Ray Tracing Paralel

Moraru Radu Andrei 462
Licu Mihai George 462

1. Introducere

Ray tracing-ul este o tehnică de generare a imaginilor care simulează interacțiunea luminii cu obiectele dintr-o scenă tridimensională pentru a crea imagini realiste. Aceasta implică urmărirea traiectoriei razelor de lumină pe măsură ce se deplasează prin spațiu și interacționează cu diverse suprafețe. Algoritmul calculează efecte vizuale complexe precum reflexii, refracții și umbre, rezultând o redare extrem de fidelă a realității. Pentru fiecare pixel din imagine se trasează o rază de lumină, a cărei culoare și traiectorie sunt afectate de obiectele întâlnite în scenă. În cazul imaginilor de înaltă rezoluție, procesul de ray tracing devine extrem de intensiv, deoarece raza poate interacționa cu multiple obiecte, necesitând calcule repetate de intersecții și proprietăți optice. Această tehnică este folosită preponderent în industriile de divertisment, grafică pentru jocuri video și cinematografie pentru generarea de efecte vizuale și animații 3D, dar și în design, arhitectură și simulări științifice.

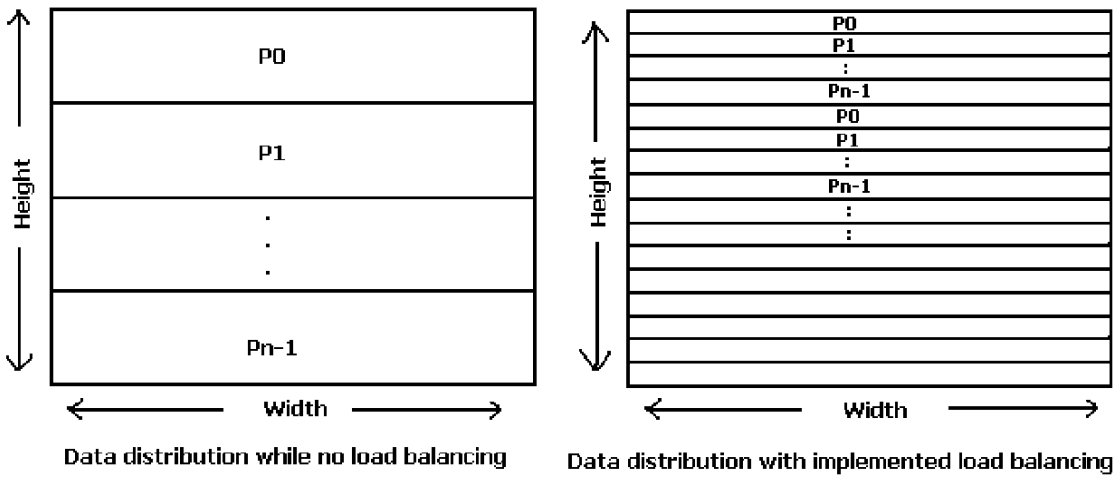
Datorită cerințelor computaționale foarte mari ale ray tracing-ului, paralelizarea este esențială pentru a obține performanțe acceptabile într-un timp util. Paralelizarea permite distribuirea sarcinii de calcul pe mai multe procesoare sau nuclee, reducând semnificativ timpul de execuție. Algoritmul de raytracing se pretează bine la paralelizare deoarece fiecare rază poate fi urmărită independent, permițând astfel reducerea semnificativă a timpului de rulare. În practică, acest algoritm este frecvent paralelizat pe GPU utilizând framework-uri specializate precum CUDA (Compute Unified Device Architecture) de la NVIDIA sau OpenCL (Open Computing Language). GPU-urile sunt ideale pentru această sarcină datorită arhitecturii lor care suportă execuția simultană a unui număr mare

de fire de execuție (threads). Acest lucru permite fiecărui nucleu al GPU-ului să proceseze traiectoria unei raze sau a unui grup de raze în paralel, reducând semnificativ timpul total de randare. Prin utilizarea GPU-urilor, se pot obține îmbunătățiri dramatice ale performanței, permițând randări în timp real și facilitând implementarea ray tracing-ului în aplicații care necesită imagini de înaltă calitate și interactivitate, cum ar fi jocurile video și simulările virtuale.

Pentru acest proiect, am implementat paralelizarea folosind MPI, utilizând scheme de partajare a datelor și comparând rezultatele obținute prin utilizarea diferitelor topologii. Am ales topologia de grid bidimensional, deoarece se potrivește foarte bine problemei noastre, și adițional topologia Ring, datorită simplității sale.

2. Load balancing

Ideea principală pe care am urmărit-o a fost să divizăm imaginea pe care trebuie să o generăm în blocuri de dimensiuni egale și să alocăm fiecare bloc unui procesor. Totuși, deoarece anumite blocuri pot prezenta caracteristici specifice, este necesară o divizare mai fină a acestora. De exemplu, un bloc poate fi complet gol, ceea ce ar face ca calculele să fie extrem de simple, iar procesorul alocat acelui bloc să termine mult mai rapid decât celelalte. Pentru a ne asigura că eficiența programului nu este compromisă, am optat pentru o schemă de partiționare a datelor mai granulară. Aceasta presupune divizarea blocurilor în unități mai mici și asignarea mai multor blocuri necontigue fiecărui procesor.



Presupunând că imaginile sunt realiste, acest lucru implică faptul că caracteristicile lor sunt distribuite neuniform. Prin alocarea procesoarelor să lucreze pe mai multe zone îndepărtate una de cealaltă, putem obține o mediere a complexității calculelor. Astfel, procesoarele nu vor fi suprasolicitate de calcule intense într-o singură regiune, ci vor avea sarcini mai echilibrate. Această abordare ajută la evitarea situațiilor în care un procesor rămâne inactiv în timp ce altele sunt încă ocupate, ceea ce ar duce la un dezechilibru în utilizarea resurselor și la timpi de execuție mai lungi.

În esență, prin implementarea acestei strategii de divizare granulară și asignare necontiguă a blocurilor, ne-am propus să îmbunătățim distribuția sarcinilor de calcul și să maximizăm utilizarea eficientă a procesorului. Această metodă asigură că toate procesoarele sunt implicate în mod echilibrat în procesul de randare, contribuind astfel la performanțe îmbunătățite ale programului nostru și la generarea rapidă și eficientă a imaginilor de înaltă calitate. Modul în care am ales sa delimitam blocurile a fost in plan orizontal, imaginea fiind apoi reconstruita prin asezarea succesiva a fragmentelor.

3. Implementare

3.1 Structura

Proiectul nostru este structurat în multiple directoare prin care se enumera următoarele:

- benchmarks; reprezinta directorul cu codul pentru rularea testelor, rezultatele testelor și codul pentru generarea graficelor
- docs; director cu enunțul problemei, aceasta documentatie si prezentarea aferenta
- include; folder cu implementările ce țin de algoritm în sine
- main.cpp; fișierul entrypoint al proiectului unde se regasesc apelurile pentru execuția algoritmului
- makefile; pentru a ușura dezvoltarea folosind utilitarul make
- test.png; imaginea rezultată în urma rulării algoritmului

3.2 Ray tracing - calcularea culorii unui pixel

3.2.1 Viewport - abstractizarea camerei

Imaginea este considerată un plan de numere reale denumit viewport. Fiecare pixel este mapat la un pătrat in viewport.

3.2.2 Anti-aliasing

Calcularea unei singure raze per pixel rezulta in imagini cu margini colturoase. Pentru a face tranziția între culori contrastante mai realista am folosit anti-aliasing.
Anti-aliasing presupune calcularea culorii unui pixel ca medie a culorii mai multor raze trasate aleator prin spațiul din viewport al pixelului.

3.2.3 Sursa și direcția razelor

Razele pornesc dintr-un punct comun, considerat centrul camerei. Vectorul care porneste din centrul camerei spre centrul viewport-ului este ortogonal cu planul definit de viewport. Distanța dintre centrul camerei si viewport este distanta focala a camerei.
Razele pornesc din centrul camerei spre punctul din viewport pentru care sunt calculate.

3.2.4 Calcularea culorii unei raze

Culoarea unei raze este determinată recursiv:

1. Se gaseste primul obiect din scena cu care se intersectează raza.
2. Dacă raza nu se intersectează cu niciun obiect, culoarea ei va fi
3. Dacă nu s-a trecut de limita de recursivitate, culoarea razei va fi determinată de culoarea unei raze nou create și culoarea obiectului cu care se intersectează. Se calculeaza culoarea razei nou create recursiv.
4. Sursa razei noi se afla pe suprafața obiectului intersectat. Este important ca în cazul reflexiei sursa sa fie în direcția normalei fata de suprafața obiectului, altfel, din cauza erorilor de floating point arithmetic, ar putea ajunge în spatele suprafeței.
5. Direcția razei noi depinde de material.

3.3 Materiale

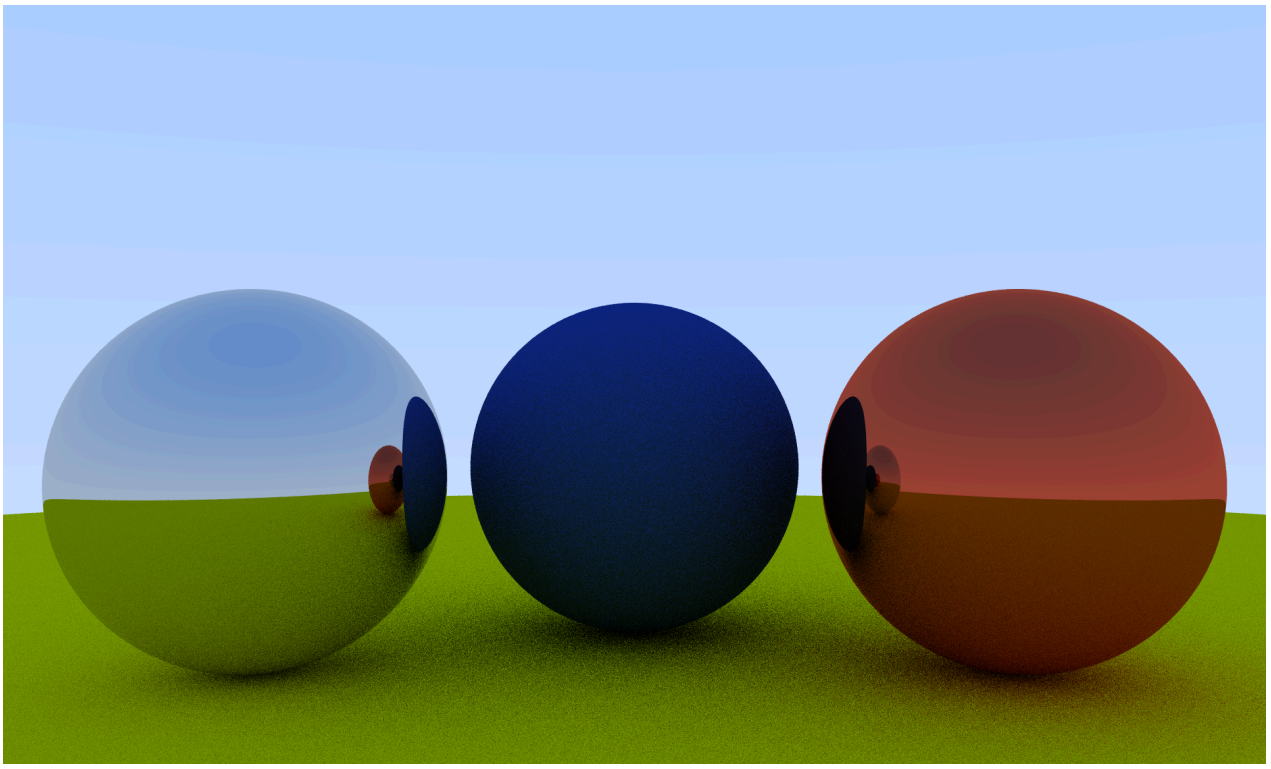
3.3.1 Lambertian

Reflexie difuză - direcția razei noi este determinată aleator. Aceasta trebuie să aibă un unghi de maxim 90 de grade fata de normala la planul obiectului.

3.3.2 Metal

Relflexie perfecta - directia razei noi este oglindita fata de normala la planul obiectului.

Ruland aceasta tehnica în mod secvențial folosind un singur nod procesor ne rezultă următoarea imagine.



Putem observa pe langa sursa de lumina avem prezente obiecte sferice din diverse materiale cu variate reflexii de lumina.

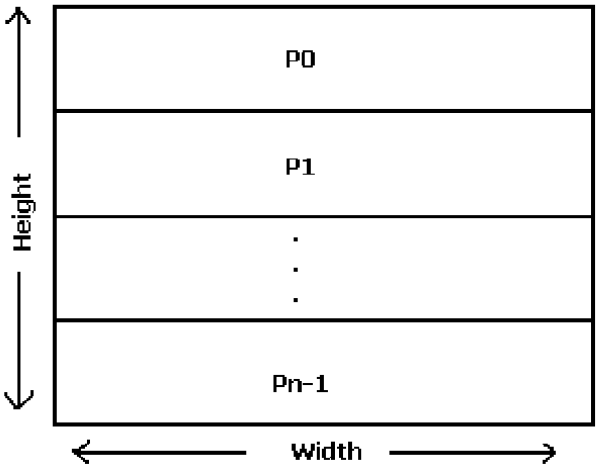
3.4 Tehnici de paralelizare

Fiecare procesor este responsabil de un număr egal (mai puțin în cazul în care dimensiunea imaginii nu se divide la numărul de

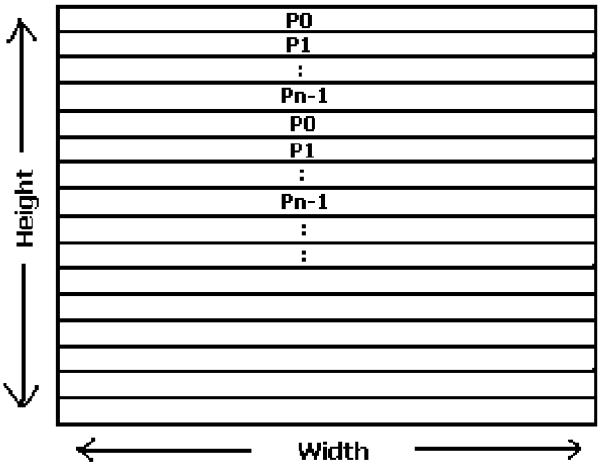
procesoare). Procesorul 0 acumuleaza rezultatele cu metoda Gather și reconstruieste imaginea. Împărțirea și reconstrucția depind de tehnica de load balancing folosită.

3.4.1 Load balancing

Cea mai simpla implementare alocă cate un bloc contigu de rânduri din imagine fiecărui procesor. Reconstructia este simpla, deoarece Gather pune rezultatele locale în receive buffer în ordinea rangului procesoarelor.
Aceasta solutie nu ia în considerare faptul ca imaginile naturale conțin zone cu intensități de procesare diferite (număr de obiecte, grad de reflexie, etc.). O soluție mai bună este alocarea rândurilor imaginii din N în N intre procesoare. Cu aceasta metoda, procesorul 0 va primi randurile i pentru care i%N este 0 în primul bloc, 1 în al doilea bloc și tot așa. Reconstruirea imaginii trebuie sa mute randurile în locul corect.



Data distribution while no load balancing



Data distribution with implemented load balancing

3.4.2 Topologii utilizate

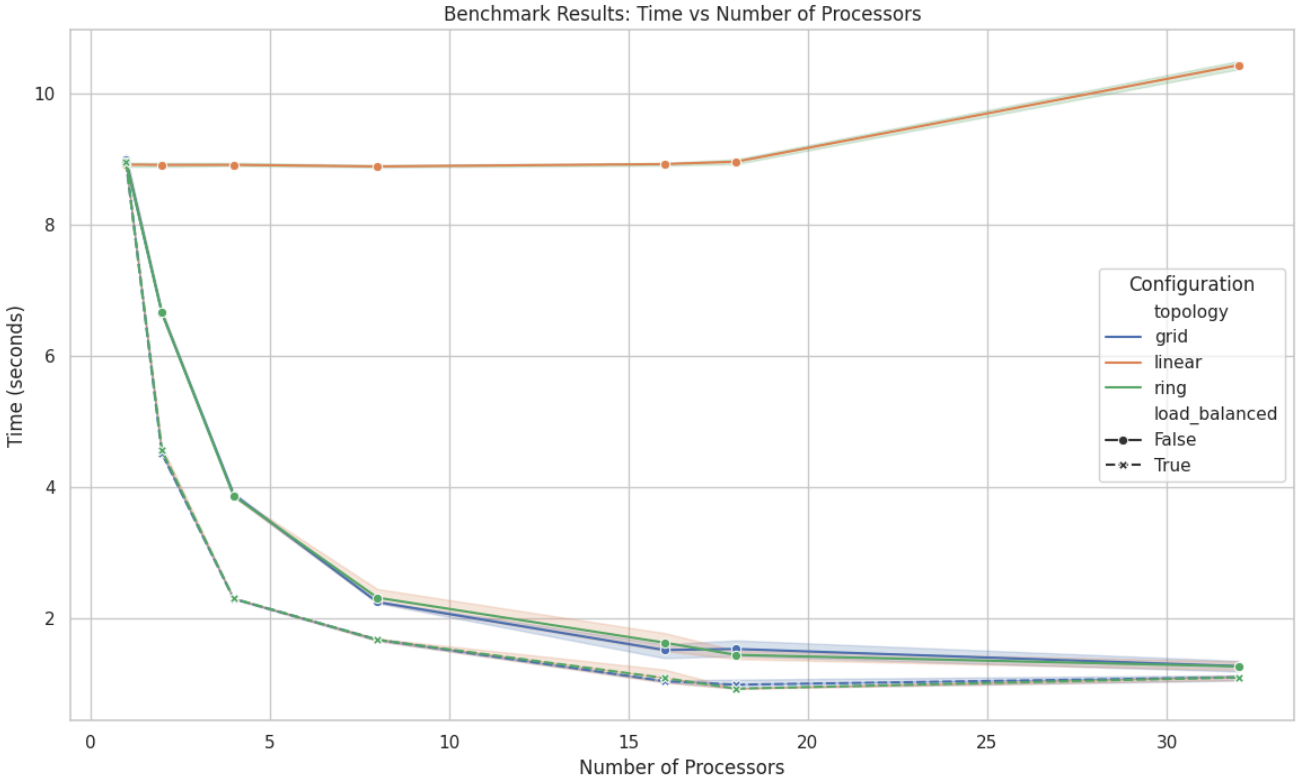
Pentru solutia noastra, topologiile nu au avut un impact asupra performanței algoritmului, deoarece complexitatea algoritmului de Gather este O(p) pentru ring/grid/hypercube. Am implementat un ring utilizând un comunicator cartezian cu o singura dimensiune și wrap around, si un grid utilizând un comunicator cartezian bidimensional.

4. Benchmarks

În acest proiect, rezoluția imaginii este fixa de 1920x1152 pixeli. Mărime aleasă pentru usurinta diviziilor egale. Blocurile de date (dimensiunea blocurilor de pixeli) sunt determinate prin împărțirea înălțimii imaginii la numărul de procesoare disponibile. Am scalat numărul de procese de la 1 la 32 în puteri de 2. Această metodă permite o distribuție eficientă a sarcinilor, asigurând o utilizare optimă a resurselor de procesare pentru a obține o randare rapidă și de înaltă calitate a imaginii.

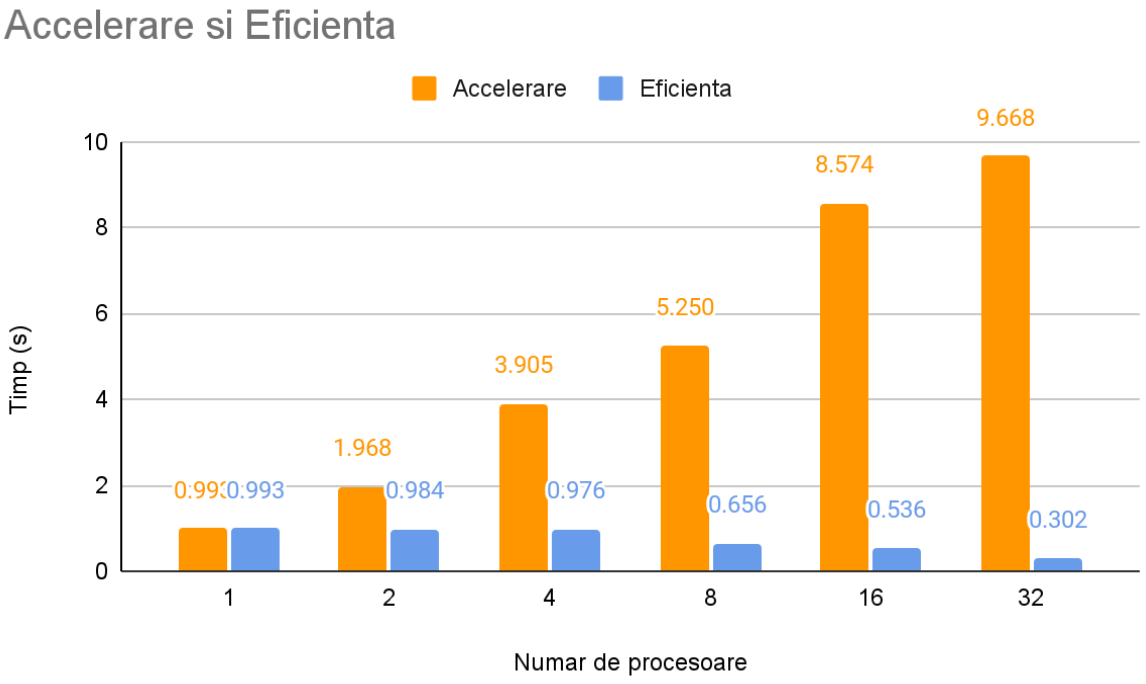
Pentru rularea benchmark-urilor am folosit un script bash ce porneste procese pentru fiecare configurare ("grid" "grid_nlb" "ring" "ring_nlb" "linear"), pentru fiecare număr de procese din lista de test (1 2 4 8 16 18 32), de cate 5 ori fiecare, iar apoi scrie aceste rezultate în fișierul benchmark_results.csv în forma de valori separate prin virgule in formatul "num_processors,run,topology,load_balanced,time".

Apoi, folosind un script python si bibliotecile pandas si seaborn am generat graficul aferent timpului de rulare raportat la numărul de procesoare pentru fiecare configurare. Rezultatul fiind acesta:



Se poate observa cum topologia nu a avut un efect considerabil în timpul rulării. In schimb, load balancing-ul a adus imbunatatiri considerabile.

Metoda secventiala prezinta un timp de executie constant.



5. Concluzie

În concluzie, experimentul nostru a demonstrat că topologia nu a avut un efect semnificativ asupra timpului de rulare, probabil datorită numărului redus de procesoare utilizate. Cu toate acestea, metoda de load balancing implementată a adus îmbunătățiri considerabile în ceea ce privește eficiența și timpul de execuție. Este important de menționat că o arhitectură SIMD/SIMT ar putea oferi rezultate considerabil mai bune, datorită memoriei shared si a capacității sale de a executa un număr mare de operații în paralel, optimizând astfel performanța generală a algoritmului de ray tracing. Aceste constatări subliniază importanța alegerii corecte a strategiilor de paralelizare și a arhitecturilor hardware pentru a maximiza eficiența în generarea de imagini realiste.

6. Bibliografie

- Shirley, P. (2016). *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html
- Khan, T. (2011). *Parallel Ray Tracing using MPI and OpenMP*. https://www.researchgate.net/profile/Tazrian-Khan-2/publication/228686336_Parallel_Ray_Tracing_using_MPI_and_OpenMP/links/54e21d740cf2c3e7d2d1de6f/Parallel-Ray-Tracing-using-MPI-and-OpenMP.pdf
- Zhang, Y. (2020). *Parallel Ray Tracing with OpenMP*. https://zongyw.io/assets/file/multicore_report.pdf