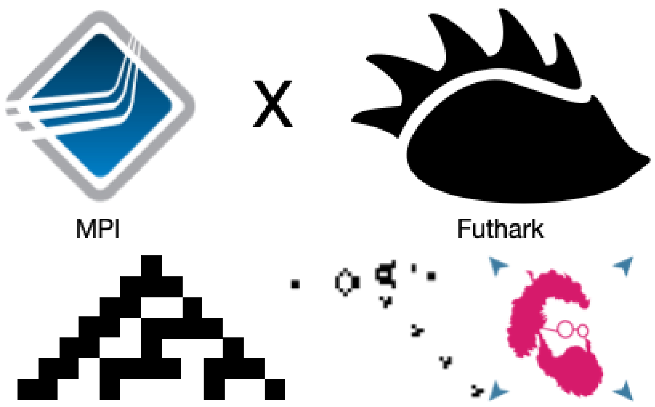
MPI-Futhark: Distributed High-Performance Computing For People



Bachelor thesis defended by

Baptiste Coudray

Information technologies engineering with a specialisation in Software and Complex Systems

August 2021

Referent HES teacher

Dr. Orestis Malaspinas

Legend and source of the cover picture: MPI x Futhark with examples Realized by Baptiste Coudray

Contents

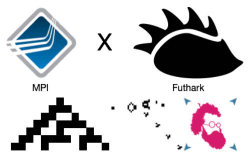
Acknowledgements

I would like to thank the people who helped me during this project:

- Dr. Orestis Malaspinas, for his supervision and help on the project,
- Michaël El Kharroubi, for his help on the project,
- Dr. Troels Henriksen, for his answers to my questions about Futhark,
- Yann Sagon, for his answers to my questions about Baobab/Yggdrasil,
- Theo Pirkl, for the model of the bachelor thesis.

Abstract

Today, most computers are equipped with Graphics Processing Units (GPUs). They provide more and more computing cores and have become fundamental embedded high-performance computing tools. In this context, the number of applications taking advantage of these tools seems low at first glance. The problem is that the development tools are heterogeneous, complex, and strongly dependent on the GPU running the code. Futhark is an experimental, functional, and architecture agnostic language; that is why it seems relevant to study it. It allows generating code allowing a standard sequential execution (on a single-core processor), on GPU (with Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) backends), on several cores of the same processor (shared memory). To make it a tool that could be used on all high-performance platforms, it lacks support for distributed computing Message Passing Interface (MPI). We create a library that distributes a cellular automaton on multiple compute nodes through MPI. The update of the cellular automaton is computed via the Futhark language using one of the four available backends (sequential, multicore, OpenCL, and CUDA). In order to test our library, we implement a cellular automaton in one dimension (Simple Cellular Automaton (SCA)), in two dimensions (Game of Life), and three dimensions (Lattice-Boltzmann Method (LBM)). Finally, with the performance tests performed, we obtain an ideal speedup in one and two dimensions with the sequential and multicore backend, but with LBM, we obtain a maximum of x42 with 128 tasks. When using the GPU backends, we obtain an ideal speedup for the three cellular automata. Parallel computing shows better performance compared to sequential or concurrent computing. For example, with the Game of Life, we are up to 15 times faster.



Candidate:
Baptiste COUDRAY
Field of study: Information Technologies Engineering

Referent teacher:
Dr. Orestis MALASPINAS

List of acronyms

API	Application Programming Interface.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
GCC	GNU Compiler Collection.
GPU	Graphics Processing Unit.
HEPIA	Haute École du Paysage, d'Ingénierie et d'Architecture de Genève.
HES-GE	Haute École Spécialisée de GEnève.
I/O	Input/Output.
LBM	Lattice-Boltzmann Method.
MPI	Message Passing Interface.
OpenCL	Open Computing Language.
POSIX	Portable Operating System Interface uniX.
SCA	Simple Cellular Automaton.

List of illustrations

List of tables

Reference of the URLs

URL01	https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg
URL02	https://commons.wikimedia.org/wiki/File:Gustafson.png
URL03	https://commons.wikimedia.org/wiki/File:Elder_futhark.png
URL04	https://commons.wikimedia.org/wiki/File:Gol-blinker1.png
URL05	https://commons.wikimedia.org/wiki/File:Gol-blinker2.png

Introduction

Today, most computers are equipped with GPUs. They provide more and more computing cores and have become fundamental embedded high-performance computing tools. In this context, the number of applications taking advantage of these tools seems low at first glance. The problem is that the development tools are heterogeneous, complex, and strongly dependent on the GPU running the code. Futhark is an experimental, functional, and architecture agnostic language; that is why it seems relevant to study it. It allows generating code allowing a standard sequential execution (on a single-core processor), on GPU (with CUDA and OpenCL backends), on several cores of the same processor (shared memory). To make it a tool that could be used on all high-performance platforms, it lacks support for distributed computing. This work aims to develop a library that can port any Futhark code to an MPI library with as little effort as possible.

To achieve that, we introduce the meaning of distributed high-performance computing, then what is MPI and Futhark. The MPI specification allows doing distributed computing, and the programming language Futhark allows doing high-performance computing by compiling our program in OpenCL, CUDA, multicore, and sequential. We decide to implement a library that can distribute cellular automaton in, one, two or three dimensions. By adding Futhark on top of MPI, the programmer will have the possibilities to compile his code in:

- distributed-sequential mode,
- distributed-multicore mode,
- distributed-OpenCL mode,
- distributed-CUDA mode.

Finally, we used this library by implementing a cellular automata in each dimension:

- a SCA in one dimension,
- the Game of Life in two dimensions,
- the LBM in three dimensions.

We perform a benchmark to ensure that each cellular automata scales correctly in the four modes.

The leading resources we used to carry out this project were Futhark and MPI user guide. We also exchanged with Futhark creator Troels Henriksen.

Working method

During this project, we use Git and put the source code on the Gitlab platform of Haute École du Paysage, d'Ingénierie et d'Architecture de Genève (HEPIA):

- Source code of the library with usage examples
  - https://gitedu.hesge.ch/baptiste.coudray/projet-de-bachelor
- Source code of this report
  - https://gitedu.hesge.ch/baptiste.coudray/projet-de-semestre/-/tree/report

Chapter 1:

Distributed High-Performance Computing

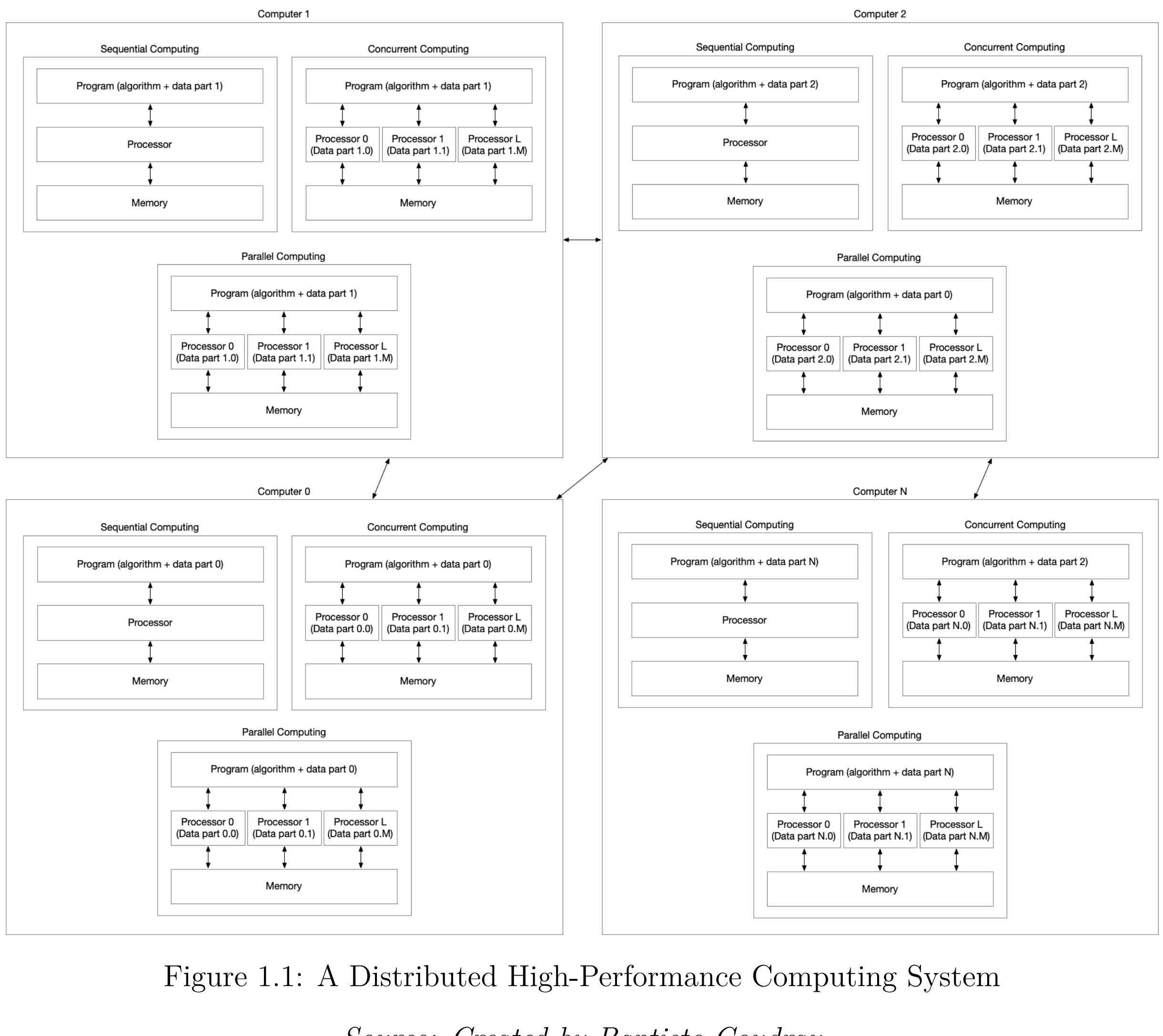

Computer 1
Sequential Computing
Program (algorithm + data part 1)
Processor
Memory
Concurrent Computing
Program (algorithm + data part 1)
Processor 0 (Data part 1.0)
Processor 1 (Data part 1.1)
Processor L (Data part 1.M)
Memory
Parallel Computing
Program (algorithm + data part 1)
Processor 0 (Data part 1.0)
Processor 1 (Data part 1.1)
Processor L (Data part 1.M)
Memory
Computer 2
Sequential Computing
Program (algorithm + data part 2)
Processor
Memory
Concurrent Computing
Program (algorithm + data part 2)
Processor 0 (Data part 2.0)
Processor 1 (Data part 2.1)
Processor L (Data part 2.M)
Memory
Parallel Computing
Program (algorithm + data part 0)
Processor 0 (Data part 2.0)
Processor 1 (Data part 2.1)
Processor L (Data part 2.M)
Memory
Computer 0
Sequential Computing
Program (algorithm + data part 0)
Processor
Memory
Concurrent Computing
Program (algorithm + data part 0)
Processor 0 (Data part 0.0)
Processor 1 (Data part 0.1)
Processor L (Data part 0.M)
Memory
Parallel Computing
Program (algorithm + data part 0)
Processor 0 (Data part 0.0)
Processor 1 (Data part 0.1)
Processor L (Data part 0.M)
Memory
Computer N
Sequential Computing
Program (algorithm + data part N)
Processor
Memory
Concurrent Computing
Program (algorithm + data part 2)
Processor 0 (Data part N.0)
Processor 1 (Data part N.1)
Processor L (Data part N.M)
Memory
Parallel Computing
Program (algorithm + data part N)
Processor 0 (Data part N.0)
Processor 1 (Data part N.1)
Processor L (Data part N.M)
Memory

Figure 1.1: A Distributed High-Performance Computing System

Source: Created by Baptiste Coudray

As we can see in 1.1, distributed systems are groups of networked computers that share a common goal. They are used to increasing computing power and solve a complex problem faster than a single machine (1). In order to do that the problem's data are divided along each computer which can be done by communicating with each other via message passing. Each computer executes the same program (which is a distributed program) but on a different data. The algorithm is applied using one of this three computing methods:

1. sequential computing,
2. concurrent computing,
3. or parallel computing.

With sequential computation, the algorithm is executed step by step, each operation is triggered only when the previous operation is completed, even when the two operations are independent.
With concurrent computing, the problem's data are once again split into smaller parts in order to be shared with the threads available on the processor. Each thread applies independently with time-slicing the algorithm on his set of data. A performance gain is noticeable when tasks are most independent of others because they do not have to wait for the progress of another task (thread) (2).
With parallel computing, data are also split again and the algorithm is applied simultaneously on the multiple processors available. Generally, we use the GPU because it contains a thousand cores while a Central Processing Unit (CPU) contains only a hundred. Thus, the primary goal of parallel computing is to increase available computation power for faster application processing and problem-solving.

So, *Distributed High-Performance Computing* means distributing a program on multiple networked computers and executing the algorithm using sequential, concurrent or parallel computing.

Chapter 2:

Message Passing Interface

In order to realize distributed programming, the standard MPI was created in 1993-1994 to standardize the passage of messages between several computers or in a computer with several processors/cores (3). MPI is, therefore, a communication protocol and not a programming language. Currently, the latest version of MPI is 4.0 which approved in 2021. There are several implementations of the standard:

- MPICH, which support for the moment, MPI 3.1,
- Open MPI, which support, for the moment, MPI 3.1

We use Open MPI throughout this project to distribute a cellular automaton across multiple compute nodes and exchange missing neighbors of a cell.

2.1. Example by imitating a token ring network

To understand the basis of MPI, let us look at an example mimicking a *token ring* network (4). This type of network forces a process to send a message to the message in the console, for example, only if it has the token in its possession. Moreover, once it has emitted its message, the process must transmit the token to its neighbor.

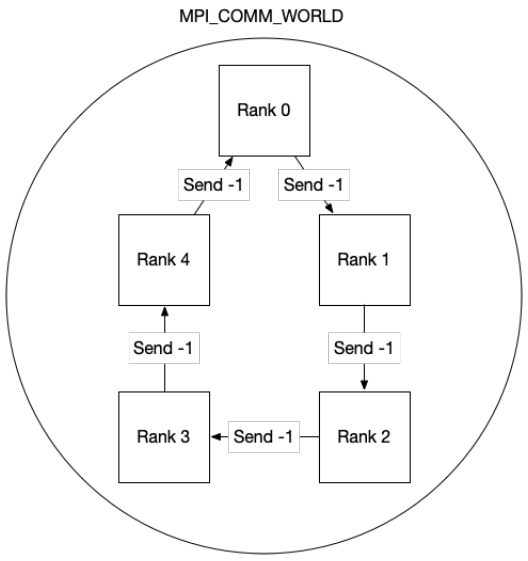


Figure 2.1: Imitation of a network in *token ring*

Source: Created by Baptiste Coudray

In this example (2.1), the node with the rank zero has first the token that it will pass to node one, then it will give it to node two, and so on. The program ends when the token is back in possession of the process zero: node four sends the token to node zero.

```
int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Find out rank, size
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int token;
    // Receive from the lower process and send to the higher process. Take care
    // of the special case when you are the first process to prevent deadlock.
    if (world_rank != 0) {
        MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Process %d received token %d from process %d\n",
               world_rank, token, world_rank - 1);
    } else {
        // Set the token's value if you are process 0
        token = -1;
    }
    MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size, 0,
    MPI_COMM_WORLD);
    // Now process 0 can receive from the last process. This makes sure that at
    // least one MPI_Send is initialized before all MPI_Recvs (again, to prevent
    // deadlock)
    if (world_rank == 0) {
        MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
        printf("Process %d received token %d from process %d\n", world_rank,
               token, world_size - 1);
    }
    return MPI_Finalize();
}
```

Any parallel program using MPI must call the `MPI_Init` function to initialize the environment, otherwise, an error message is displayed when using another MPI function. Then, `MPI_Comm_rank` allows us to retrieve our ID (the node number we have), and then with the `MPI_Comm_size` function, we get the number of nodes on which our program is running, in this case five. Thanks to the node number, the node with the identifier zero, sends the token to its neighbor via the `MPI_Send` function.

So, once sent, it waits for node four to send the token via the function `MPI_Recv`. Then, the other nodes are waiting to receive the token from their neighbor to pass the token in turn. The nodes communicate through the communicator `MPI_COMM_WORLD`, a macro-constant designating all nodes associated with the current program.

The function `MPI_Send` and `MPI_Recv` can be blocking or non-blocking, depending on the size of the message that we send. Indeed, there is an internal message buffer; while it is not full, the functions will be non-blocking; otherwise, they will be blocking.

Finally, every program must terminate with the `MPI_Finalize` function; otherwise, the execution ends with an error message.

```
mpicc ring.c -o ring
mpirun -n 5 ./ring
```

To compile a MPI program, you have to go through the `mpicc` program, which is a wrapper around GNU Compiler Collection (GCC). Indeed, `mpicc` automatically adds the correct compilation parameters to the GCC program. Next, our compiled program must be run through `mpirun` to distribute our program to compute nodes. Finally, the `-n` parameter is used to specify the number of processes to run.

```
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 4 received token -1 from process 3
Process 0 received token -1 from process 4
```

Thus, we can see that the processes exchange the token each in turn until node zero receives the token again.

Chapter 3:

Introduction to the language Futhark

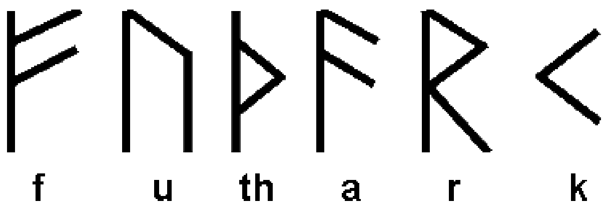


Figure 3.1: Futhark

Source: Taken from https://commons.wikimedia.org/, ref. URL03

Futhark is a purely functional programming language for producing parallelizable code on CPU or GPU. It was designed by Troels Henriksen, Cosmin Oancea and Martin Elsman at the University of Copenhagen. The main goal of Futhark is to write generic code that can compile into either:

- OpenCL,
- CUDA,
- multi-threaded Portable Operating System Interface uniX (POSIX) C,
- sequential C,
- sequential Python.

Although a Futhark code can compile into an executable, this feature reserves for testing purposes because there is no Input/Output (I/O). Thus, the main interest is to write particular functions that you would like to speed up thanks to parallel programming and compile in library mode. In this mode, the compiler convert our Futhark code in sequential C, multi-threaded POSIX C, OpenCL, or CUDA code depending on which backend we want.

3.1. Example in pure Futhark

To better understand Futhark, here is a simple example: calculating the factorial of a number (5).

```
let fact (n: i32): i32 = reduce (*) 1 (1...n)
let main (n: i32): i32 = fact n
```

As we can see, the function `fact` defines the factorial of a number as the successive multiplication of numbers from one to `n` through the function `reduce`. The program's entry point, `main`, takes as parameter a number `n` and calls the function `fact` with this number.

```
futhark opencl fact.fut
echo 12 | ./fact
```

To compile the Futhark code, we have to specify a backend; Here we compile in OpenCL to run the program on the graphics card, and we run the program with the number 12 as the parameter.

```
479001600i32
```

The program calculates the factorial of 12 and therefore returns 479 001 600.

3.2. Example using Futhark C API

In this other example, we use Futhark in a C program to perform a very specific operation, in this case to calculate the factorial of a number. We use the library mode of the Futhark compiler, and we use the OpenCL backend to convert the Futhark code.

```
entry fact (n: i32): i32 = reduce (*) 1 (1...n)
```

In a Futhark file (`fact.fut`), we define the function `fact` like the previous example. In library mode, the compiler convert all functions preceded with the keyword `entry`. Thus, the `fact` function is respecting this rule in order to use it in a C code.

```
futhark opencl --lib fact.fut
```

Then you have to convert the Futhark code in library mode and specify the backend. Here, the factorial program is converted in OpenCL. Finally, it generates a `fact.h` and `fact.c` file, which can be included in a C program.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include "fact.h"

int main(int argc, char **argv) {
    int number = atoi(argv[1]);

    /* Futhark Init */
    struct futhark_context_config *futcfg = futhark_context_config_new();
    struct futhark_context *futctx = futhark_context_new(futcfg);

    /* Main Code */
    int result;
    futhark_entry_fact(futctx, &result, number);
    printf("%d\n", result);

    /* Futhark Free */
    futhark_context_config_free(futcfg);
    futhark_context_free(futctx);
    return 0;
}
```

The program initializes a Futhark configuration and a Futhark context (`futhark_context_config_new`, `futhark_context_new`); then, the program calls the `fact` function, which has been converted in OpenCL. It is called via the function `futhark_entry_fact`, which takes as arguments the Futhark context, an integer pointer to store the result, and the number whose factorial desire.

```
futhark opencl --library fact.fut
gcc fact.c -c
gcc main.c -o fact fact.o -lOpenCL -lm
```

```
./fact 12
479001600
```

After compiling in library mode our Futhark code, we compile the generated code with our main code via GCC. The program's execution with the factorial of 12 returns the correct value, i.e. 479 001 600.

Chapter 4:

Cellular Automaton

A cellular automaton consists of a regular grid of cells, each in one of a finite number of states. The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state (time $t = 0$) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time (6).

The neighborhood of a cell is defined either by the Moore neighborhood or by the Von Neumann neighborhood. The first one defines that a cell has in a two-dimensional cellular automaton four neighbors while the second one, eight.

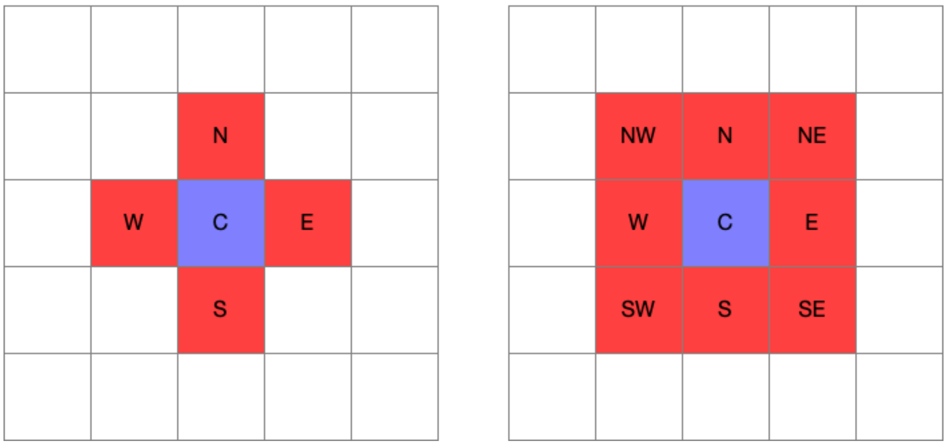


Figure 4.1: Comparison between Von Neumann (left) and Moore (right) neighborhoods

Source: Created by Baptiste Coudray

The grid on the left represents the Von Neumann neighborhood, i.e., the four neighbors of a cell. These are denoted by the four cardinal points (north, west, south, east). The grid on the right represents Moore's neighborhood, i.e., the eight neighbors of a cell. These are denoted by the four cardinal points and the four inter-cardinal points (northwest, southwest, southeast, northeast).

The cellular automaton will have to use the Moore neighborhood, which means that each cell has:

- two neighbors in one dimension,
- eight neighbors in two dimensions,
- 26 neighbors in three dimensions.

These values are valid for a cellular automaton of dimension two and a Chebyshev distance of one. We can generalize the number of neighbors that a cell has via the formula $(2r + 1)^d - 1$, where r is the Chebyshev distance, and d is the dimension.

4.1. MPI x Futhark

Our library allows distributing cellular automata automatically so that the programmer only has to write the Futhark function to update his cellular automaton. Our library supports cellular automata of one, two, and three dimensions and with any types of data. The use of the Futhark language allows to quickly update the state of the cellular automaton thanks to the different backend available. Therefore, several modes are available:

- distributed-sequential, the Futhark code executes sequentially,
- distributed-multicore, the Futhark code executes concurrently to POSIX threads,
- distributed-OpenCL/CUDA, the Futhark code executes on the graphics card.

a) Communication

Communication between the different MPI tasks is necessary to recover the missing neighbors and recreate the complete cellular automaton. Therefore, we create a virtual Cartesian topology. "*A virtual topology is a mechanism for naming the processes in a communicator in away that fits the communication pattern better. The main aim of this is to make sub-sequent code simpler. It may also provide hints to the run-time system which allow it to optimise the communication or even hint to the loader how to configure the processes. The virtual topology might also gain us some performance benefit.*" (7)

One dimension

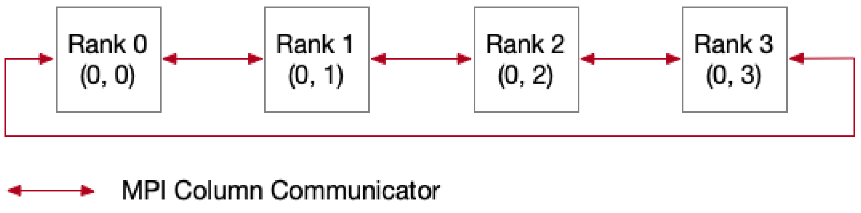


Figure 4.2: Example of Cartesian virtual topology in one dimension

Source: Created by Baptiste Coudray

In a one-dimensional Cartesian topology like 4.2, we notice that the rows can communicate directly with their left and right neighbors even if they are at the ends of the network. Indeed, the MPI communicator is defined as cyclic, which avoids having to traverse the $N - 2$ neighbors that separate them.

Two dimensions

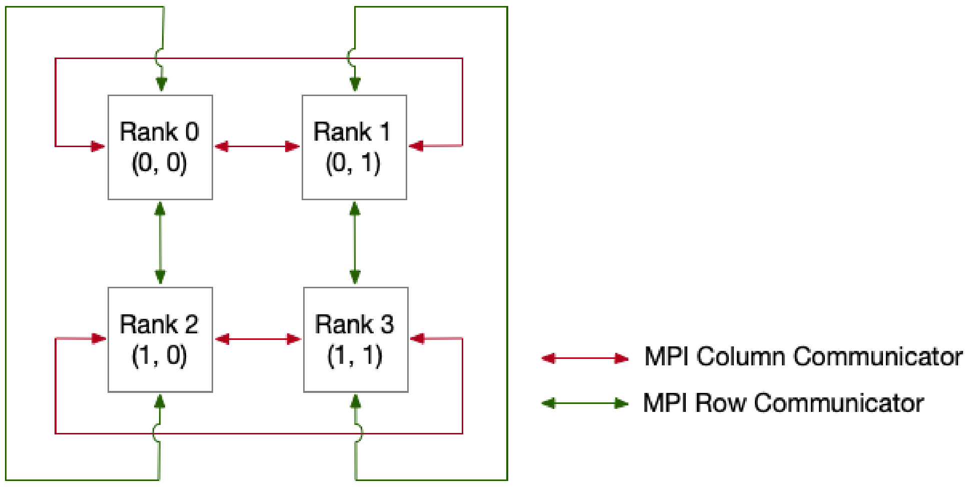


Figure 4.3: Example of Cartesian virtual topology in two dimensions

Source: Created by Baptiste Coudray

In a two-dimensional Cartesian topology like 4.3, we notice that each rank can communicate directly with their left, right, top, and bottom neighbors. For each row there is an MPI communicator containing the ranks available in it:

- first communicator has `Rank 0` and `Rank 1`,
- second communicator has `Rank 2` and `Rank 3`.

For each column, there is an MPI communicator containing the ranks available in it:

- first communicator has `Rank 0` and `Rank 2`,
- second communicator has `Rank 1` and `Rank 3`.

Each communicator created is cyclic, so the first rank in each the communicator created can communicate with the last rank available in it.

When a rank needs to communicate with its diagonal neighbor, we use the default communicator (`MPI_COMM_WORLD`) to communicate directly with each other without going through a neighbor.

b) Three dimensions

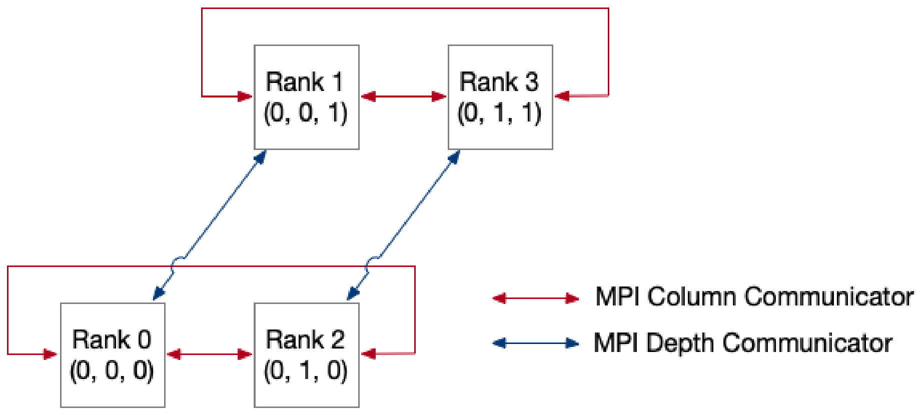


Figure 4.4: Example of Cartesian virtual topology in three dimensions

Source: Created by Baptiste Coudray

In a three-dimensional Cartesian topology like 4.4, we notice that the rows have the same communication capabilities as a two-dimensional topology, but, in addition, they can communicate with their front and back neighbors. Thus, two depth communicators are created:

- one containing `Rank 0` and `Rank 1`,
- one containing `Rank 2` and `Rank 3`.

c) Data dispatching

The cellular automaton is shared as equally possible among the available tasks to perform more or less the same amount of work. Thus, each task has a chunk that is a part of the cellular automaton. This chunk match the dimension of the cellular automaton, so it can be of dimensions one, two, or three.

One dimension

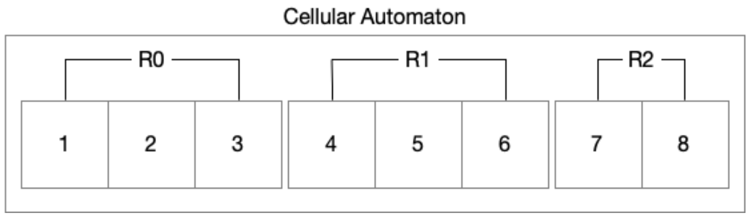


Figure 4.5: Example of sharing a cellular automaton in one dimension

Source: Created by Baptiste Coudray

In this example (4.5), a cellular automaton of dimension one, size eight, is split between three processes. As the division of the cellular automaton is not an integer, rank two have only two cells, unlike the others, which have three.

Two dimensions

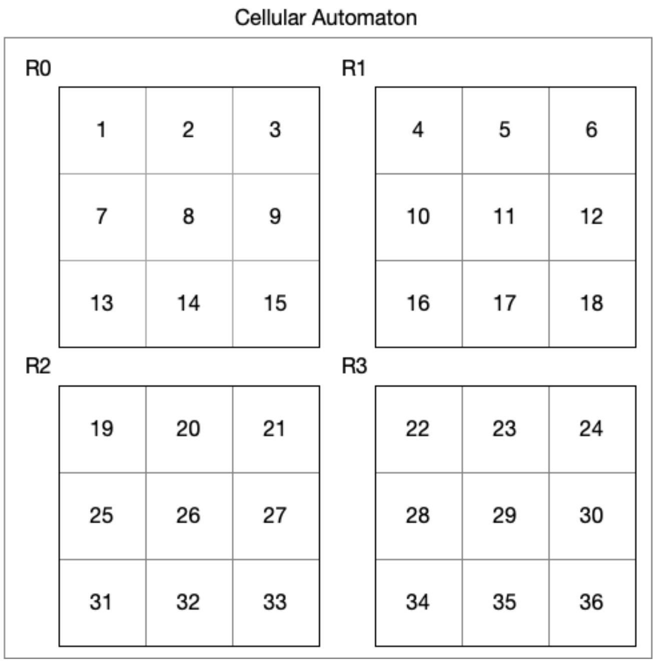


Figure 4.6: Example of sharing a cellular automaton in two dimensions

Source: Created by Baptiste Coudray

In this example (4.6), the cellular automaton is in two dimensions and of size 6×6. With four tasks available, it can be separated into four sub-matrices of 3×3.

Three dimensions

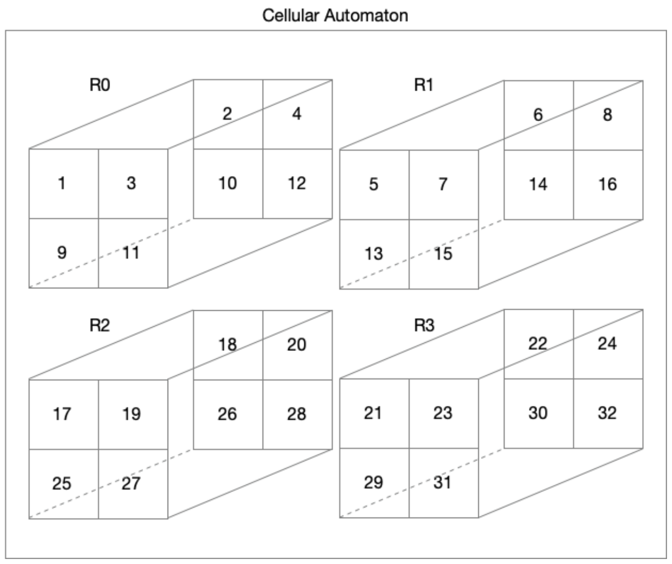


Figure 4.7: Example of sharing a cellular automaton in three dimension

Source: Created by Baptiste Coudray

In this example (4.7), the cellular automaton is in three dimensions and of size $4 \times 4 \times 2$. With four tasks available, it can be separated into four sub-cubes of $2 \times 2 \times 2$.

d) Envelope

The envelope of a chunk represents the missing neighbours (at a Chebyshev distance chose by the programmer) of the cells at the extremities of the chunk. These missing cells are needed to compute the next iteration of the chunk of the cellular automaton that the process has.

One dimension

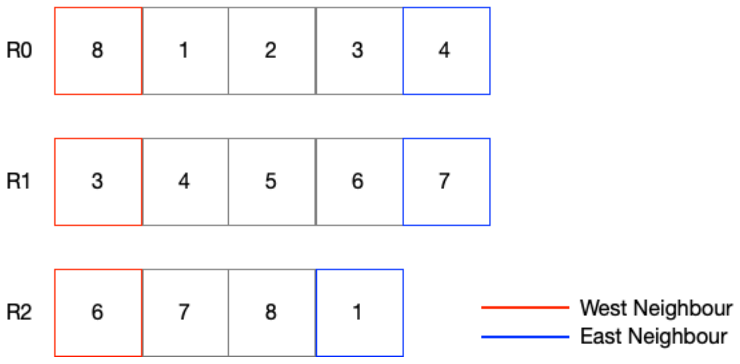


Figure 4.8: Example of the envelope of a chunk in one dimension

Source: Created by Baptiste Coudray

Using this one dimension cellular automaton (4.5), the Moore neighborhood of a cell includes the west-neighbor and the east-neighbor. We notice that the envelope of R_n includes the last cell of $R_{(n-1)\,\%\,N}$ and the first cell of $R_{(n+1)\,\%\,N}$.

For example, the envelope for R0 is the west-neighbor of one, so eight and the east neighbor of three, so four. The envelope for R1 is the west-neighbor of four, so three and the east neighbor of six, so seven. Finally, the envelope for R2 is the west-neighbor of seven, so six and the east neighbor of eight, so one. As we can see, the Chebyshev distance of all these envelopes is one. The ranks exchange data via MPI using the Cartesian virtual topology.

Two dimensions

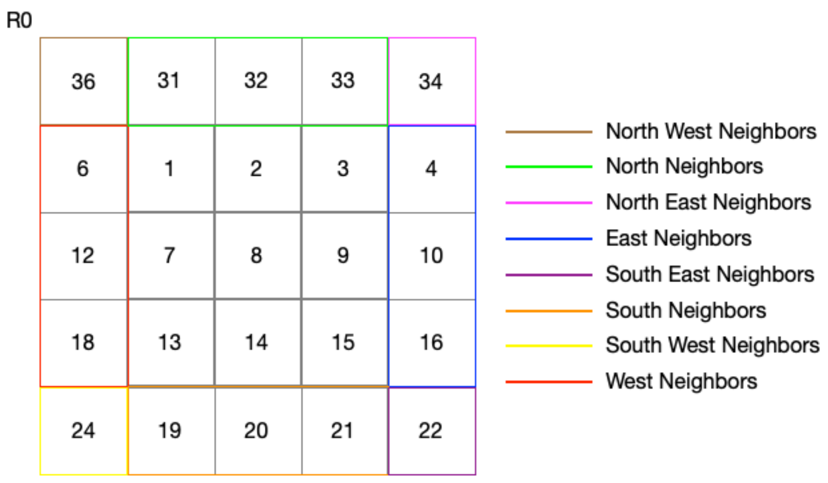


Figure 4.9: Example of the envelope of a chunk in two dimensions

Source: Created by Baptiste Coudray

Using the two-dimensional cellular automaton described above (4.6), the chunk envelope of R0 requires eight communications. This example uses a Cartesian topology of size 2×2 (m×n), the neighbors are recovered as follows:

1. `North West Neighbors` are sent by $R_{((y-1)\,\%\,m,\,(x-1)\,\%\,n)}$,
2. `North Neighbors`, are sent by $R_{((y-1)\,\%\,m,\,x)}$,
3. `North East Neighbors`, are sent by $R_{((y-1)\,\%\,m,\,(x+1)\,\%\,n)}$,
4. `East Neighbors`, are sent by $R_{(y,\,(x+1)\,\%\,n)}$,
5. `South East Neighbors`, are sent by $R_{((y+1)\,\%\,m,\,(x+1)\,\%\,n)}$,
6. `South Neighbors`, are sent by $R_{((y+1)\,\%\,m,\,x)}$,
7. `South West Neighbors`, are sent by $R_{((y+1)\,\%\,m,\,(x-1)\,\%\,n)}$,
8. `West Neighbors`, are sent by $R_{(y,\,(x-1)\,\%\,n)}$.

As we can see, the Chebyshev distance of this envelope is one.

Three dimensions

With a three-dimensional cellular automaton like (4.7, the envelope of a chunk requires 26 MPI communications because we send:

- the eight vertices,
- the twelve edges,
- and the six faces around the chunk.

Chapter 5:

Simple Cellular Automaton

The simplest non-trivial cellular automaton that can be conceived consists of a one-dimensional grid of cells that can take only two states (“0” or “1”), with a neighborhood consisting, for each cell, of itself and the two cells adjacent to it (6).

There are $2^3 = 8$ possible configurations (or patterns, rules) of such a neighborhood. In order for the cellular automaton to work, it is necessary to define what the state must be at the next generation of a cell for each of these patterns. The eight rules/configurations defined is as follows:

Table 5.1: Evolution rules for a cellule in a one dimensional cellular-automaton

Rule n°	East neighbour state	Cell state	West neighbour state	Cell next state
1	0	0	0	0
2	0	0	1	1
3	0	1	0	1
4	0	1	1	1
5	1	0	0	1
6	1	0	1	0
7	1	1	0	0
8	1	1	1	0

5.1. Example

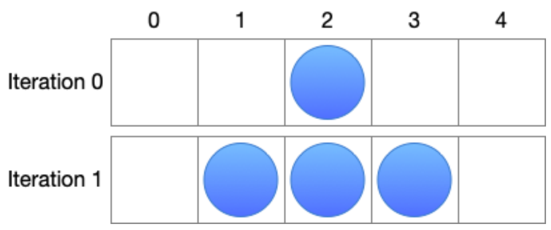


Figure 5.1: First and second state of a SCA

Source: Created by Baptiste Coudray

Iteration 0 is the initial state and only cell two is alive. To perform the next iteration:

- the cell (one) is born because of rule n°2,
- the cell (two) stays alive because of rule n°3,
- the cell (three) stays alive because of rule n°6.

5.2. Distributed version

With the created library, we implement this SCA previously described. To do this, we create a Futhark `elementary.fut` file, which is used to calculate the next state of a part of the cellular automaton.

```
let get_neighbours [n] (elem: [n]i8) : [n](i8,i8) = ...

entry next_chunk_elems [n] (chunk_elems :[n]i8) :[]i8 =
    let neighbours = get_neighbours chunk_elems
    let next_elems = ...
    in next_elems[1:n-1]
```

Therefore, the `elementary.fut` file contains only a function that applies the rules on the cellular automaton. As we can see, `next_chunk_elems` is the primary function that takes a chunk of the cellular automaton as a parameter. This function applies the rules defined before on every cell and returns the new value of each cell without the envelope.

```
void init_chunk_elems(chunk_info_t *ci) {
    int8_t *data8 = ci->data;
    for (int i = 0; i < ci->dimensions[1]; ++i) {
        data8[i] = rand() % 2;
    }
}
```

```c
void compute_next_chunk_board(struct dispatch_context *dc,
        struct futhark_context *fc, chunk_info_t *ci) {
    struct futhark_i8_1d *fut_chunk_with_envelope =
            get_chunk_with_envelope(dc, fc, 1, futhark_new_i8_1d);

    struct futhark_i8_1d *fut_next_chunk_elems;
    futhark_entry_next_chunk_elems(fc, &fut_next_chunk_elems,
                                   fut_chunk_with_envelope);
    futhark_context_sync(fc);

    futhark_values_i8_1d(fc, fut_next_chunk_elems, ci->data);
    futhark_context_sync(fc);

    /* ... Free resources ... */
}

int main(int argc, char *argv[]) {
    /* ... MPI & Futhark Init ... */
    const int N_ITERATIONS = 100;
    int elems_dimensions[1] = {600};
    struct dispatch_context *disp_context =
            dispatch_context_new(elems_dimensions, MPI_INT8_T, 1);
    chunk_info_t ci = get_chunk_info(disp_context);
    init_chunk_elems(&ci);

    for (int j = 0; j < N_ITERATIONS; ++j) {
        compute_next_chunk_elems(disp_context, fut_context, &ci);
        int8_t *sca = get_data(disp_context);
    }
    /* ... Free resources ... */
}
```

Finally, a C file `main.c` is needed to create the program's entry point. We initialize the MPI and Futhark environment. Then, our library, via the function `dispatch_context_new`, by specifying the size of the cellular automaton (600), its data type by using a predefined type in MPI (`MPI_INT8_T`), and the number of dimensions (one in this case). It returns a dispatch context that will need to be provided when calling other functions of our library.

The function `get_chunk_info` is called to get the chunk of the cellular automaton attributed to the current rank. Thus, we initiate it with the values that we want with the function `init_chunk_elems`.

In the temporal loop, we update our chunk using the created function `compute_next_chunk_elems`. In this function, we call the Application Programming Interface (API) function `get_chunk_with_envelope` from our library with the following parameters:

- the dispatch context, obtained from `dispatch_context_new`,
- the Futhark context, obtained from `futhark_context_new`,
- the Chebyshev distance,
- and a pointer to a Futhark function that converts a C array to a Futhark array.

Our API function handles the MPI communication to exchange each chunk's missing and needed neighbors to create the envelope of the current process's chunk.

Thus, we call our Futhark function `futhark_entry_next_chunk_elems` to compute the new values. Given that the function is asynchronous, the function `futhark_context_sync` waits for the action to finish. Finally, we retrieve the new values with `futhark_values_i8_1d`.

After all that, it is possible to retrieve the entire cellular automaton on the root node by calling the function `get_data` on each task. The variable `sca` is not `NULL` if it is the root node, and it points to the entire cellular automaton.

5.3. CPU Benchmarks

We perform benchmarks to validate the scalability of our one-dimensional distribution when compiling in sequential, multicore, OpenCL, or CUDA mode. The benchmarks are performed on the Haute École Spécialisée de GEnève (HES-GE) cluster (Baobab/Yggdrasil). The sequential and multicore benchmarks are performed as follows:

- the cellular automaton is $300,000,000$ cells in size,
- the number of tasks varies between 2^0 and 2^7,
- 15 measurements are performed, one measurement corresponds to one iteration,
- the iteration is computed 100 times.

Table 5.2: Results for the distributed-sequential version of SCA

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	657.866 [s]	± 14.977 [s]	x1.0	15
2	332.771 [s]	± 2.814 [s]	x2.0	15
4	161.963 [s]	± 7.309 [s]	x4.1	15
8	87.602 [s]	± 2.918 [s]	x7.5	15
16	42.743 [s]	± 0.039 [s]	x15.4	15
32	20.938 [s]	± 0.007 [s]	x31.4	15
64	11.071 [s]	± 0.024 [s]	x59.4	15
128	5.316 [s]	± 0.191 [s]	x123.7	15

This table contains the results obtained by using the backend `c` of Futhark.

Table 5.3: Results for the distributed-multicore version of SCA

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	708.689 [s]	± 16.036 [s]	x1.0	15
2	358.007 [s]	± 4.037 [s]	x2.0	15
4	138.523 [s]	± 3.773 [s]	x5.1	15
8	71.077 [s]	± 1.280 [s]	x10.0	15
16	34.697 [s]	± 0.834 [s]	x20.4	15
32	25.776 [s]	± 0.725 [s]	x27.5	15
64	12.506 [s]	± 0.554 [s]	x56.7	15
128	5.816 [s]	± 0.045 [s]	x121.8	15

This table contains the results obtained by using the backend `multicore` of Futhark.

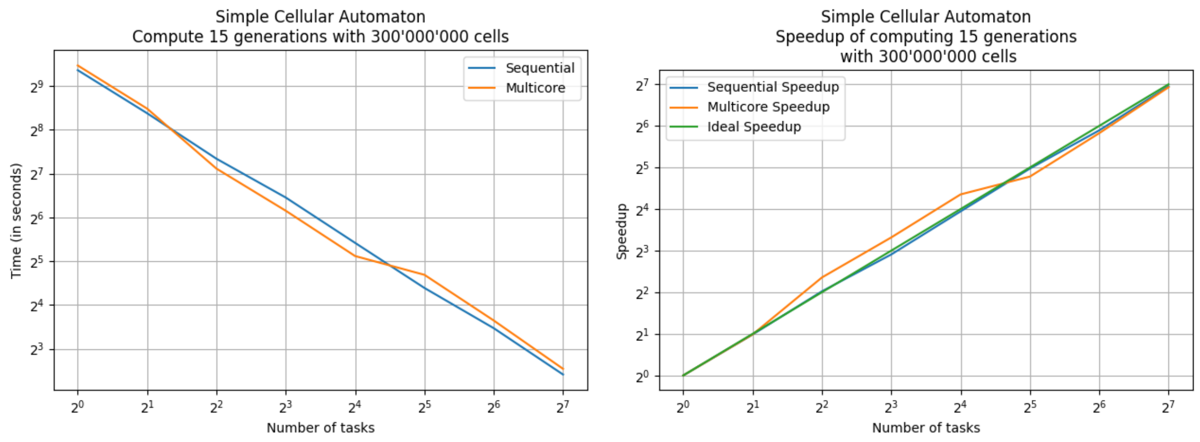


Figure 5.2: Benchmarks of the SCA in distributed-sequential/multicore

Source: Realized by Baptiste Coudray

We compare the average computation time for each number of tasks and each version (sequential and multicore) on the left graph. On the right graph, we compare the ideal speedup with the distributed-sequential and multicore version speedup. The more we increase the number of tasks, the more the execution time is reduced. Thus, the distributed-sequential or multicore version speedup follows the curve of the ideal speedup. We can see that concurrent computing does not provide a significant performance gain over sequential computing because of the overhead of creating threads.

5.4. GPU Benchmarks

The OpenCL and CUDA benchmarks are performed as follows:

- the cellular automaton has 300′000′000 cells,
- the number of tasks varies between 2^0 and 2^3.
- 15 measurements are performed, one measurement corresponds to one iteration,
- the iteration is computed 50′000 times,
- an NVIDIA GeForce RTX 3090 is allocated for each task.

Table 5.4: Results for the distributed-OpenCL version of SCA

Number of tasks	Number of GPUs	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	1	166.086 [s]	± 0.096 [s]	x1.0	15
2	2	83.339 [s]	± 0.099 [s]	x2.0	15
4	4	42.122 [s]	± 0.078 [s]	x3.9	15
8	8	21.447 [s]	± 0.031 [s]	x7.7	15

This table contains the results obtained by using the backend `opencl` of Futhark.

Table 5.5: Results for the distributed-CUDA version of SCA

Number of tasks	Number of GPUs	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	1	160.291 [s]	± 0.062 [s]	x1.0	15
2	2	80.434 [s]	± 0.094 [s]	x2.0	15
4	4	40.640 [s]	± 0.073 [s]	x3.9	15
8	8	20.657 [s]	± 0.046 [s]	x7.8	15

This table contains the results obtained by using the backend `cuda` of Futhark.

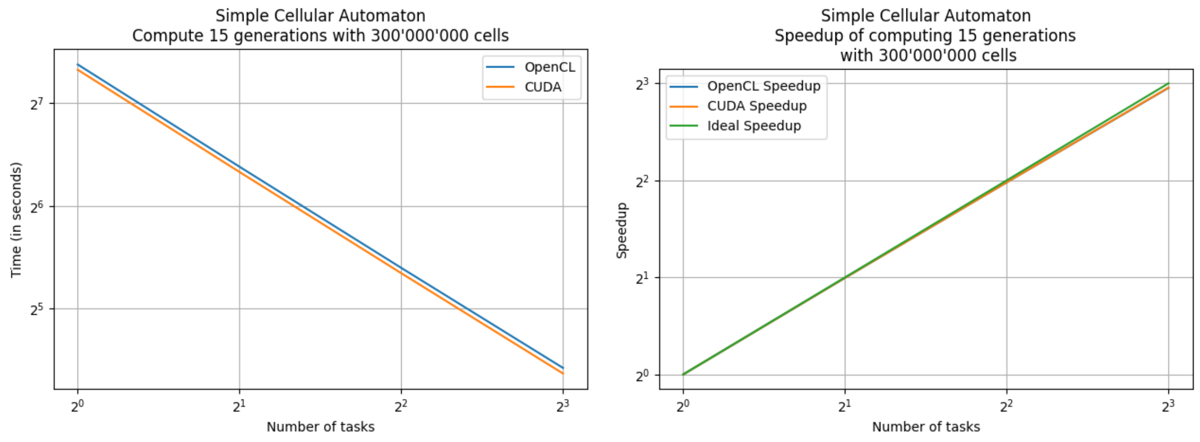


Figure 5.3: Benchmarks of the SCA in distributed-OpenCL/CUDA

Source: Realized by Baptiste Coudray

With this performance test (5.3), we compare the average computation time for each number of tasks/GPUs and each version (OpenCL and CUDA) on the left graph. On the right graph, we compare the ideal speedup with the `distributed-opencl` and cuda version speedup. We notice that the computation time is essentially the same in OpenCL as in CUDA. Moreover, the distributed follows the ideal speedup curve. Finally, we notice that parallel computation is up to four times faster than sequential/concurrent computation when executing with a single task/graphical card.

Chapter 6:

Game of Life

The Game of Life is a zero-player game designed by John Horton Conway in 1970. It is also one of the best-known cellular automata. The game does not require the interaction of a player for it to evolve, it evolves thanks to these extremely simple rules:

1. a cell has eight neighbors,
2. a cell can be either alive or dead,
3. a dead cell with exactly three living neighbors becomes alive,
4. a living cell with two or three living neighbors stays alive; otherwise, it dies (8).

6.1. Example with the blinker

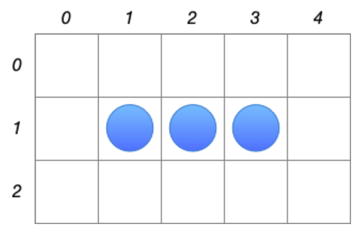


Figure 6.1: First state of blinker

Source: Taken from https://commons.wikimedia.org/, ref. URL04. Re-created by Baptiste Coudray

A basic example is a blinker:

- the cell (one, one) and (one, three) die because they have seven dead neighbors and one living neighbor (rule n°4),
- the cell (zero, two) and (two, two) are born because they have three living neighbors (rule n°3),
- the cell (one, two) stays alive because it has two living neighbors (rule n°4).

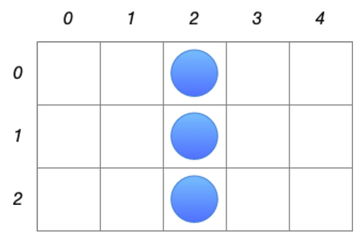


Figure 6.2: Second state of blinker

Source: Taken from https://commons.wikimedia.org/, ref. URL05. Re-created by Baptiste Coudray

Thus, after the application of the rules, the horizontal line becomes a vertical line. Then, at the next iteration, the vertical line becomes a horizontal line again.

6.2. Distributed version

We create the game of life with our library to test it with a two-dimensional cellular automaton. The code is relatively the same as the previous example; therefore, it is not explained, but you can find it in the Git repository.

```
let count_neighbours [n][m] (board: [n][m]i8) :[n][m]i8 = ...
let compute_next_board [n][m] (chunk_board :[n][m]i8)
                              (neighbours: [n][m]i8) :[][]i8 = ...

entry next_chunk_board [n][m] (chunk_board :[n][m]i8) :[][]i8 =
    let neighbours = count_neighbours chunk_board
    let next_board = compute_next_board chunk_board neighbours
    in next_board[1:n-1, 1:m-1]
```

This is a sneak peek of the `gol.fut` file. Like the SCA, we only update our cellular automaton and return it without the envelope.

```
void compute_next_chunk_board(struct dispatch_context *dc,
        struct futhark_context *fc, chunk_info_t *ci) {
    struct futhark_i8_2d *fut_chunk_with_envelope =
            get_chunk_with_envelope(dc, fc, 1, futhark_new_i8_2d);
    struct futhark_i8_2d *fut_next_chunk_board;
    futhark_entry_next_chunk_board(fc, &fut_next_chunk_board,
                                   fut_chunk_with_envelope);
    /* ... Sync, Get values & Free resources ... */
}
```

```c
int main(int argc, char *argv[]) {
    /* ... MPI & Futhark Init ... */
    int board_dimensions[2] = {800, 800};
    struct dispatch_context *disp_context =
            dispatch_context_new(board_dimensions, MPI_INT8_T, 2);
    /* ... */
    const int N_ITERATIONS = 100;
    for (int i = 0; i < N_MEASURES; ++i) {
        compute_next_chunk_board(disp_context, fut_context, &ci);
    }
    /* ... Free resources ... */
}
```

In the C file `main.c`, we can see this almost the same code as for the SCA example, but when calling `dispatch_context_new` we specify that this is a two-dimensional cellular automaton. In the `compute_next_chunk_board` function, we call `get_chunk_with_envelope` from our library with a different conversion function. It transforms a C 2D array to a Futhark 2D array.

6.3. CPU Benchmarks

We perform benchmarks to validate the scalability of our two-dimensional distribution when compiling in sequential, multicore, OpenCL, or CUDA mode. The benchmarks are performed on the HES-GE cluster (Baobab/Yggdrasil).

The sequential and multicore benchmarks are performed as follows:

- the cellular automaton is 900′000′000 cells in size,
- the number of tasks varies between 2^0 and 2^7,
- 15 measurements are performed, one measurement corresponds to one iteration,
- the iteration is computed 100 times.

Table 6.1: Results for the distributed-sequential version of Game of Life

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	3471.723 [s]	± 47.092 [s]	x1.0	15
2	1140.064 [s]	± 56.780 [s]	x3.0	15
4	790.365 [s]	± 10.501 [s]	x4.4	15
8	398.093 [s]	± 13.438 [s]	x8.7	15

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
16	221.687 [s]	± 4.152 [s]	x15.7	15
32	100.422 [s]	± 0.068 [s]	x34.6	15
64	55.986 [s]	± 1.587 [s]	x62.0	15
128	28.111 [s]	± 0.263 [s]	x123.5	15

This table contains the results obtained by using the backend `c` of Futhark.

Table 6.2: Results for the distributed-multicore version of Game of Life

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	2154.686 [s]	± 198.122 [s]	x1.0	15
2	1160.921 [s]	± 77.230 [s]	x1.9	15
4	502.860 [s]	± 3.465 [s]	x4.3	15
8	206.818 [s]	± 4.179 [s]	x10.4	15
16	106.103 [s]	± 0.450 [s]	x20.3	15
32	71.463 [s]	± 0.485 [s]	x30.2	15
64	39.116 [s]	± 0.489 [s]	x55.1	15
128	14.008 [s]	± 0.335 [s]	x153.8	15

This table contains the results obtained by using the backend `multicore` of Futhark.

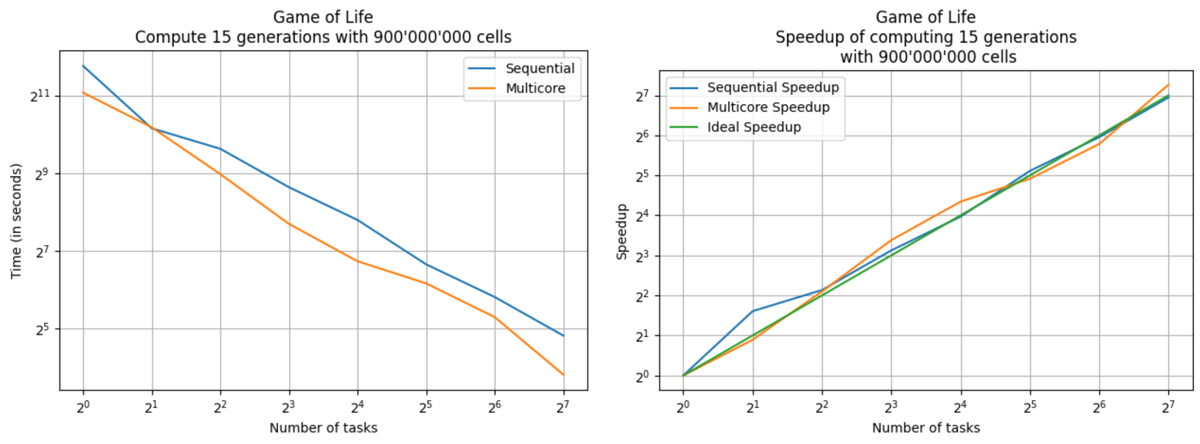


Figure 6.3: Benchmarks of the game of life in distributed-sequential/multicore

Source: Realized by Baptiste Coudray

We notice an apparent difference between the distributed-sequential and multicore version when there is only one task. The multicore version is 1.6 times faster than the sequential version. Nevertheless, both versions have a perfect speedup. The multicore version even gets a maximum speedup of x154 with 128 tasks. This performance can be explained by the caching of data in the processor and the use of threads.

6.4. GPU Benchmarks

The OpenCL and CUDA benchmarks are performed as follows:

- the cellular automaton has 900′000′000 cells,
- the number of tasks varies between 2^0 and 2^3.
- 15 measurements are performed, one measurement corresponds to one iteration,
- the iteration is computed 8′000 times,
- an NVIDIA GeForce RTX 3090 is allocated for each task.

Table 6.3: Results for the distributed-OpenCL version of Game of Life

Number of tasks	Number of GPUs	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	1	230.144 [s]	± 0.225 [s]	x1.0	15
2	2	115.400 [s]	± 0.070 [s]	x2.0	15
4	4	58.019 [s]	± 0.104 [s]	x4.0	15
8	8	29.157 [s]	± 0.061 [s]	x7.9	15

This table contains the results obtained by using the backend `opencl` of Futhark.

Table 6.4: Results for the distributed-CUDA version of Game of Life

Number of tasks	Number of GPUs	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	1	218.807 [s]	± 0.057 [s]	x1.0	15
2	2	109.598 [s]	± 0.109 [s]	x2.0	15
4	4	55.039 [s]	± 0.100 [s]	x4.0	15
8	8	27.737 [s]	± 0.050 [s]	x7.9	15

This table contains the results obtained by using the backend `cuda` of Futhark.

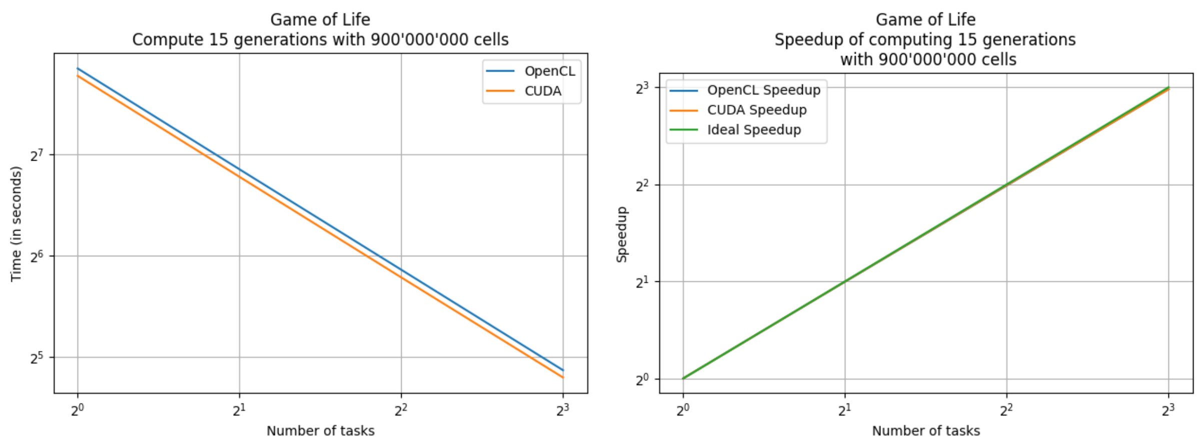


Figure 6.4: Benchmarks of the game of life in distributed-OpenCL/CUDA

Source: Realized by Baptiste Coudray

With this performance test (6.4), we notice that the computation time is essentially the same in OpenCL as in CUDA. Moreover, the distribution follows the ideal speedup curve. Furthermore, we notice that parallel computation is up to 15 times faster than sequential/concurrent computation when executing with a single task/graphical card.

Chapter 7:

Lattice-Boltzmann Method

"*The lattice Boltzmann method (LBM) has established itself in the past decades as a valuable approach to Computational Fluid Dynamics (CFD). It is commonly used to model time-dependent, incompressible or compressible flows in a regime of Direct Numerical Simulation (DNS) or Large Eddy Simulation (LES). One of its strengths lies in the ability to easily represent complex physical phenomena, ranging from multi-phase flows to reactive and suspension flows. The method originates in a molecular description of a fluid, based on the Boltzmann equation, and can directly incorporate physical terms stemming from a knowledge of the interaction between molecules. It is therefore an invaluable tool in fundamental research, as it keeps the cycle between the elaboration of a theory and the formulation of a corresponding numerical model short.*" (9)

7.1. Distributed version

We implement the Lattice-Boltzmann Method with our library to test it with a three-dimensional cellular automaton. The code is almost the same as the previous example, so we focus on custom types in this example. A cell in this cellular automaton is of type of array containing 27 floats. In MPI, there is no predeclared type of array, so we need to create one. Thanks to our library, it is possible to do that with a minimum code.

```c
typedef struct lbm_values {
    float values[NB_VALUES];
} lbm_values_t;

int main(int argc, char *argv[]) {
    /* ... MPI & Futhark Init ... */
    int count = 1;
    int block_lengths[] = {NB_VALUES};
    MPI_Aint displacements[] = {offsetof(struct lbm_values, values)};
    MPI_Datatype types[] = {MPI_FLOAT};

    MPI_Datatype lbm_type = create_type(count, block_lengths, displacements,
                                        types);

    int lbm_dimensions[3] = {400, 400, 400};
    struct dispatch_context *disp_context =
            dispatch_context_new(lbm_dimensions, lbm_type, 3);
    /* ... Temporal loop & Free resources ... */
}
```

We need to declare a structure specifying a cell, so `lbm_values` is a struct containing an array of 27 floats. After that, in the `main` function, we describe the structure as follows:

- there is one field (`values`),
- the first field is containing 27 values,
- we compute where is located the field `values` in the structure,
- `values` is of type `MPI_FLOAT`.

Finally, we call our library function `create_type` with the previous parameters, and it returns an `MPI_Datatype` that can be pass to the function `dispatch_context_new`.

7.2. CPU Benchmarks

We perform benchmarks to validate the scalability of our three-dimensional distribution when compiling in sequential, multicore, OpenCL, or CUDA mode. The benchmarks are performed on the HES-GE cluster (Baobab/Yggdrasil). The sequential and multicore benchmarks are performed as follows:

- the cellular automaton is 27′000′000 cells in size,
- the number of tasks varies between 2^0 and 2^7,
- 15 measurements are performed, one measurement corresponds to one iteration,
- the iteration is computed 100 times.

Table 7.1: Results for the distributed-sequential version of LBM

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	716.133 [s]	± 5.309 [s]	x1.0	15
2	363.166 [s]	± 3.482 [s]	x2.0	15
4	185.430 [s]	± 0.847 [s]	x3.9	15

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
8	93.994 [s]	± 0.566 [s]	x7.6	15
16	81.266 [s]	± 8.947 [s]	x8.8	15
32	41.040 [s]	± 1.590 [s]	x17.4	15
64	22.188 [s]	± 0.321 [s]	x32.3	15
128	17.415 [s]	± 4.956 [s]	x41.1	15

This table contains the results obtained by using the backend `c` of Futhark.

Table 7.2: Results for the distributed-multicore version of LBM

Number of tasks	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	695.675 [s]	± 8.867 [s]	x1.0	15
2	352.925 [s]	± 4.293 [s]	x2.0	15
4	181.736 [s]	± 0.695 [s]	x3.8	15
8	237.983 [s]	± 0.271 [s]	x2.9	15
16	79.360 [s]	± 2.185 [s]	x8.8	15
32	46.285 [s]	± 0.138 [s]	x15.0	15
64	24.059 [s]	± 0.061 [s]	x28.9	15
128	16.614 [s]	± 1.088 [s]	x41.9	15

This table contains the results obtained by using the backend `multicore` of Futhark.

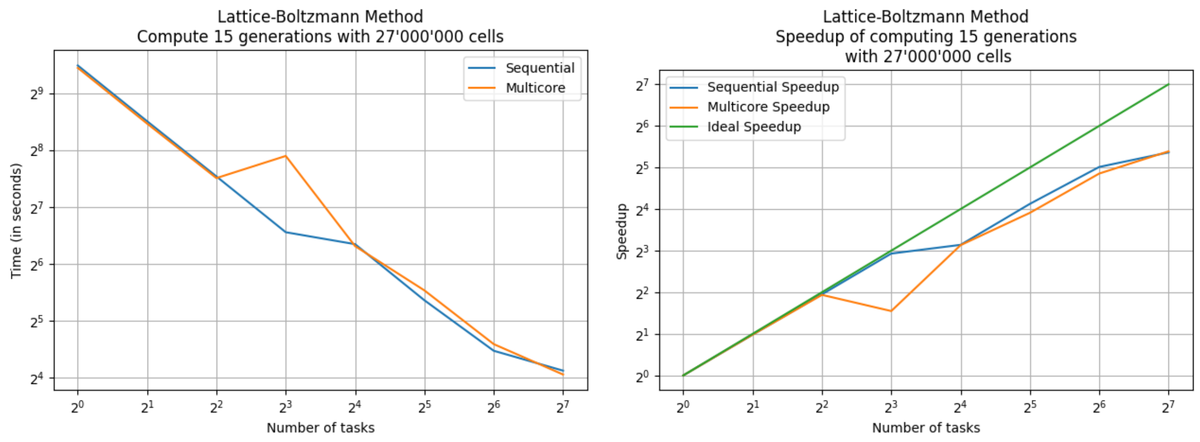


Figure 7.1: Benchmarks of the LBM in distributed-sequential/multicore

Source: Realized by Baptiste Coudray

Contrary to the previous benchmarks, the speedups do not follow the ideal speedup curve. Indeed, whether in sequential or multicore, we obtain a maximum speedup with 128 tasks of x41 when we were hoping to have a speedup of x128.

7.3. GPU Benchmarks

The OpenCL and CUDA benchmarks are performed as follows:

- the cellular automaton has 27′000′000 cells,
- the number of tasks varies between 2^0 and 2^3.
- 15 measurements are performed, one measurement corresponds to one iteration,
- the iteration is computed 3′000 times,
- an NVIDIA GeForce RTX 3090 is allocated for each task.

Table 7.3: Results for the distributed-OpenCL version of LBM

Number of tasks	Number of GPUs	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	1	210.347 [s]	± 0.096 [s]	x1.0	15
2	2	99.677 [s]	± 0.038 [s]	x2.1	15
4	4	40.710 [s]	± 0.076 [s]	x5.2	15
8	8	20.800 [s]	± 0.031 [s]	x10.1	15

This table contains the results obtained by using the backend `opencl` of Futhark.

Table 7.4: Results for the distributed-CUDA version of LBM

Number of tasks	Number of GPUs	Average [s]	Standard Derivation [s]	Speedup	Number of measures
1	1	207.683 [s]	± 0.249 [s]	x1.0	15
2	2	99.177 [s]	± 0.056 [s]	x2.1	15
4	4	40.240 [s]	± 0.074 [s]	x5.2	15
8	8	20.459 [s]	± 0.037 [s]	x10.2	15

This table contains the results obtained by using the backend `cuda` of Futhark.

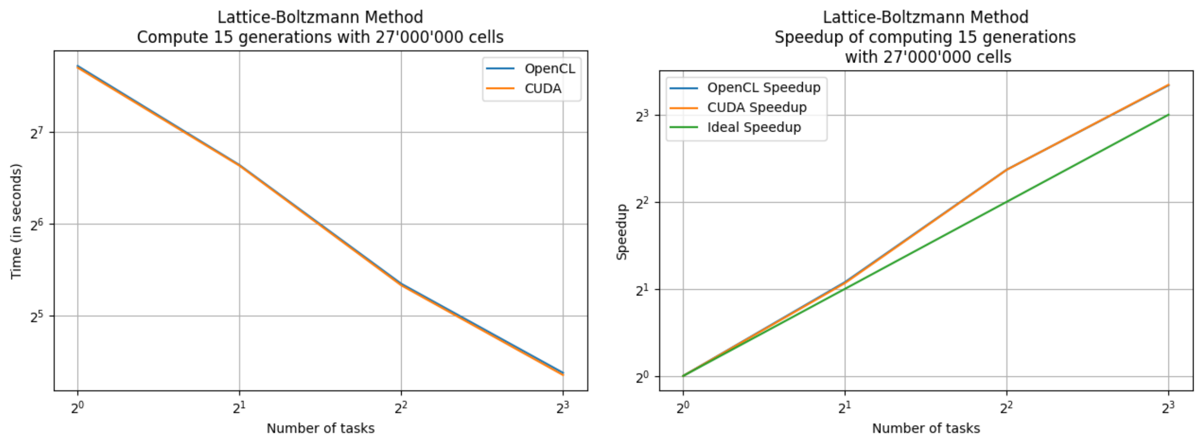


Figure 7.2: Benchmarks of the LBM in distributed-OpenCL/CUDA

Source: Realized by Baptiste Coudray

Like the other benchmarks (5.3, 6.4), there is very little difference between the OpenCL and CUDA versions (computation time and speedup). We get a more than ideal speedup with 2, 4, and 8 tasks/GPUs (x2.1, x5.2, and x10.2, respectively). Finally, we notice that parallel computation is up to 3 times faster than sequential/concurrent computation when executing with a single task/graphical card.

Chapter 8:

Conclusion

In this project, we created a library allowing to distribute a one, two or three dimensional cellular automaton on several computation nodes via MPI. Thanks to the different Futhark backends, the update of the cellular automaton can be done in sequential, concurrent or parallel computation. Thus, we compared these different modes by implementing a cellular automaton in one dimension (SCA), in two dimensions (Game of Life) and in three dimensions (LBM). Benchmarks for each backend were performed to verify the scalability of the library. We obtained ideal speedups with the cellular automata in one and two dimensions and with the use of the sequential and multicore Futhark backend. With these two backends and a three-dimensional cellular automaton, we had a maximum speedup of x41 with 128 tasks. Concerning the OpenCL and CUDA backends, they show no difference in performance between them and for the three cellular automata, the speedup is ideal. Parallel computing has consistently shown better performance compared to sequential or simultaneous computing. For example, with the Game of Life, we are up to 15 times faster.

During this work, I learnt the importance to make unit tests to valid my implementation. Indeed, I was able to narrowing down multiple bugs that I made and make sure that my library was still functioning when I was adding cellular automaton in two and three dimension.

The library can be improved to obtain an ideal speedup in three dimensions with the CPU backends. Moreover, the support of the Von Neumann neighborhood to manage other cellular automata.

Documentary references

1. Distributed computing. *Wikipedia* [online]. 2021. [Accessed on 23 July 2021]. Retrieved from : https://en.wikipedia.org/w/index.php?title=Distributed_computing

2. ORACLE. Chapter 1 covering multithreading basics (multithreaded programming guide). [online]. 2010. [Accessed on 23 July 2021]. Retrieved from : https://docs.oracle.com/cd/E19455-01/806-5257/6je9h0329/index.html

3. Message Passing Interface. *Wikipédia* [online]. 2021. [Accessed on 22 July 2021]. Retrieved from : https://fr.wikipedia.org/w/index.php?title=Message_Passing_Interface

4. KENDALL, Wess. MPI send and receive · MPI tutorial. MPI tutorial. [online]. 18 May 2018. [Accessed on 22 July 2021]. Retrieved from : https://mpitutorial.com/tutorials/mpi-send-and-receive/

5. HENRIKSEN, Troels. Basic usage with the factorial function. The futhark programming language. [online]. 11 April 2021. [Accessed on 22 July 2021]. Retrieved from : https://futhark-lang.org/examples/fact.html

6. Automate cellulaire. *Wikipédia* [online]. 2021. [Accessed on 22 July 2021]. Retrieved from : https://fr.wikipedia.org/w/index.php?title=Automate_cellulaire

7. MACDONALD, Neil, MINTY, Elspeth, HARDING, Tim and BROWN, Simon. Writing message-passing parallel programs with MPI. pp. 92.

8. Jeu de la vie. *Wikipédia* [online]. 2021. [Accessed on 22 July 2021]. Retrieved from : https://fr.wikipedia.org/w/index.php?title=Jeu_de_la_vie

9. LATT, Jonas, MALASPINAS, Orestis, KONTAXAKIS, Dimitrios, PARMIGIANI, Andrea, LAGRAVA, Daniel, BROGI, Federico, BEN BELGACEM, Mohamed, THORIMBERT, Yann, LECLAIRE, Sébastien, LI, Sha, MARSON, Francesco, LEMUS, Jonathan, KOTSALOS, Christos, CONRADIN, Raphaël, COREIXAS, Christophe, PETKANTCHIN, Rémy, RAYNAUD, Franck, BENY, Joel and CHOPARD, Bastien. Palabos: Parallel lattice boltzmann solver. *Computers & Mathematics with Applications.* 1 April 2020. Vol. 81. DOI 10.1016/j.camwa.2020.03.022.