ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Παράλληλα Συστήματα-Παράλληλα Υπολογιστικά Συστήματα

Conway's Game of Life

MPI, MPI/OPENMP, CUDA

Ανδρέας Παππάς 1115201500201

Εργασία 2019-2020

Table Of Contents

Introduction

Conway's Game of Life and cellular automata

The cellular automaton is an important tool in science that can be used to model a variety of natural phenomena. Cellular automata can be used to simulate brain tumor growth by generating a 3-dimensional map of the brain and advancing cellular growth over time [1]. In ecology, cellular automata can be used to model the interactions of species competing for environmental resources [2]. These are just two examples of the many applications of cellular automata in many fields of science, including biology [3][4], ecology [5], cognitive science [6], hydrodynamics [7], dermatology [8], chemistry [9][10], environmental science [11], agriculture [12], operational research [13], and many others.

Cellular automata are so named because they perform functions automatically on a grid of individual units called cells. One of the most significant and important examples of the cellular automaton is John Conway's Game of Life, which first appeared in [15]. Conway wanted to design his automaton such that emergent behavior would occur, in which patterns that are created initially grow and evolve into other, usually unexpected, patterns. He also wanted to ensure that individual patterns within the automaton could dissipate, stabilize, or oscillate. Conway's automaton is capable of producing patterns that can move across the grid (gliders or spaceships), oscillate in place (flip-flops), stand motionless on the grid (still lifes), and generate other patterns (guns).

Conway established four simple rules that describe the behavior of cells in the grid. At each time step, every cell in the grid has one of two particular states: ALIVE or DEAD. The rules of the automaton govern what the state of a cell will be in the next time step.

Like all cellular automata, the rules in Conway's Game of Life pertain to cells and their "neighbors", or the cells to which a cell is somehow related (usually spatially). The collection of a cell's neighbors is known as the cell's "neighborhood". Two examples of neighborhoods are shown in Figure 1. The code in this module uses the latter of these two, the "Moore neighborhood", in which the 8 cells immediately adjacent to a cell constitute the cell's neighbors.

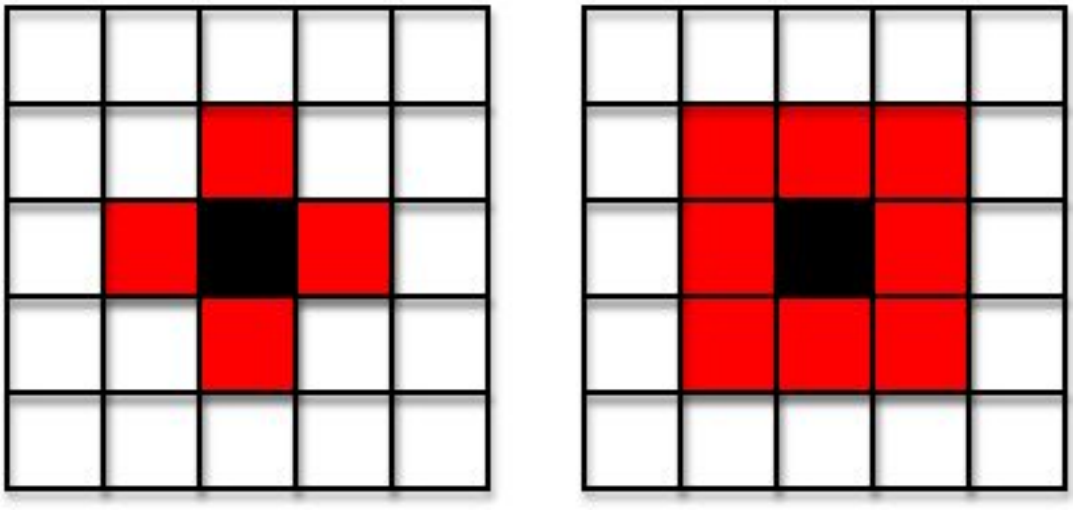


Figure 1: Two types of neighborhoods of a black cell, consisting of red neighbor cells: von Neumann neighborhood (left) and Moore neighborhood (right).

An issue arises for the cells on the edges of the grid, since they will not have a full neighborhood. Cells on the sides only have 5 neighbors, and those in the corners only have 3 neighbors. There are numerous ways to solve this problem. In this module, we resolve the issue by modeling the grid not as a rectangle, but as a torus that wraps around on the top, bottom, and sides. In this arrangement, the cells in the top row have the cells in the bottom row as their neighbors to the north, the cells in the bottom row have those in the top row as their neighbors to the south, the cells in the left column have the cells in the right column as their neighbors to the west, and the cells in the right column have the cells in the left column as their neighbors to the east. A toroidal grid can be simplified by including "ghost" rows and columns, which are copies of the rows and columns on the opposite sides of the grid from the edge rows and columns. A ghost corner is determined by copying the corner cell that is opposite the ghost corner on the diagonal. The method of using ghost rows and columns is depicted in Figure 2.

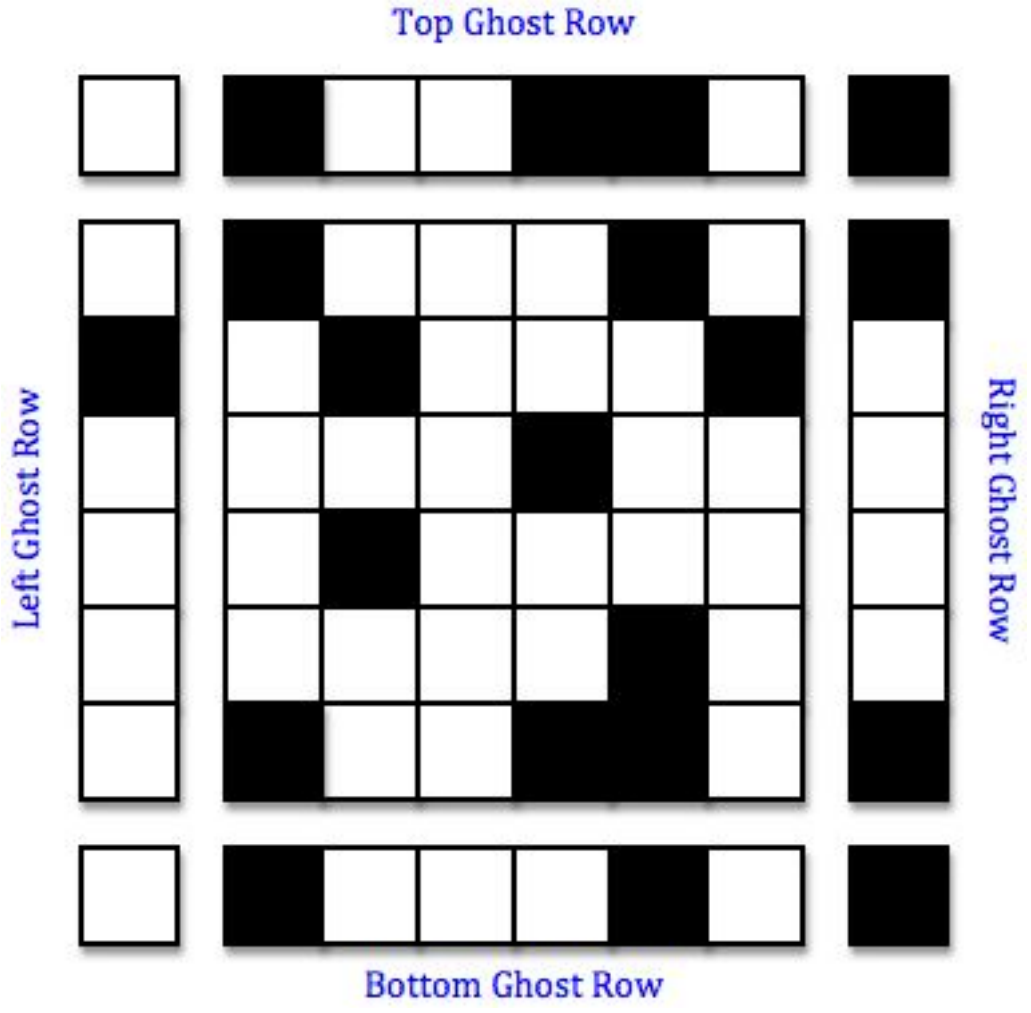


Figure 2: A toroidal grid simplified with ghost rows and columns

The rules of Conway's Game of Life are as follows:

- If a cell has fewer than 2 ALIVE neighbors, it will be DEAD in the next time step.
- If an ALIVE cell has 2 or 3 ALIVE neighbors, it will be ALIVE in the next time step.
- If a cell has more than 3 ALIVE neighbors, it will be DEAD in the next time step.
- If a DEAD cell has 3 ALIVE neighbors, it will be ALIVE in the next time step.

The automaton begins by initializing the states of the cells randomly and ends after a certain number of time steps have elapsed.

For a small board, a cellular automaton can be simulated with pencil and paper. For even a reasonably small board, however, working by hand becomes cumbersome and the use of a computer is needed to run the simulation in an acceptable amount of time. A coded implementation of Conway's Game of Life running on a single computer can simulate a fairly large grid in a very small amount of time.

To simulate an even bigger grid, one needs to employ more processing power than is available on a single processor. The concept of parallel processing can be used to leverage the computational power of computing architectures with multiple or many processors working together.

Conway's Game of Life is an interesting problem to parallelize because of the boundary conditions involving ghost rows and columns and because neighbor cells are tightly-coupled; that is, neighbors relate to each other directly and require calculations to be performed on groups of them. This leads to interesting parallel communication patterns and requires a modest amount of computation as well.

Compilation & Execution

In this module we're using the ARGO supercomputer provided by our University. (University of Athens, department of Informatics & Telecommunications). That particular cluster contains: **80 cores!** As a first step we needed a good sequential program that will be used as a basis in the study of parallel programs. The basic need created in a multi-process program is to exchange data with each other, so we used MPI which is a message transmission standard. It offers many options, several of which are being considered in order to arrive at the most efficient. The next step was to develop a hybrid approach with mpi and threads from openmp, with the aim of using fewer processes for the same number of cores, thus reducing communication. We compare these two approaches to see which one best fits the problem we are trying to solve. In the final stage, we made measurements with the CUDA program to compare the differences of behavior in the CPU programs with the GPU program and check the difference in performance.

Compilation, Execution & Project Structure

The project is structured as follows:

- First there is the main directory named: GoL_AndreasPappas_1115201500201
- Inside the main directory there are 2 sub-directories:
  - GoL_All_Source_Files
  - GoL_CUDA_Source_Files
  - Also there is The main README Documentation.pdf file that you're currently reading.
- Each of these sub-directories contain the source files that implement each project + a README + a Makefile for the compilation of each project. Also each of these directories contain a PBSScript.sh for inputing a job to the ARGO cluster and measure the performance of our programs.

So, **how to compile**: There are 2 ways:

1. Use the makefile by providing the command: **make mpi**
2. Use the following command: **mpicc -O3 -g life.c -L$MPIP_DIR/lib -lmpiP -lbfd -lunwind -o life.x** (The above command, presupposes that you're using the ARGO cluster. If not then simply use the **make** command).

Now, **how to execute**: Again there are 2 ways:

1. Providing the following command: **mpirun -np <num procs> life [-r Rows] [-c Columns] [-t Generations]**
   a. Example: **mpirun -np 4 ./life.mpi -r 144 -c 144 -t 100**
2. Or supposing that you're using ARGO:
   a. First you load the PBSScript by providing: **qsub PBSScript.sh**
   b. Then you type: **qstat**
   c. And then after the job is complete you can check the 2 output files myJob.o<id> for the output and myJob.e<id> for the errors.

At this point it is important to mention that the above instructions are meant for the MPI program! Furthermore it is worth to note that I will not provide any instructions for the Serial, OpenMP, Hybrid, CUDA programs about the ARGO cluster because knowing the above ones, are enough to adapt to the other programs + the ARGO cluster is only available on UoA registered students and they're given instructions already at the course site on the Argo-InstructionsMPInPBS.pdf file.

We'll continue with the compilation & execution instructions on the other programs.

As contained in the README:

*The code can be made into serial, MPI, OpenMP, and hybrid executables. To compile each of these, enter '**make serial**', '**make mpi**', '**make openmp**', or '**make hybrid**', respectively. To remove all executables from the directory, enter '**make clean**'. To re-make all of the executables at once, enter '**make all**'.*

CUDA:

- **Compile:**
  - [user@name]$ **nvcc life.cu -o life**
- **Run:**
  - [user@name]$ **./life [dimensions] [generations]**
  - **Example: ./life 10 5**
    - *The above will generate a 10x10 grid of 5 Time Steps.*
- **Or you can use the PBSScript.sh on ARGO cluster**

At this point we're done with the compilation & execution instructions!

Introduction to Parallelism

In parallel processing, instead of a single program executing tasks in a sequence, the program is split among multiple "execution flows" executing tasks in parallel, i.e. at the same time. The term "execution flow" refers to a discrete computational entity that performs processes autonomously[1]. Execution flows have more specific names depending on the flavor of parallelism being utilized. In "distributed memory" parallelism, in which execution flows keep their own private memories (separate from the memories of other execution flows), execution flows are known as "processes". In order for one process to access the memory of another process, the data must be communicated, commonly by a technique known as "message passing". The standard of message passing considered in this module is defined by the "Message Passing Interface (MPI)", which defines a set of primitives for packaging up data and sending them between processes.
In another flavor of parallelism known as "shared memory", in which execution flows share a memory space among them, the execution flows are known as "threads". Threads are able to read and write to and from memory without having to send messages.[2] The standard for shared memory considered in this module is OpenMP, which uses a series of constructs for specifying parallel regions of code to be executed by threads.[3]
A third flavor of parallelism is known as "hybrid", in which both distributed and shared memory are utilized. In hybrid parallelism, the problem is broken into tasks that each process executes in parallel; the tasks are then broken further into subtasks that each of the threads execute in parallel. After the threads have

[1] A common synonym is "execution context"; "flow" is chosen here because it evokes the stream of instructions that each entity processes.
[2] It should be noted that shared memory is really just a form of fast message passing. Threads must communicate, just as processes must, but threads get to communicate at bus speeds (using the front-side bus that connects the CPU to memory), whereas processes must communicate at network speeds (ethernet, infiniband, etc.), which are much slower.
[3] Threads can also have their own private memories, and OpenMP has constructs to define whether variables are public or private to threads.

executed their sub-tasks, the processes use the shared memory to gather the results from the threads, use message passing to gather the results from other processes, and then move on to the next tasks.
In terms of what is actually being parallelized, there are two main types of parallelism that can occur. In the first, “data parallelism”, data is split up into chunks and assigned to the execution flows, and each execution flow performs the same task on its chunk of data as the other execution flows on their chunks of data. In the second, “task parallelism”, all execution flows operate on the same set of data but perform different tasks on it. Many parallel applications will leverage both data and task parallelism.

Parallel Hardware

In order to use parallelism, the underlying hardware needs to support it. The classic model of the computer, first established by John von Neumann in the 20th century, has a single CPU connected to memory. Such an architecture does not support parallelism because there is only one CPU to run a stream of instructions. In order for parallelism to occur, there must be multiple processing units running multiple streams of instructions. There are a few ways to achieve this. One way is to split the CPU into multiple processing units called “cores”. Each core is capable of executing instructions concurrently with the
other cores, which means they can work in parallel. Cores on a chip will typically share RAM among them. Another way to achieve parallelism is to link multiple computers together over a network (which is what we had at our University before ARGO). In this case the computers can also work in parallel, but they do not share RAM. These computers may themselves have multi-core CPUs, which allows for hybrid parallelism: shared memory between the cores and distributed memory among the compute nodes. A diagram of these different parallel hardware models is shown in Figures 3-5.

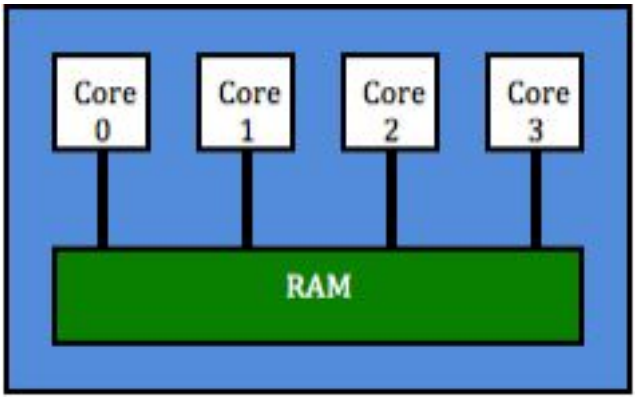


Figure 3: Shared memory, multiple cores on one chip.

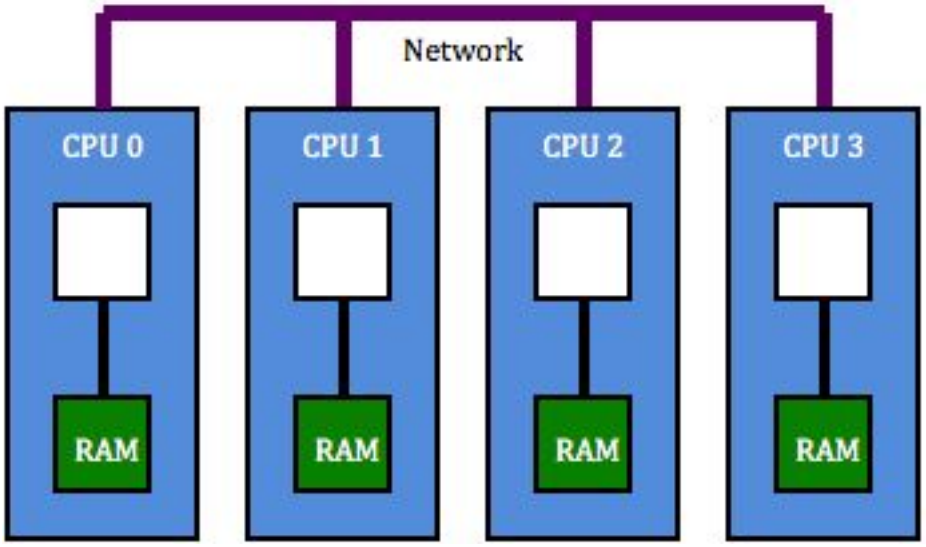


Figure 4: Distributed memory, multiple CPUs connected via network.

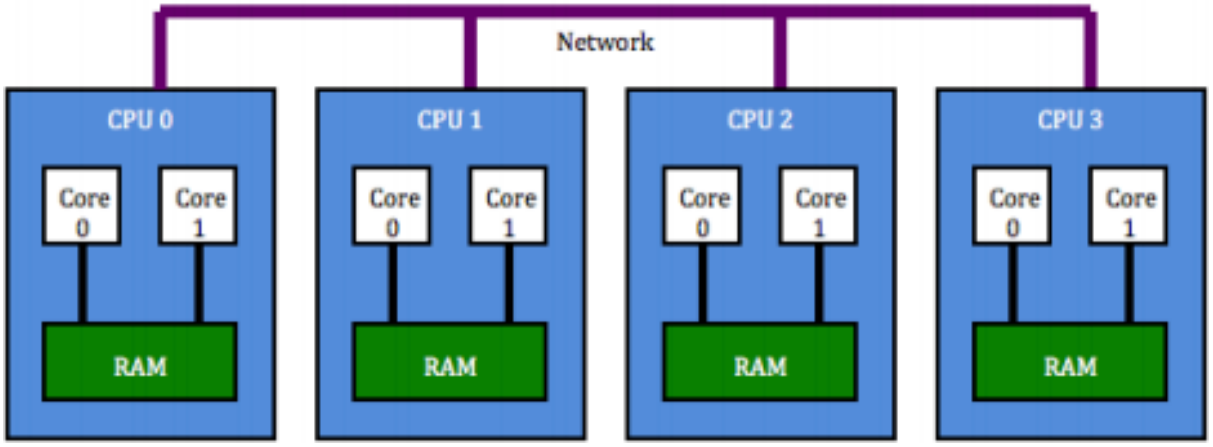


Figure 5: Hybrid, multiple multi-core CPUs connected via network.

Why use Parallelism

There are compelling advantages for using parallelism. The three motivations considered in this module are speedup, accuracy, and scaling.

"Speedup" is the idea that a program will run faster if it is parallelized as opposed to executed serially. The advantage of speedup is that it allows a problem to be modeled[4] faster. If multiple execution flows are able to work at the same time, the work will be finished in less time than it would take a single execution flow. Speedup is an enticing advantage.

"Accuracy" is the idea of forming a better solution to a problem. If more processes are assigned to a task, they can spend more time doing error checks or other forms of diagnostics to ensure that the final result is a better approximation of the problem that is being modeled. In order to make a program more accurate, speedup may need to be sacrificed.

"Scaling" is perhaps the most promising of the three motivations. Scaling is the concept that more parallel processors can be used to model a bigger problem in the same amount of time it would take fewer parallel processors to model a smaller problem. A common analogy to this is that one person in one boat in one hour can catch a lot fewer fish than ten people in ten boats in one hour. There are issues that limit the advantages of parallelism; we will address two in particular. The first, communication overhead, refers to the time that is lost waiting for communications to take place before and after calculations. During this time, valuable data is being communicated, but no progress is being made on executing the algorithm. The communication overhead of a program can quickly overwhelm the total time spent modeling the problem, sometimes even to the point of making the program less efficient than its serial counterpart. Communication overhead can thus mitigate the advantages of parallelism.

A second issue is described in an observation put forth by Gene Amdahl and is commonly referred to as "Amdahl's Law". Amdahl's Law says that the speedup of a parallel program will be limited by its serial regions, or the parts of the algorithm that cannot be executed in parallel. Amdahl's Law posits that as the number of processors devoted to the problem increases, the advantages of parallelism diminish as the serial regions become the only part of the code that takes significant time to execute. Amdahl's Law is represented as an equation in Figure 6.

$$Speedup = 1 / (1 - P + (P/N))$$

Where,

P = the proportion of the program that can be made parallel 1 – P = the proportion of the program that cannot be made parallel €
N = the number of processors

[4] It should be emphasized that this module refers to "modeling" a problem, not "solving" a problem. This follows the computational science credo that algorithms running on computers are just one tool used to develop *approximate* solutions (models) to a problem. Finding an actual solution may involve the use of many other models and tools.

Figure 6: Amdahl's Law

Amdahl's Law provides a strong and fundamental argument against utilizing parallel processing to achieve speedup. However, it does not provide a strong argument against using it to achieve accuracy or scaling. The latter of these is particularly promising, as it allows for bigger classes of problems to be modeled as more processors become available to the program. The advantages of parallelism for scaling are summarized by John Gustafson in Gustafson's Law, which says that bigger problems can be modeled in the same amount of time as smaller problems if the processor count is increased. Gustafson's Law is represented as an equation in Figure 7.

$$\text{Speedup}(N) = N - (1 - P) * (N - 1)$$

where

N = the number of processors

(1 – P) = the proportion of the program that cannot be made parallel

Figure 7: Gustafson's Law

Amdahl's Law reveals the limitations of what is known as "strong scaling", in which the number of processes remains constant as the problem size increases. Gustafson's Law reveals the promise of "weak scaling", in which the number of processes increases along with the problem size.

The Algorithm

Our parallel Game of Life algorithm is designed with four scales in mind: serial, shared memory, distributed memory, and hybrid. All four come from the same piece of source code; this is accomplished by surrounding the parts of the code that require shared and distributed memory by #ifdef guards that the pre-processor uses to determine which sections of the code need to be compiled. There are also sections of the algorithm that are executed a certain way if distributed memory is enabled and a different way if distributed memory is disabled. In this description, we can see some of the data structures that we will need by finding the nouns (and the adjectives and prepositional phrases associated with them) in our description. An example of a list of data structures is shown in Table 1.

Data Structures

Noun	Data Structure
A grid of cells	A 2-dimensional array of cells
Cell	An Integer representing a stage (ALIVE or DEAD)
Each Time Step	An integer count of the number of time steps that have elapsed
Some Number of Time Steps	An integer representing the total number of time steps
ALIVE neighbors	An integer count of the number of ALIVE neighbors
The next Time Step	A 2-dimensional array of cells representing the grid in the next time step.

Table 1: Data structures

Notice that we keep an integer count of the number of time steps. We could also keep a separate grid for each time step, but in order to save memory we choose to only keep the two grids that are most important: the current grid and the next grid (we need to keep at least two grids because we do not want to overwrite the cells in one grid before we have referenced them all to determine the cells of the new grid).

There are also data structures that control the parallelism. Any parallel algorithm is likely to have data structures like the ones listed in Table 2.

Data structures that control parallelism	
Written Representation	**Name**
The rank[5] of a process	OUR_RANK
The number of processes	NUMBER_OF_PROCESSES
The thread number[6]	MY_THREAD_NUMBER
The number of threads	NUMBER_OF_THREADS

Table 2 : Data structures that control parallelism

5 Rank is the terminology used by MPI to indicate the ID number of a process.

6 Thread number is the terminology used by OpenMP to indicate the ID of a thread

The next question to ask is, “Where does data parallelism occur?” Recall that data parallelism refers to splitting up data among all the execution flows. If we use hybrid parallelism, there will be data split among processes that are further split among threads. To start, we identify the data that will be split among the processes and upon which the processes will perform a task in parallel. Of the data structures in Table 1, only two have data that can be parallelized: the 2-dimensional array of cells representing the grid in the current time step and the 2-dimensional array of cells representing the grid in the next time step. A grid is 2-dimensional because it consists of rows and columns, so the grid can be split into chunks of rows, chunks of columns, sub-grids, or various other configurations. In this module, we choose to split the grid into chunks of rows.

We must now ask, “For how many rows is each process responsible?” There are many ways to determine a process’s “workload”, or the amount of data for which it is responsible. One common method is to have a single process assign data to the other processes (so-called “master/worker” methods). In this module, we instead choose to have each process determine its own workload.

The next question is, “For how many columns is each thread responsible?” This is again a question of load balancing, but it turns out that OpenMP will take care of the load balancing in this case. Recall that when OpenMP encounters a parallel section, it spawns threads to perform that section in parallel. If the section is enclosed in a loop, it
dynamically assigns iterations of the loop to threads until all iterations of the loop have been executed. In our algorithm, columns are dealt with by looping over them, so we can leave the load balancing of threads to OpenMP.[7]
Now that we have determined how the data is parallelized, the next question to ask is, “what parallel task is performed on the data”? Each process has some number of rows (hereafter referred to as a sub-grid) for which it is responsible to find the state of the cells in the next sub-grid. For each cell in the sub-grid, the state of the neighbor cells must be determined, and from this information the state of the cell in the new sub-grid is set.

The next question is, “What additional values must be established in order to perform this parallel task?” A good method for identifying these is to draw and label a picture of the task. An example of such a picture is in Figure 8. Since we are taking a thread-centric view of our algorithm, we pretend that the thread is labeling the diagram. Things labeled “I, me, and my” refer to the threads. Things labeled “we, us, and our” refer to the processes, since processes are collections of threads.

[7] This assumes that OpenMP schedules iterations of the loop to threads dynamically. In some paradigms, the loop iterations are scheduled to threads statically; in which case it is known ahead of time which threads will execute which iterations. Static scheduling is simpler, as it requires only one schedule at the beginning of the loops, whereas dynamic scheduling requires constant checking to see if threads are available to do work. On the other hand, dynamic scheduling can produce better performance if the loop iterations do not divide evenly among the threads.

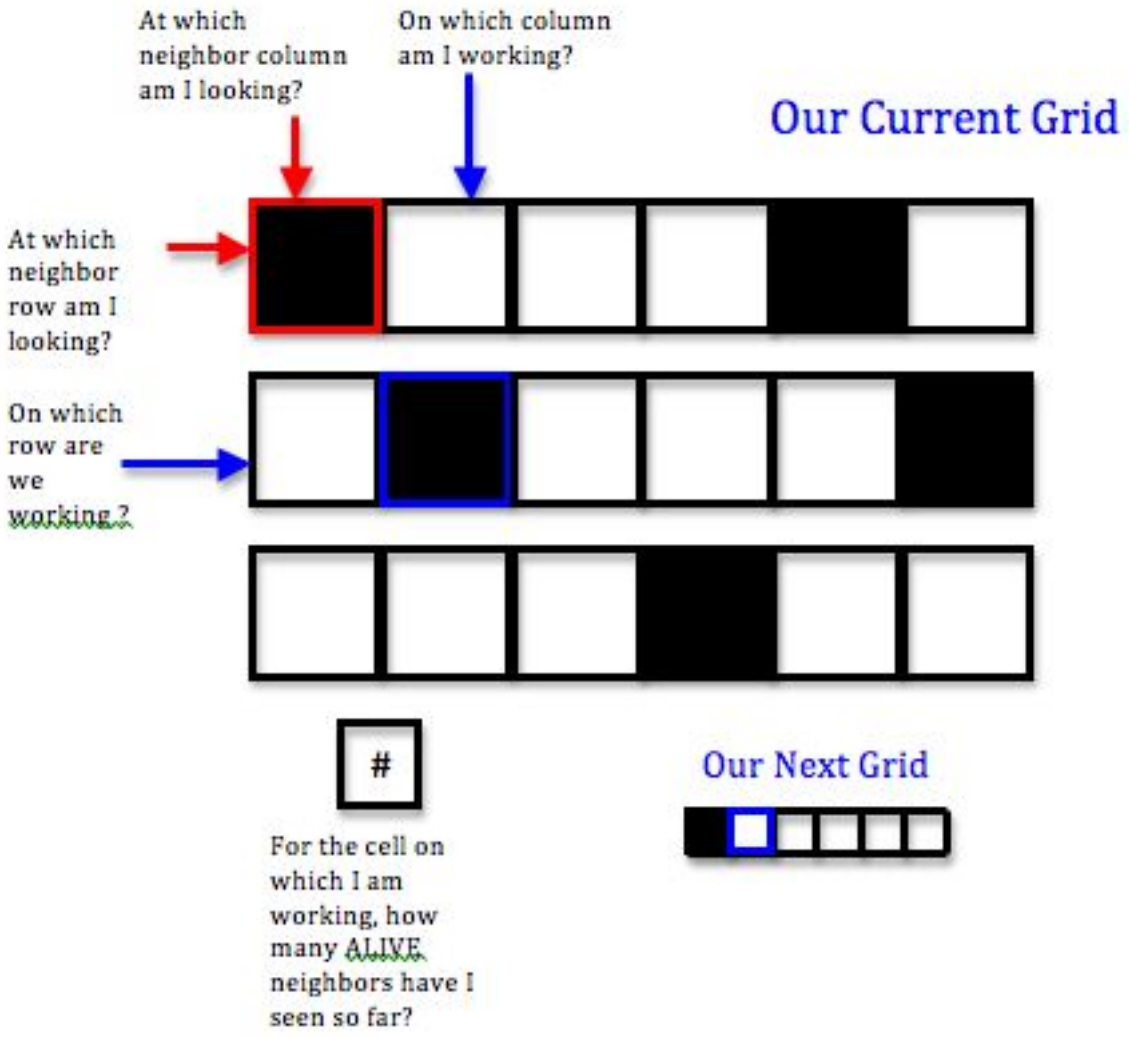


Figure 8: A picture of the parallel task at hand

From Figure 8, we can see that the process is employing the use of ghost rows. How is it able to access these rows? It must receive them from the other processes. Recall that processes communicate data to each other by passing messages. We have established from Figure 8 that the processes must pass ghost rows to other processes that need to access them. To better understand this, it is again helpful to draw a picture. Figure 9 shows an example of such a picture.

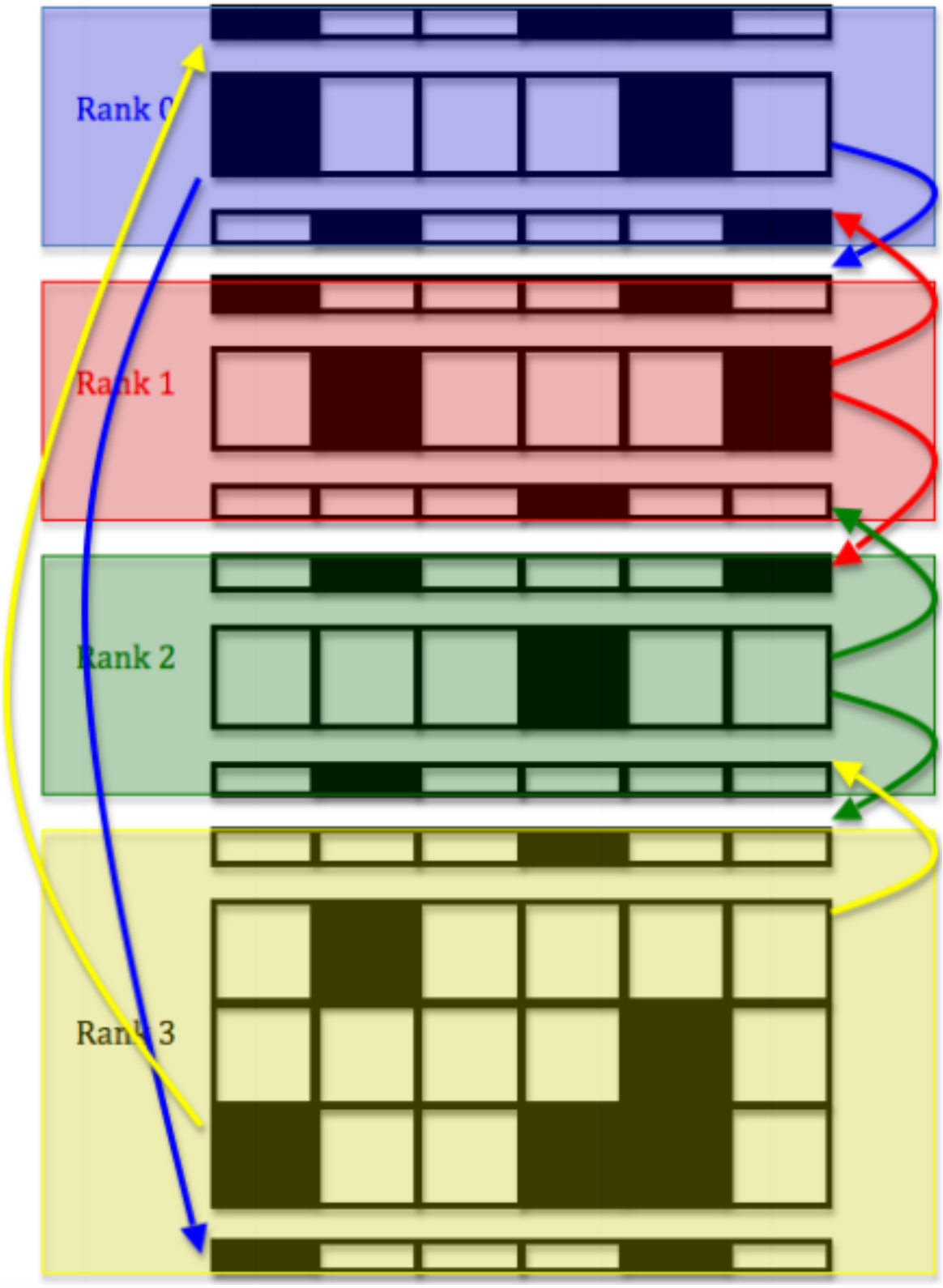


Figure 9: Message passing

It should be noted that each process now has two ghost rows in addition to the rows it was assigned. We can work this into our data structures by making the first row the top ghost row and the last row the bottom ghost row.

The order of sends and receives is relevant. The MPI calls we use in this module send messages using “non-blocking” methods but receive them using “blocking” methods. The distinction between these is that blocking methods stop the execution of the program until an event occurs, whereas non-blocking methods allow the program to continue regardless. This means that messages must be

ready to be received if the program is to continue, which only works if the messages are sent first. MPI also offers methods for blocking sends and non-blocking receives, but these will not be discussed in this module.

Can the message passing be parallelized? Put another way, can multiple threads in a process pass messages at once? The answer depends on whether thread support is enabled for the MPI methods being used. Some MPI implementations do not support threaded message passing, and some implementations are compiled with it disabled. This module assumes that threaded message passing is disabled, as is often the case[8].

We can now return to Figure 8. Recall that we are trying to identify the new values that are needed to execute the parallel task. We can see from Figure 8 that certain questions are being asked; these can be answered by creating new values. Table 3 shows an example of this.

Values for the parallel task	
Written Representation	**Name**
Our current grid	our_current_grid[ ][ ]
Our next grid	Our_next_grid[ ][ ]
On which row are we working?	our_current_row
At which neighbor row am I looking?	my_neighbor_row
At which neighbor column am i looking?	my_neighbor_column
For the cell on which i am working, how many ALIVE neighbors have i seen so far?	my_number_of_alive_neighbors
On which column am i working?	my_current_column

Table 3: Written representation of values

In the next page we can see the algorithm written in pseudocode.

[8] If threaded message passing is available, the program should be initialized with MPI_Init_thread and passed the MPI_THREAD_MULTIPLE argument. See the man page of MPI_Init_thread for details.

All processes do the following:

- Initialize the distributed memory environment
- Parse command line arguments
- Make sure we have enough rows, columns, and time steps
- Exit if we don't

- Determine our number of rows
- Allocate enough space in our current grid and next grid
- for the number of rows and the number of columns, plus
- the ghost rows and columns
- Initialize the grid (each cell gets a random state)
- Determine the process with the next-lowest rank
- Determine the process with the next-highest rank

- Run the simulation for the specified number of time steps
- Set up the ghost rows

  /* If distributed memory is defined */

  - 1. Send our second-from-the-top row to the process
  - with the next-lowest rank
  - 2. Send our second-from-the-bottom row to the
  - process with the next-highest rank
  - 3. Receive our bottom row from the process with the
  - next-highest rank
  - 4. Receive our top row from the process with the
  - next-lowest rank

  /* Otherwise (if distributed memory is not defined)*/

  - 1. Set our top row to be the same as our second-to-
  - last row
  - 2. Set our bottom row to be the same as our second-
  - to-top row
- Set up the ghost columns
  - 1. The left ghost column is the same as the
  - farthest-right, non-ghost column
  - 2. The right ghost column is the same as the
  - farthest-left, non-ghost column
- Display the current grid
- Determine our next grid – for each row, do the following:
  - 1. For each column, spawn threads to do the Following:
    - a. Initialize the count of ALIVE neighbors to 0
  - b. For each row of the cell's neighbors, do the following:
    - i. For each column of the cell's
    - neighbors, do the following:
      - If the neighbor is not the cell
      - itself, and the neighbor is ALIVE,
      - do the following:
        - Add 1 to the count of the
        - number of ALIVE neighbors
  - c. Apply Rule 1 of Conway's Game of Life
  - d. Apply Rule 2 of Conway's Game of Life
  - e. Apply Rule 3 of Conway's Game of Life
  - f. Apply Rule 4 of Conway's Game of Life

- Spawn threads to copy the next grid into the current
- grid
- Deallocate data structures
- Finalize the distributed memory environment

Figure 10: The Algorithm

Notes about the algorithm

In order for a process to determine its workload, it needs to know 1) how many total processes will be working and 2) how much work needs to be done. In this example, the number of processes is determined by MPI at run-time, and the amount of work to be done is equal to the number of rows in the grid. There are numerous ways to spread out the work, but in this module we choose to assign an even chunk of work to each process. Thus, each process will be responsible for the number of rows obtained by dividing the total number of rows by the number of processes. If the number of processes does not divide evenly into the number of rows, then there will be some remainder of rows left over. This can be arbitrarily assigned to a process; we choose to assign it to the last process. An example of this load balancing is Figure 11, in which MPI Ranks 0, 1, and 2 are each responsible for one row, and Rank 3 is responsible for three rows.

Figure 11: Load balancing among 4 processes

We are storing the grids as 2-D arrays, so we allocate enough space for NUMBER_OF_ROWS + 2 (because we have 2 ghost rows) times NUMBER_OF_COLUMNS + 2 (because we have 2 ghost columns). In C, we allocate the overall arrays using double pointers (int**) and then allocate each of the sub-arrays (rows) using single pointers (int*).

Code Implementation:

Now that the pseudo-code has been developed, the code can be implemented. It should first be noted that MPI processes each execute the entire code, but OpenMP threads only execute the sections of code for which they are "spawned", or created. This means that sections of code will only be executed with shared memory parallelism if they are contained within parallel OpenMP constructs, but all code will be executed with distributed memory parallelism (except for sections of the code contained in conditionals that are only executed by

a process of a certain rank).

The MPI functions that are used in the code are listed and given explanations in Table 4. The relevant OpenMP construct is listed and given an explanation in Table 5.

MPI Function	Explanation
MPI_Init()	This initializes the MPI environment. It must be called before any other MPI functions.
MPI_Comm_rank()	This assigns the rank of the process in the specified communicator to the specified variable.
MPI_Comm_size()	This assigns the number of processes in the specified communicator to the specified variable.
MPI_Send()	This sends a message of a certain length to a certain process.
MPI_Recv()	This receives a message of a certain length from a certain process.
MPI_Finalize()	This cleans up the MPI environment. No other MPI functions may be called after this one.

Table 4: MPI Functions used by the code

OpenMP pragma	Explanation

#pragma omp parallel for
private(...)
for(...) {
…
}
	This spawns a pre-defined number of threads and parallelizes the for loop. Each thread has a private copy of each variable specified.

Table 5: OpenMP construct used by the code

When using library functions such as malloc and calloc and the MPI functions, it is a good idea to check the return codes of the functions to ensure that the functions operated successfully. If an error occurs in these functions it is a good idea to print an error message explaining what went wrong and then exit the program, returning an error code. The code in this module uses a custom function

called exit_if, which prints an error message and exits the program if something is true, namely if a function returns a value that does not match the expected value.

Scaling.

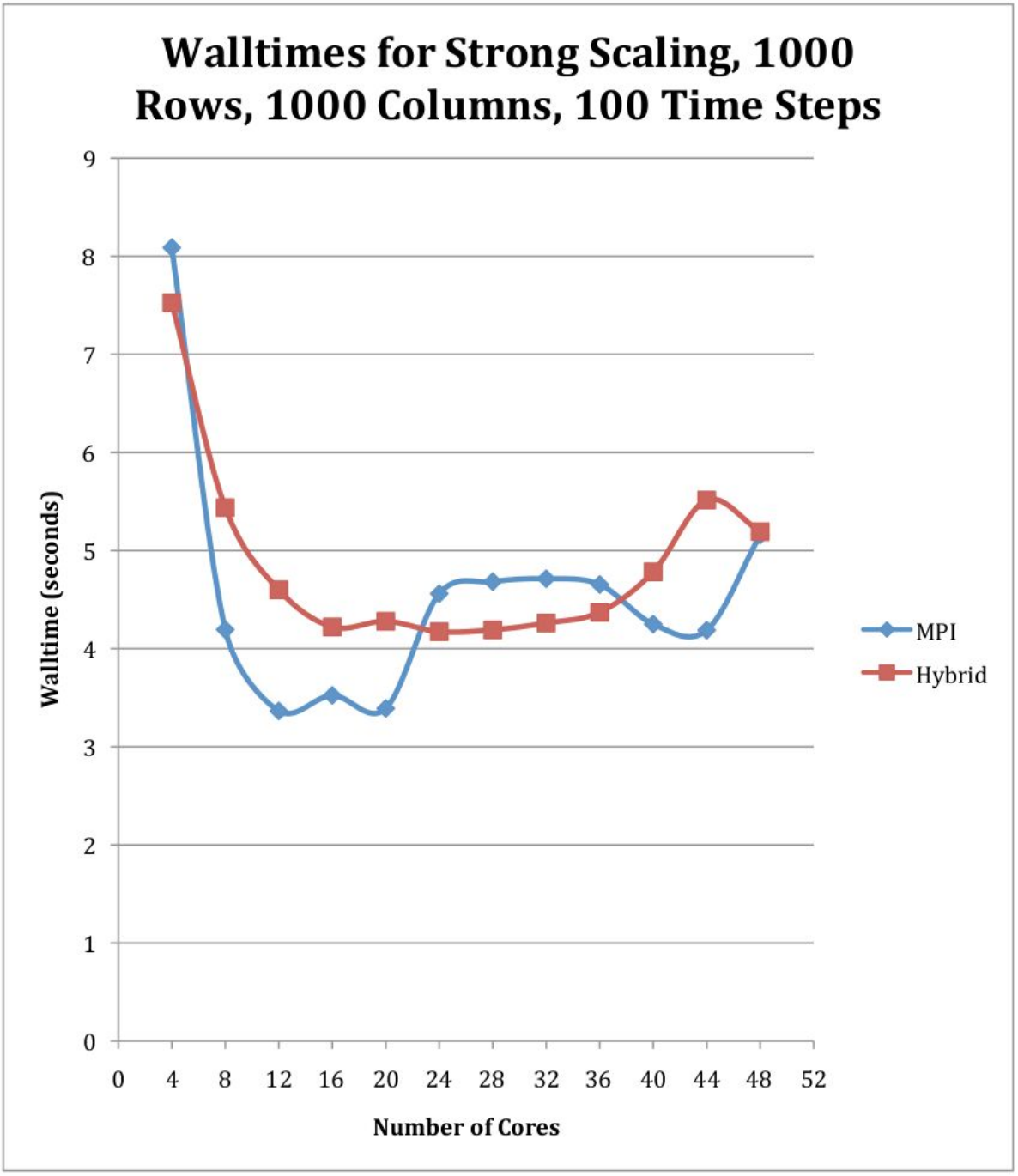
Now that we have developed a parallel algorithm, a natural next question is, "does the algorithm scale?" Because of the limitations revealed by Amdahl's Law, we can be assured that the algorithm will not scale far if we merely increase the number of cores devoted to the problem (strong scaling); the code will initially run faster but will see diminished returns as communication overhead overwhelms the total amount of time spent running the code. However, Gustafson's Law promises that if we increase the problem size as we increase the core count (weak scaling), we will be able to model a bigger problem in the same amount of time. The goal of this section is to show that strong scaling of our parallel code fails as predicted by Amdahl's Law and to show that weak scaling of our parallel code succeeds as predicted by Gustafson's Law.

Walltimes for Strong Scaling, 1000 rows, 1000 columns, and 100 Time Steps

# of nodes used	Total # of cores	Serial	OpenMP	MPI	Hybrid
1	4	17.941	12.63	8.088	7.525
2	8			4.194	5.437
3	12			3.365	4.6
4	16			3.524	4.221
5	20			3.391	3.232
6	24			4.56	4.173
7	28			4.683	4.191
8	32			4.713	4.261
9	36			4.654	4.371
10	40			4.25	4.783
11	44			4.188	5.515
12	48			5.159	5.192

There are a few things to notice about this data.

First of all, the speedup is dramatic when moving from shared memory to distributed memory, cutting the running time almost in half. Note also that the difference in runtimes between hybrid and MPI is not large, and that in some cases the hybrid program took longer than the pure MPI program to run. A graph of the data is shown below. Note that all these are taken within the ARGO cluster, using the PBSScript.sh



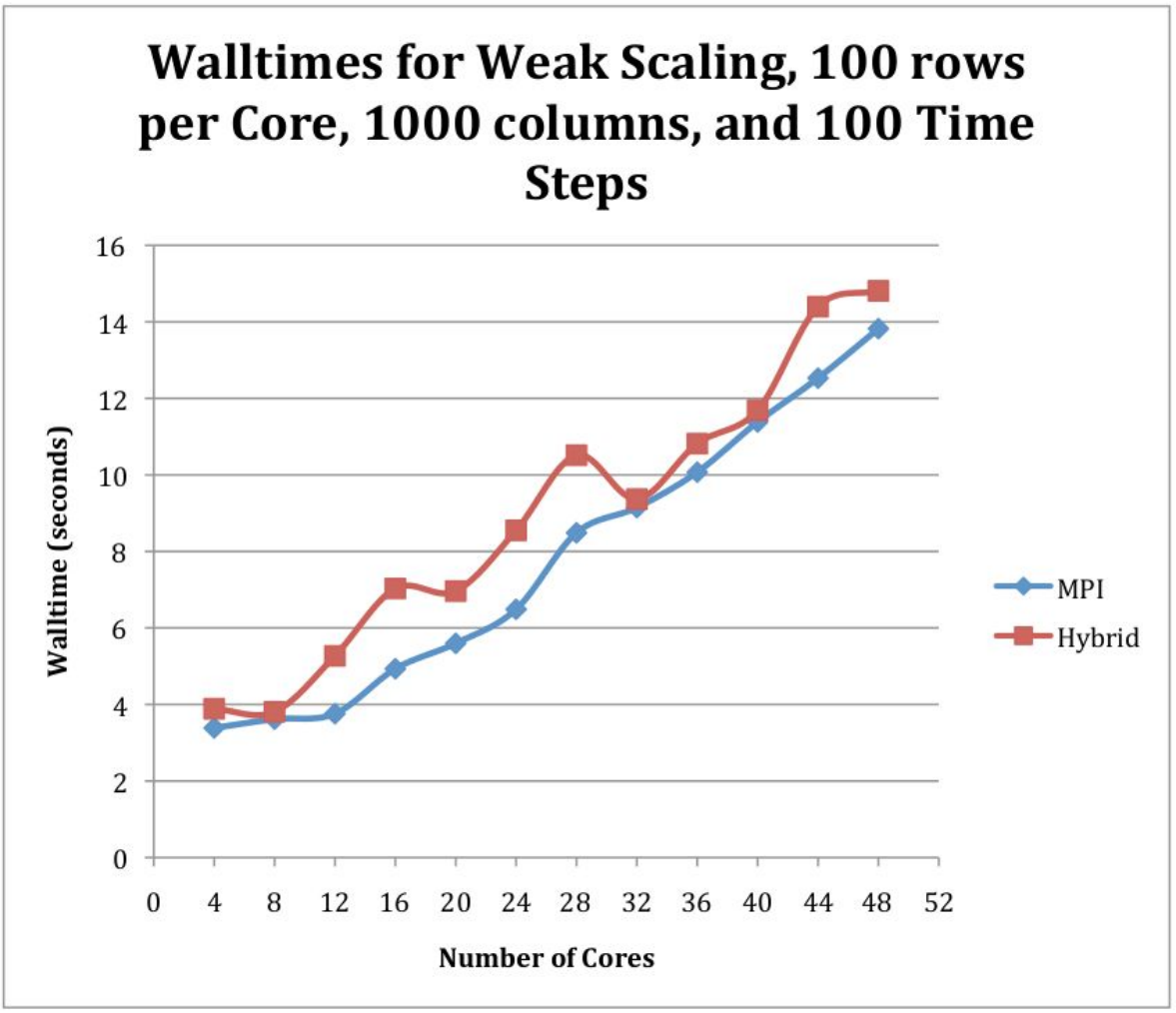
Note that both curves have a sharp decrease followed by a level, in the case of Hybrid, and an increase, in the case of MPI. The dramatic decrease is indicative of Amdahl's Law. As we add more cores, we see less and less speedup, and in fact it eventually takes more time to run with more cores than with fewer cores. Speedup is only noticeable up to about 16 cores.

Below you can see the performance measurements and a graph for the weak scaling.

Walltimes for Weak Scaling, 100 rows per Core, 1000 columns, 100 Time Steps

# of nodes used	Total # of cores	Total # of rows	Serial	OpenMP	MPI	Hybrid
1	4	400	7.187	5.282	3.387	3.887
2	8	800			3.616	3.808
3	12	1.200			3.759	5.27
4	16	1.600			4.933	7.032
5	20	2.000			5.597	6.963
6	24	2.400			6.487	8.55
7	28	2.800			8.486	10.515
8	32	3.200			9.146	9.369
9	36	3.600			10.071	10.817
10	40	4.000			11.383	11.691
11	44	4.400			12.529	14.396
12	48	4.800			13.822	14.811

Keep in mind that with weak scaling we are increasing the problem size as we increase the number of cores. Because of communication overhead between the processes, the walltime will gradually increase as we add more cores. This is visible in the upward trend of the graph below.



We can simulate Game of Life for grids ranging from 400 rows to 4,800 rows using 4-48 cores in under 14 seconds for MPI and in under 16 seconds for Hybrid. Notice that at no point does the Hybrid program run faster than the pure MPI program. This would lead us to believe that there is no advantage to using the hybrid program over the MPI program.

CUDA

For the cuda code I used the examples that we were given on the course. First, copy the data from CPU memory to GPU memory and then on each loop we call the kernel code. Again we're using macro commands inside the source code, and we're implementing the functions inside the same source code for better code efficiency. Compiling the program will generate the executable which when we run it, it will give us information about the grid and the generations and the time taken and also it will generate an output .txt file for the grid display. Else you can choose to print the grid on stdout on it's starting and end state by uncommenting the define macros at the beginning of the file.

Everything is well commented for better understanding of the code. The structure and philosophy overall is similar to our MPI program structure.

CUDA Performance

On the table below we can see the results of the CUDA Performance (speedup) measured and calculated on the ARGO's cluster GPU.

We can see the speedup in relation to the matrix size and the # of threads.

	Threads			
Matrix Size	**128**	**256**	**512**	**1024**
144 x 144	68.01	63.698	63.696	48.333
720 x 720	93.143	94.226	88.564	74.213
1440 x 1440	95.856	97.031	92.146	78.39
1728 x 1728	95.85	97.42	92.3	78.4
2880 x 2880	94.9	96.9	91.01	77.31

From the table above we can see that, on 256 threads we got the greatest speedup. Also, as we increase the size of the problem the speedup improves until it reaches a bound. Also, as we were taught in the lab as we increase 4x times the matrix also the time gets increased by 4x.

Comparing with MPI

As we saw, with the CUDA implementation we got a speedup of 97.42 compared with the serial program which is extraordinary. As expected it is greatly higher than the performance of the MPI compared with the serial program, which is anticipated because the GPU can parallelize the problem much more than the CPU and each thread of the GPU calculates 1 cell of the grid which results in a much better performance compared to the MPI.

Conclusion

In general, we observed that as we increase the problem size the speedup improves. Parallelizing the problem with the use of a GPU, using CUDA, results in the maximum overall speedup and performance. Of course with the MPI parallelization we got great results and even greater as the grids were getting bigger, reaching really close to the ideal 1. Also as studied in this module the scalability is also great! And that's the overall point of this course / module and parallelization overall as discussed on the first pages. To improve the performance of an algorithm / problem and get faster solutions and results compared to a program / problem that does not involve parallelization!

References

[1] Kansal, A. R., Torquato, S., Harsh, G. R. IV, Chiocca, E. A., & Deisboeck, T. S. (April 2000). Simulated Brain Tumor Growth Dynamics Using a Three-Dimensional Cellular Automaton. *Journal of Theoretical Biology, 203*(4), 367-382. doi:10.1006/jtbi.2000.2000

[2] Silvertown, Jonathan, Holtier, Senino, Johnson, Jeff, & Dale, Pam. (September 1992). Cellular Automaton Models of Interspecific Competition for Space--The Effect of Pattern on Process. *Journal of Ecology, 80*(3), 527-533. Retrieved from http://www.jstor.org/stable/2260696

[3] Wimpenny, Julian W.T. and Colasanti, Ric. (1997). A Unifying Hypothesis for the Structure of Microbial Biofilms Based on Cellular Automaton Models. *FEMS Microbiology Ecology, 22*(1), 1-16. doi:10.1111/j.1574-6941.1997.tb00351.x

[4] Beauchemin, Catherine, Samuel, John, & Tuszynski, Jack. (January 2005). A Simple Cellular Automaton Model for Influenza A Viral Infections. *Journal of Theoretical Biology, 232*(2), 223-234. doi:10.1016/j.jtbi.2004.08.001.

[5] Sirakoulis, G.Ch., Karafyllidis, I., & Thanailakis A. (September 2000). A Cellular Automaton Model for the Effects of Population Movement and Vaccination on Epidemic Propagation. *Ecological Modelling, 133*(3), 209-223. doi:10.1016/S0304- 3800(00)00294-5.

[6] Bedau, Mark A. (November 2003). Artificial Life: Organization, Adaptation and Complexity from the Bottom Up. *Trends in Cognitive Sciences, 7*(11), 505-512. doi:10.1016/j.tics.2003.09.012

[7] Orszag, Steven A. and Yakhot, Victor. (April 1986). Reynolds Number Scaling of Cellular-Automaton Hydrodynamics. *Physical Review Letters, 56*(16), 1691–1693. doi:10.1103/PhysRevLett.56.1691

[8] Smolle, Josef et al. (1995). Computer Simulations of Histologic Patterns in Melanoma Using a Cellular Automaton Provide Correlations with Prognosis. *Journal of Investigative Dermatology, 105*, 797–801. doi:10.1111/1523-1747.ep12326559

[9] Marx V., Reher, F.R., & Gottstein, G. (March 1999). Simulation of Primary Recrystallization Using a Modified Three-dimensional Cellular Automaton. *Acta Materialia, 47*(4), 1219-1230. doi:10.1016/S1359-6454(98)00421-2

[10] Khalatur, Pavel G., Shirvanyanz, David G., Starovoitova, Nataliya Yu., & Khokhlov, Alexei R. (March 2000). Conformational Properties and Dynamics of Molecular Bottle-brushes: A Cellular-automaton-based

Simulation.*Macromolecular Theory and Simulations, 9*(3), 141-155, doi:10.1002/(SICI)1521-3919(20000301)9:3<141::AID-MATS141>3.0.CO;2-3

[11] Berjak, S. G. and Hearn, J. W. (February 2002). An Improved Cellular Automaton Model for Simulating Fire in a Spatially Heterogeneous Savanna System. *Ecological Modelling, 148*(2), 133-151. doi:10.1016/S0304-3800(01)00423-9

[12] Balmann, A. (1997). Farm-based Modelling of Regional Structural Change: A Cellular Automata Approach. *European Review of Agricultural Economics, 24*(1-2), 85-108. doi:10.1093/erae/24.1-2.85

[13] Wahle, J., Annen, O., Schuster, Ch., Neubert, L., & Schreckenberg, M. (June 2001). A Dynamic Route Guidance System Based on Real Traffic Data. *European Journal of Operational Research, 131*(2), 302-308. doi:10.1016/S0377-2217(00)00130-2

[14] Asanovic, K. et al. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. University of California at Berkeley. Technical Report No. UCB/EECS-2006-183.

[15] Gardner, M. (October 1970). The Fantastic Combinations of John Conway's New Solitaire Game “Life”. *Scientific American, 223*, 120-123.

Outro

At last I would like to thank our professor Mr. Kotronis for providing us with this exciting course and his well structured notes and directions provided in the course website: https://eclass.uoa.gr/courses/D36/

Thank you,

Andrew Pappas

1115201500201

September, 2020, ©All Rights Reserved!