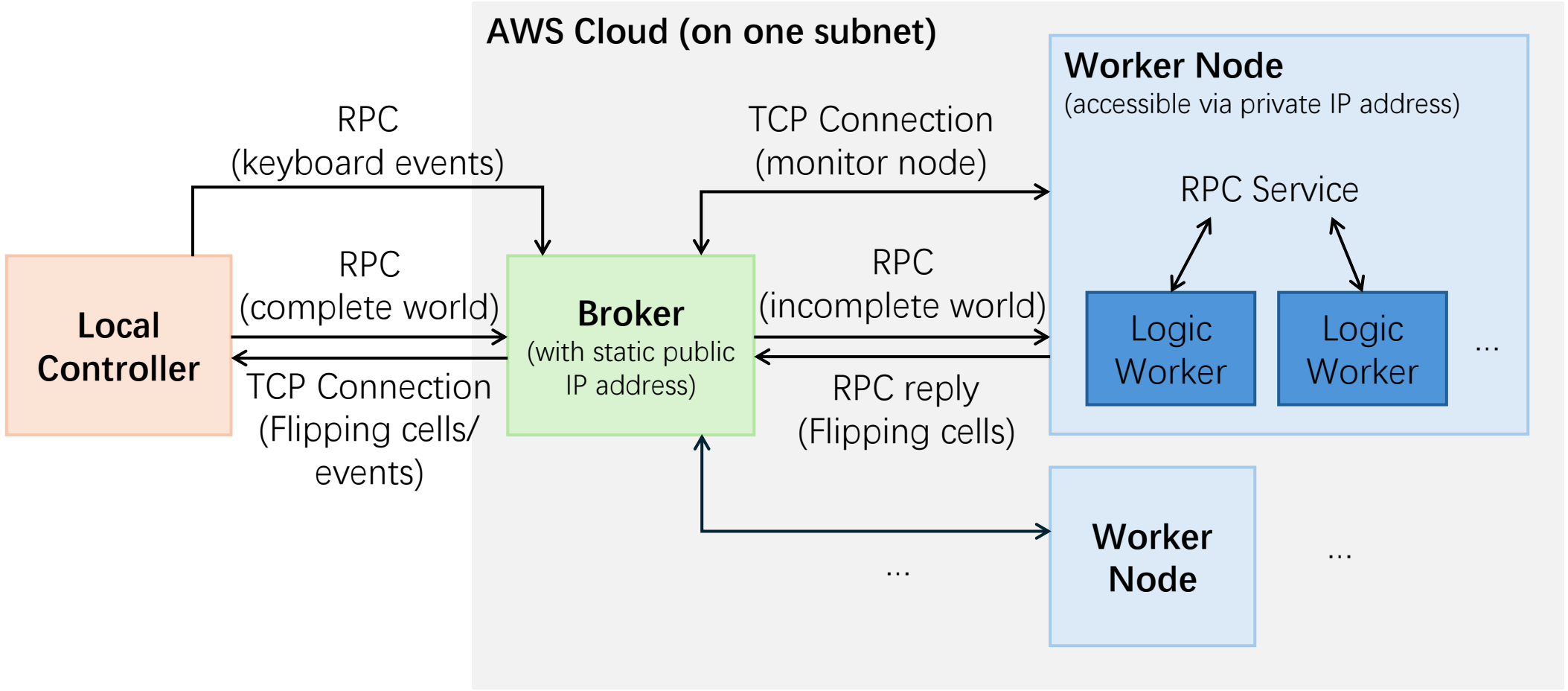
Parallel: Benchmarks



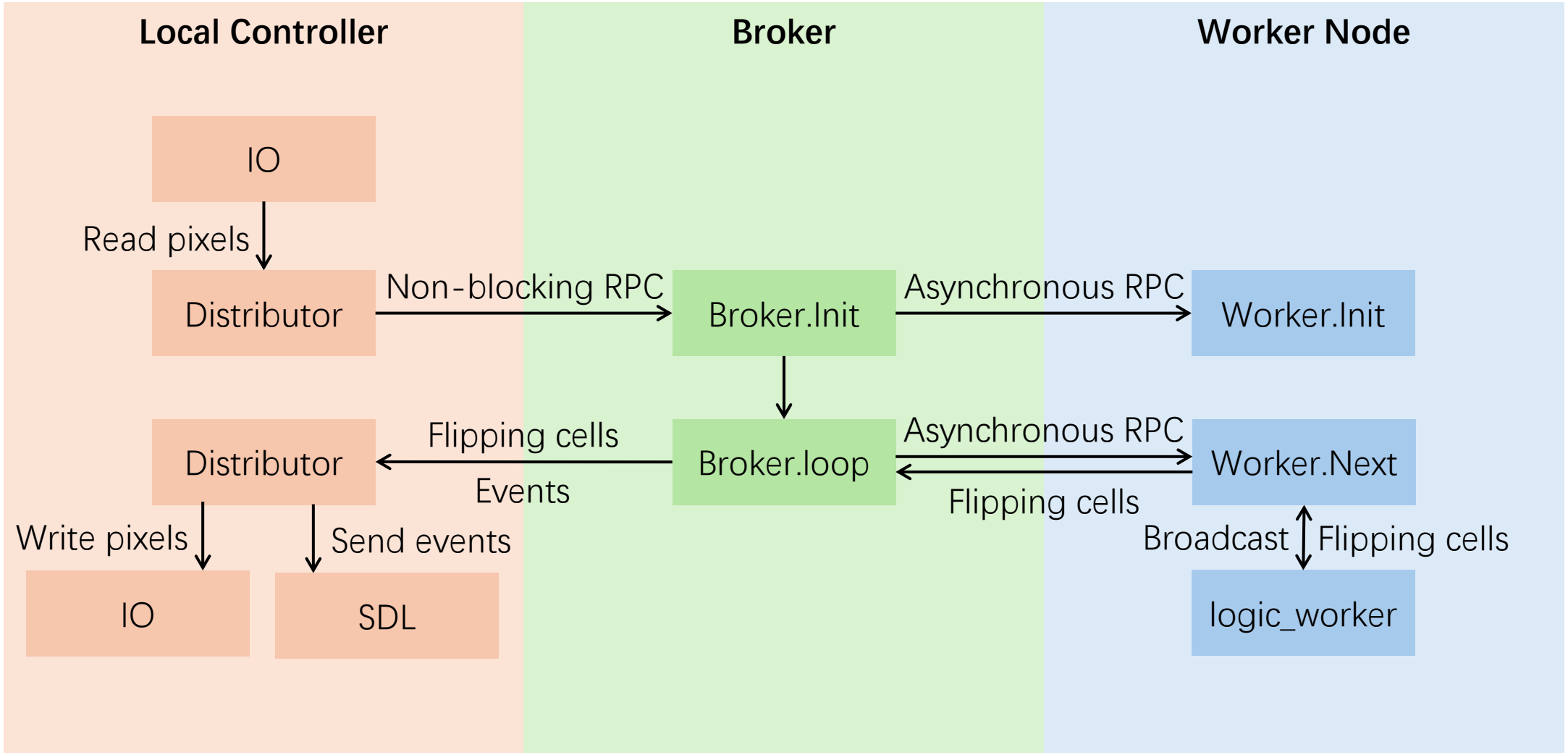
Parallel: Benchmarks



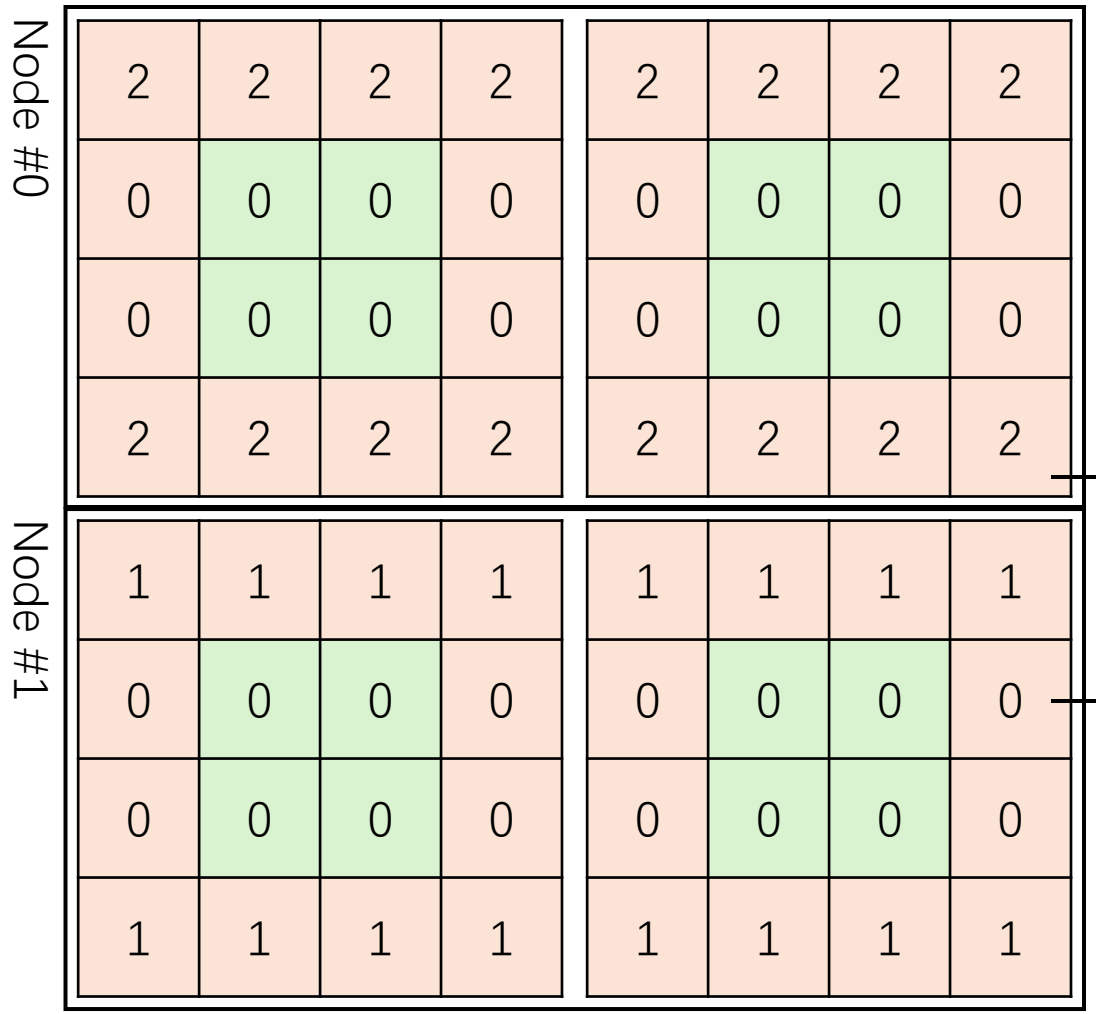
Distributed: Architecture



Distributed: Evaluation Steps



Distributed: Exchange Graph



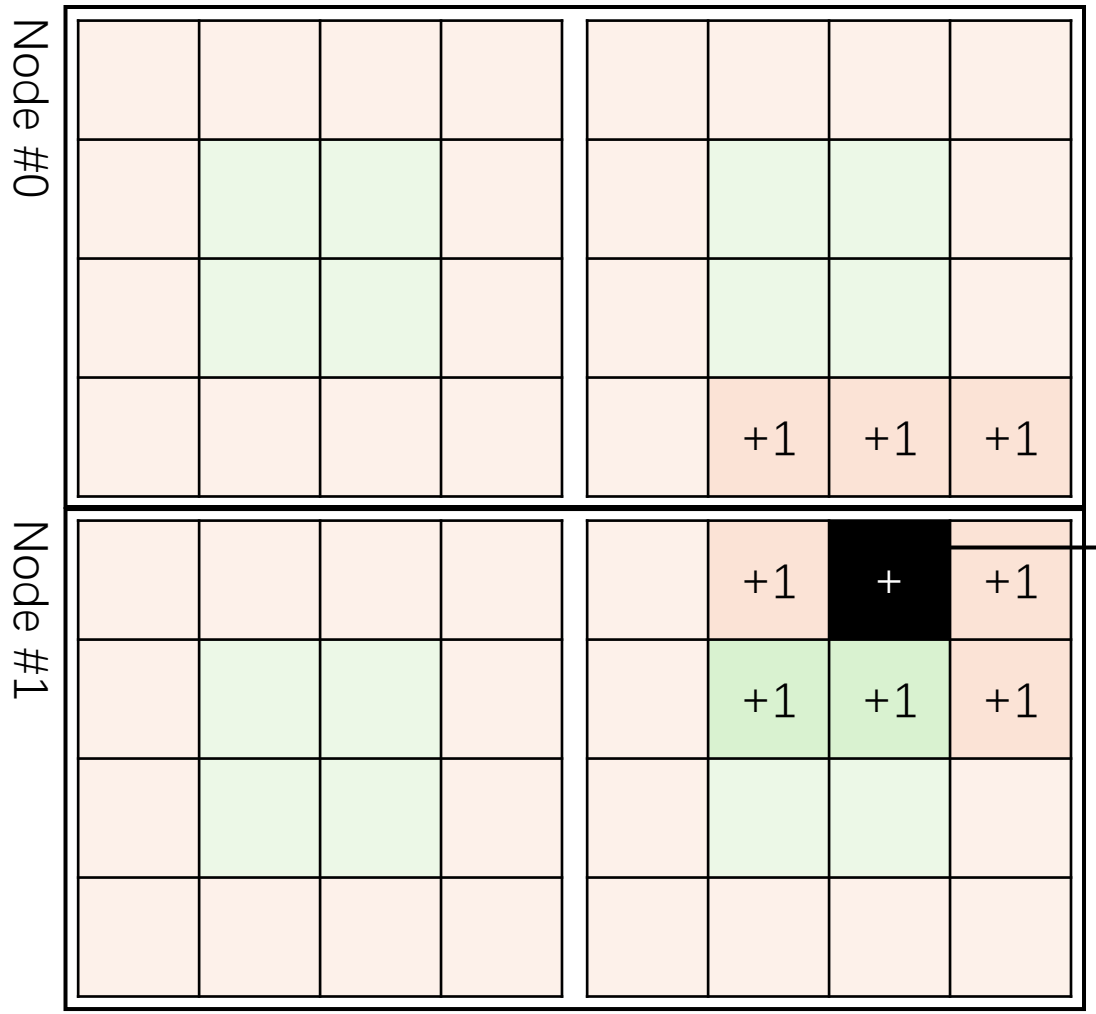
Example: 8x8 grid, 4 threads, 2 worker nodes

Numbers in graph are neither pixel values nor counts of surrounding alive cells. They are indicating exchange targets. For example, 6 is 0b00000110, which means flipping event of this cell is sent to node #1 and node #2.

2 means exchange target is node #1
Because three of its surrounding cells live in partition assigned to node #1

0 means this cell is excluded for exchanging
Because all its surrounding cells are in the same partition (although they may live in different blocks)

Distributed: Adjustments



Example: 8x8 grid, 4 threads, 2 worker nodes

Each worker node cannot access pixels outside assigned partition. The only way they know what's happening outside the partition is adjustment. It contains slices of cells flipped that affects this partition.

When this cell flipped

Node #1 first update surrounding counts in partition, the broker send this cell as an adjustment to Node #0 so its surrounding cells in partition of Node #0 will be updated.

Distributed: Keyboard Events

All keyboard events are sent to the broker, and the broker sends them back to local controller to make sure broker is in the same state as local controller.

Events are sent via RPC, all these exposed functions push events into event channel, which then be read by the main loop in broker.

Save (s) -> `broker.Save`

Pause (p) -> `broker.Pause`

Resume (p) -> `broker.Resume`

Quit (q) -> `broker.Quit`

Kill (k) -> `broker.Kill`

Distributed: Compression

All data transmission in the whole system is compressed, including RPC and TCP connection.

The initial matrix can be compressed in two ways, and slice of cells flipped can be compressed in the second way:

- Pack 8 pixels into one byte since each pixel only takes two values.
- Compress each integer so that each integer only occupies the minimum number of bytes to represent the range of width and height.

The minimum number of bytes is calculated by a helper function in compression.go, namely `getSizeInt`.

Distributed: Scalability

Currently our system supports up to 8 worker nodes (limited by the data type of exchange graph).

Each worker node launched on the same subnet will automatically try to register itself to the broker. The broker keeps a set of available workers so it will be retrieved for task allocations in the following tasks.

The private IP address of worker is retrieved from TCP connection object to establish RPC service.

Distributed: Data Streaming

To enable SDL live view, results for each turn are sent back via TCP connection.

1. TCP connection is established from local controller before initialising task in the broker.
2. All flipping cells events sent by worker nodes to broker are sent to local controller.
3. There may be several flipping cells event received by local controller in each turn.
4. Each turn ended with a turn completion event.

TCP connection is not reusable for following tasks since there may be unread data for last task but read by next task.

Distributed: Data Encoding

Both cells flipping events and keyboard events are sent through the same connection.

We specify how such events are encoded into messages:

- The first byte is always event type enumeration.
- If the event is flipping cells event, then the following 8 bytes represents the length of slice of cells flipped, in bytes. And the remaining part is compressed slice of cells flipped, using said compression method.

Distributed: Fault Tolerance

We have considered the following failures:

1. Unexpected connection lost between local controller and broker

   Local controller will panic when trying to read from connection, the remote system stops evaluation and turn to idle.

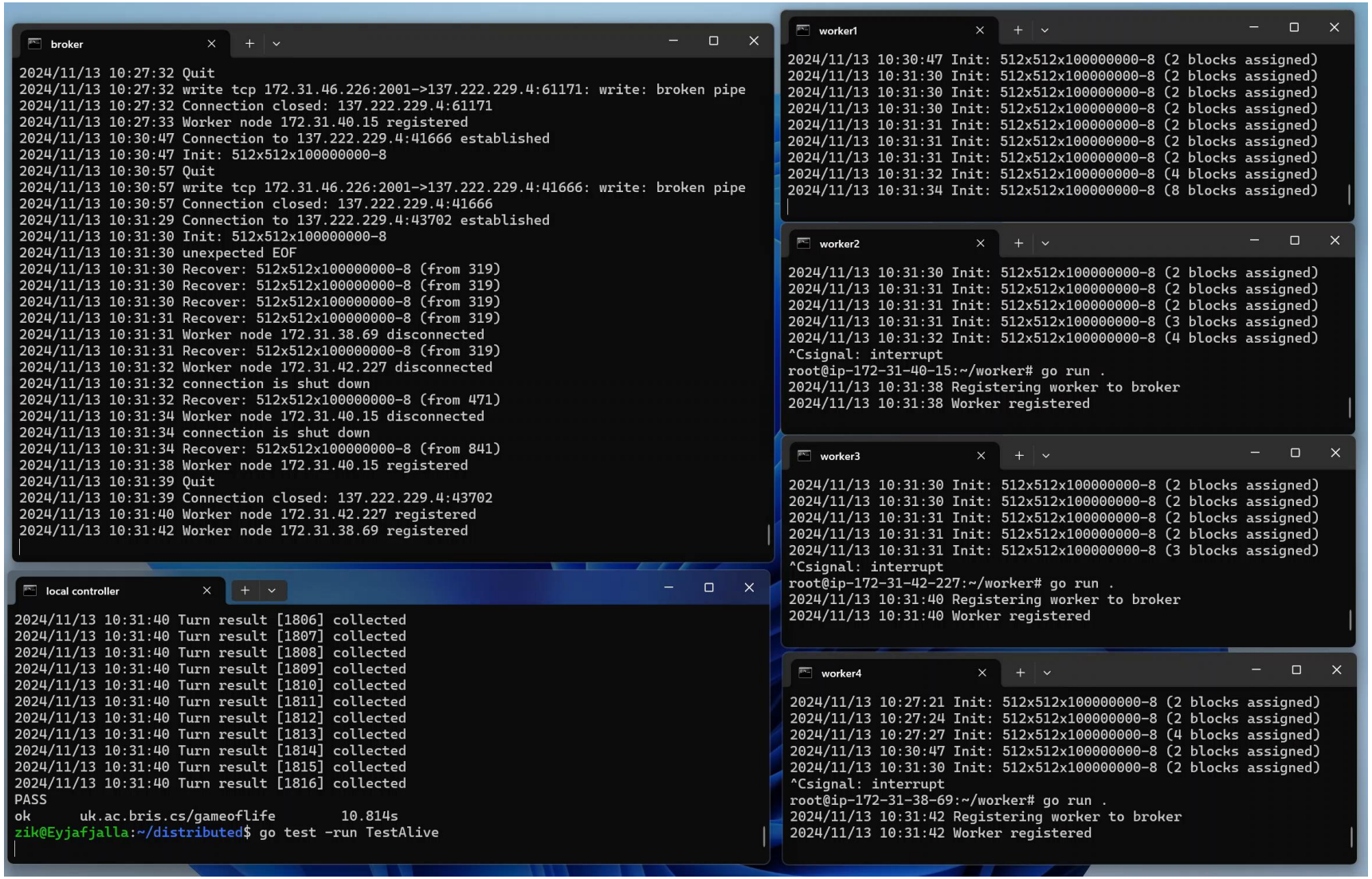
2. Unexpected connection lost between broker and worker node

   That worker node is removed from available node list. There is no effects to the system except that performance is reduced.

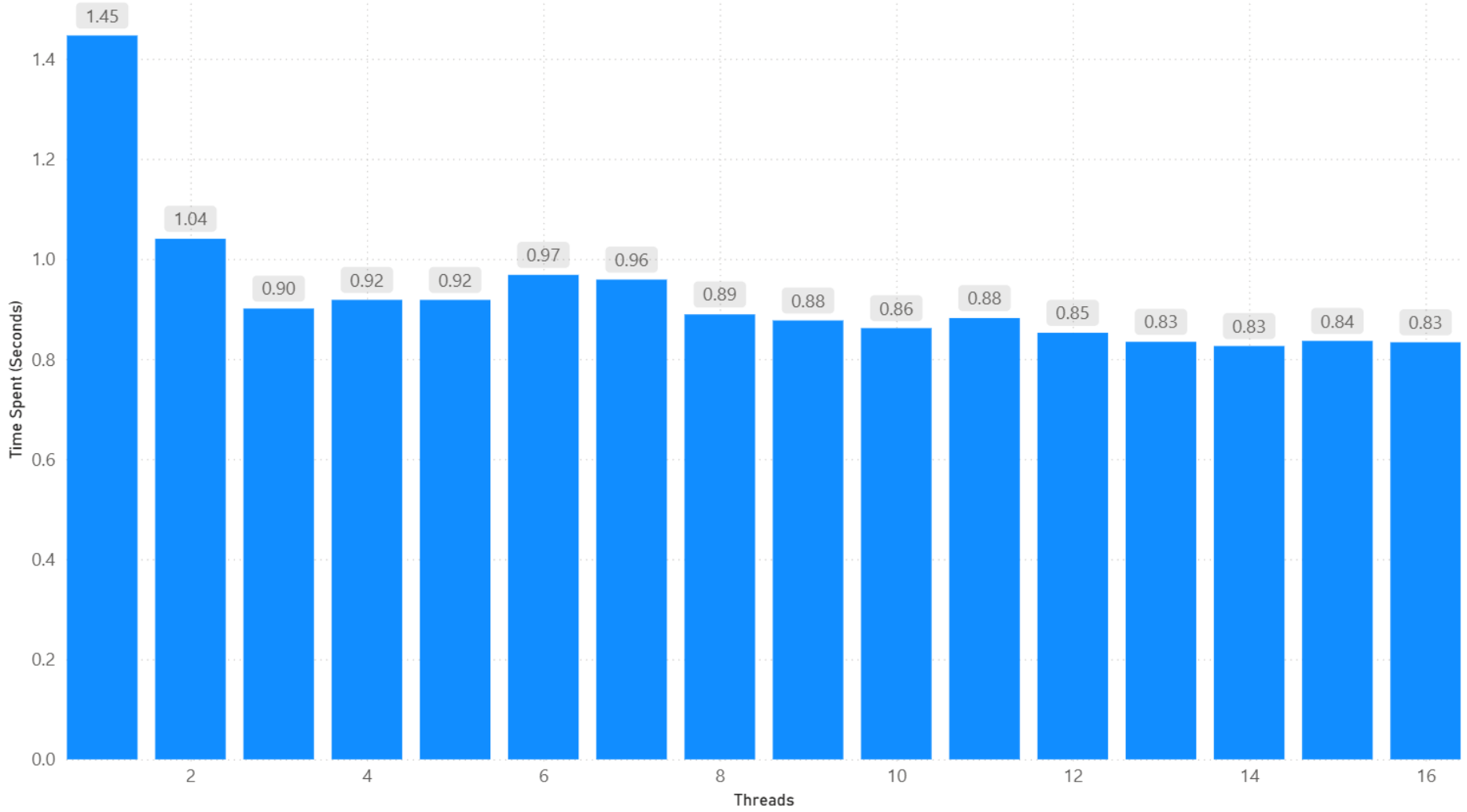
3. Unexpected connection lost between broker and worker node when a task is running

   All partial results for current turn is ignored. Current task is replaced by a new task from current turn to final turn, using available nodes. Original TCP connection is kept for the new task. No actions is required for local controller.

Distributed: Fault Tolerance



Distributed: Benchmarks



Distributed: Benchmarks