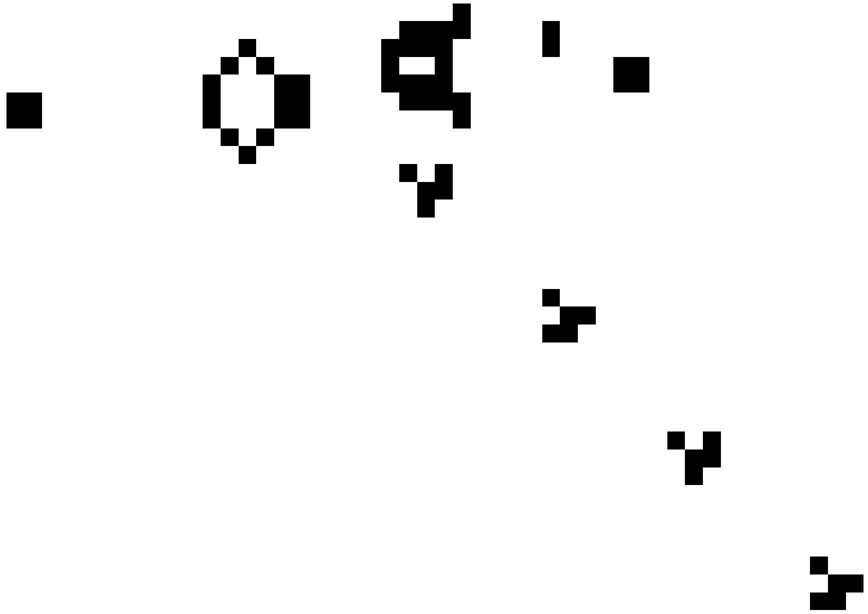
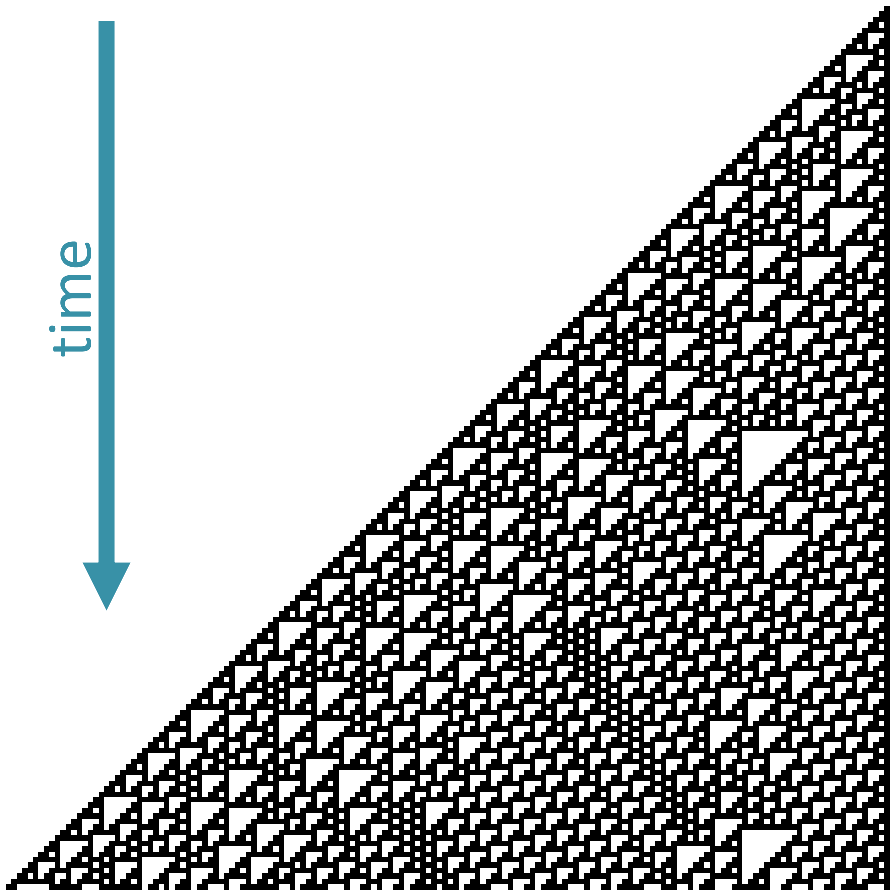
Assignment 3 - Stencil

Motivation: LIFE by John Conway



- The game of LIFE
  - 1970
  - A cellular automaton
    - 2D grid of cells
    - Two states: Dead, Alive
  - Each cell reproduces depending on its state and the state of its 8 neighbor cells
    - Born if exactly 3 neighbors alive
    - Survives if 2 or 3 neighbors alive
  - Turing complete!
    - Rendell, 2002
- Author: John Conway
  - Born Dec 26, 1937
  - Died Apr 11, 2020, COVID-19

Motivation: Rule 110 by Stephen Wolfram



- Rule 110
  - 1985
  - 1D grid of cells, two states
  - One of 256 rules possible for 1D-neighborhood two-state cells

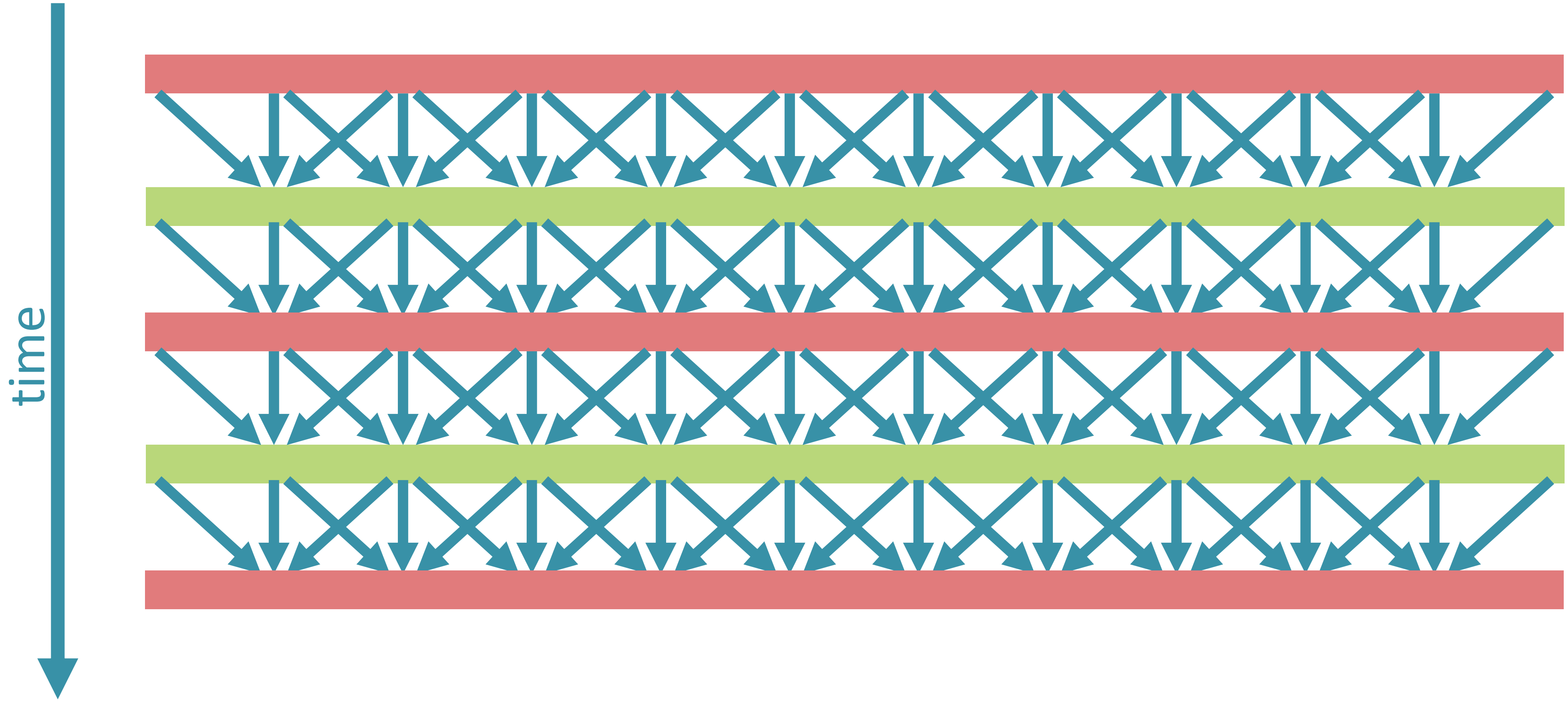
```
a[i] = (110>>(4*a[i-1]+2*a[i]+a[i+1]))&1;
```

  - Turing complete!
    - Cook, 2005
- Author: Stephen Wolfram
  - Born Aug 29, 1959
  - www.wolframalpha.com

Motivation

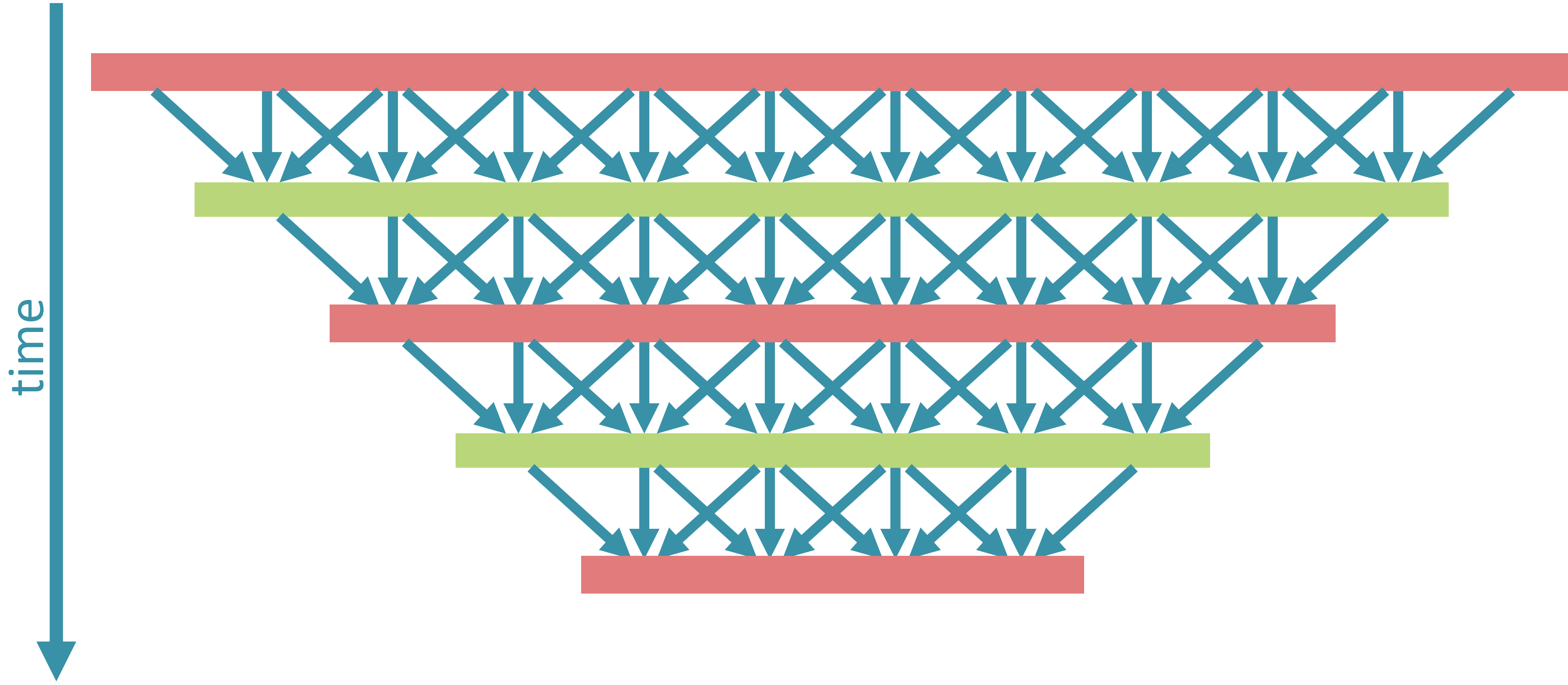
- Many mathematical models work as follows:
  - A (multi-dimensional) array of cells
    - Each cell has a state
  - A global clock
    - In each tick, each cell is updated based on the (previous) states of its neighbors
    - The update formula is usually independent of position and time
- Examples:
  - Cellular automata
    - Ulam & von Neumann, 1948
    - Finite number of states of each cell
    - 1D: Rule 110; 2D: LIFE
  - Stencils
    - Emmons, 1944
    - Numeric solution of partial differential equations (finite difference method)
      - Heat transfer, hydrodynamics, ...
    - State of each cell consists of several real or complex numbers, depending on the problem

Serial evaluation of a stencil



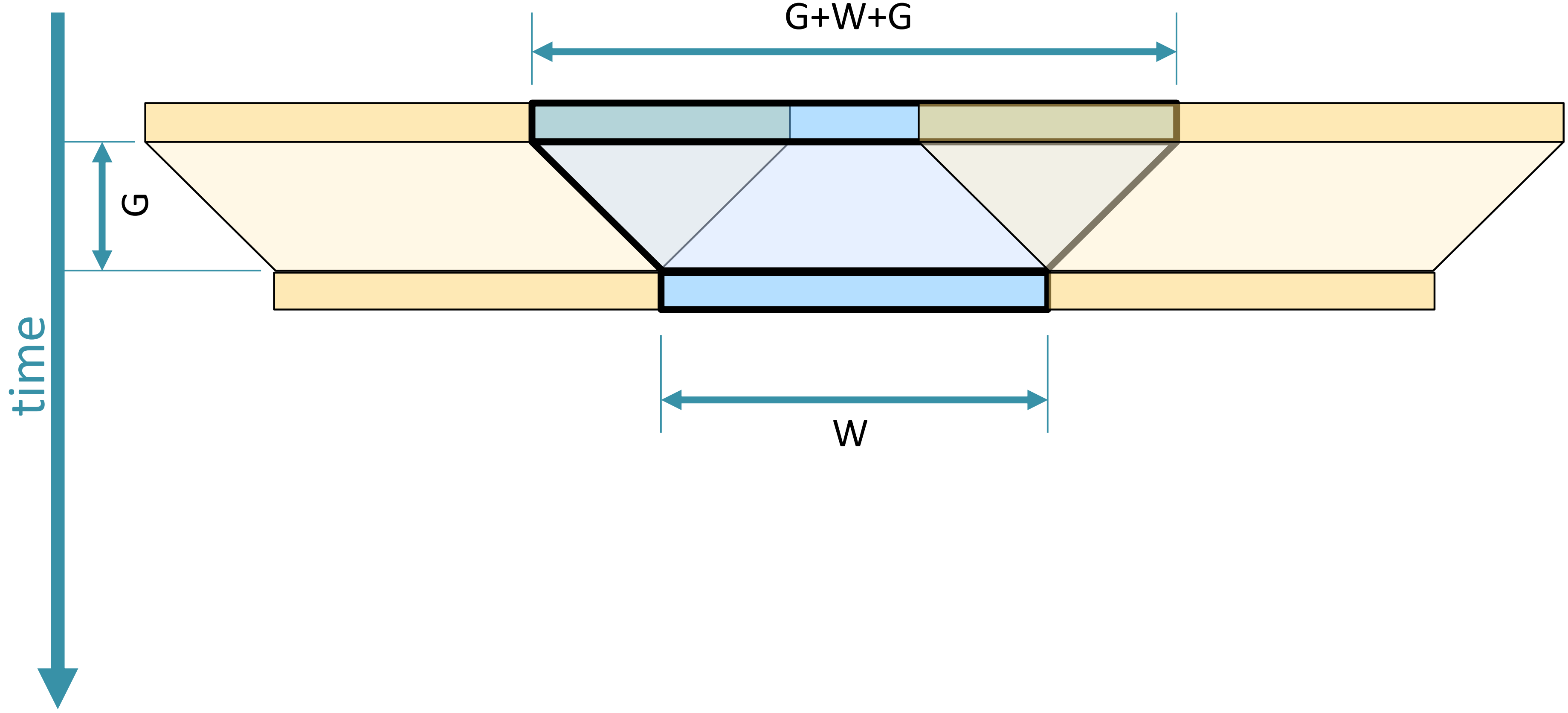
- Two alternating buffers required
  - Stencils propagate state in both directions
  - The new states must always be computed from the old states
- Boundary cells may require special handling
  - Wrapping around is the simplest (but usually not physically correct)

Evaluation of a part of the space



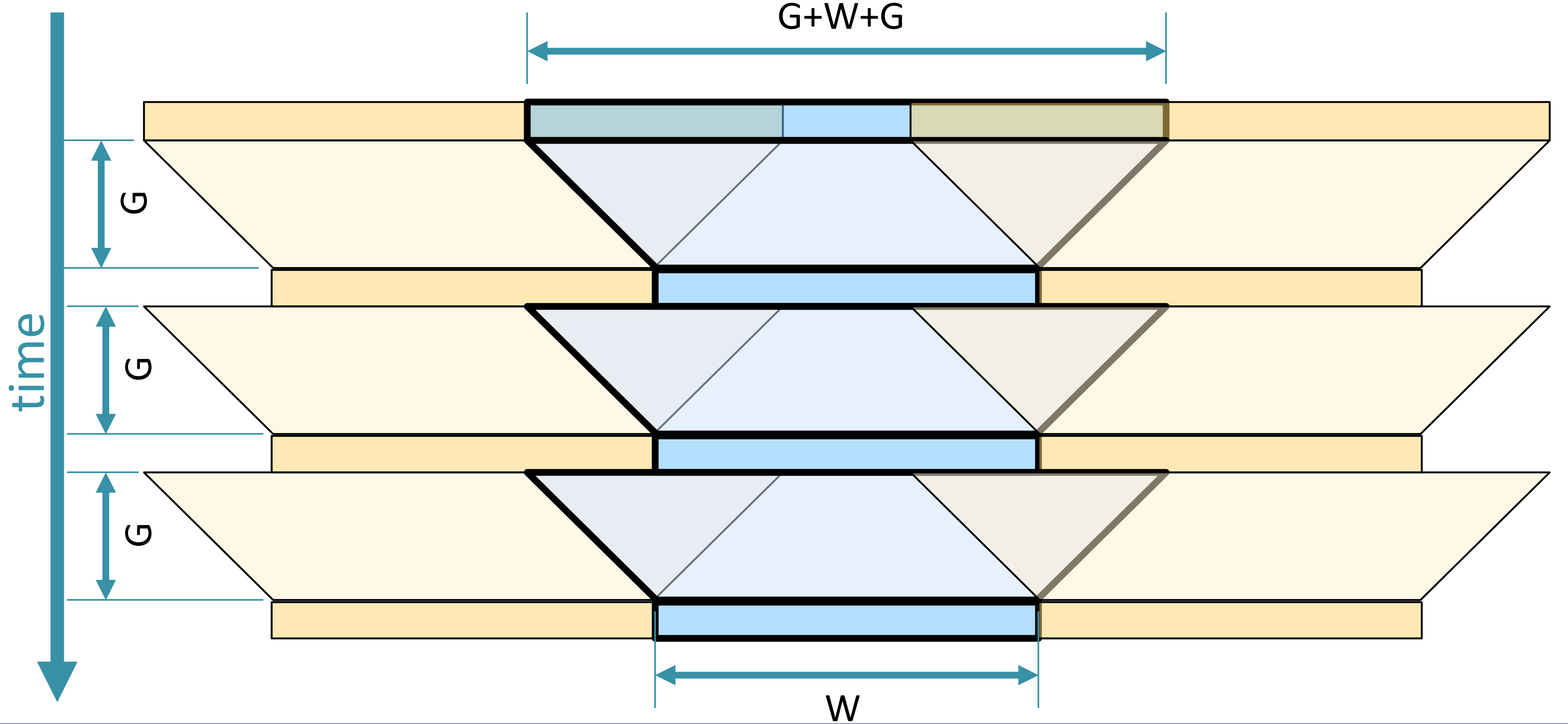
- If the state is not known outside a part of the space, the result is valid only for a subset of the space
  - Shrinking with time

Parallel evaluation of a stencil



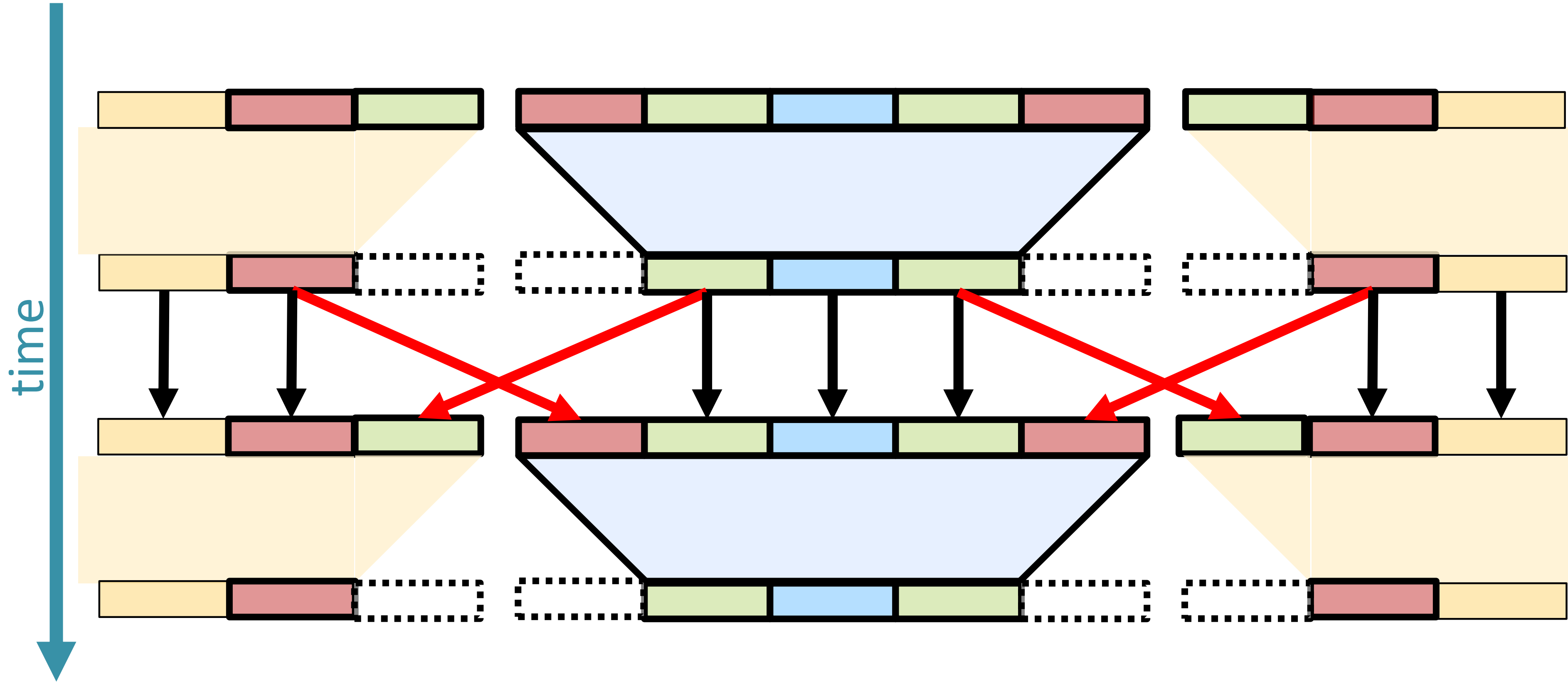
- A running thread cannot continuously observe the results produced by adjacent threads
  - Synchronization in every generation would be too expensive
- An individual task is to compute G generations, producing W results
  - The input to such a task is G+W+G wide
  - A part of the work is duplicated in adjacent tasks (threads)

Parallel evaluation of a stencil



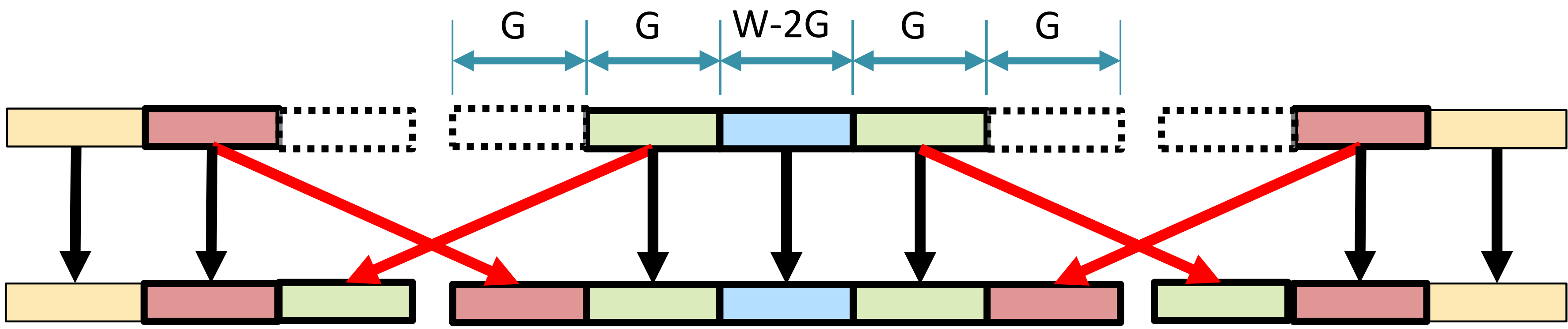
- An individual task is to compute G generations, producing W results
  - The input to such a task is G+W+G wide
- The same thread may continue with subsequent generations
  - Synchronization with adjacent threads is required each G generations

Data exchanged between threads



- The same thread may continue with subsequent generations
  - Synchronization with adjacent threads is required each G generations
  - Two buffers of size G are sent from each thread to its neighbors
  - Two buffers of size G are received by each thread from its neighbors

Data exchanged between threads



- Every G generations, synchronization is required
  - Before synchronization, each thread holds W valid elements
    - Plus two buffers of size G, containing out-of-date elements
  - Two buffers of size G must be copied from each thread to its neighbors
- Two possible approaches
  - The receiving thread copies the data from adjacent threads to its own empty buffers
    - The adjacent threads may copy in parallel but they must not start computing further generations
    - Synchronization (rendezvous) required both before and after copying!
  - The sending thread copies the data and then sends the data including ownership of the buffer
    - The out-of-date buffers may be used, provided they are separable
    - Only one synchronization required (waiting for the messages)

Data exchanged between threads

- Synchronization primitives in C++17
  - Starting and joining threads
    - Requires a master thread
    - Thread start is significantly slower than synchronization between running threads
  - Mutex
    - Not suitable for waiting for an event
  - Promise-future
    - Not reusable!
    - Additional thread-safe mechanism for dispatching promises (or futures) required
  - Condition variable
    - Difficult to understand, coupled with a mutex
    - Spurious triggers
    - Reusable, does not involve allocation on each use

Data exchanged between threads

- Emulating rendezvous (between two threads)
  - Corresponds to std::barrier in C++20
  - Could be emulated by a pair of std::binary_semaphore in C++20
    - But you only have C++17
  - With std::condition_variable:
    - Lock a mutex
    - Increment a shared counter
    - Notify a condition variable
    - While the counter is less than 2, wait for the condition variable
      - Waiting internally unlocks the mutex, allowing the other thread to increment
      - Spurious wakeups may occur – repeated testing of the counter is required
    - Unlock the mutex
    - BEWARE: A thread-safe way of resetting the counter is required
      - Resetting must appear after all the threads safely exited the rendezvous but before any of them enters the rendezvous again
      - In our case, we need a rendezvous both before and after the copying – resetting one of them may be done inside the other (if controlled by the same mutex)

Data exchanged between threads

- Emulating message passing
  - Easily emulated by a std::counting_semaphore in C++20
    - But you only have C++17
  - With std::condition_variable:
    - Send
      - With a mutex locked, push the message to a queue
      - Notify a condition variable
    - Receive
      - Lock the mutex
      - While the queue is empty, wait for the condition variable
      - Pop the message from the queue
      - Unlock the mutex
  - In our case
    - Each pair of adjacent threads needs two message channels
    - Each message channel may contain up to two messages
      - A general (dynamically allocating) queue is not needed

The assignment

- Your job is to implement:
  - A generic structure used to hold the states of cells
    - 1-dimensional, wrapped-around = circle
  - A generic method which executes a given stencil function
    - For a given number of generations
    - In parallel, using only the C++17 standard library
  - Submit into recodex as "stencil1d.hpp"
- The testing framework will apply two stencil functions:
  - Rule 110
    - State of a cell = bool
  - Lemmings
    - State of a cell is a structure containing some small numbers
- The framework will test correctness by printing some states
  - Using recodex to compare against the desired output
- Beware: Recodex will also apply some not-so-infinite time limits
- The framework and the desired outputs are available outside Recodex
  - //www.ksi.mff.cuni.cz/teaching/nprg051-web/DU/du-1920-3.framework.zip

The required interface

```
template< typename ET> class circle {
  public:

  circle(std::size_t s);

  std::size_t size() const;

  void set(std::ptrdiff_t x, const ET& v);

  REF get(std::ptrdiff_t x) const;

  template< typename SF>
  void run(
    SF&& sf,
    std::size_t g,
    std::size_t thrs =
     std::thread::hardware_concurrency());

};
```

- Class **circle<ET>**
  - **ET** is the type representing the status of one cell
  - The constructor allocates the space for the desired number of cells
    - also returned by **size**()
    - each cell initialized as ET()
  - The function **set/get** serve for writing/reading individual cells
    - The cells are circularly arranged – index overflows are handled by wrapping
    - The **set/get** function must support indexes (x) in the range <-size(), 2*size()-1>
    - BEWARE: the built-in operator% does not perform mathematical modulo for negative numbers
    - get may return by const reference but beware of std::vector<bool>

The required interface

```
template< typename ET> class circle {
  public:

  circle(std::size_t s);

  std::size_t size() const;

  void set(std::ptrdiff_t x, const ET& v);

  REF get(std::ptrdiff_t x) const;

  template< typename SF>
  void run(
    SF&& sf,
    std::size_t g,
    std::size_t thrs =
     std::thread::hardware_concurrency());

};
```

- The function **run** does the parallel evaluation of the stencil
  - The stencil is specified as a function/functor/lambda with the following interface:

```
ET sf(ET left, ET mine, ET right);
```

    - The arguments may also be passed by const reference; therefore, the run function shall avoid copying the cell states when calling sf
  - The run function performs **g** generations, modifying the contents of the underlying class
    - Any auxiliary resources like threads or work buffers shall be freed upon return
  - The **thrs** argument specifies the desired number of threads, the default is as shown
    - Other parameters, like suitable values of W and G, must be determined internally
    - Beware of non-divisibility