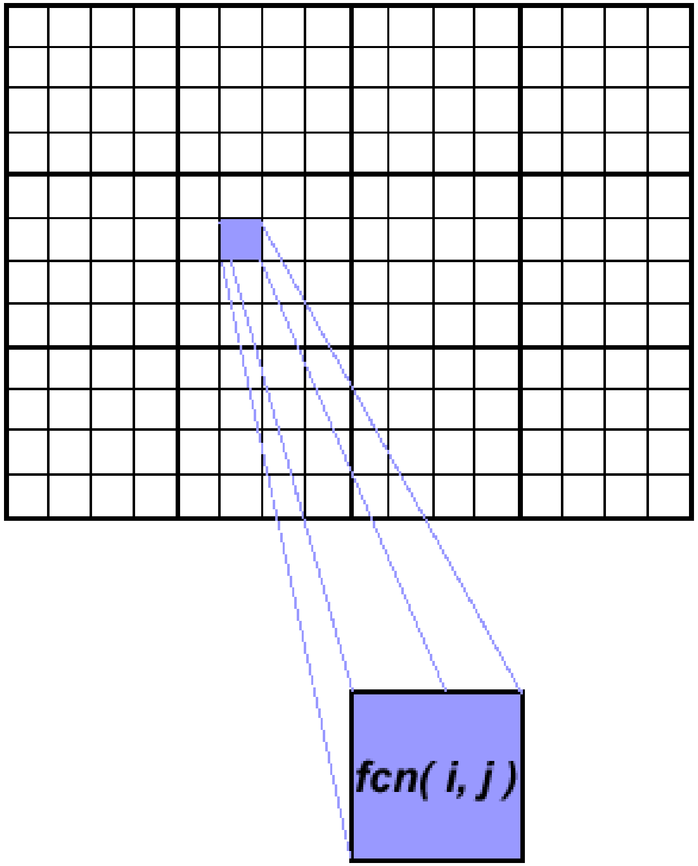
Examples for Parallel Program Design

https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial

Examples: Parallel Array Processing

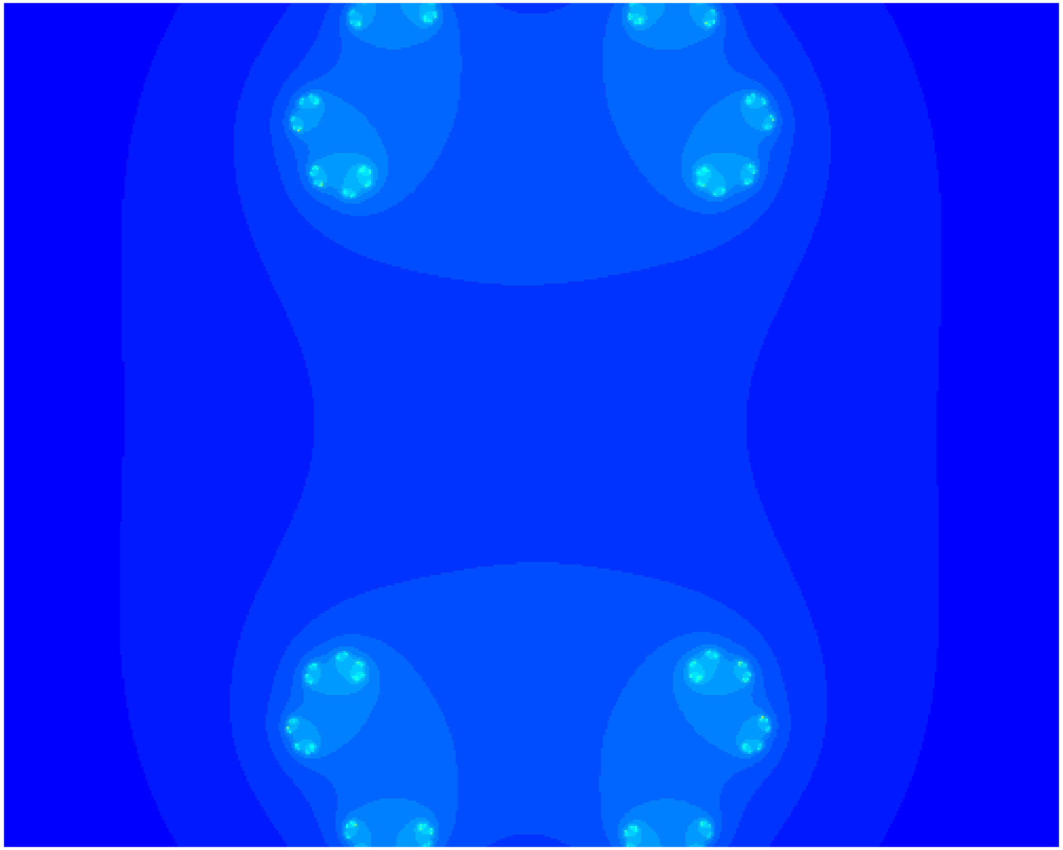
The serial code could be of the form

```
for (i = 0; i < m: i++)
     for (j = 0; j < n; j++)
         a[i][j] = fcn(i,j);
```



• The calculation of elements is independent from each other - this leads to a **embarissingly parallel** situation

Example: Julia set fractals



Julia set fractals are generated by initializing a complex number

$\mathbf{z = x + yi}$ **where** $\mathbf{i^2 = -1}$ and x and y are image pixel coordinates in the range of about -2 to 2.

z is **repeatedly** updated using:

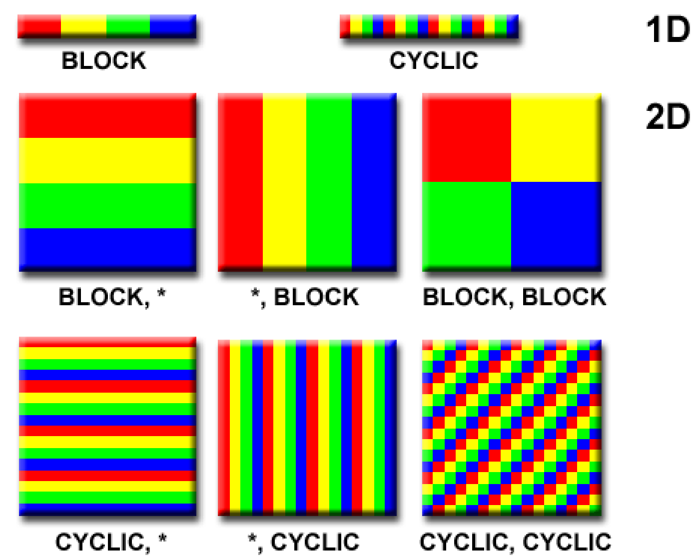
$\mathbf{z = z^2 + c}$ where c is another complex number that gives a specific Julia set. After numerous iterations, if the magnitude of **z <2** we say that pixel is in the Julia set and color it accordingly.

https://en.wikipedia.org/wiki/Julia_set

https://www.geeksforgeeks.org/fractals-in-cc/

Array Processing Parallel Solution 1

- The elements of the array are distributed so that each processor has a portion of an array (**sub-array**)
- **Independent calculation** of elements of the array ensure that it does not require communications between tasks (<u>embarrassingly parallel</u>).
- **The pattern of distribution is chosen by other criteria**, for example, the **unit stride** (stride 1) between the subarrays. The stride unit **maximizes the use of the cache / memory**



Array Processing Parallel Solution 1

After the array is distributed, each task executes the portion of the loop corresponding to its data. For example, with the block distribution:

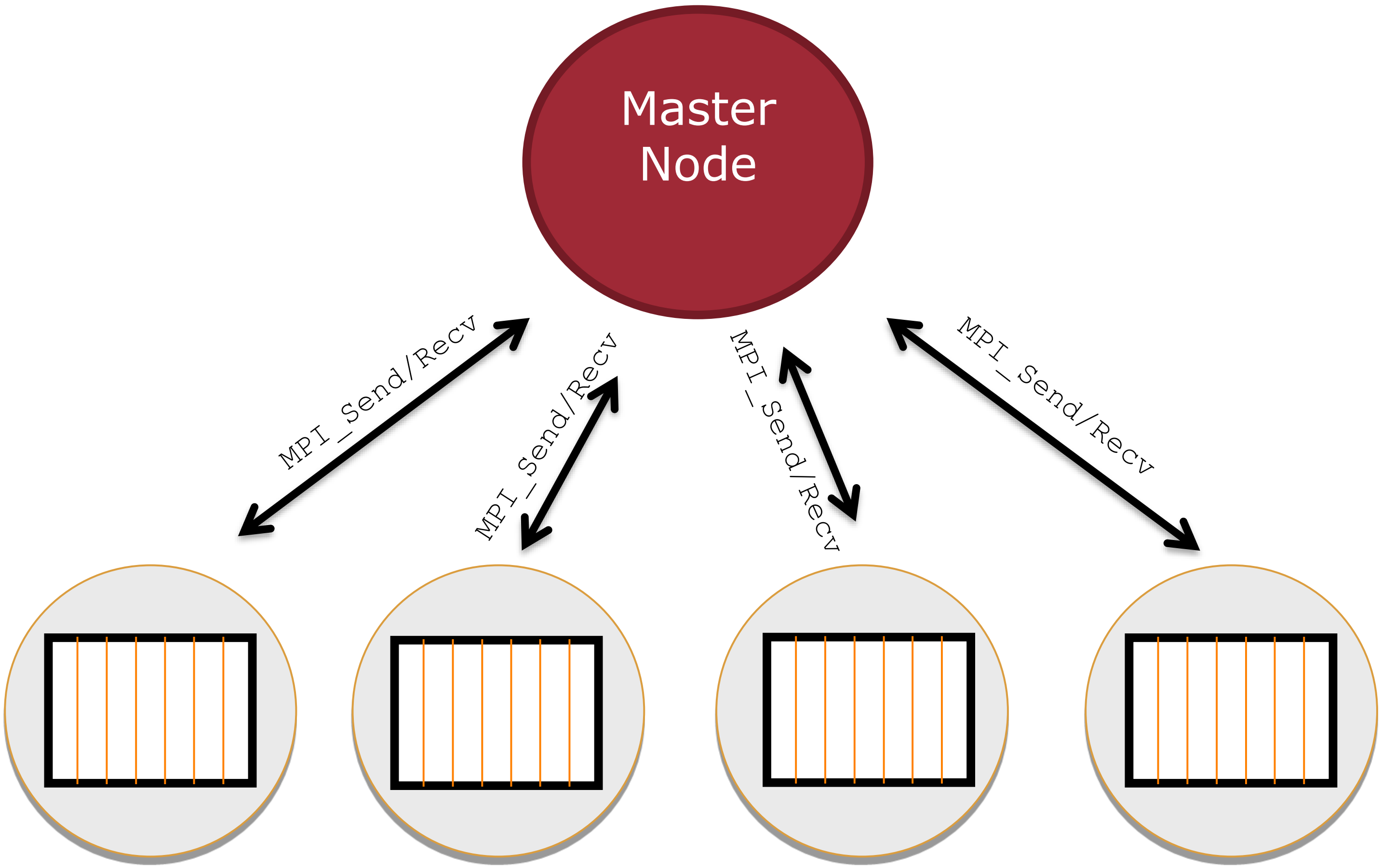
```
for (i = mystart; i <= myend: i++)
     for (j = 0; j < n; j++)
          a[i][j] = fcn(i,j);
```

Array Processing
Parallel Solution 1 (Static Master-slave)

A possible solution:

- **SPMD** model implementation
- The **Master process** initializes the array, sends info to the worker and receives the results.
- The **Worker processes** receive the info, perform portion of the computation and send the results to the master
- We use the block diagram array distribution

Master Slave pattern



SPMD paradigm (branching)

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array
  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

# calculate my portion of array
for (i = mystart; i <= myend: i++)
  for (j = 0; j < n; j++)
    a[i][j] := fcn(i,j);

  send MASTER results
endif
```

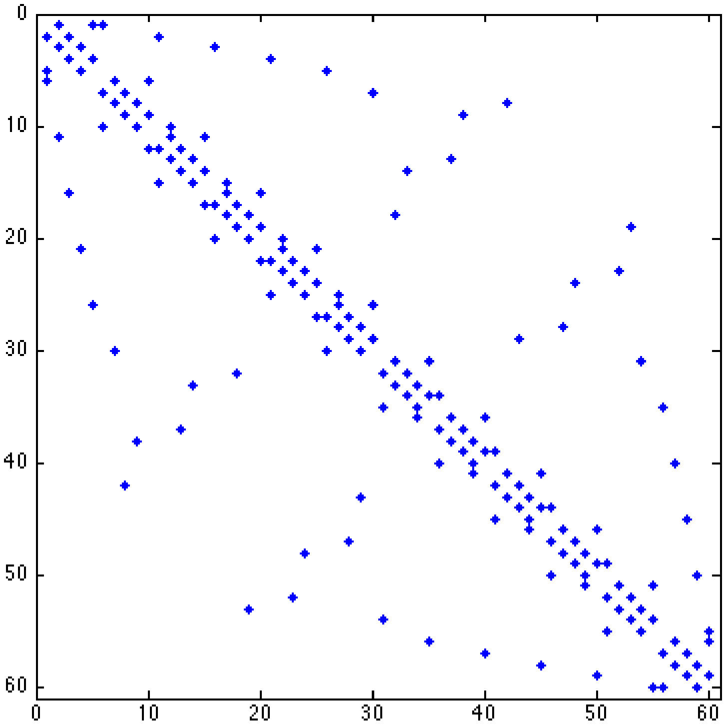
In red "parallel" changes respect to the sequential version

Array Processing
Parallel Solution 2: Pool of Tasks

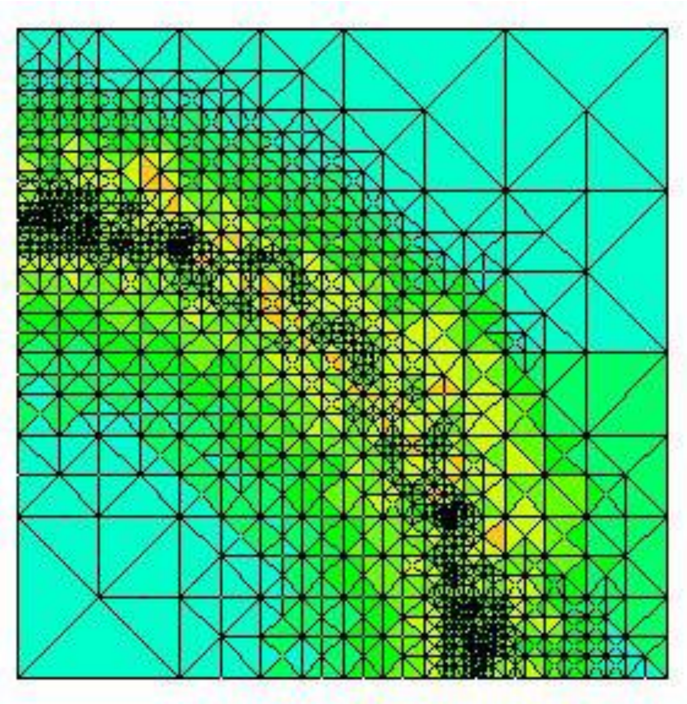
- The previous solution shows a **static load balancing**:
  - Each task has a fixed amount of work to do
  - The slower tasks determine the overall performance of the system
  - However, this problem is not serious when all tasks have more or less the same job on identical machines

- If you have **load balance issues** (some tasks work faster than others) it is better to use the "pool of tasks" model

Load Balancing



Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros"



Adaptive grid methods - some tasks may need to refine their mesh while others don't

Pool of Tasks

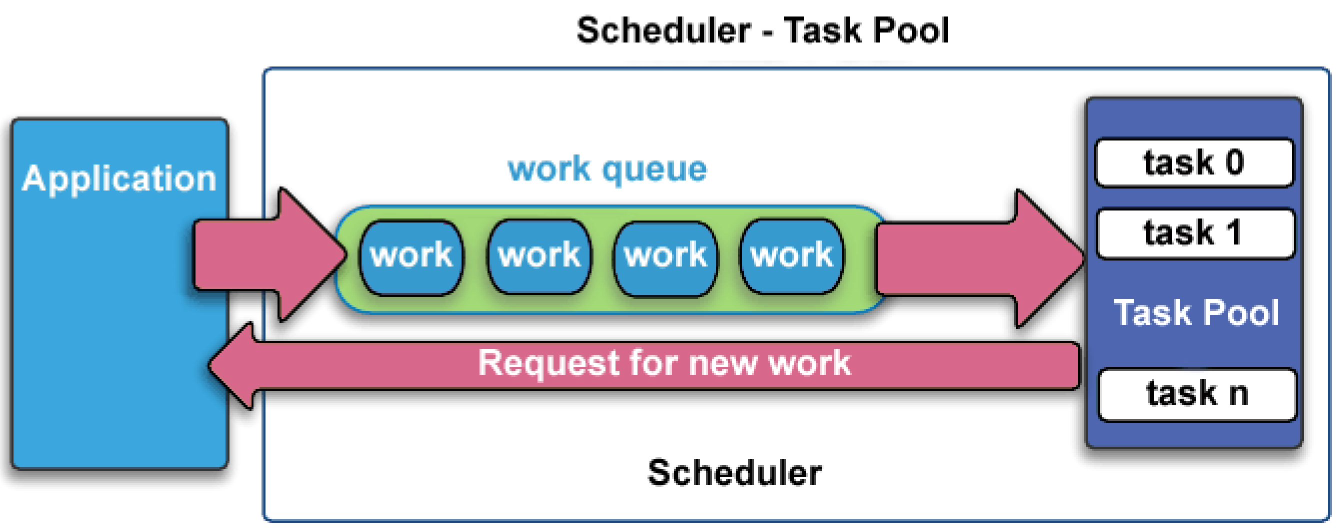
Master Process:
- It keeps the pool of tasks that need to run slave processes
- Sends a task to a slave when required
- Collects results from the slaves

Worker Process: runs in sequence:
- Pick up a task from the master process
- Performs computation
- Send the results to the master

The slave processes do not know what portion of the array they will handle before runtime or how many tasks will be performed on it

Dynamic Load balancing occurs at run time: Faster tasks (or processes) will get more work

Scheduler-task pool approach



As each task finishes its work, it receives a new piece from the work queue

SMPD paradigm (branching)

```
find out if I am MASTER or WORKER

if I am MASTER

  do until no more jobs
    send to WORKER next job
    receive results from WORKER
  end do

  tell WORKER no more jobs

else if I am WORKER

  do until no more jobs
    receive from MASTER next job
    calculate array element: a[i,j] = fcn(i,j)
    send results to MASTER
  end do

endif
```

π computation

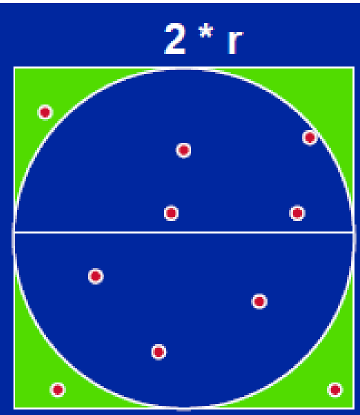
The calculation of π can be carried out in different ways. We use a Monte Carlo method (algorithms that use statistical sampling for the resolution):

- Inscribe a circle in a square
- Generate random points in the square
- Determines the N number of points in the square that are ALSO in the circle

- The area of the circle is Ac, As is the one of the square
- We can say that $\pi \sim 4 * (Ac / As)$
- Note that more points are generated, the better the approximation

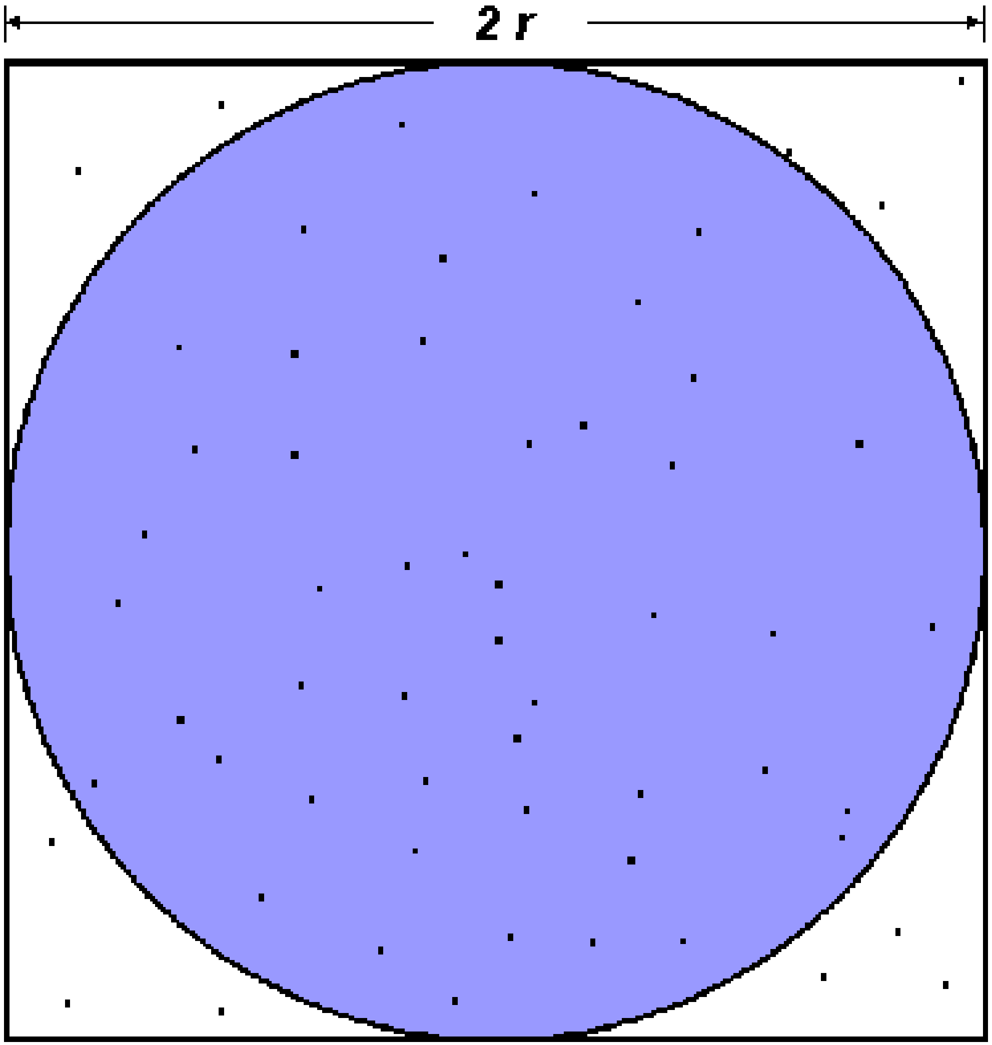
π computation

- **Monte Carlo Calculations**: Using Random numbers to solve **<u>tough</u>** problems
  - Sample a problem domain to estimate areas, compute probabilities, find optimal values, etc
  - Example: Computing pi with a digital dart board:



N= 10	π = 2.8
N=100	π = 3.16
N= 1000	π = 3.148

- **Throw darts at the circle/square.**
- **Chance of falling in circle is proportional to ratio of areas:**
  $A_c = r^2 * \pi$
  $A_s = (2*r) * (2*r) = 4 * r^2$
  $P = A_c/A_s = \pi /4$
- **Compute π by randomly choosing points, count the fraction that falls in the circle, compute pi.**



$$A_S = (2r)^2 = 4r^2$$

$$A_C = \pi r^2$$

$$\pi = 4 \times \frac{A_C}{A_S}$$

pseudo-Serial Code

```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1 ; ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

Observation

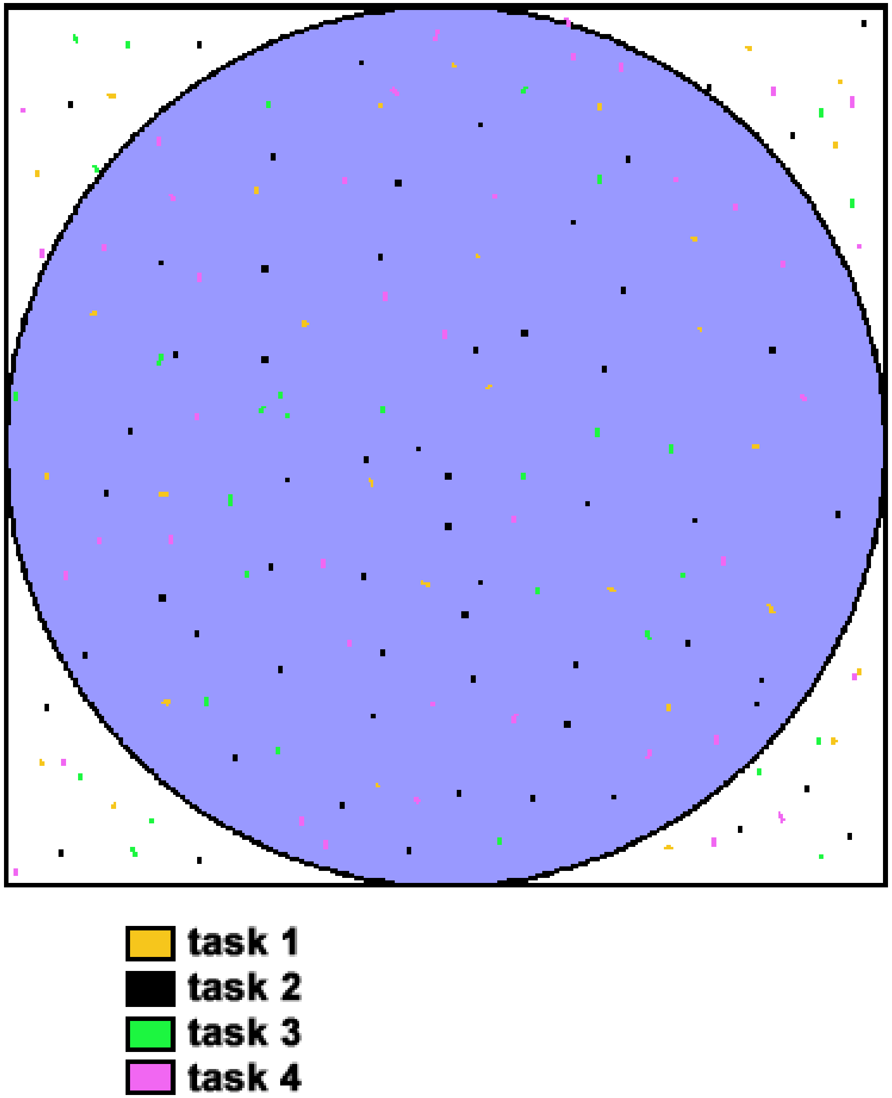
- Most of the computation is carried out for the execution of the loop
- Leads to a "embarrassingly parallel" solution
- Computationally “hard"
- Minimum communication
- Minimum I / O

π Computation Parallel Solution

- **Parallel strategy: break the loop into portions that can be executed by the task**
- Therefore:
  - Each task executes its portion of the cycle a number of times ...
  - **Note**: Each task can do computation WITHOUT requiring information from other tasks (there are **no data dependencies**)
  - It uses the SPMD model. In addition, a task acts as master and collects the results

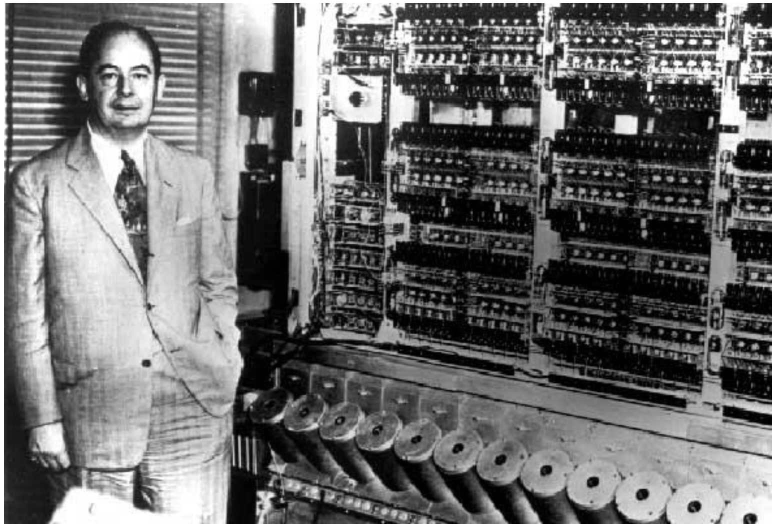
```
npoints = 10000
circle_count = 0

p = number of tasks
num = npoints/p

find out if I am MASTER or WORKER

do j = 1,num
  generate 2 random numbers between 0 and 1
  xcoordinate = random1 ; ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER

  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)

else if I am WORKER

send to MASTER circle_count

endif
```

Be careful to the random seed!



Use rand_r() in C, better than rand() !

Modelling and Simulation
Cellular Automata



- Cellular Automata (CA) are discrete **parallel computational models**, widely utilized for modeling and simulating complex Systems
- Invented by John von Neumann and Stanislaw Ulam at Los Alamos National Lab (early 1950s)
- Based on work by Alan Turing
- Most basic research on CA in the 1950s and 60s
- Three major events in CA research
  - John von Neumann's self-reproducing automaton
  - John Conway's Game of Life
  - Stephen Wolfram's classification of cellular automata

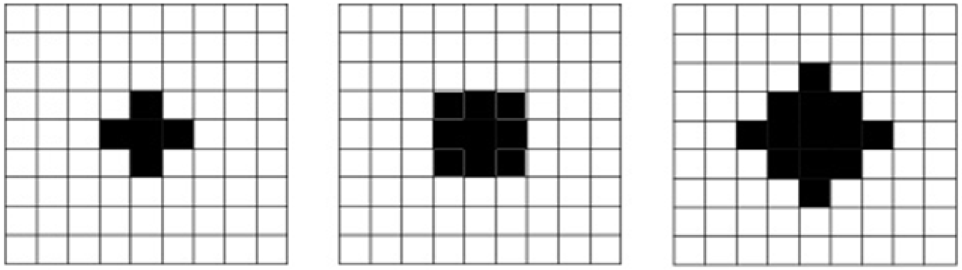
Cellular Automata

- Conceived in the 50s by John von Neumann for the study of self-reproducing issues (von Neumann, 1966)
- They are a parallel computational model, discrete in space and time
- CA can be described as a matrix of simple processing units, the cells, each one interacting with its neighboring ones

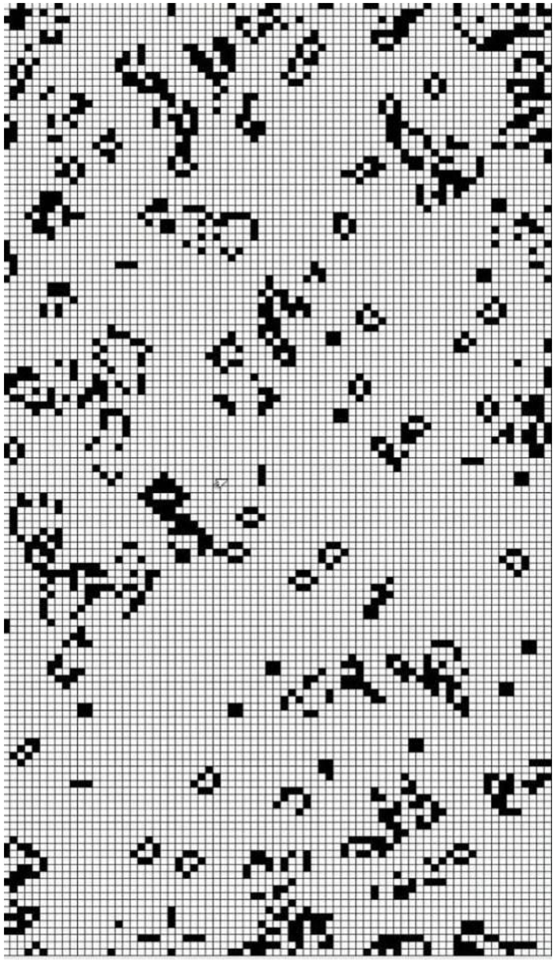
Cellular Automata

- **CA** = a lattice of cells identified by points in a Euclidean space
- **X={ξ_1,.......ξ_{m-1}}** is the neighbourhood index so that, given a generic cell **c** the set **N(X,c)** of the neighbouring cells is:

  N(X,c) = N(c) = {c, c+ξ_1,...c+ξ_{m-1}}
- **For each cell c**
  - **S(c)** is the finite set of possible states.
  - **σ(c, N(c)): $S^m \rightarrow S$** is the transition function

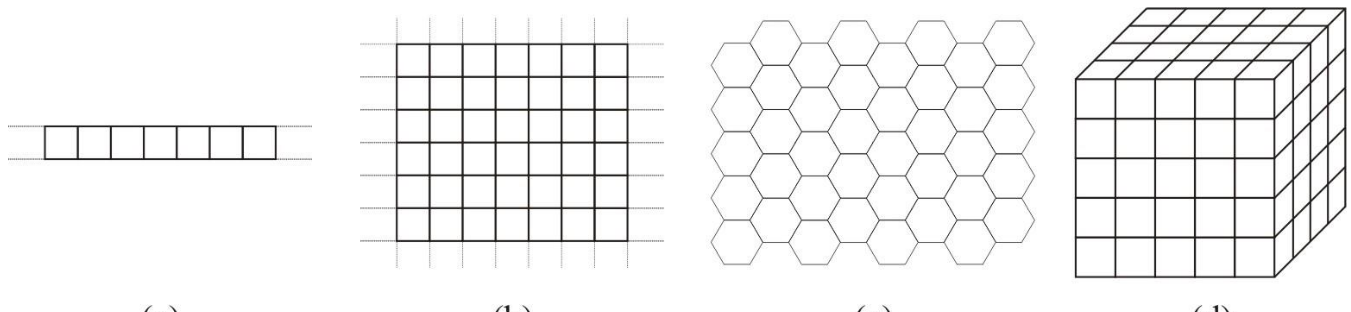


von Neumann Moore von Neumann



Cellular Automata

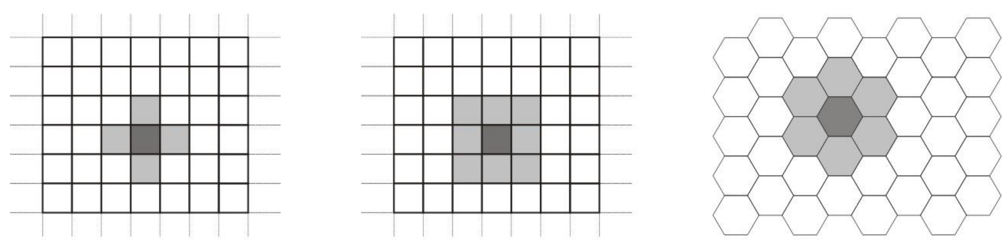
Cellular spaces



(a) (b) (c) (d)

Neighborhoods

(a) (b)



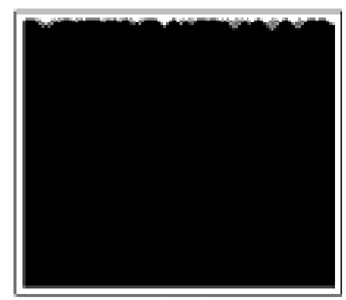
(a) (b) (c)

Cellular Automata

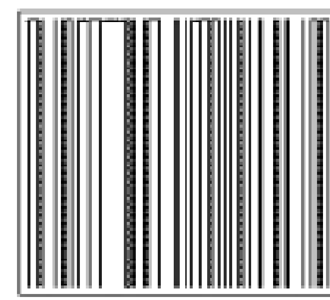
- At time $t=0$, cells are in arbitrary states which define the **initial condition** of the system
- CA evolves by changing states of cells at discrete steps by applying simultaneously to each cell the same **transition function**, so that its evolution is determined by local interactions among their constituent parts
- The overall dynamics **emerges** as a consequence of the simultaneous applications of the transition function to each cell

CA Dynamics

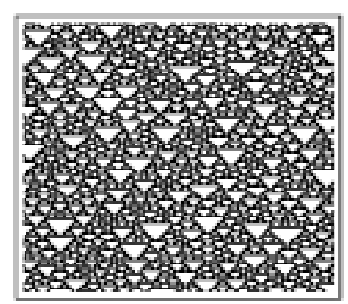
- Wolfram's Classification of 1-D CA behavior
  1. Spatially stable
  2. Sequence of stable or periodic structures
  3. Chaotic aperiodic behavior
  4. Complicated localized structures
- Wolframs classification most popular
- Problem: Class membership of a given rule is undecidable



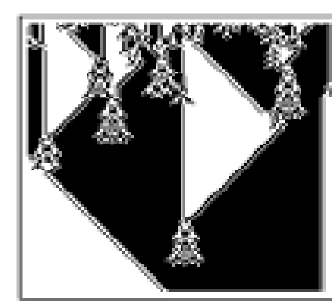
class 1



class 2



class 3



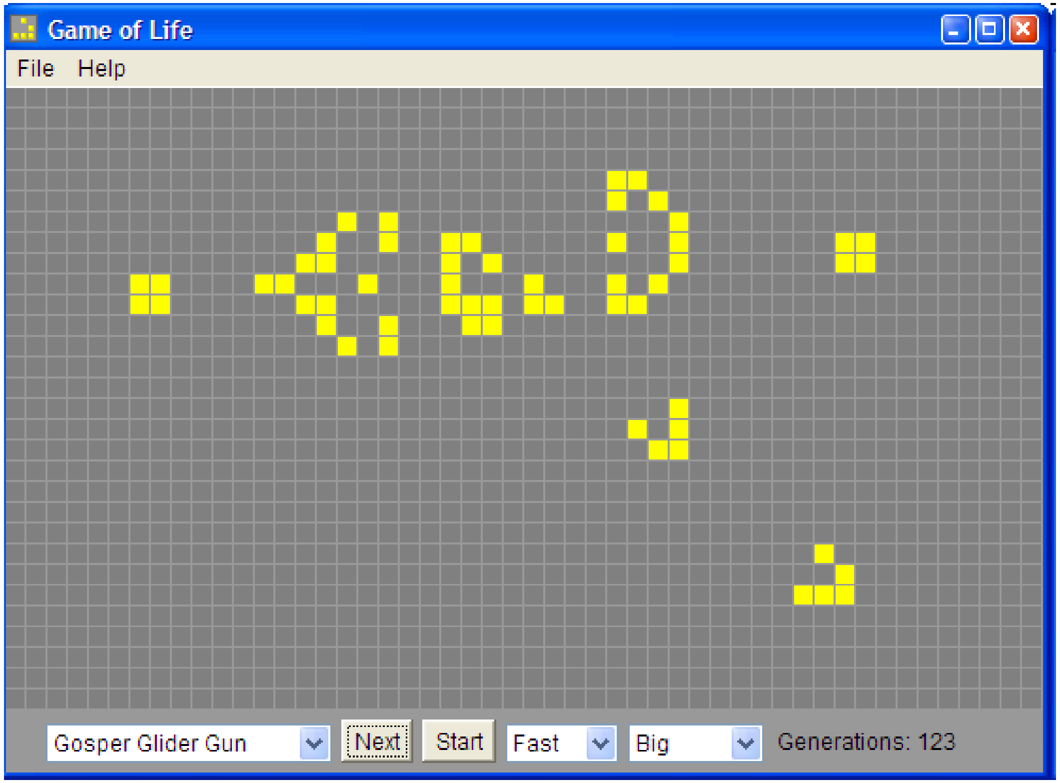
class 4

CA and Complex System Theory

- Game of life
- Developed by John H. Conway in 1970
- Simple rules → complex behavior
- Rules
  - Survival: 2 or 3 live neighbors
  - Birth: exactly 3 live neighbors
  - Death: all other cases

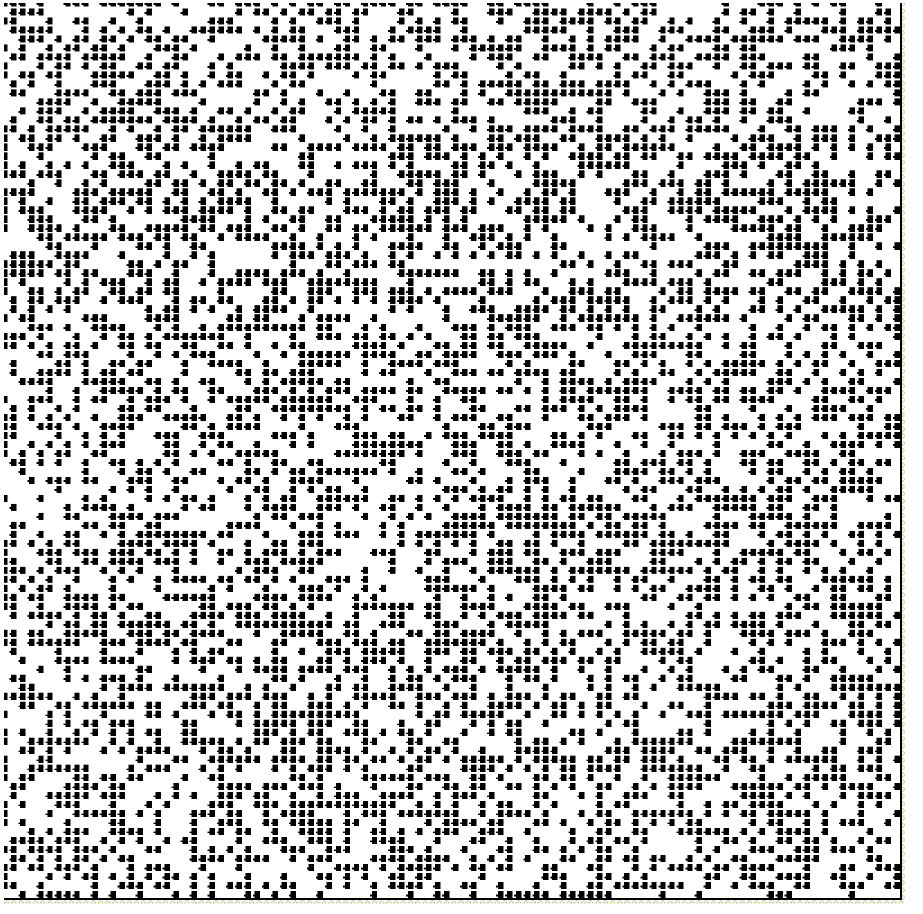
http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Look at Golly Software

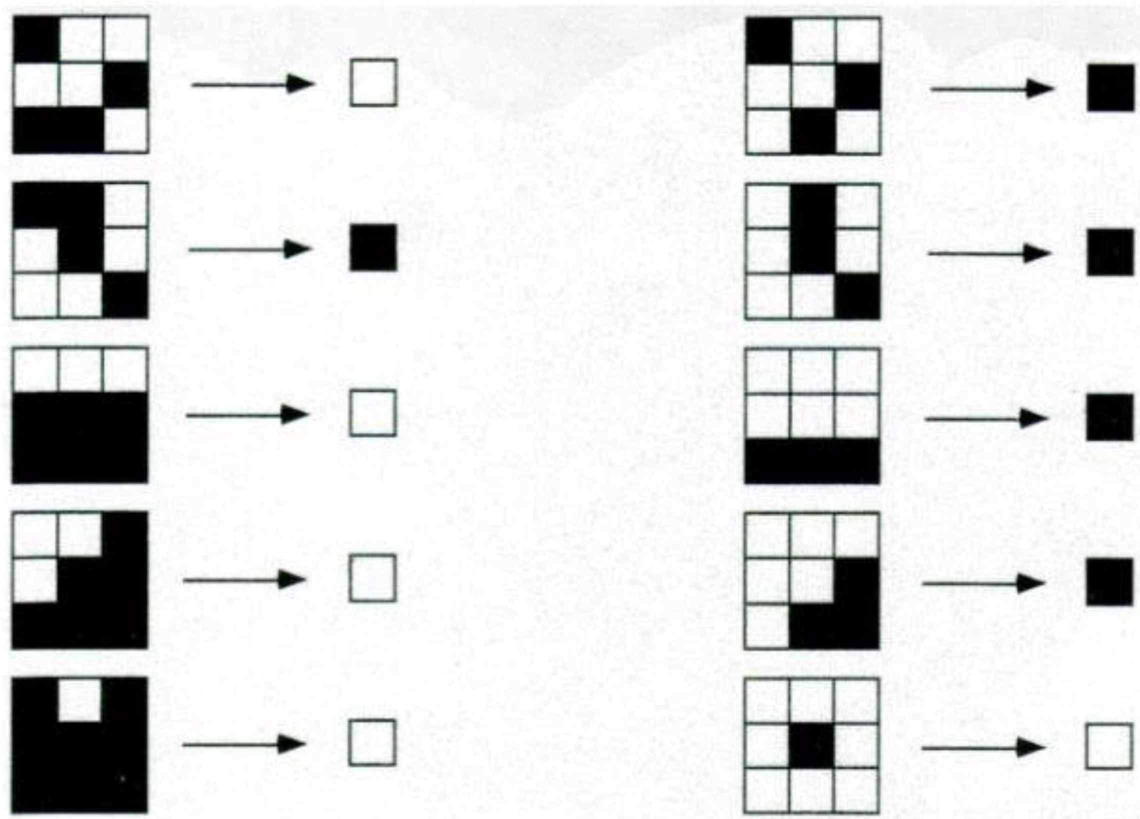


The Game of Life

- The **Game of Life** is a cellular automaton developed by the English mathematician **John Conway** (1927-2020). Its purpose is to show how **complex behaviors can emerge from simple rules** and many-body interactions, a principle that underlies eco-biology, which also refers to the **theory of complexity.**

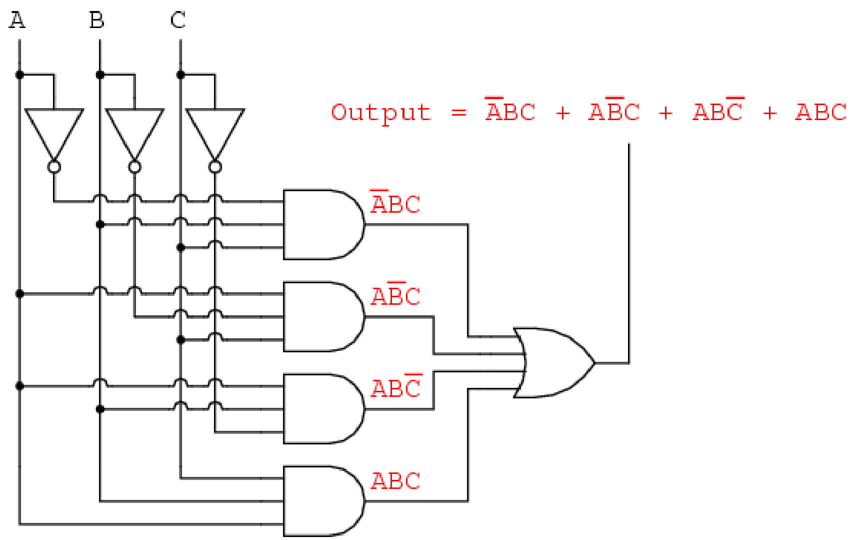


Simple rules, executed at each time step

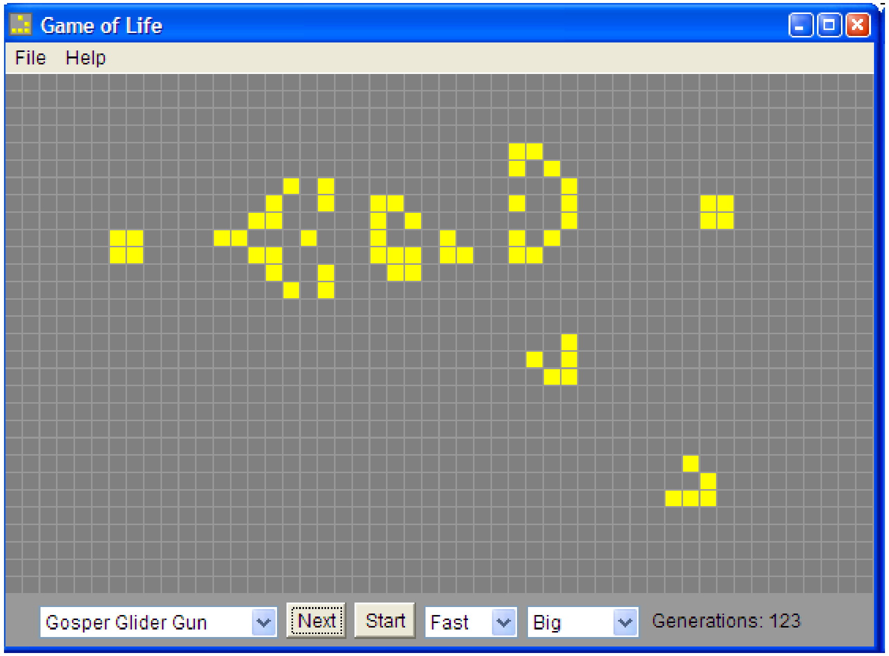


- A live cell stays alive (survives) if it has 2 or 3 live neighbors, otherwise it dies.
- A dead cell comes to life (birth) if it has exactly 3 live neighbors, otherwise it stays dead.

Turing equivalence of the Game of Life



Logic gates



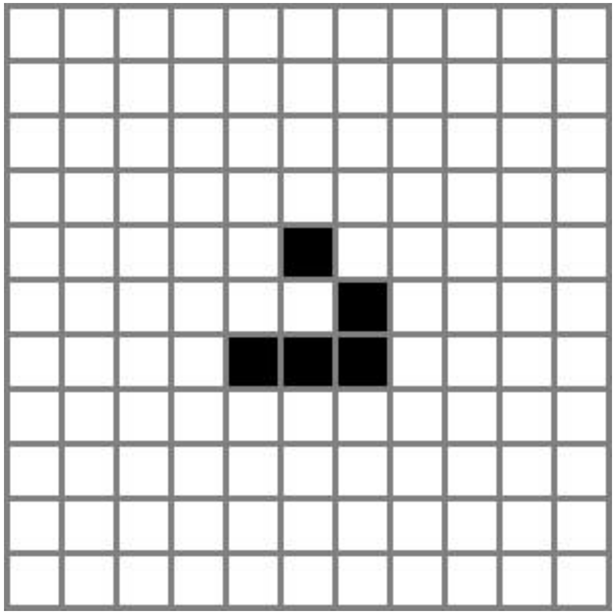
Glider gun

It has been proved that the Game of Life has the same computional power of a Turing Machine (es. http://rendell-attic.org/gol/tm.htm)

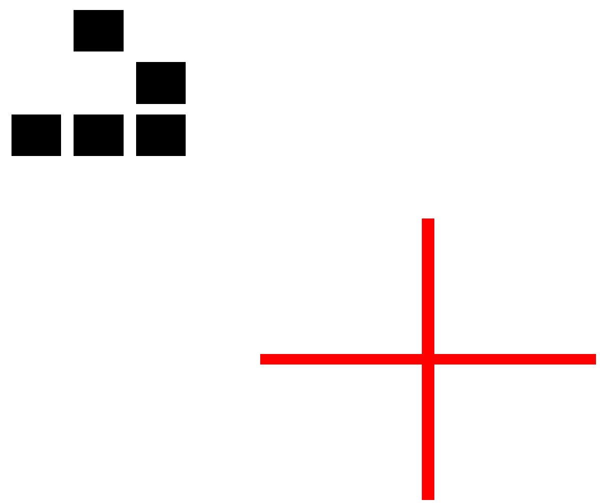
Alan Mathison Turing was a British mathematician, logical and cryptographer, considered one of the fathers of computer science and one of the greatest mathematicians of the twentieth century

Is it alive?

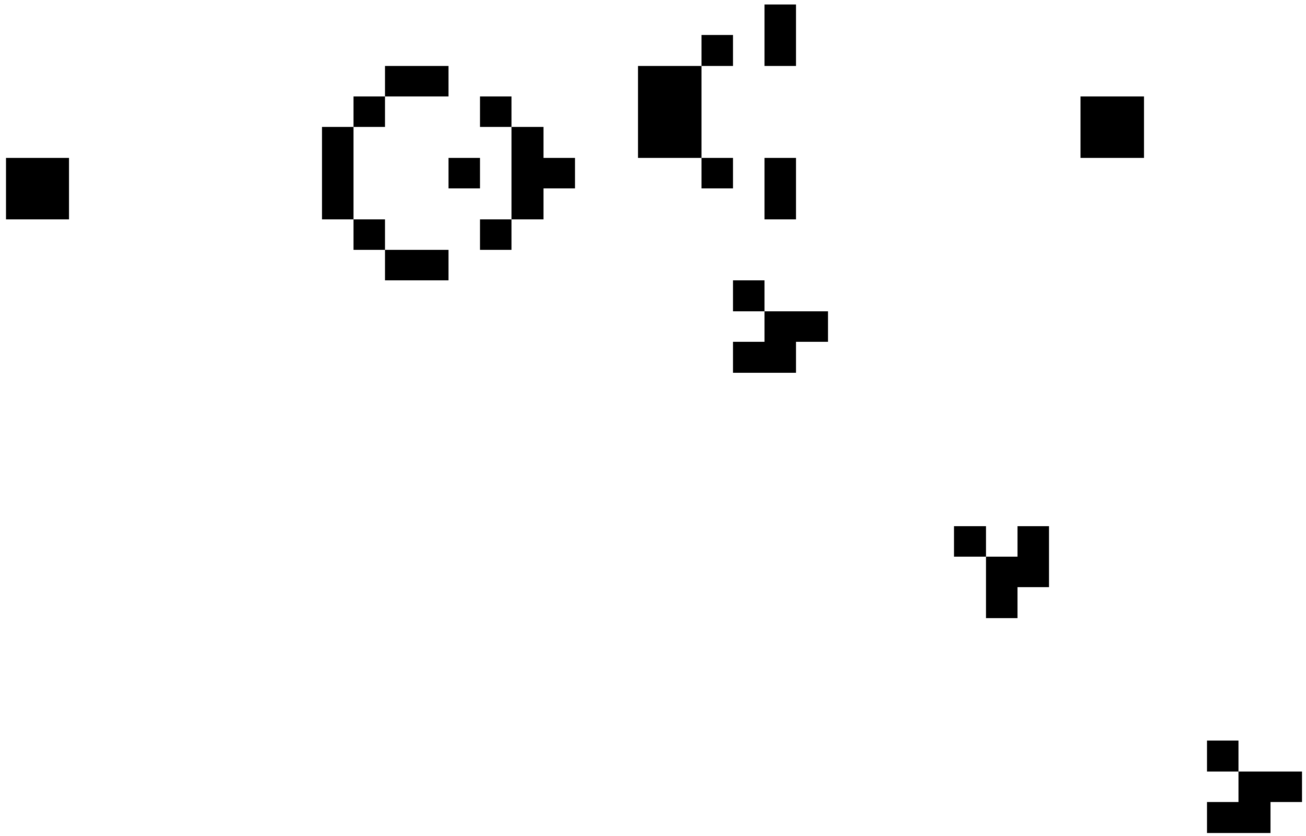
- Compare it to the definitions…



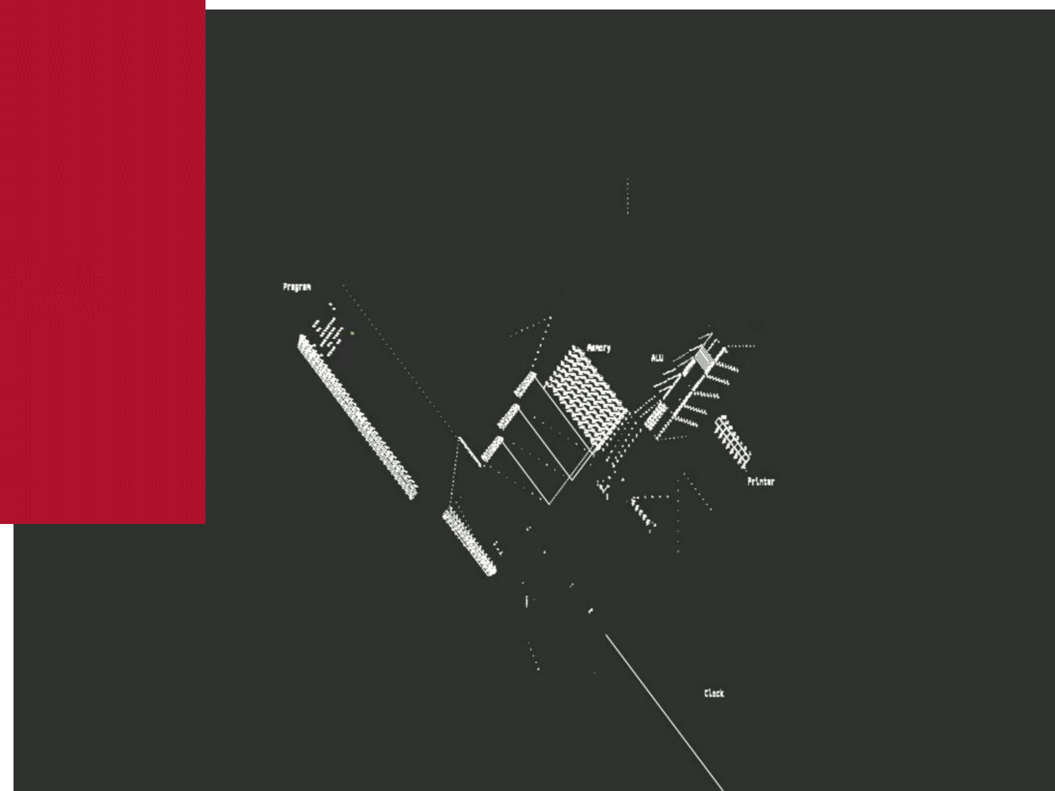
Glider



A Glider Gun



Un computer virtuale basato su GoL!



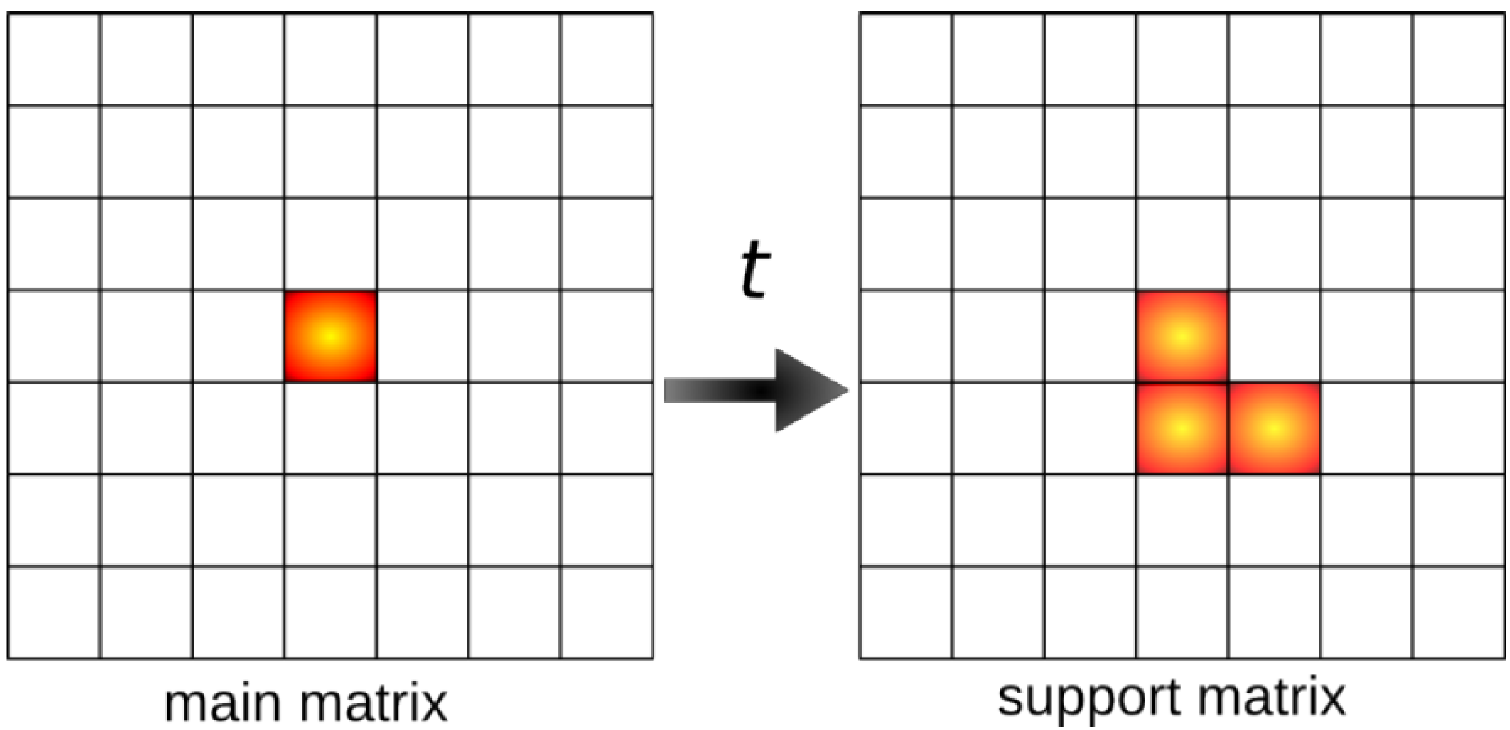
Golly software

http://golly.sourceforge.net/

https://sourceforge.net/projects/golly/

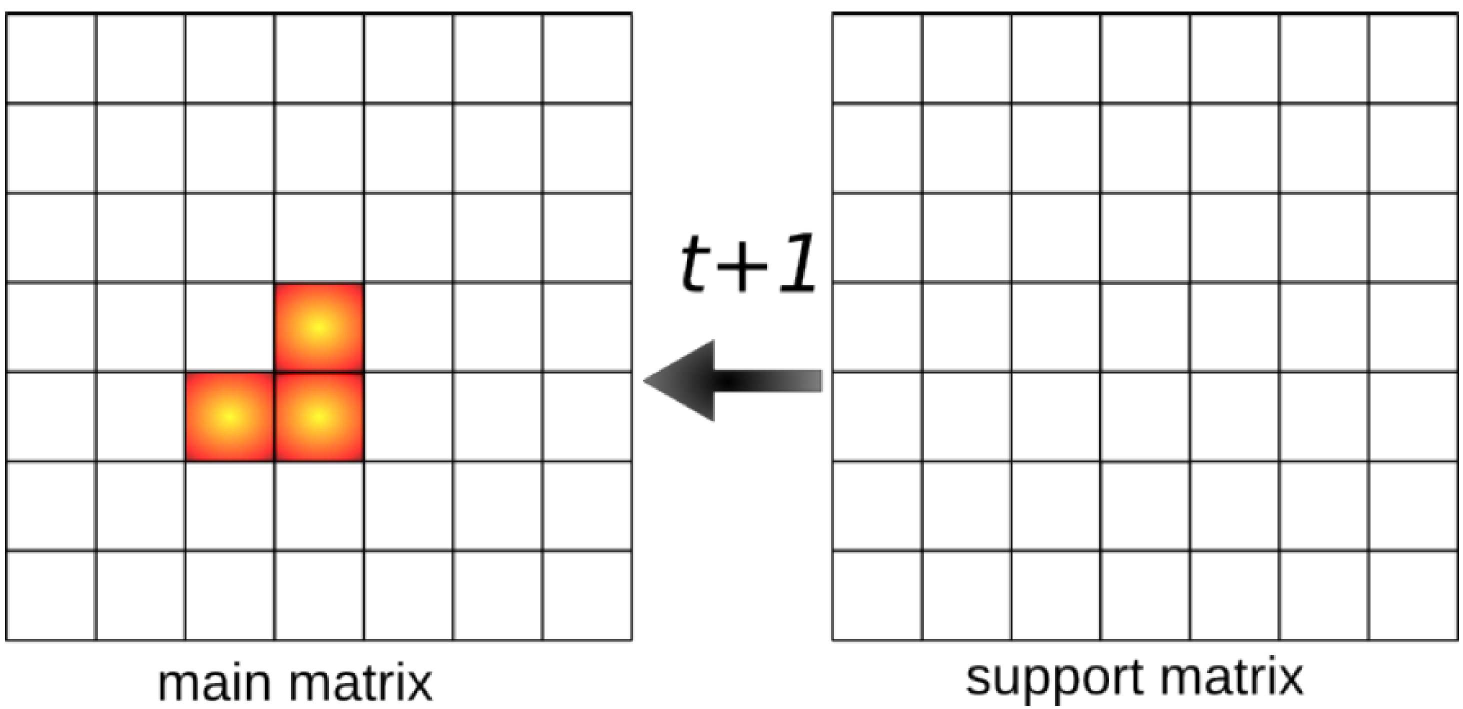
Serial Implementation

- To maintain actual cells states during the CA step, a double matrix data structure is considered for each substate:
  - actual values are read from the main matrix
  - new values are stored on the support matrix
- At the end of every CA step, the support matrix becomes the main one and the process continues



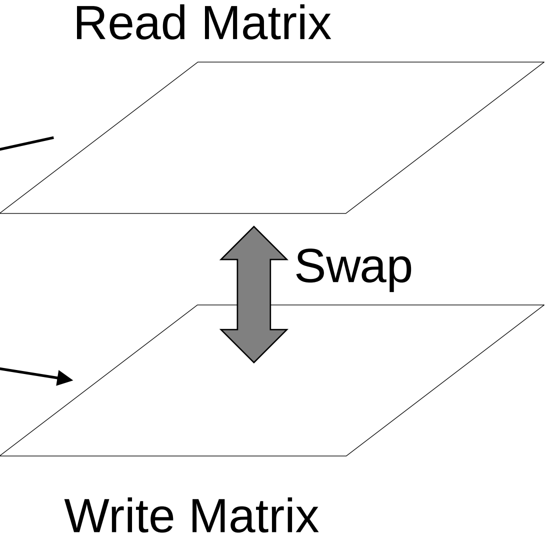
Serial Implementation

- To maintain actual cells states during the CA step, a double matrix data structure is considered for each substate:
  - actual values are read from the main matrix
  - new values are stored on the support matrix
- At the end of every CA step, the support matrix becomes the main one and the process continues

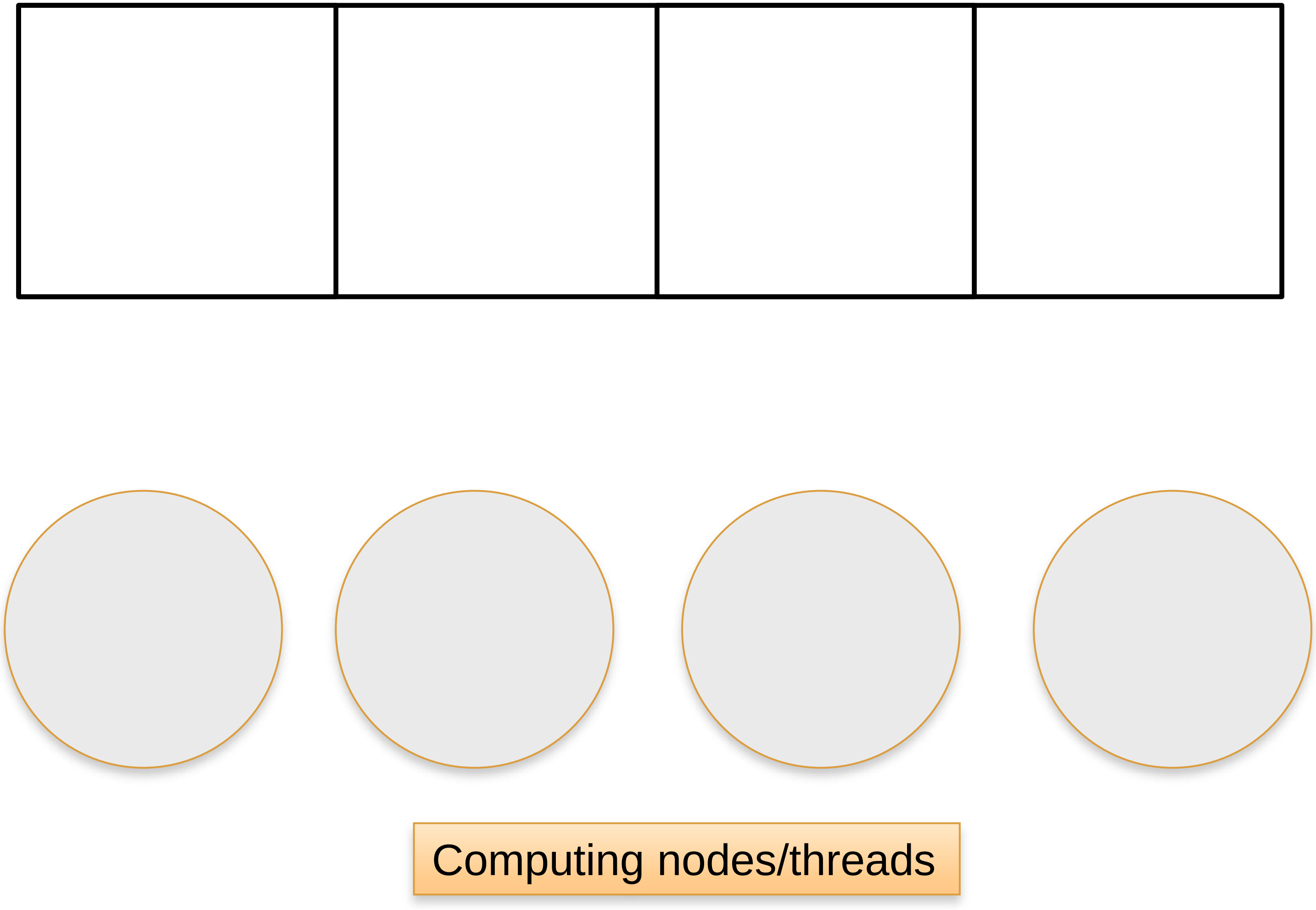


Parallel Execution of CA

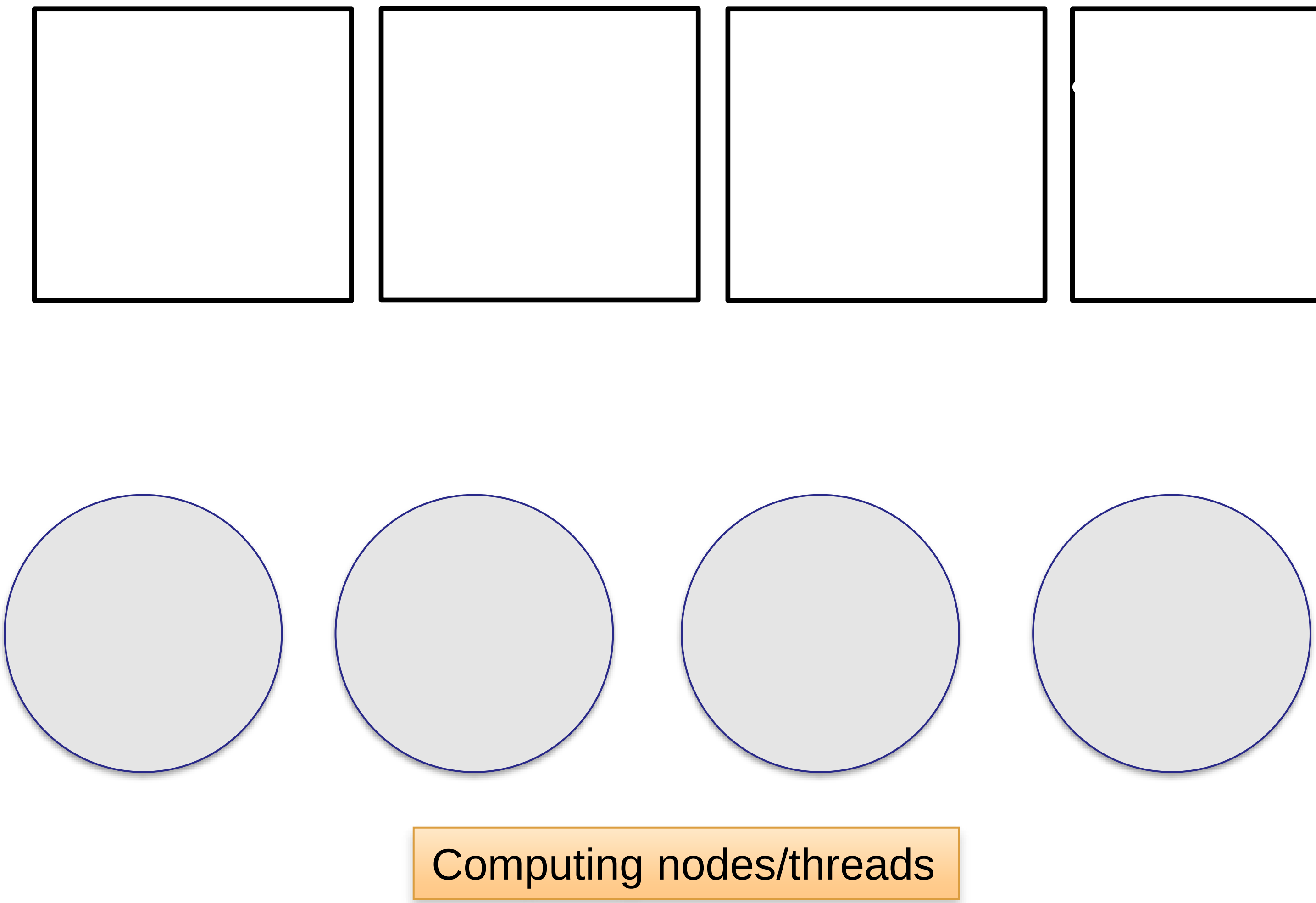
```
For each step {
  For each cell c {
    N(c)=getNeighbourhood()
    S(c)=σ(c,N(c))
  }
  swap matrices
}
```



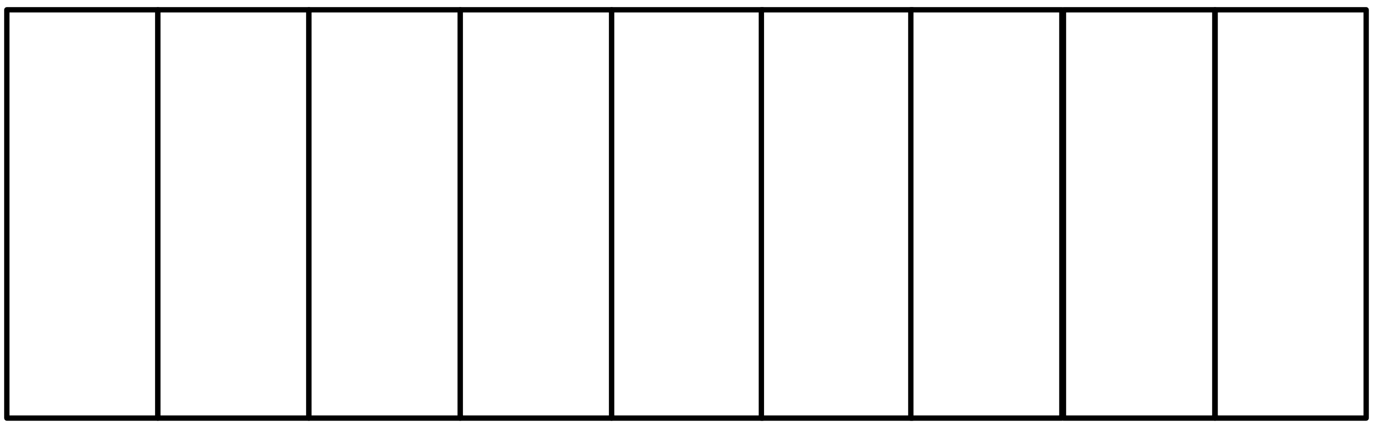
Parallel Implementation
Space Partitioning



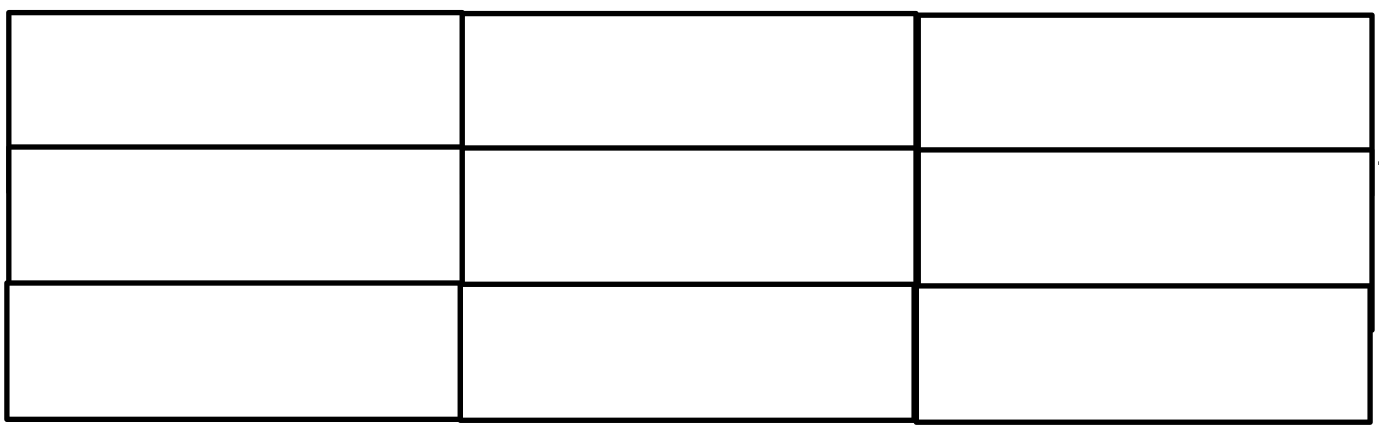
Parallel Implementation
Space Partitioning



Mono-dimensional vs two-dimensional partitioning

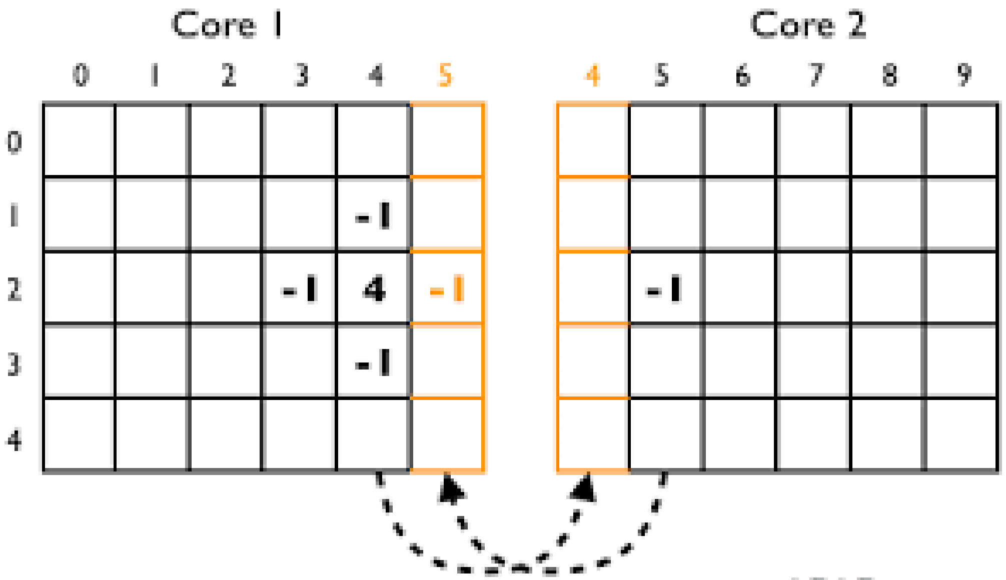


Mono-dimensional
Partitioning



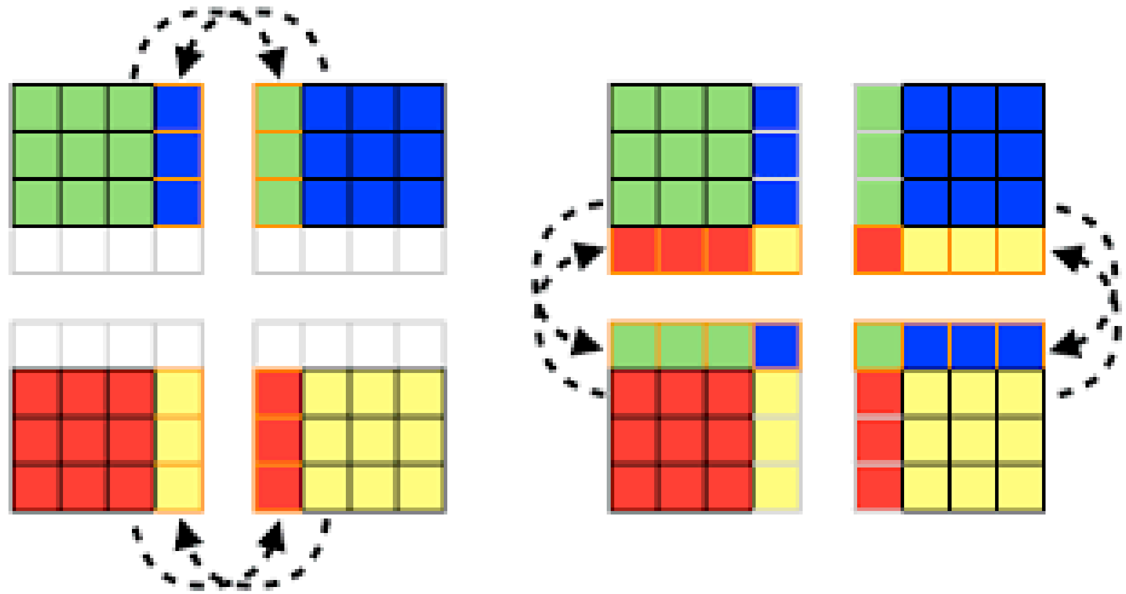
Two-dimensional
Partitioning

Ghost Cells

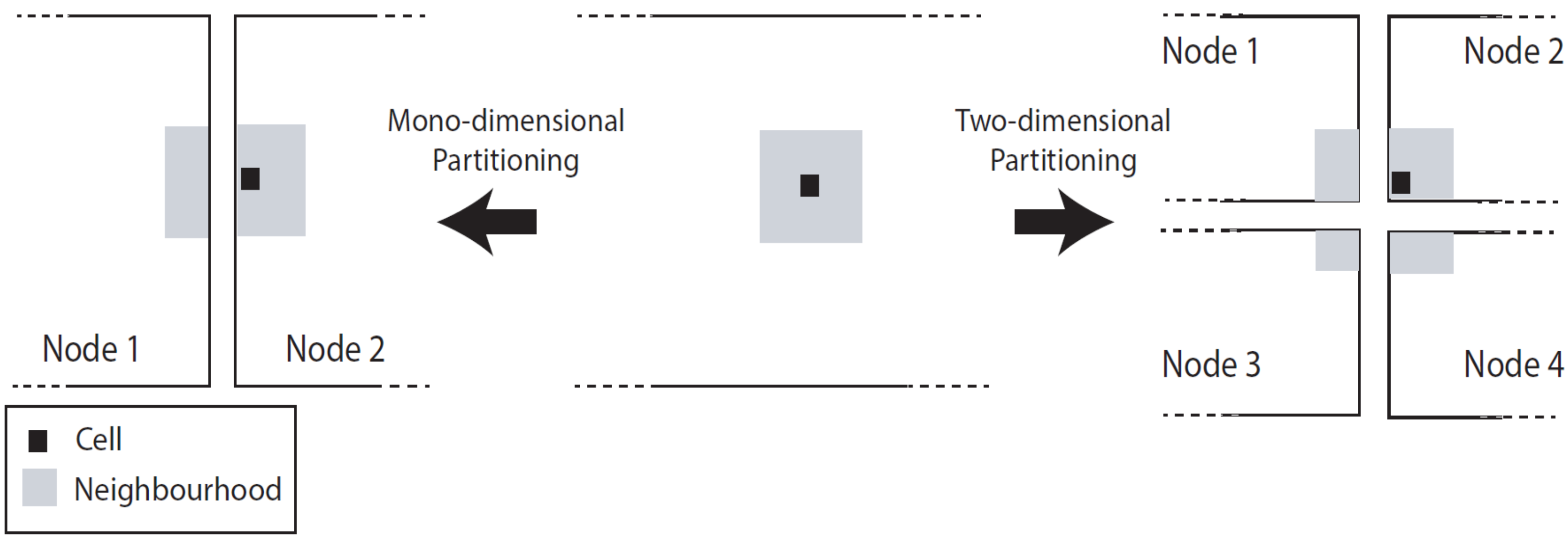


1D Partitioning

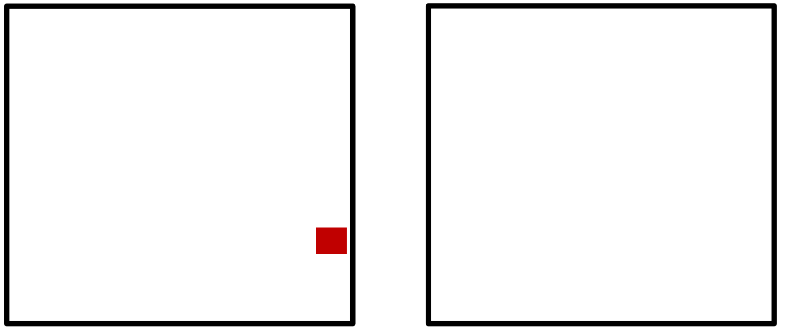
2D Partitioning



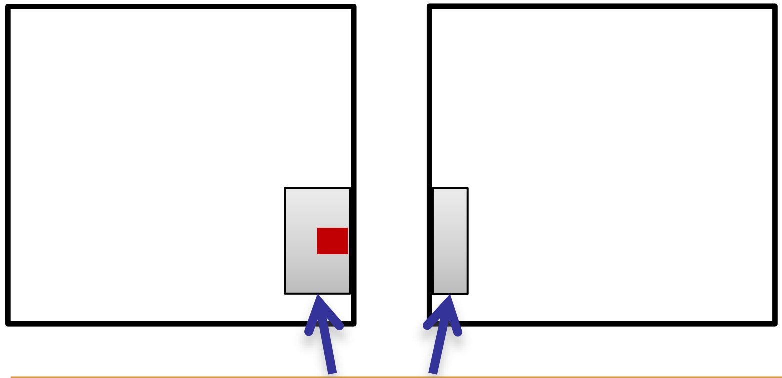
Exchange Borders



Border of a region

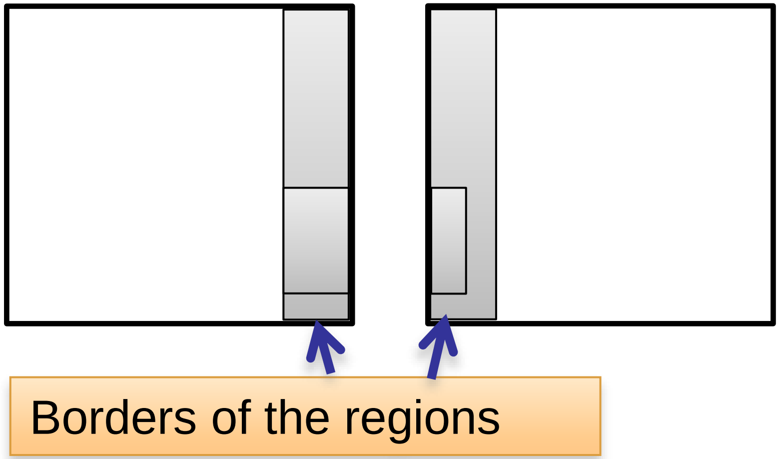


Border of a region

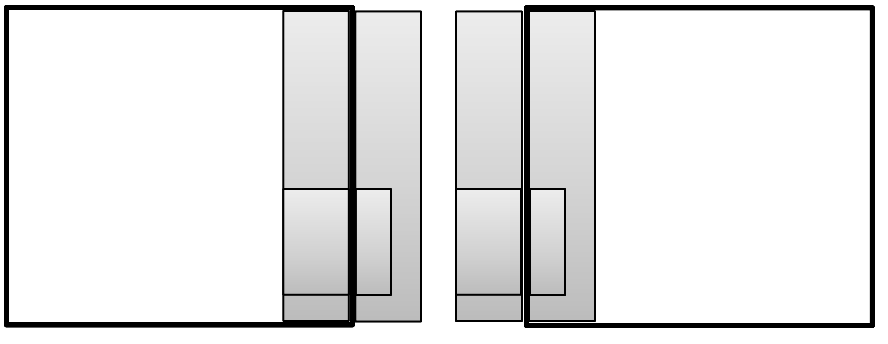


Cell’s neighbourhood overlaps with another region

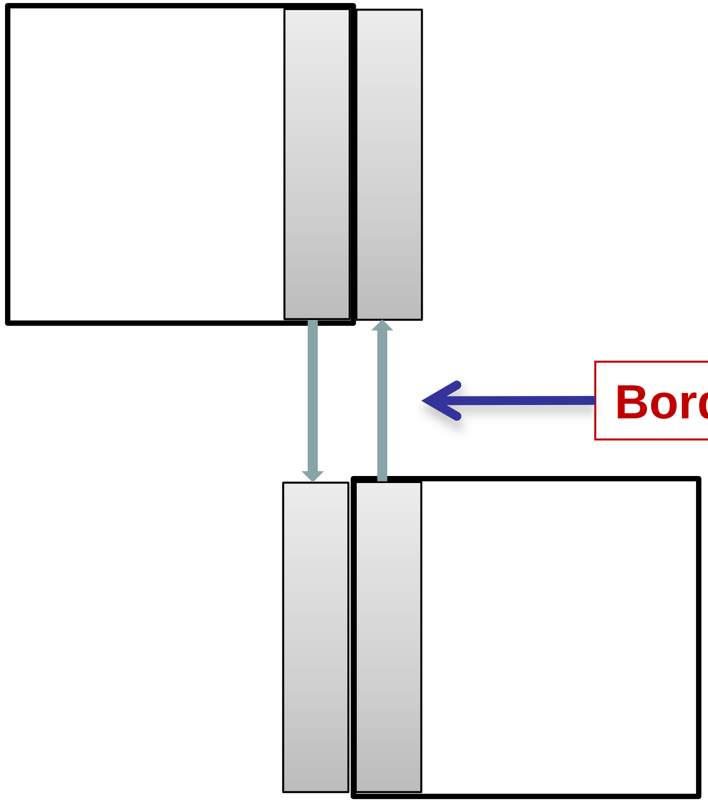
Border of a region



Border of a region



Border of a region



Borders must be kept aligned at each step

- And in OpenMP? Simple!
  - All data is shared
  - No Halo/Borders are needed!
  - Parallelize loops!
- And in MPI? Not Simple!
  - Ghost cells
  - Blocking and non-blocking messages
  - We will see later during MPI classes

Heat equation

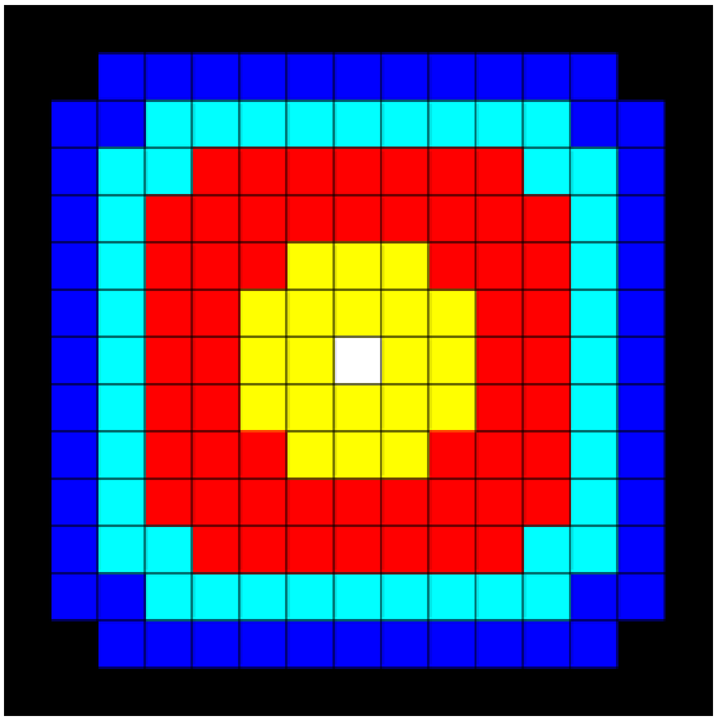
- Cellular Automata ☺

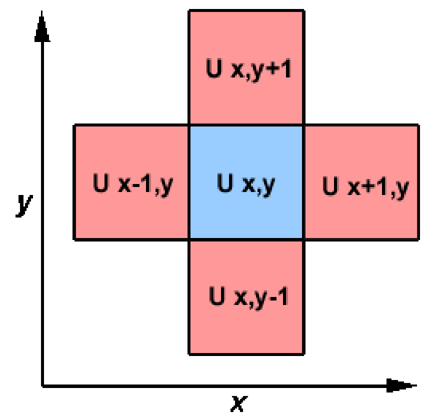
- As we know, most of the problems in parallel computing require (unfortunately) communications between tasks. A number of them require further communication with the "neighbors‟ task
- The **heat equation is a partial differential equation** that describes the temperature change over time (on a plate for example), given the initial distribution of temperature and boundary conditions
- A **finite differences method (FDM) scheme** is used to solve the equation numerically on a square region
- The initial temperature is set to zero at the edges and high in the middle
- The temperature boundary is maintained at zero (so as to simulate air, for instance)
- An **iterative algorithm** is used. The elements of a two-dimensional array representing the temperatures at points in the square

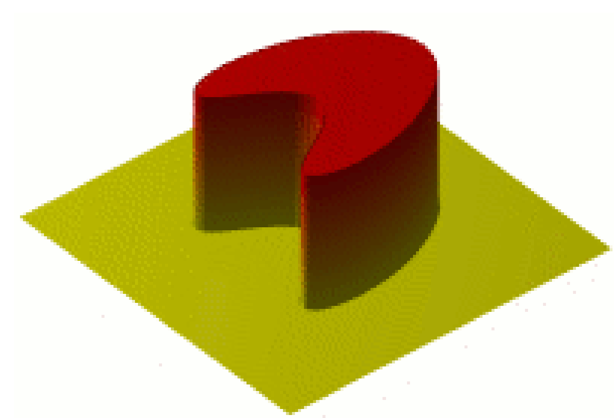
The calculation of the elements DEPENDS on the value of the neighboring elements

Heat equation

$$\frac{\partial T(t,x)}{\partial t} = \kappa \frac{\partial^2 T(t,x)}{\partial x^2}$$







https://en.wikipedia.org/wiki/Heat_equation

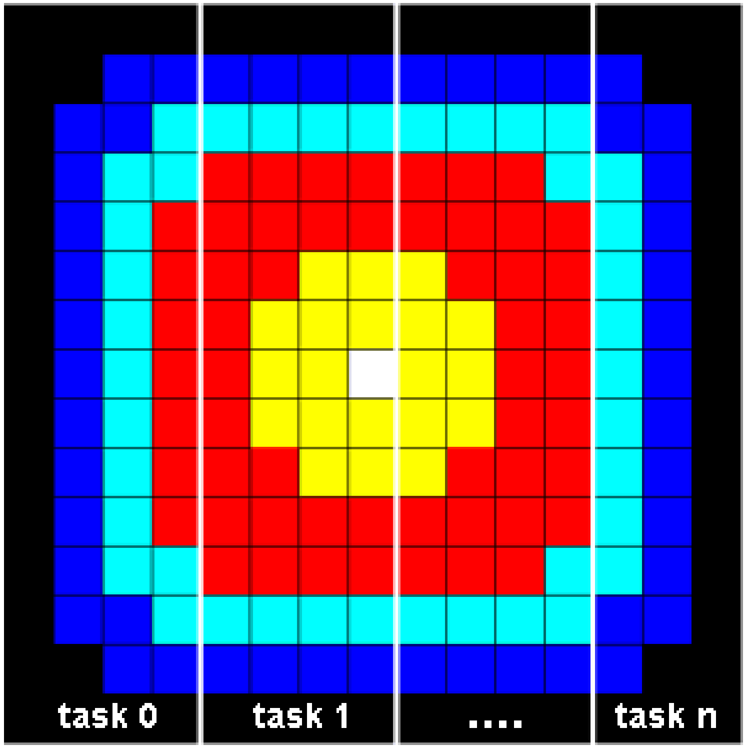
Heat equation- Serial Program

The calculation of an element depends on the values of the neighbors
u2 = current step
u1 = previous step

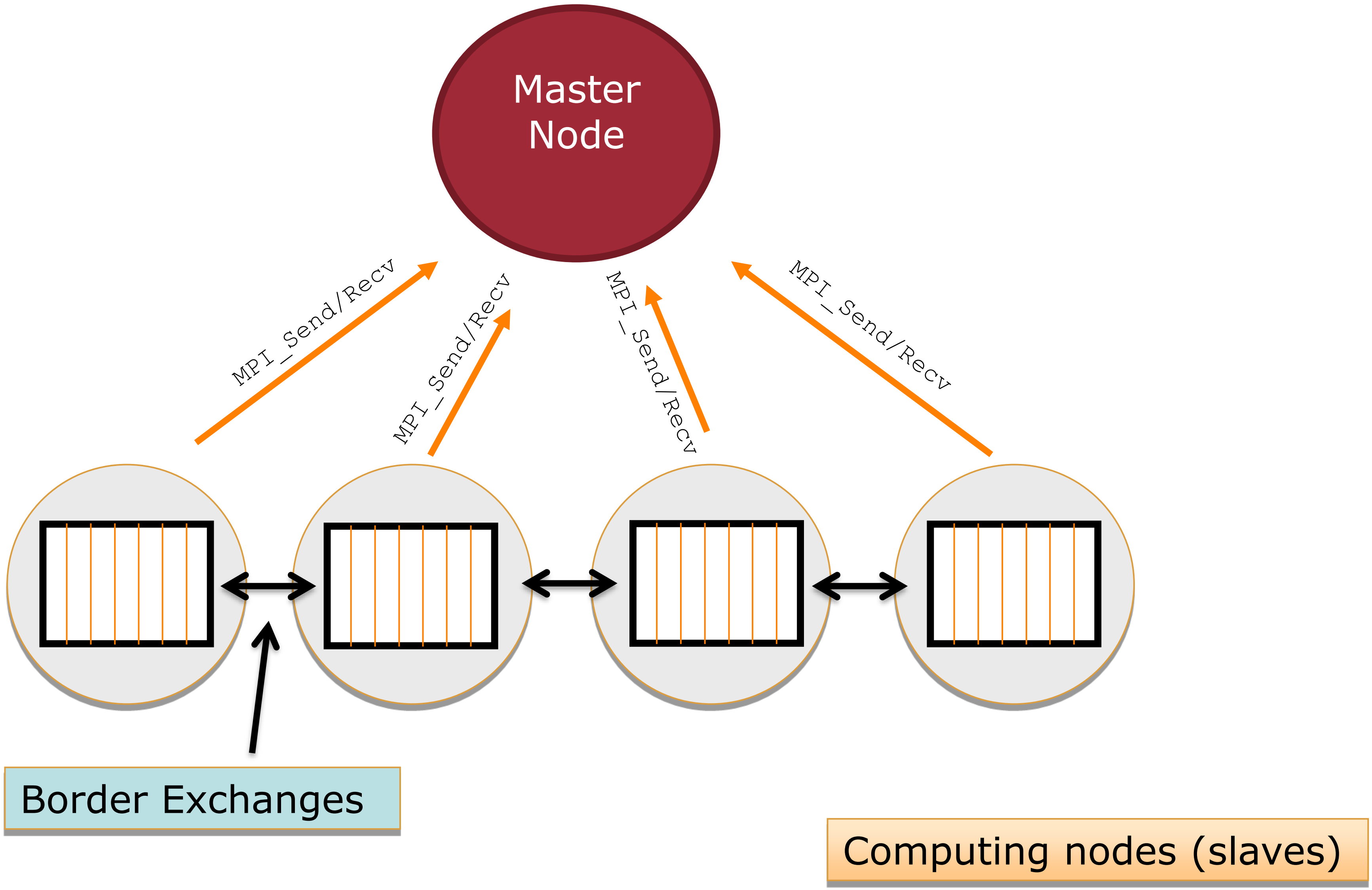
```
for (iy = 1; iy< ny; iy++){
  for (ix = 1; ix< nx; ix++){
    u2(ix, iy) = u1(ix, iy)
      + cx * (u1(ix+1,iy) + u1(ix-1,iy) - 2.*u1(ix,iy))
      + cy * (u1(ix,iy+1) + u1(ix,iy-1) - 2.*u1(ix,iy));
   }
}
```

Parallel Solution 1

- We use the **SPMD** and **Data parallelism** model
- The entire array is **partitioned** and distributed as subarrays to the task. Each task has a portion of the entire array
- We determine data dependencies
  - **Internal elements** that belong to tasks independent of other tasks
  - **Border elements** depend on data elements of the neighbors, so you need to communicate ...
- The **master process** sends initial data to a slave, controls the convergence and collect results
- The **slave processes** compute the solution, indicating when and where necessary, with the neighboring processes


task 0
task 1
....
task n

Master Slave pattern (Heat Equation)



```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time
    send neighbors my border info
    receive from neighbors their border info

    update my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```

Parallel Solution 2: Overlap Communication with Computation

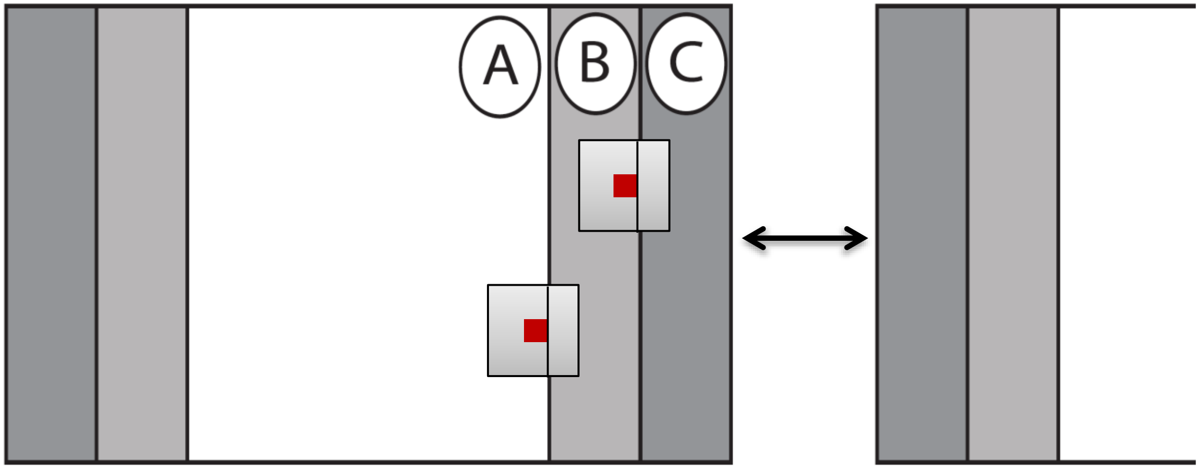
- In the previous solution, it is assumed the use of **blocking communication** for the slaves. Blocking Communications wait for the communicating process the "completion" before executing the next instruction

- In the previous solution, the neighbor processes **BEFORE** communicate the border details, **THEN** update their portion of the array

- **Communication times** can be drastically reduced (complicating the code though!) through the use of non-blocking communications. Non-blocking communications allow the execution of computation WHILE communication is in progress

- In this second solution, each process updates the internal part of its own array while the communication of the board is in place, updating its border portion AFTER that the communication is completed.

Parallel Solution 2: Overlap Communication with Computation



Computing Node

(A) Neighbourhood cells can fall in B

(B) Neighbourhood cells can fall in C

(C) Border replica

Execution loop

```
For each step {
    SendBorder()
    For each cell c in A {
        N(c)=getNeighbourhood()
        S(c)=σ(c,N(c))
    }
    ReceiveBorder()
    For each cell c in B{
        N(c)=getNeighbourhood()
        S(c)=σ(c,N(c))
    }
}
```

```
find out if I am MASTER or WORKER

if I am MASTER
  initialize array
  send each WORKER starting info and subarray

  do until all WORKERS converge
    gather from all WORKERS convergence data
    broadcast to all WORKERS convergence signal
  end do

  receive results from each WORKER

else if I am WORKER
  receive from MASTER starting info and subarray

  do until solution converged
    update time

    non-blocking send neighbors my border info
    non-blocking receive neighbors border info

    update interior of my portion of solution array
    wait for non-blocking communication complete
    update border of my portion of solution array

    determine if my solution has converged
      send MASTER convergence data
      receive from MASTER convergence signal
  end do

  send MASTER results

endif
```