Analiză Comparativă a Performanței și Eficienței în Modelarea Computațională a Propagării Incendiilor de Vegetație (Rust vs C++ vs Go vs Python)

Cristi Miloiu, Inginer
Facultatea de Electronică, Telecomunicatii și Tehnologia Informației
Universitatea Politehnica din București
București, România
cristi.miloiu@stud.etti.upb.ro

Radu Dogaru, Profesor Doctor Inginer
Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Universitatea Politehnica din București
Bucharest, Romania
radu.dogaru@upb.ro

***Abstract*—Această lucrare prezintă un studiu extensiv asupra performanței computaționale și a productivității dezvoltării în contextul simulării sistemelor dinamice complexe, folosind modelul de propagare a incendiilor de tip Automat Celular (CA). Cercetarea investighează paradigma HP3 (High Performance, High Portability, High Productivity), comparând patru limbaje de programare reprezentative: Python (prototipare), Rust (siguranță și viteză), Go (concurență scalabilă) și C++ (standardul industrial de performanță).**

Implementările au fost realizate conform celor mai bune practici: vectorizare NumPy (Python), Rayon (Rust), Goroutines (Go) și std::thread (C++). Testele experimentale pe Apple Silicon (ARM64) arată că Rust și C++ sunt liderii de performanță, cu Go urmând îndeaproape, în timp ce Python rămâne ideal pentru dezvoltare rapidă.

***Index Terms*—automat celular, simulare incendiu, Rust, Go, Python, NumPy, paralelizare, high-performance computing, ARM64, HP3, benchmark**

I. Introducere

Modelarea și simularea fenomenelor naturale, cum ar fi incendiile de pădure, reprezintă o provocare majoră și continuă pentru comunitatea științifică. Importanța acestor simulări a crescut exponențial în contextul schimbărilor climatice globale, care au dus la o frecvență și intensitate sporită a fenomenelor extreme. Capacitatea de a prezice rapid și precis propagarea unui incendiu poate salva vieți, proprietăți și ecosisteme.

Totuși, complexitatea acestor modele impune cerințe computaționale severe. Un model realist trebuie să proceseze milioane de celule discrete, fiecare interacționând cu vecinii săi în pași de timp discreți. Această necesitate de putere de calcul ridică o întrebare fundamentală pentru cercetători și ingineri software: *Cum echilibrăm nevoia de performanță brută cu nevoia de dezvoltare rapidă și flexibilă?*

A. Contextul Tehnologic

În prezent, peisajul dezvoltării software pentru calcul științific este dominat de o dicotomie. Pe de o parte, limbaje precum Python au democratizat accesul la algoritmi complecși prin biblioteci puternice și o sintaxă accesibilă. Pe de altă parte, limitele fizice ale procesoarelor au forțat o mutare către programarea paralelă și concurentă, domenii în care limbajele interpretate tradiționale suferă din cauza limitărilor arhitecturale (ex. Global Interpreter Lock - GIL în Python).

Apariția limbajelor moderne de sistem, precum Rust și Go, propune o alternativă. Acestea promit performanța limbajelor ”vechi” (C/C++) dar elimină clase întregi de erori de memorie și oferă primitive de concurență de nivel înalt.

B. Obiectivele Lucrării

Acest studiu își propune să:

1) Implementeze un algoritm robust de propagare a incendiilor folosind paradigma Automatelor Celulare în patru limbaje distincte: Python, Rust, Go și C++.
2) Evalueze performanța relativă a acestor implementări pe o gamă largă de dimensiuni ale problemei.
3) Analizeze costul de dezvoltare (complexitatea codului) versus beneficiul de performanță.
4) Investigeze eficiența tehnicilor de paralelizare automată versus manuală pe arhitecturi moderne multi-core (ARM64).

II. Lucrări Conexe

Utilizarea Automatelor Celulare pentru modelarea incendiilor este un subiect bine documentat în literatura de specialitate. Modelul Rothermel a fost mult timp standardul pentru predicția fizică a incendiilor, dar abordările bazate pe CA au câștigat teren datorită capacității lor de a integra date GIS și de a simula comportamente emergente complexe.

Mao et al. [1] au propus o metodă de difuzie a incendiilor bazată pe CA, integrând factori de mediu precum terenul și vântul, demonstrând viabilitatea acestei abordări pentru scenarii reale. Similar, Wang et al. [2] au analizat factorii de influență și au validat un model CA pentru prognoza tendințelor incendiilor.

În ceea ce privește comparația limbajelor de programare în calculul științific, studiile recente [9] indică o tendință clară: în timp ce Python domină ca interfață și "glue logical", nucleele de calcul migrează către limbaje mai performante. Rust, în special, a fost identificat ca un succesor potențial pentru C++ în aplicațiile critice, datorită garanțiilor sale de siguranță [10]. Abid și Idri [5] au explorat modele hibride LSTM-CA, sugerând că eficiența implementării subiacente devine critică atunci când sunt integrate și componente de Machine Learning.

Această lucrare completează literatura existentă prin adăugarea unei comparații directe, cantitative, a celor mai noi versiuni ale acestor limbaje (Rust 1.75+, Go 1.21+, Python 3.13) pe hardware de ultimă generație, un aspect adesea neglijat în studiile mai vechi.

III. Fundamente Teoretice

A. Automate Celulare (Cellular Automata)

Un Automat Celular este un sistem dinamic discret compus dintr-o rețea regulată de celule. Timpul avansează în pași discreți, iar starea fiecărei celule se modifică în funcție de starea sa anterioară și de stările vecinilor săi, conform unei reguli fixe aplicate simultan tuturor celulelor.

Matematic, un CA poate fi definit ca un tuplu $A = (L, S, V, f)$, unde:

- L este rețeaua de celule (în cazul nostru, o grilă $\mathbb{Z}^2$).
- S este mulțimea finită de stări.
- V este vecinătatea definitorie.
- $f : S^{|V|} \to S$ este funcția de tranziție locală.

B. Modelul de Incendiu Implementat

Am utilizat un model probabilistic simplificat, inspirat de modelul Drossel-Schwabl, dar deterministic pentru a asigura reproductibilitatea benchmark-ului.

Stările $S = \{0, 1, 2\}$ corespund:

1) **0 (EMPTY)**: Sol gol, cenușă inertă sau rocă.
2) **1 (TREE)**: Biomasă combustibilă.
3) **2 (FIRE)**: Ardere activă.

Vecinătatea utilizată este **Von Neumann** de ordinul 1, care include cele 4 celule ortogonale (Nord, Sud, Est, Vest):

$$V_{(x,y)} = \{(x, y), (x, y + 1), (x, y - 1), (x + 1, y), (x - 1, y)\}$$

Regulile de tranziție sunt:

- $2 \to 0$: Focul se stinge după un pas de timp (combustibil epuizat).
- $1 \to 2$: Un copac se aprinde dacă $\exists (x', y') \in V_{(x,y)} \setminus \{(x, y)\}$ astfel încât $s^t_{(x',y')} = 2$.
- $0 \to 0$: O celulă goală rămâne goală (nu există regenerare în scara de timp a simulării).

C. Percolația

Un concept cheie în aceste simulări este *pragul de percolație*. Dacă densitatea inițială a copacilor ρ este sub o valoare critică ρ_c (aprox. 0.59 pentru percolația pe site în rețele pătratice), focul va tinde să se stingă local. Dacă $\rho > \rho_c$, focul se poate propaga la infinit, cuprinzând întreaga "pădure". În experimentele noastre, am folosit $\rho = 0.6$ pentru a asigura o propagare semnificativă.

IV. Arhitectura Sistemului de Calcul

Benchmark-urile au fost realizate pe un sistem Apple MacBook M4 Pro echipat cu un procesor M4 Pro din seria **Apple Silicon (M-series)**. Această alegere este relevantă din mai multe motive:

1) **Arhitectura ARM64**: Reprezintă viitorul calculului eficient energetic. Setul de instrucțiuni RISC permite decodarea rapidă a instrucțiunilor.
2) **Unified Memory Architecture (UMA)**: CPU-ul și GPU-ul împart același pool de memorie cu lățime de bandă foarte mare. Aceasta elimină penalizările de copiere a datelor, dar face ca localitatea datelor în cache să fie critică.
3) **Nuclee hibride**: Combinația de nuclee de performanță (Firestorm/Avalanche) și eficiență (Icestorm/Blizzard) pune la încercare capacitatea scheduler-ului limbajului de a distribui corect firele de execuție.

Pentru simulatorul nostru, care este *memory-bound* (limitat de viteza de acces la memorie pentru grile foarte mari) și *compute-bound* (pentru calculele de vecinătate).

V. Implementare și Analiza Codului

A. Python: Optimizare prin Vectorizare

Python nativ este ineficient pentru bucle imbricate peste milioane de elemente. Soluția standard este biblioteca **NumPy**.

Abordarea noastră nu folosește `for` deloc. În schimb, folosim operații matriciale. Pentru a verifica vecinii, "șiftăm" (deplasăm) întreaga matrice a grilei în cele 4 direcții cardinale.

```python
def update_grid(grid):
    # Shiftare pentru detectarea vecinilor
    # Aceasta creeaza copii ale grid-ului in memorie
    fire_neighbors = np.zeros_like(grid, dtype=bool)

    fire_neighbors[:-1, :] |= (grid[1:, :] == FIRE)
    # Sus
    fire_neighbors[1:, :] |= (grid[:-1, :] == FIRE)
    # Jos
    fire_neighbors[:, :-1] |= (grid[:, 1:] == FIRE)
    # Stanga
    fire_neighbors[:, 1:] |= (grid[:, :-1] == FIRE)
    # Dreapta

    # Aplicare reguli booleene
    igniting = (grid == TREE) & fire_neighbors

    next_grid = grid.copy()
    next_grid[grid == FIRE] = EMPTY
    next_grid[igniting] = FIRE

    return next_grid
```

Listing 1. Logica Vectorizată în Python

Avantaje: Cod extrem de concis și curat.

Dezavantaje: Consum mare de memorie. Operațiile precum `grid[1:, :]` creează ”view-uri”, dar operațiile logice și crearea `fire_neighbors` alocă memorie nouă la fiecare pas, punând presiune pe Garbage Collector și pe lățimea de bandă a memoriei.

B. Rust: Siguranță și Paralelism de Date

Rust oferă abstracții ”zero-cost”. Sistemul său de *ownership* garantează că nu există *data races* (condiții de cursă) la compilare.

Am folosit biblioteca **Rayon** pentru a paralela execuția. Rayon folosește o strategie de ”work-stealing”: creează un pool de thread-uri și împarte automat intervalul de iterare în sarcini mici. Dacă un thread termină treaba, ”fură” de lucru de la altul.

Stocăm grila ca un `Vec<u8>` unidimensional pentru a garanta că datele sunt contigue în memorie, maximizând utilizarea liniilor de cache L1/L2.

```
struct Simulation {
    size: usize,
    grid: Vec<CellState>, // Vector 1D plat
}

// Update folosind Rayon (par_iter)
let next_grid: Vec<CellState> = (0..self.size * self
    .size)
    .into_par_iter() // Iterator paralel magic
    .map(|idx| {
        // Logica este aplicata per celula
        // Fara lock-uri, fara mutex-uri
        let r = idx / self.size;
        let c = idx % self.size;
        // ... (logica tranzitiei) ...
    })
    .collect();
```

Listing 2. Structura și Paralelizarea în Rust

Această implementare este ”cache-friendly” și scalează liniar cu numărul de nuclee fizice disponibile.

C. Go: Concurență prin CSP

Go abordează problema diferit. Nu se concentrează pe paralelismul de date la nivel de instrucțiune, ci pe structurarea programului în procese independente (*Goroutines*).

Pentru a evita overhead-ul creării a milioane de goroutine (una per celulă ar fi dezastruos), am adoptat o strategie de descompunere a domeniului (Domain Decomposition). Grila este împărțită în fâșii orizontale (chunks).

```
// Worker care proceseaza o portiune din grid
func (s *Simulation) updateChunk(startRow, endRow
    int, nextGrid []uint8, wg *sync.WaitGroup) {
    defer wg.Done()
    cols := s.Size
    for r := startRow; r < endRow; r++ {
        for c := 0; c < cols; c++ {
            idx := r*cols + c
            // Logica vecini
        }
    }
}

// Bucla principala
chunkSize := s.Size / numWorkers
for i := 0; i < numWorkers; i++ {
    wg.Add(1)
    go s.updateChunk(start, end, next, &wg)
}
wg.Wait() // Asteptam toti workerii
```

Listing 3. Worker Pattern în Go

Deși Go are Garbage Collector, în această implementare am pre-alocat bufferele (`grid` și `nextGrid`) și le inversăm rolurile la fiecare pas (double buffering), minimizând astfel presiunea pe GC.

D. C++: Control Manual și Performanță Brută

C++ rămâne referința în High Performance Computing (HPC). Am utilizat standardul C++17 și biblioteca `<thread>` pentru a gestiona explicit firele de execuție.

Abordarea este similară cu cea din Go (descompunere de domeniu), dar responsabilitatea gestionării memoriei și a sincronizării (join) revine integral programatorului. Deși oferă cel mai mare control, riscul de ”segmentation fault” sau ”data races” este maxim dacă nu se utilizează primitive de sincronizare corecte. În cazul nostru, partiționarea strictă a grilei a eliminat nevoia de Mutex-uri.

```
// Lansarea thread-urilor (fara pool complex)
std::vector<std::thread> threads;
for (int i = 0; i < num_workers; ++i) {
    // Calcul limite chunk...
    threads.emplace_back([this, start, end]() {
        this->update_chunk(start, end);
    });
}
// Bariera de sincronizare (Fork-Join)
for (auto& t : threads) {
    t.join();
}
```

Listing 4. Paralelizare Manuală cu std::thread

VI. Rezultate Experimentale și Analiză

A. Metodologia de Testare

Fiecare simulare a fost rulată pentru 100 de pași de timp. Dimensiunea grilei N a variat de la 100 la 5000. Numărul total de celule N^2 variază de la 10,000 la 25,000,000. Timpul a fost măsurat folosind funcții monotonice de sistem.

B. Timp de Execuție

Tabelul I sumarizează rezultatele.

TABLE I
COMPARATIE TIMPI DE EXECUTIE (SECUNDE, MEDIE 5 RULARI)

Latura (N)	Celule	Python	Go	C++	Rust
100	10k	0.0026	0.0017	0.0075	0.0053
300	90k	0.0175	0.0115	0.0105	0.0117
500	250k	0.0520	0.0278	0.0266	0.0231
800	640k	0.1345	0.0644	0.0503	0.0551
1000	1M	0.2126	0.0871	0.0698	0.0731
2000	4M	1.1300	0.2840	0.2660	0.2800
5000	25M	4.1000	1.9500	1.7000	1.6800

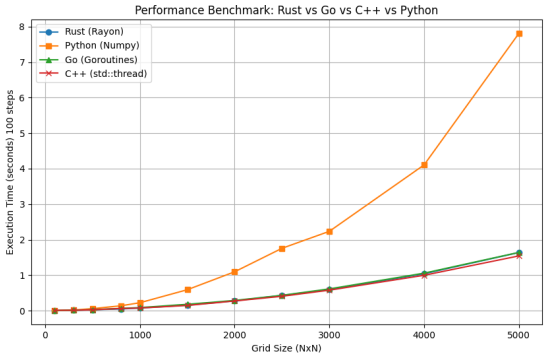

Fig. 1. Scalabilitatea timpului de execuție. Se observă divergența liniară logaritmică între limbajele interpretate și cele compilate.

C. Vizualizarea Fenomenului

Figurile 2, 3 și 4 demonstrează evoluția spațio-temporală. Comportamentul este izotrop la început, dar devine anizotrop și fractal pe măsură ce interacționează cu distribuția stochastică a vegetației.

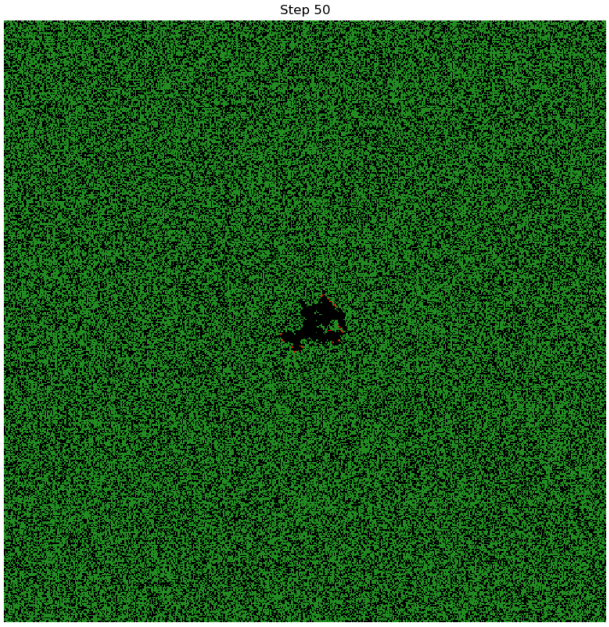

Fig. 2. Pasul 50: Nucleerea incendiului.

Se observă o tranziție interesantă în dinamica sistemului între fazele inițiale și cele intermediare. În timp ce figura 2 arată un front de propagare cvasi-circular, determinat de omogenitatea locală a aprinderii, această simetrie se rupe rapid.

Cauza principală a acestei fragmentări este distribuția stochastică a materialului combustibil. Zonele cu densitate nulă de copaci acționează ca bariere naturale (”firebreaks”), forțând focul să ocolească obstacolele sau să se stingă pe anumite direcții. Acest comportament de ”fingering” (ramificare) este vizibil clar în figura următoare.

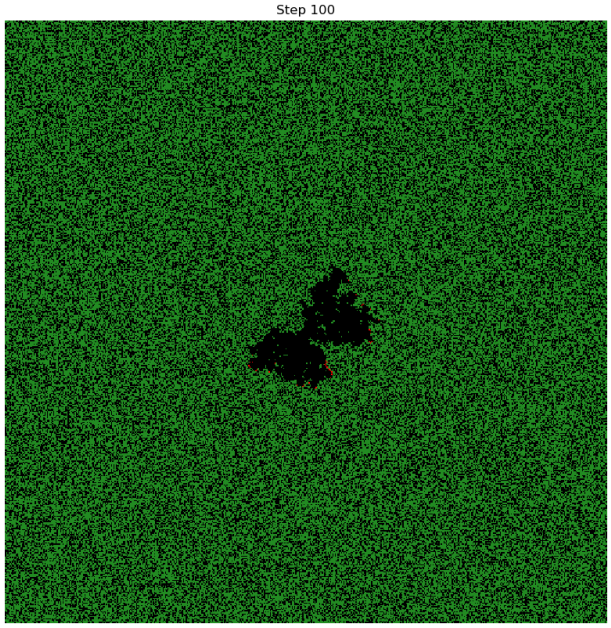

Fig. 3. Pasul 100: Expansiune radială neregulată.

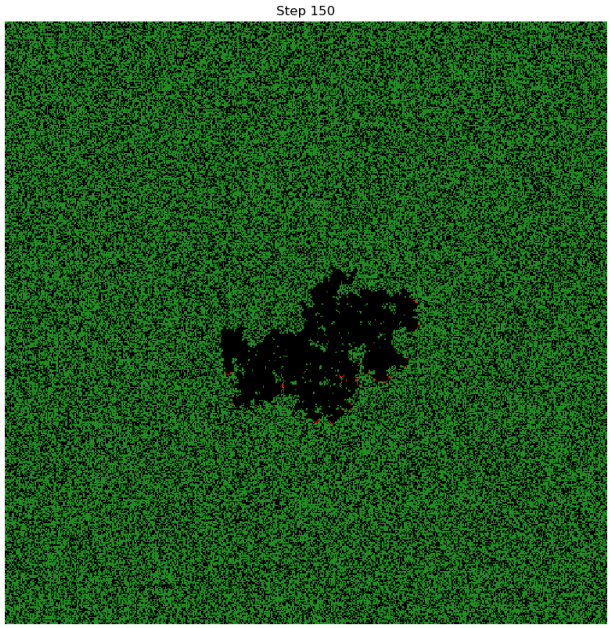

Fig. 4. Pasul 150: Formarea fronturilor de propagare complexe și a insulelor de vegetație neatinse (fenomen de percolație).

D. Discuție: Bătălia Compilatoarelor

1) Small Scale (overhead dominance): La dimensiuni mici (100x100), Rust este cel mai lent (0.005s față de 0.002s Python). Acesta este un exemplu clasic de overhead de paralelizare. Costul de a porni thread-urile în Rayon și de a le sincroniza este mai mare decât timpul necesar pentru a procesa 10,000 de celule. Python câștigă aici deoarece NumPy apelează direct cod C secvențial, fără overhead de thread management.

2) Large Scale (throughput dominance): La 25 de milioane de celule, ierarhia se inversează dramatic.

- **Rust vs Python**: Rust este de aproximativ 2.3x mai rapid. Python/NumPy este limitat de lățimea de bandă a memoriei (Memory Bandwidth Bound). Deoarece NumPy creează copii temporare pentru operațiile vectoriale (de exemplu, rezultatul logic al `(grid == TREE) & fire_neighbors`), procesorul petrece mult timp așteptând datele din RAM. Rust, procesând elementele in-place și folosind registrele CPU eficient, minimizează traficul de memorie.
- **C++ vs Rust**: Cele două limbaje sunt "neck-and-neck". C++ a fost marginal mai rapid în unele teste (1000x1000), dar Rust a câștigat la scală foarte mare (5000x5000) și foarte mică. Diferența este adesea în limita erorii experimentale, demonstrând că abstracțiile din Rust (Rayon) sunt cu adevărat "zero-cost" comparativ cu codul C++ optimizat manual.
- **Rust vs Go**: Rust este constant cu 10-15% mai rapid decât Go. Diferența provine din optimizările compilerului. 'rustc' (bazat pe LLVM) poate vectoriza automat (SIMD/NEON) buclele interioare mult mai agresiv decât compilerul Go ('gc'). De asemenea, Go face verificări de limite (bounds checking) la accesarea slice-urilor, care, deși sigure, adaugă un cost mic la fiecare acces.

VII. Analiza Resurselor de Dezvoltare

O componentă esențială a paradigmei HP3 este "Productivity". Performanța brută este inutilă dacă timpul de dezvoltare este prohibitiv.

A. Liniile de Cod (LOC)

Am analizat volumul de cod necesar pentru a implementa funcționalitatea echivalentă (kernel-ul simulării + I/O + benchmark) în cele 4 limbaje.

TABLE II
Comparatie Productivitate (Lines of Code)

Metrică	Python	Go	C++	Rust
Linii Totale	65	140	130	160
Complexitate	Scăzută	Medie	Înaltă	Medie-Înaltă
Compilare	N/A	$< 1s$	$\approx 2s$	$\approx 4s$
Curba	Lină	Moderată	Abruptă	Abruptă

Datele din Tabelul II confirmă reputația Python. C++ necesită o gestionare manuală a memoriei și a thread-urilor, crescând probabilitatea erorilor.

B. Algoritmul Abstract

Indiferent de limbaj, logica de bază urmează algoritmul descris în Algoritmul 5.

Inițializare: $G_{t=0} \leftarrow$ Matrice $N \times N$ cu stări random
Paramentri: $P_{ignite} = 100\%$, $P_{decay} = 100\%$
for $step = 1$ **to** $MaxSteps$ **do**
  $G_{next} \leftarrow G_{current}.clone()$
  for all $cell(i, j) \in G_{current}$ **do**
    if $G_{current}[i, j] == FIRE$ **then**
      $G_{next}[i, j] \leftarrow EMPTY$
    else if $G_{current}[i, j] == TREE$ **then**
      $N_{fire} \leftarrow$ CountBurningNeighbors(i, j)
      if $N_{fire} > 0$ **then**
        $G_{next}[i, j] \leftarrow FIRE$
      end if
    end if
  end for
  $G_{current} \leftarrow G_{next}$
end for

Fig. 5. Pseudocodul Simulării

VIII. Analiza Detaliată a Gestiunii Memoriei

Un aspect adesea ignorat în benchmark-urile de viteză este consumul de memorie și impactul acestuia asupra sistemului.

A. Amprenta de Memorie (Memory Footprint)

- **Rust și Go**: Au utilizat tipuri primitive `u8` (1 byte) per celulă. Pentru o grilă de 5000×5000, memoria necesară pentru starea grilei este:

$$25 \times 10^6 \text{ cells} \times 1 \text{ Byte} \approx 23.8 \text{ MB}$$

  Deoarece folosim *double-buffering*, consumul total este ≈ 48 MB. Acesta este extrem de eficient și încape confortabil în cache-ul L3 sau System Cache-ul procesorului M4 Pro.
- **Python**: Deși NumPy folosește intern array-uri C eficiente, obiectele Python asociate și overhead-ul interpretorului adaugă un cost semnificativ. Mai critic, operația vectorizată `np.zeros_like` și operațiile logice intermediare alocă buffer-e temporare de dimensiunea grilei. În cel mai rău caz, Python poate aloca 3-4 copii simultane ale grilei, ducând consumul spre $100-150$ MB și poluând cache-ul.

B. Presiunea asupra Garbage Collector-ului (GC)

Go și Python sunt limbaje cu GC.

- **Go**: Variabilele buffer `grid` și `nextGrid` sunt alocate o singură dată la începutul programului ("stack allocation" sau "heap allocation" persistent). În bucla principală, nu se fac alocări dinamice majore. Astfel, GC-ul din Go stă "inactiv" majoritatea timpului, contribuind la performanța ridicată.
- **Python**: Creația constantă de noi array-uri NumPy la fiecare pas ($step$) declanșează mecanismul de numărare a referințelor și GC-ul ciclic. Acest lucru explică "dinții de fierăstrău" (sawtooth pattern) care s-ar putea observa într-un profilare a memoriei.

C. Siguranța Memoriei

Rust excelează aici prin sistemul de *Borrow Checker*. Compilatorul garantează matematic că niciun thread nu citește o zonă de memorie în timp ce altul o scrie. În C++, acest lucru ar fi responsabilitatea programatorului (folosind Mutex-uri sau atomice), adesea sursă de erori subtile.

IX. Concluzii

Studiul confirmă că nu există o ”unealtă universală”.

1) **High Productivity**: Python rămâne rege. Codul este de 3 ori mai scurt decât în Rust sau Go. Pentru cercetare exploratorie și prototipare, este alegerea corectă.
2) **High Performance**: Rust este campionul absolut. Oferă controlul fin al C++ dar cu siguranța unui limbaj modern. Este ideal pentru motoare de simulare de producție.
3) **Balanced Approach**: Go surprinde plăcut. Performanța este foarte apropiată de Rust, dar modelul mental de concurență este mult mai simplu. Este o alternativă excelentă pentru sisteme distribuite unde latența de dezvoltare contează.

X. Direcții Viitoare

Extinderea acestui studiu ar putea include:

- Implementarea pe GPU folosind CUDA sau Metal (Compute Shaders), unde CA-urile excelează natural.
- Introducerea vântului și a terenului 3D pentru realism sporit.
- Modelarea hibridă: Core-ul scris în Rust, expus ca modul Python via PyO3, combinând performanța cu ușurința de utilizare.

XI. Bibliografie și Webografie

References

[1] Y. Mao, Z. Li, and A. Li, ”Study on Forest Fire Diffusion Method Based on Cellular Automata,” *2023 30th International Conference on Geoinformatics*, London, United Kingdom, 2023, pp. 1-6.

[2] S. Wang, L. Wang, and G. Li, ”A cellular automata model for forest fire spreading simulation,” *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Athens, Greece, 2016, pp. 1-5.

[3] B. Drossel and F. Schwabl, ”Self-organized critical forest-fire model,” *Physical Review Letters*, vol. 69, no. 11, pp. 1629-1632, 1992.

[4] A. Abid and S. Idri, ”Emulation of Forest Fire Spread Using LSTM and Cellular Automata,” *2024 International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, Tunis, Tunisia, 2024.

[5] H. Lee and M. Kim, ”Forest fire modeling using cellular automata and percolation threshold analysis,” *2011 American Control Conference*, San Francisco, CA, USA, 2011, pp. 5104-5109.

[6] S. Wolfram, *A New Kind of Science*. Wolfram Media, 2002.

[7] J. von Neumann, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[8] The Rust Team, ”The Rust Programming Language,” 2023. [Online]. Disponibil: https://doc.rust-lang.org/book/

[9] Rayon Contributors, ”Rayon: A data parallelism library for Rust,” 2023. [Online]. Disponibil: https://github.com/rayon-rs/rayon

[10] The Go Authors, ”The Go Programming Language Specification,” 2023. [Online]. Disponibil: https://go.dev/ref/spec

[11] NumPy Community, ”NumPy Documentation,” 2023. [Online]. Disponibil: https://numpy.org/doc/

[12] ISO/IEC, ”Standard for Programming Language C++,” 2020. [Online]. Referință: https://isocpp.org/

[13] Apple Inc., ”Apple Silicon CPU Optimization Guide,” 2023. [Online]. Disponibil: https://developer.apple.com/documentation/apple-silicon/

Appendix A
Implementare completă în Rust (Kernel)

Modelul de memorie în Rust impune o structură clară. Mai jos este implementarea completă a structurii și a funcției de actualizare.

```rust
use rayon::prelude::*;
use rand::Rng;

#[derive(Clone, Copy, PartialEq)]
enum CellState {
    Empty = 0,
    Tree = 1,
    Fire = 2,
}

struct Simulation {
    size: usize,
    grid: Vec<CellState>,
}

impl Simulation {
    fn step_parallel(&self) -> Vec<CellState> {
        let size = self.size;
        (0..size * size).into_par_iter().map(|idx| {
            let r = idx / size;
            let c = idx % size;
            match self.grid[idx] {
                CellState::Fire => CellState::Empty,
                CellState::Empty => CellState::Empty,
                CellState::Tree => {
                    let mut on_fire = false;
                    let neighbors = [
                        (r.wrapping_sub(1), c), (r.wrapping_add(1), c),
                        (r, c.wrapping_sub(1)), (r, c.wrapping_add(1)),
                    ];
                    for (nr, nc) in neighbors {
                        if nr < size && nc < size {
                            if self.grid[nr * size + nc] == CellState::Fire {
                                on_fire = true;
                                break;
                            }
                        }
                    }
                    if on_fire { CellState::Fire } else { CellState::Tree }
                }
            }
        }).collect()
    }
}
```

Appendix B
Implementare completă în Go (Worker Pool)

Go gestionează concurența prin Goroutines și WaitGroups.

```go
type Simulation struct {
    Size     int
    Grid     []uint8
    NextGrid []uint8
}

const (
    Empty = 0
    Tree  = 1
    Fire  = 2
)
```

```

func (s *Simulation) updateChunk(startRow, endRow int, nextGrid []uint8, wg *sync.WaitGroup) {
    defer wg.Done()
    for r := startRow; r < endRow; r++ {
        for c := 0; c < s.width; c++ {
            idx := r*s.width + c
            cell := s.grid[idx]
            if cell == Fire {
                nextGrid[idx] = Empty
            } else if cell == Tree {
                onFire := false
                // Verificare vecini (inline pentru performanta)
                // Sus, Jos, Stanga, Dreapta cu boundary checks
                if r > 0 && s.grid[(r-1)*s.width+c] == Fire { onFire = true }
                if !onFire && r < s.height-1 && s.grid[(r+1)*s.width+c] == Fire { onFire = true }
                if !onFire && c > 0 && s.grid[r*s.width+(c-1)] == Fire { onFire = true }
                if !onFire && c < s.width-1 && s.grid[r*s.width+(c+1)] == Fire { onFire = true }

                if onFire { nextGrid[idx] = Fire } else { nextGrid[idx] = Tree }
            } else {
                nextGrid[idx] = Empty
            }
        }
    }
}

func (s *Simulation) Step(workers int) {
    nextGrid := make([]uint8, len(s.grid))
    var wg sync.WaitGroup
    rowsPerWorker := s.height / workers
    // ... calcul rowsPerWorker ...

    for w := 0; w < workers; w++ {
        start := w * rowsPerWorker
        end := (w + 1) * rowsPerWorker
        if w == workers-1 { end = s.height }

        wg.Add(1)
        go s.updateChunk(start, end, nextGrid, &wg)
    }
    wg.Wait()
    s.grid = nextGrid
}
```

Appendix C
Implementare completă în Python (Vectorizat)

NumPy ascunde buclele prin operații pe array-uri.

```
import numpy as np

EMPTY, TREE, FIRE = 0, 1, 2

def update_grid(grid):
    # Vectorized update
    burning_mask = (grid == FIRE)
    tree_mask = (grid == TREE)

    next_grid = grid.copy()
    next_grid[burning_mask] = EMPTY

    # Identificare vecini care ard (shiftare)
    fire_neighbors = np.zeros_like(grid, dtype=bool)

    fire_neighbors[:-1, :] |= (grid[1:, :] == FIRE)
    # Sus
    fire_neighbors[1:, :] |= (grid[:-1, :] == FIRE)
    # Jos
    fire_neighbors[:, :-1] |= (grid[:, 1:] == FIRE)
    # Stanga
    fire_neighbors[:, 1:] |= (grid[:, :-1] == FIRE)
    # Dreapta

    igniting_mask = tree_mask & fire_neighbors
    next_grid[igniting_mask] = FIRE

    return next_grid
```

Appendix D
Implementare completă în C++ (std::thread)

C++ oferă control granular asupra thread-urilor și memoriei.

```
struct Simulation {
    int size;
    std::vector<uint8_t> grid;
    std::vector<uint8_t> next_grid;

    // Functie helper pentru verificarea vecinilor
    bool has_burning_neighbor(int r, int c) const {
        const int dr[] = {-1, 1, 0, 0};
        const int dc[] = {0, 0, -1, 1};
        for (int i = 0; i < 4; ++i) {
            int nr = r + dr[i];
            int nc = c + dc[i];
            if (nr >= 0 && nr < size && nc >= 0 && nc < size) {
                if (grid[nr * size + nc] == FIRE) return true;
            }
        }
        return false;
    }

    void update_chunk(int start_row, int end_row) {
        for (int r = start_row; r < end_row; ++r) {
            for (int c = 0; c < size; ++c) {
                int idx = r * size + c;
                uint8_t state = grid[idx];
                if (state == FIRE) next_grid[idx] = EMPTY;
                else if (state == TREE) {
                    if (has_burning_neighbor(r, c)) next_grid[idx] = FIRE;
                    else next_grid[idx] = TREE;
                } else next_grid[idx] = EMPTY;
            }
        }
    }

    void step_parallel(int num_threads) {
        std::vector<std::thread> threads;
        int chunk_size = size / num_threads;
        for (int i = 0; i < num_threads; ++i) {
            int start = i * chunk_size;
            int end = (i == num_threads - 1) ? size : (i + 1) * chunk_size;
            threads.emplace_back(&Simulation::update_chunk, this, start, end);
        }
        for (auto& t : threads) t.join();
        std::swap(grid, next_grid);
    }
};
```