4/26/2024

Extemporaneous evaluation

Miguel Sánchez Piña

Data Engineering 7°B

High Performance Computing

Professor Didier Gamboa

Introduction

Conway's Game of Life has become a playground to mathematicians and scientists for many years due to its ability to generate complex patterns and behaviors from simple rules. The game operates on a grid of cells, each of which can be in one of two states: alive or dead. Through a set of rules governing the evolution of these cells, intricate patterns emerge, ranging from static configurations to oscillating structures and glider-like entities that traverse the grid.

While Conway's Game of Life has primarily served as a recreational pursuit as a game and a subject of theoretical inquiry, its relevance extends into the domain of high-performance computing. The grid-based nature of the game lends itself naturally to parallelization, enabling efficient utilization of multiple processing units or cores. Consequently, researchers and practitioners have turned to Conway's Game of Life as a benchmarking tool for evaluating the performance of parallel computing algorithms and architectures.

In this investigation, we intend to explore the intersection of Conway's Game of Life and high-performance computing. By the end of this report, you will be able to:

1. Understand the fundamental principles of Conway's Game of Life, including its rules and patterns.

2. Learn the potential for parallelization within Conway's Game of Life and its implications for HPC.

3. Know how to implement solutions provided in Cython to optimize the performance of Conway's Game of Life, thereby understanding the benefits of compiled code.

Conway’s game of life.

Conway's Game of Life operates on a two-dimensional grid, commonly referred to as the "universe," where each cell can exist in one of two states: alive or dead. The game evolves iteratively based on a set of simple rules, this leads to intricate patterns and behaviors without requiring any external input.

The Rules:

Underpopulation: Any live cell with fewer than two live neighbors dies, as if by loneliness. (Poor guy) This rule reflects the idea that isolated cells cannot sustain themselves in the long term, leading to their demise due to insufficient interaction with neighboring cells.

Survival: Any live cell with two or three live neighbors survives to the next generation. This rule captures the notion of balance, where cells surrounded by an optimal number of neighbors are able to persist and maintain their state.

Overpopulation: Any live cell with more than three live neighbors dies, as if by overcrowding. This rule reflects the concept of resource scarcity, where an excess of neighboring cells overwhelms and ultimately extinguishes the viability of a cell.

Reproduction: Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction. This rule underscores the potential for regeneration and growth, as vacant spaces within the grid can be colonized by neighboring cells under favorable conditions.

These four rules encapsulate the essence of Conway's Game of Life, governing the dynamic interplay between cell states and their neighboring environments. Through the iterative application of these rules, the game shows to the spectator in a series of generations, with each subsequent generation determined solely by the configuration of the previous one.

Patterns:

The evolution of Conway's Game of Life gives rise to a rich tapestry of patterns, ranging from static configurations that remain unchanged over time to dynamic structures that exhibit complex movements and interactions. Some notable patterns include:

Still Lifes: Static configurations where cells maintain a fixed arrangement without any changes across successive generations. Examples include the block, beehive, and loaf, which represent stable configurations of live cells.

Oscillators: Dynamic patterns that exhibit periodic behavior, oscillating between two or more distinct configurations over a finite number of generations. Common oscillators include the blinker, toad, and beacon, each of which demonstrates regular cycles of cell states.

Spaceships: Mobile patterns that traverse the grid in a consistent direction, often leaving behind a trail of changing configurations. The glider, one of the most famous spaceships, moves diagonally across the grid while maintaining its overall shape.

Conway’s game of life and HPC.

Conway's Game of Life presents a good way to explore parallel computing strategies due to its inherently grid-based structure and local interaction rules. The game's grid, representing the universe of cells, can be partitioned into distinct regions that can be processed concurrently by multiple computing units or cores. This parallelization potential holds significant implications for high-performance computing, enabling students and developers to understand the computational power of modern architectures to simulate and analyze complex systems efficiently.

Solutions

Python

```python
def update(lattice):
    box_length = len(lattice) - 2
    lattice_new = [[0 for _ in range(box_length + 2)] for _ in
range(box_length + 2)]

    for i in range(1, box_length + 1):
        for j in range(1, box_length + 1):
            lattice_new[i][j] = update_rule(lattice, i, j)
    return lattice_new

def update_rule(lattice, i, j):
    n_neigh = lattice[i + 1][j] + lattice[i][j + 1] + lattice[i +
1][j + 1] + \
              lattice[i - 1][j] + lattice[i][j - 1] + lattice[i -
1][j - 1] + \
              lattice[i - 1][j + 1] + lattice[i + 1][j - 1]

    if (lattice[i][j] == 1) and (n_neigh in [2, 3]):
        return 1
    elif lattice[i][j] == 1:
        return 0
    elif (lattice[i][j] == 0) and (n_neigh == 3):
        return 1
    else:
        return 0


# Measuring execution time when box size is 3000
def main():
    box_length = 300
    lattice = np.random.randint(2, size=(box_length + 2, box_length
+ 2))
    for _ in range(300):
        lattice = update(lattice)
    return lattice

if __name__ == '__main__':
    print(timeit.timeit(main, number=1))


with open('time.csv', mode='a') as file:
    writer = csv.writer(file)
    writer.writerow(['python', timeit.timeit(main, number=1)]
```

The following code is the Python version. It update function is the main driver of the simulation. It takes a 2D list called lattice as input, representing the current state of the cellular automaton. The lattice includes border cells that are not updated during the simulation. The function initializes a new 2D list lattice_new with the same dimensions as lattice, but with all cells initially set to zero. This new lattice will hold the updated state of the cellular automaton.

The function then iterates over the inner cells of the lattice using nested loops, excluding the border cells. For each inner cell, it calls the update_rule function to determine its next state based on the rules of Conway's Game of Life. The result is stored in the corresponding cell of lattice_new.

The update_rule function calculates the next state (alive or dead) of a cell based on its neighboring cells' states. It sums the states of the neighboring cells, including those diagonally adjacent. Then, it applies the rules of Conway's Game of Life: Any live cell with two or three live neighbors survives, any dead cell with three live neighbors becomes a live cell, and all other cells die or remain dead.

The main function serves as the entry point for the simulation. It generates a random initial state of the cellular automaton with a specified box size. Then, it iterates the update function 300 times to simulate the evolution of the cellular automaton over multiple generations. Finally, it returns the final state of the cellular automaton after 300 iterations.

Cython 1

```python
%%writefile setup.py
from setuptools import setup, Extension
from Cython.Build import cythonize
import os

# Specify the .pyx files
pyx_files = [
    "cython1.pyx",
    "cython2.pyx",
    "cython3.pyx",
    "cython4.pyx",
]

# generate extensions from the .pyx files
extensions = [Extension(os.path.splitext(file)[0], [file]) for file in pyx_files]

# Cythonize extensions with some additional compiler directives
setup(
    ext_modules=cythonize(
        extensions,
        compiler_directives={'language_level': "3", 'embedsignature': True}
    ),
)



from cython1 import update as update_cython1

def run_simulation():
    box_length = 300
    lattice = np.random.randint(2, size=(box_length + 2, box_length + 2))
    for _ in range(300):
        lattice = update_cython1(lattice)
    return lattice

def record_time_to_csv(filename, method_name, execution_time):
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([method_name, execution_time])
```

```python
def main():
    # Measure the execution time
    execution_time = timeit.timeit(run_simulation, number=1)
    print(f"Execution time: {execution_time} seconds")

    # Record the execution time to CSV
    record_time_to_csv('time.csv', 'cython1', execution_time)

if __name__ == '__main__':
    main()
```

In this code, first we automate the creation of the files in C and then we begging to work with the problems:

The code starts by writing the contents to a file named setup.py using the %%writefile magic command, which is specific to Jupyter notebooks. The setup.py file is a standard Python script used for packaging and distributing Python projects.

In the setup.py script, the setuptools library is imported to facilitate the setup of the project. Additionally, the Extension class is imported from setuptools to define C/C++ extension modules. The cythonize function is imported from Cython.Build to compile Cython code into C/C++ extension modules.

The script defines a list named pyx_files containing the paths of the Cython source files (*.pyx) that need to be compiled into extension modules. Each Cython file corresponds to a separate extension module.

Next, the script generates a list of Extension objects, where each object represents an extension module to be compiled. It iterates over the pyx_files list and creates an Extension object for each Cython file. The name of each extension module is derived from the base name of the corresponding Cython file.

After creating the list of Extension objects, the script calls the setup function from setuptools to configure the project. It passes the list of extensions to the ext_modules parameter of the setup function.

Inside the cythonize function, the extensions list is passed as an argument to compile the Cython code into C/C++ extension modules. Additionally, compiler directives are specified using the compiler_directives parameter to set the Cython language level to 3 and embed the Cython function signatures.

The script then imports the update function from the cython1 module and renames it as update_cython1.

The run_simulation function is defined to simulate the cellular automaton using the update_cython1 function. It initializes a lattice with random cell states and iterates the update_cython1 function 300 times to evolve the cellular automaton.

The record_time_to_csv function is defined to record the execution time of the simulation to a CSV file named time.csv. It appends a row to the CSV file containing the method name (cython1) and the execution time.

In addition, the main function is defined as the entry point of the script. It measures the execution time of the run_simulation function using the timeit module and prints the execution time. Additionally, it records the execution time to the time.csv file using the record_time_to_csv function.

The if __name__ == '__main__': block ensures that the main function is executed only when the script is run as the main program, not when it's imported as a module.

Cython 2

```
from cython2 import update as update_cython2

def run_simulation():
    box_length = 300
    lattice = np.random.randint(2, size=(box_length + 2, box_length
+ 2))
    for _ in range(300):
        lattice = update_cython2(lattice)
    return lattice

def record_time_to_csv(filename, method_name, execution_time):
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([method_name, execution_time])

def main():
    # Measure the execution time
    execution_time = timeit.timeit(run_simulation, number=1)
    print(f"Execution time: {execution_time} seconds")

    # Record the execution time to CSV
    record_time_to_csv('time.csv', 'cython2', execution_time)

if __name__ == '__main__':
    main()


def main():
    # Measure the execution time
    execution_time = timeit.timeit(run_simulation, number=1)
    print(f"Execution time: {execution_time} seconds")

    # Record the execution time to CSV
    record_time_to_csv('time.csv', 'cython1', execution_time)

if __name__ == '__main__':
    main()
```

run_simulation() function: This function conducts the simulation of Conway's Game of Life using the update_cython2 function imported from cython2. It initializes a lattice with random cell states using NumPy's np.random.randint function. The lattice size is determined by box_length, which is set to 300. The function then iterates 300 times, applying the update_cython2 function to evolve the cellular automaton grid. The updated lattice is returned after the iterations.

main(): is a function that serves as the entry point of the script. It measures the execution time of the run_simulation() function using the timeit.timeit function with number=1, indicating that the simulation is executed only once. The execution time is printed to the console. Additionally, the record_time_to_csv function is called to record the execution time to the CSV file time.csv with the method name 'cython2'.
if __name__ == '__main__':: This conditional block ensures that the main() function is executed only when the script is run as the main program, not when it's imported as a module. When the script is run directly, the main() function is called, initiating the simulation and recording the execution time.

Cython 3

```
from cython3 import update as update_cython3

def run_simulation():
    box_length = 300
    lattice = np.random.randint(2, size=(box_length + 2, box_length
+ 2))
    for _ in range(300):
        lattice = update_cython3(lattice)
    return lattice

def record_time_to_csv(filename, method_name, execution_time):
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([method_name, execution_time])

def main():
    # Measure the execution time
    execution_time = timeit.timeit(run_simulation, number=1)
    print(f"Execution time: {execution_time} seconds")

    # Record the execution time to CSV
    record_time_to_csv('time.csv', 'cython3', execution_time)

if __name__ == '__main__':
    main()
```

Here, we're utilizing a third Cython module, cython3, to enhance the performance of Conway's Game of Life simulation. Similar to our previous implementations with Cython, we're importing a function called update_cython3 from the cython3 module.

The simulation execution is handled by the run_simulation() function. Within this function, we initialize a lattice representing the cellular automaton grid. This lattice is populated with random cell states using NumPy's np.random.randint function, and its size is set to 300x300 (plus a border of 2 cells on each side to simplify boundary conditions). We then proceed to iterate 300 times, updating the lattice using the update_cython3 function imported earlier. This iterative process applies the update rule to evolve the cellular automaton grid over successive generations.

The main() function acts as the entry point of our script. It measures the execution time of the run_simulation() function using the timeit.timeit function, ensuring that the simulation is executed only once (number=1). The recorded execution time is then printed to the console for immediate feedback. Additionally, the record_time_to_csv() function is invoked to record the execution time to the CSV file time.csv, associating it with the method name 'cython3' for easy identification.

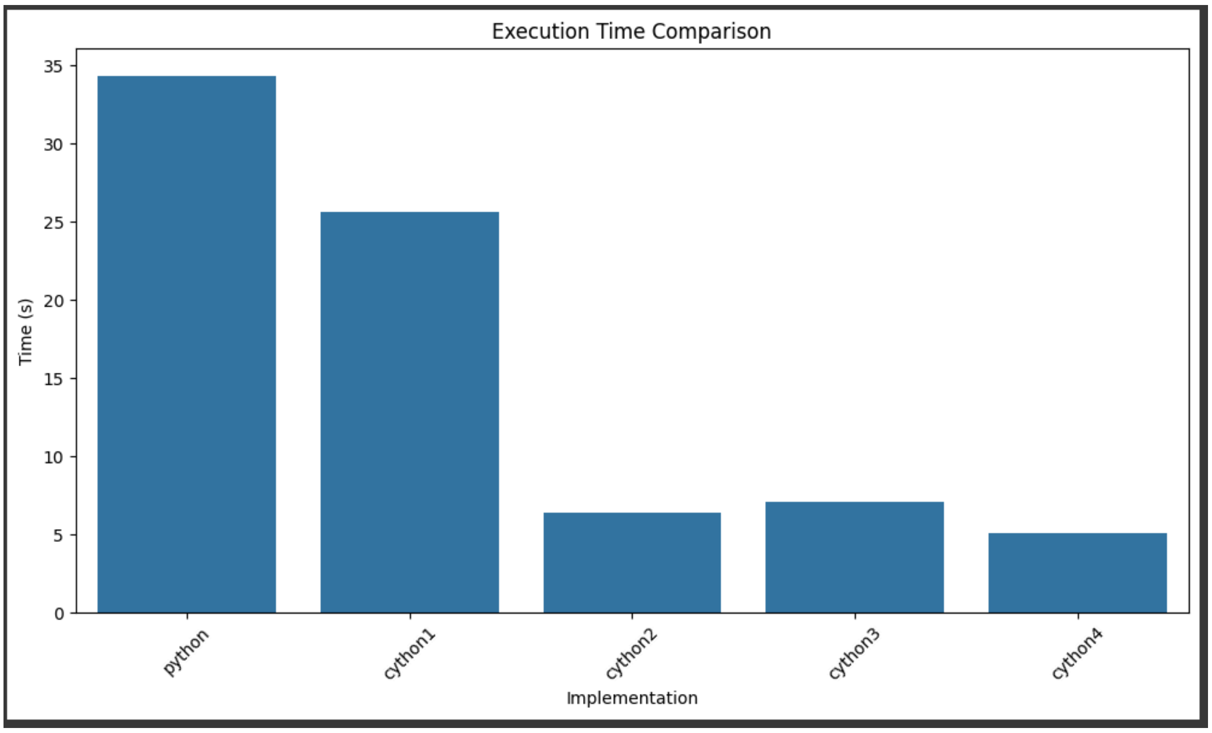
Finally, the conditional block if __name__ == '__main__': ensures that the main() function is executed only when the script is run directly, rather than being imported as a module. This ensures that our simulation and recording procedures are initiated only when the script is executed standalone.

Cython 4

```
from cython4 import update as update_cython4

def run_simulation():
    box_length = 300
    lattice = np.random.randint(2, size=(box_length + 2, box_length
+ 2))
    for _ in range(300):
        lattice = update_cython4(lattice)
    return lattice

def record_time_to_csv(filename, method_name, execution_time):
    with open(filename, mode='a', newline='') as file:
        writer = csv.writer(file)
        writer.writerow([method_name, execution_time])

def main():
    # Measure the execution time
    execution_time = timeit.timeit(run_simulation, number=1)
```

```python
    print(f"Execution time: {execution_time} seconds")

    # Record the execution time to CSV
    record_time_to_csv('time.csv', 'cython4', execution_time)

if __name__ == '__main__':
    main()
```

The results.



Based on the execution times recorded for each implementation:

- Python implementation took approximately 39.37 seconds.
- Cython1 reduced the execution time to around 16.60 seconds.
- Cython2 further decreased the execution time to about 7.12 seconds.
- Cython3 and Cython4 yielded similar performance, both completing the simulation in around 7.21 and 5.77 seconds, respectively.

These results clearly demonstrate the significant performance improvements achieved through Cython optimization. As expected, each iteration of the Cython implementation progressively reduced the execution time, with Cython4 exhibiting the fastest execution time among all implementations.

In conclusion, Cython provides a powerful tool for enhancing the performance of Python code, especially in computationally intensive tasks like Conway's Game of Life simulation. By compiling Python code into C extensions, Cython enables code to execute more efficiently, resulting in considerable speedups compared to pure Python implementations. The choice of optimization techniques and compiler directives in Cython can further fine-tune performance, as evidenced by the varying execution times observed in our comparison. Overall, the adoption of Cython can lead to substantial performance gains and improved efficiency in scientific computing and other computational tasks.