

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAȘI

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Optimizarea problemelor dificile

Gheorghiță-Tonel Nastacă

Sesiunea: iulie, 2017

Coordonator științific

Lect.Dr Cristian Traian Vidrașcu

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAȘI

FACULTATEA DE INFORMATICĂ

Optimizarea problemelor dificile

Abstract. Scopul principal al acestei lucrări este de a prezenta importanța majoră pe care o are optimizarea. Un alt obiectiv îl reprezintă descrierea anumitor tehnici de eficientizare. Prin optimizare un produs software capătă robustețe, fiabilitate și, cel mai important, putere de soluționare a unei problem într-un timp redus.

Keywords. Optimizare, Procesare pe nuclee, Procesare pe cluster, Algoritmi genetici, Probleme dificil

Gheorghiță-Tonel Nastacă

Sesiunea: iulie, 2017

Coordonator științific

Lect.Dr Cristian Traian Vidrașcu

DECLARAȚIE PRIVIND ORIGINALITATEA ȘI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „*Optimizarea problemelor dificile*” este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- ❖ toate fragmentele de text reproduse exact, chiar și în traducerea proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- ❖ reformularea în cuvinte proprii a textelor scrise de către alți autori deține referință precisă;
- ❖ codul sursă, imaginile etc. Preluate din proiecte open-source sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- ❖ rezumarea ideilor altor autori precizează referință precisă la textul original;

Iași, *data*

Absolvent Gheorghiță Tonel Nastacă

(semnătura în original)

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezentare declar că sunt de acord ca Lucrarea de licență cu titlu „*Optimizarea problemelor dificile*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezenței lucrări de licență.

Iași, *data*

Absolvent Gheorghiță Tonel Nastacă

(semnătura în original)

Cuprins

Lista de figuri.....	6
1 Motivație.....	7
2 Introducere.....	8
3 Descrierea în detaliu a problemei (Comisul Voiajor).....	9
3.1 Utilizarea problemei în viața reala	11
3.1.1 Transport.....	11
3.1.2 Construcția plăcilor cu circuite.....	11
3.1.3 Optimizarea aparatelor CNC	11
4 Soluția utilizată.....	12
4.1 Inițializarea algoritmului genetic	13
4.1.1 Dimensiunea populației	13
4.1.2 Rata de mutație	14
4.1.3 Campionatul	16
4.1.4 Elitismul	17
4.1.5 Epoci.....	17
4.2 Căutarea soluției.....	18
4.3 Condiția de oprire.....	18
5 Tehnologii utilizate.....	19
5.1 C++.....	19
5.2 Java.....	19
5.3 JavaFx.....	19
5.4 SSLSocket	20
5.5 TCP.....	20
5.6 JSON	20
5.7 Amazon Elastic Compute Cloud (EC2)	20
6 Framework-ul OpenMP.....	21
6.1 Procesarea simplă a programului	23
6.2 Procesarea paralelă a programului	24
7 Arhitectura aplicației	26
7.1 Introducere	26
7.2 Crearea aplicației.....	26
7.2.1 Nucleul aplicației.....	26
7.2.2 Procesarea pe nuclee.....	27
7.2.3 Rularea pe un cluster de calculatoare	29

7.2.4	Conectare PC cluster	30
7.2.5	Orchestrare cluster.....	31
7.2.6	Comunicarea client – aplicație	32
8	Detalii de implementare.....	33
8.1	Implementarea aplicației nucleu.....	33
8.2	Implementare cluster de calculatoare	37
8.3	Server	38
9	Performanțe obținute	40
9.1	Rezolvarea problemei cu un algoritm genetic general	40
9.2	Rezolvarea problemei utilizând un algoritm genetic specializat.....	41
9.3	Algoritm genetic plus procesare pe nuclee	42
9.4	Implementarea pe un cluster de calculatoare	43
9.5	Evidențierea diferențelor dintre versiuni.....	43
	Concluzii.....	48
	Direcții de dezvoltare.....	49
	Bibliografie	50

Lista de figuri

Figură 1:Exemplu TSP rezultat greșit.....	9
Figură 2:Exemplu TSP rezultat corect.....	10
Figură 3:Funcționarea algoritmului genetic.....	12
Figură 4:Procesare multicore (timpul de execuție).....	22
Figură 5:Procesare normală (timpul de execuție).....	22
Figură 6:Reprezentarea nucleelor în task manager 1	24
Figură 7:Reprezentarea nucleelor în task manager 2.....	25
Figură 8:Arhitectura aplicației nucleu	27
Figură 9:Procesarea muticore a algoritmului genetic	28
Figură 10:Arhitectura aplicație ce implică procesare pe un cluster de calculatoare.....	30
Figură 11:Diagramă de secvență manager cluster	31
Figură 12:Diagrama de clase a aplicație nucleu	33
Figură 13:Diagrama de clase pentru server:	38
Figură 14:Exemplu de traseu inițial.....	40
Figură 15:Diferența dintre algoritmul genetic specializat și cel general	42
Figură 16:Instanțele de EC2	43

1 Motivație

În dezvoltarea unui produs software de calitate se iau în considerare foarte mulți factori. Un factor cu o mare greutate îl reprezintă optimizarea. În informatică, prin optimizare se înțelege capacitatea softului de a rezolva o problemă în timp cât mai scurt, utilizând cât mai puține resurse. Deseori aceste două proprietăți nu se pot obține simultan. În multe cazuri se preferă alegerea optimizării timpului în detrimentul resurselor (Fog, 2015). Acest lucru este datorat faptului că, în zilele noastre, resursele sunt mult mai la îndemână. În plus, optimizarea mai presupune eficientizarea anumitor aspecte, cum ar fi modul de tratare al erorilor, transferul de date etc. Optimizarea asigură faptul că programul va funcționa în parametri normali în diferite situații.

Principalele avantaje ale optimizării:

- Execuție rapidă – prin execuție rapidă se înțelege că algoritmul returnează rezultatul scontat într-un timp cât mai scurt. Optimizarea softului permite descoperirea de particularități care pot fi îmbunătățite. Un alt lucru foarte important îl reprezintă căutarea acelor componente care limitează performanța. În acest mod se pot aplica diferite abordări astfel încât aceste nereguli să fie depășite.
- Evitarea pierderilor de memorie – dacă programul nu a fost optimizat pot interveni situații neașteptate prin care memoria este afectată. Acest lucru are impact și asupra celorlalte aplicații care rulează în paralel. Memoria fiind una dintre resursele principale ale calculatorului. Optimizarea asigură utilizarea memoriei într-un mod eficient.
- Evitarea apariției codului neutilizat – cu toate că pare puțin ciudat acest lucru se întâmplă deseori. Acest fapt conduce la o înțelegere mai dificilă a programului. Apare o anumită ambiguitate în proiect
- Prin optimizare se asigură faptul că aplicație va rula corespunzător atât pe dispozitivele puternice cât și pe dispozitivele cu o putere computațională mai modestă.

În concluzie, putem afirma că optimizarea face diferența dintre o aplicație obișnuită și o aplicație robustă. Orice aplicație trebuie să fie optimizată.

2 Introducere

Prin această lucrare se urmărește evidențierea și conștientizarea importanței pe care o are optimizarea asupra aplicației. Se va prezenta faptul cum prin optimizare se poate minimiza timpul de rulare al aplicației cu mai mult de 80%. Totodată se va demonstra că nu este suficient să deținem un calculator puternic pentru a crea un program bun. Este foarte important să știm să valorificăm cât mai bine puterea pe care o oferă un asemenea dispozitiv. Un dispozitiv computațional modest, cu un program optimizat, poate oricând să întreacă un calculator cu caracteristici mai bune dar cu un program care lasă de dorit. Toate acestea se vor realiza aplicând diferite tehnici de eficientizare.

Pentru a realiza obiectivul propus într-un mod cât mai vizibil se va folosi o problemă NP-hard. O problemă este NP-hard dacă există cel puțin o altă problemă care face parte din clasa NP¹, și care se reduce polinomial la problema inițială. Este foarte greu de determinat soluția perfectă pentru problemele de acest tip. Nu poate fi creat un algoritm care să determine o rezolvare sau acesta este mult prea dificil pentru a putea fi pus în practică. Așadar se apelează la algoritmi euristici. Aceste modele încearcă să determine o soluție cât mai fiabilă, dar nu garantează că determină soluția perfectă.

Noi vom optimiza cunoscuta problemă Comisul Voiajor. Această problemă se încadrează în clasa NP-hard. Comisul Voiajor sau Traveling Salesman Problem este o problemă de grafuri. TSP constă în determinarea ciclului hamiltonian de cost minim. Se știe că problemele de acest tip necesită putere computațională foarte mare. Pentru a rezolva această problemă apelăm la algoritmi genetici. Algoritmi genetici se bazează pe teoria evoluționistă.

Pentru optimizare vom folosi conceptul de programare paralelă și distribuită, plus multe alte tehnici foarte utile.

¹ Clasa problemelor care pot fi rezolvate de algoritmi nedeterministi in timp polinomial - <https://fmse.info.uaic.ro/getpagefile/14/np-complexitate.pdf/>

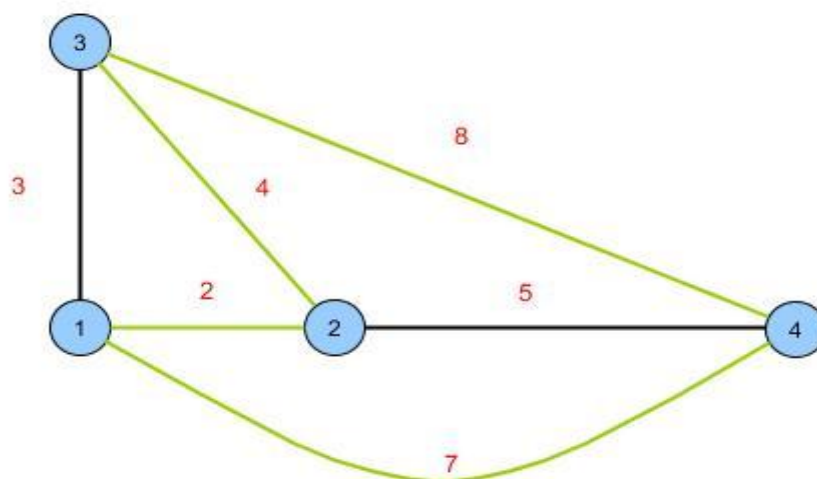
3 Descrierea în detaliu a problemei (Comisul Voiajor)

Cercetând puțin enunțul putem să intuim în ce constă problema. Comis-Voiajor reprezintă o persoană care se ocupă cu negoțul. Acesta se deplasează în diferite locații pentru a căuta anumiți cumpărători. Astfel a apărut următoare întrebare: ”Având o hartă cu diferite orașe și distanțele dintre ele, care este cea mai scurtă rută pentru a vizita toate orașele o singură dată, iar la final să se ajungă în punctul inițial?”

Definiție: Fie G un graf neorientat, complet (fiecare nod are legătură cu fiecare). Să se determine un ciclu de cost minim care pornește dintr-un nod oarecare, trece prin toate nodurile o singură dată iar apoi se ajunge la nodul de plecare (Rai, Madan, & Anand, 2014). Costul unui asemenea circuit este determinat de suma tuturor dimensiunilor laturilor. Soluția unei astfel de probleme nu se poate determina în timp polinomial.

Pentru a prezenta problema cât mai bine am creat următoarele exemple:

Figură 1: Exemplu TSP rezultat greșit

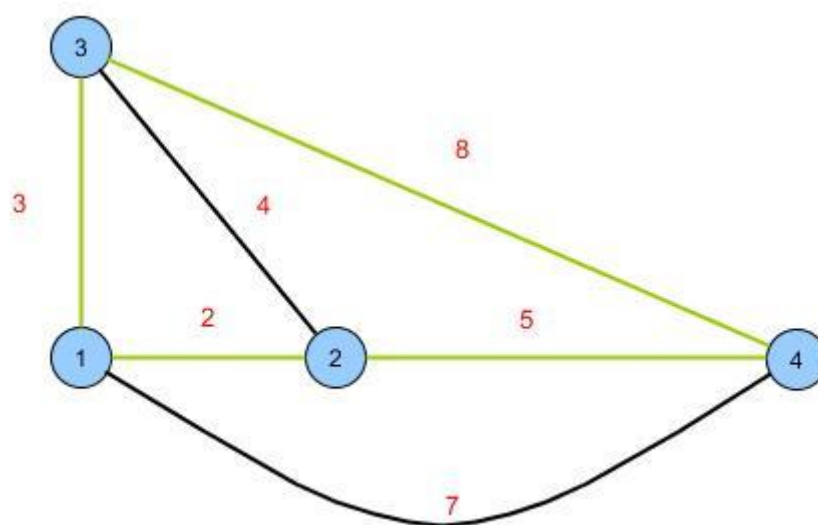


În mod normal, la o primă vedere, suntem tentați să folosim metoda greedy pentru a determina ciclul de cost minim, dar după cum se observă, analizând circuitul verde, acest lucru conduce la o soluție greșită. Trebuie să menționăm că tehnica greedy (lacom) este utilizată pentru a rezolva această problemă, dar nu asigură faptul că duce la o soluție care se apropie măcar de optim.

O altă tehnică folosită pentru a rezolva această problemă este Backtracking. Această tehnică funcționează rezonabil doar dacă numărul de noduri din graf este de dimensiuni reduse. La o

scurtă analiză putem observa că numărul total de trasee posibile este egal cu $(n - 1)!$, unde n este numărul de noduri.

Pentru cazul prezentat în Figura 1 această metodă ajunge la următorul rezultat.



Figură 2: Exemplu TSP rezultat corect

După cum se observă, soluția prezentată în Figura 2 este cea optimă. Acest lucru este asigurat de backtracking, care va returna mereu soluția cea mai bună atunci când n are valori rezonabile. Totuși, pentru a rezolva problema Comisului Voiajor se vor folosi algoritmi euristici. Metodele euristice se utilizează la problemele care nu se pot rezolva urmând un anumit șablon sau soluțiile existente sunt mult prea complexe pentru a putea fi „explicate” calculatorului. Problema pe care o tratăm în această lucrare este de acest tip. Nu există un model matematic fiabil care să funcționeze pentru orice tip de input. Avem soluții care funcționează foarte bine pentru un anumit set de date, dar acestea nu pot fi generalizate pentru orice tip de date de intrare deoarece complexitatea timp ar depăși limita rezonabilă de așteptare. Așadar, din dorința de a utiliza un algoritm care să se potrivească pentru orice input am apelat la metodele euristice. Rolul acestor metode este de a găsi o soluție aproape de optim într-un timp rezonabil (Melanie, 1999).

3.1 Utilizarea problemei în viața reală

3.1.1 Transport.

Problema Comisul Voiajor îți găsește, în general, aplicativitate atunci când vine vorba de firme de transport. Un exemplu deloc neglijabil îl reprezintă o firmă de curierat internațională. Este foarte important să se asigure că traseul pe care îl va parcurge șoferul este unul optim. Orice companie dorește să fie cât mai eficientă în realizarea țelului propus.

3.1.2 Construcția plăcilor cu circuite

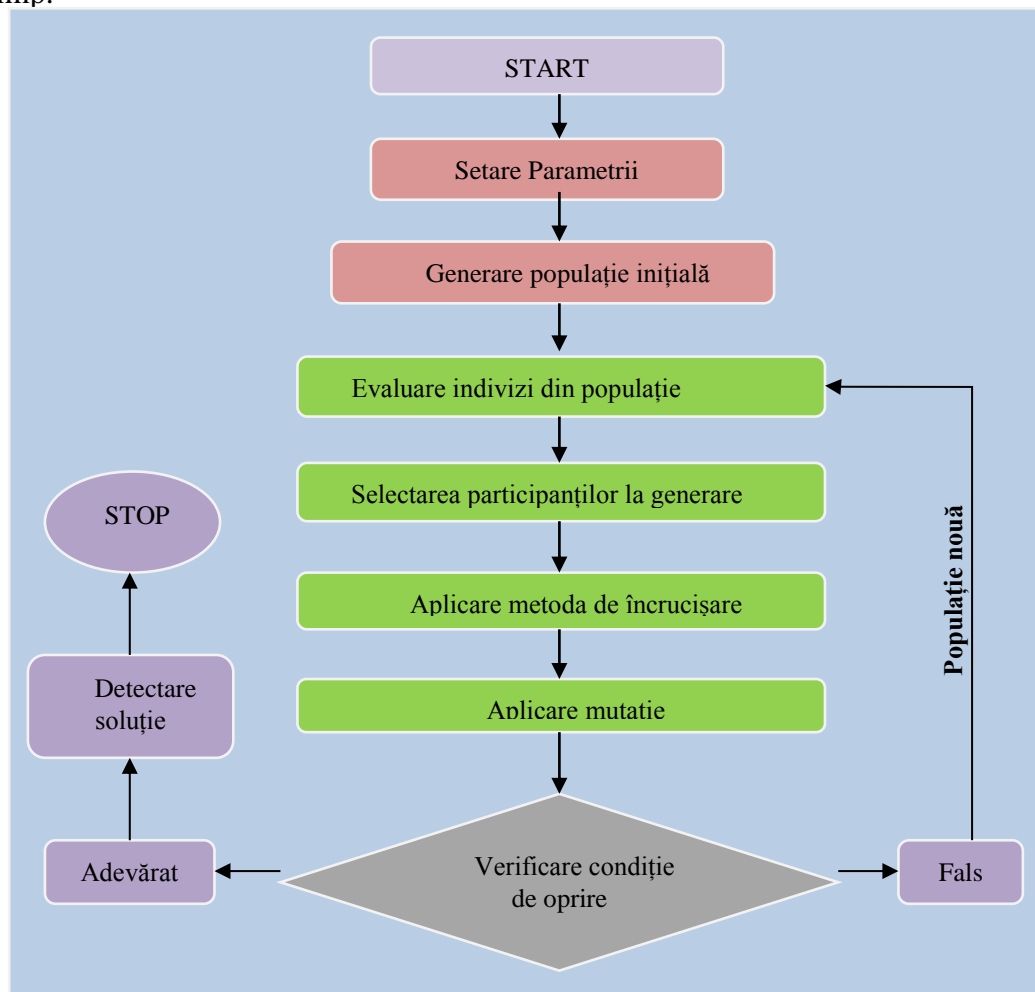
O placă sofisticată poate să necesite numeroase orificii de diferite dimensiuni, iar aceasta poate să devină o problemă costisitoare. Distanța dintre două puncte poate fi văzută ca distanța dintre două orașe. Este adevărat că distanța este mult mai mică, dar având în vedere că se construiesc cantități impresionante de asemenea plăci, realizarea eficientă a orificiilor devine o problemă reală. Pentru a rezolva această problemă se poate apela imediat la TSP.

3.1.3 Optimizarea aparatelor CNC

Aparatele de tip CNC sunt construite pentru a realiza desene, sculpturi în lemn etc. Aceste dispozitive sunt folosite foarte mult în industrie. Este important ca aparatele CNC să fie cât mai eficiente, astfel se poate salva timp prețios ce poate fi folosit în chestiuni mai utile.

4 Soluția utilizată

Pentru a rezolva această problemă se vor folosi algoritmi genetici. Acești algoritmi încearcă să imite procesul de dezvoltare al naturii. Mai exact se simulează teoria propusă de britanicul Charles Darwin (Breabăn, 2004). În urma cercetărilor realizate, acesta a ajuns la concluzia că formele de viață inferioară au evoluat către forme de viață superioară odată cu trecerea timpului. Creatorul acestor algoritmi este John Holland. El a depistat această soluție în jurul anului 1970. Această metodă este utilizată deseori în cazul în care găsirea soluției optime necesită foarte mult timp.



Figură 3: Funcționarea algoritmului genetic

În următoarele rânduri, pe baza diagramei din Figura 3 se va descrie modul de funcționarea al algoritmilor genetici. Se va prezenta în detaliu ce se realizează la fiecare pas, începând cu inițializarea până la a defini condiția de oprire. Este imperios necesar ca algoritmul să aibă definită o condiție de oprire validă. În continuare se va descrie algoritmul:

4.1 Inițializarea algoritmului genetic

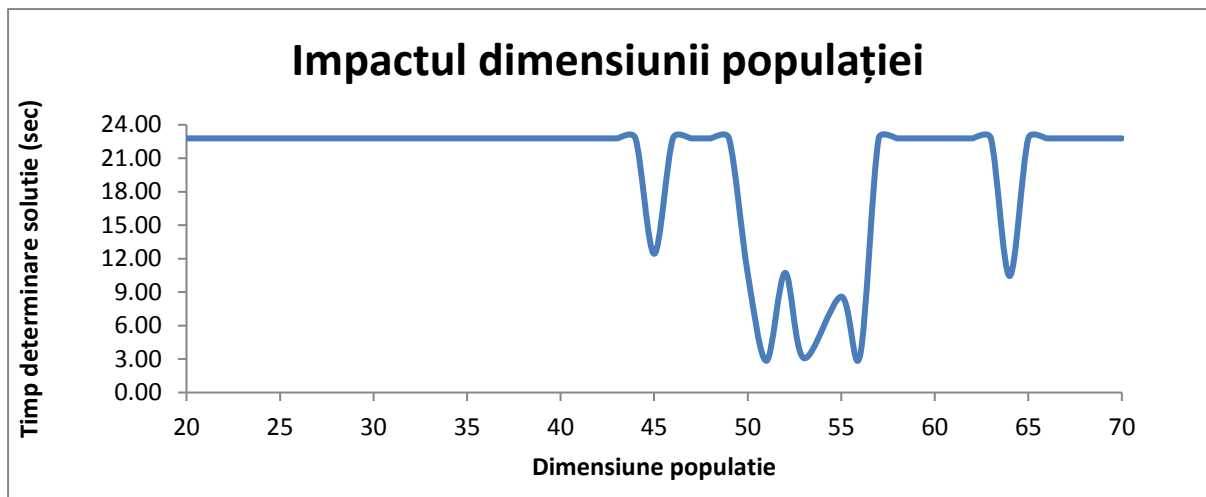
Inițializarea algoritmului genetic are un rol foarte important. Selectarea parametrilor potriviți pentru problema care urmează a fi tratată poate să reducă timpul de execuție considerabil. Alegerea corectă a parametrilor nu rezolvă doar problema legată de timp, ci conduce și la o soluție calitativă.

Parametrii care necesită inițializați:

- Dimensiunea populației
- Rata de mutație
- Dimensiunea campionatului de selecție a unui individ
- Elitism
- Numărul de generații

4.1.1 Dimensiunea populației

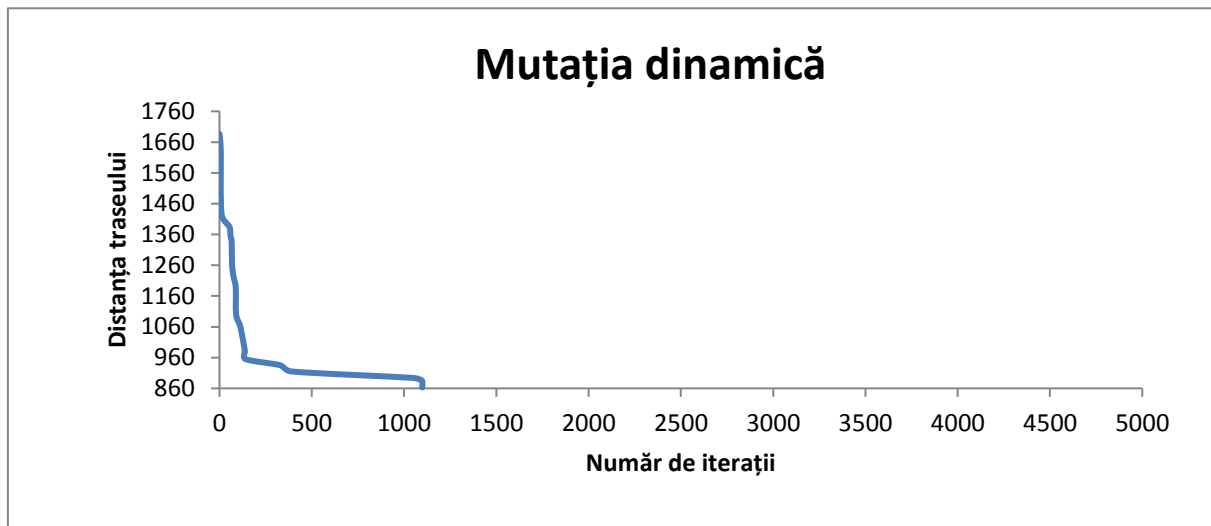
Dimensiunea populației reprezintă numărul de indivizi care formează populația. Dacă facem comparație cu viața reală acest parametru reprezintă numărul de persoane dintr-un oraș. Numărul de indivizi rămâne constant pe parcursul rulării algoritmului. În mod obișnuit această dimensiune se selectează prin încercări repetate. Se selectează un anumit interval, iar pentru fiecare număr din intervalul respectiv se accesează algoritmul genetic. Peste rezultatele obținute se creează o statistică și se selectează numărul cel mai bine cotate. Trebuie să specificăm că acest parametru nu este valabil pentru orice set de date al problemei. Dimensiunea populației este strict legată de datele de intrare. În cazul nostru, dacă selectăm un input cu 50 de orașe și cu dimensiunea populației de 300, iar apoi un input cu 5 orașe se observă clar că numărul de indivizii din populație trebuie să fie mult mai mic. Spațiul de soluție oferit de setul cu 5 orașe ($4!$) este mai mic decât numărul de indivizii selectat anterior (300), ceea ce înseamnă că vom avea indivizi identici dacă mergem pe aceeași dimensiune de populație. Așadar trebuie să existe o corelație între dimensiunea setului de date și numărul de indivizii din populație.



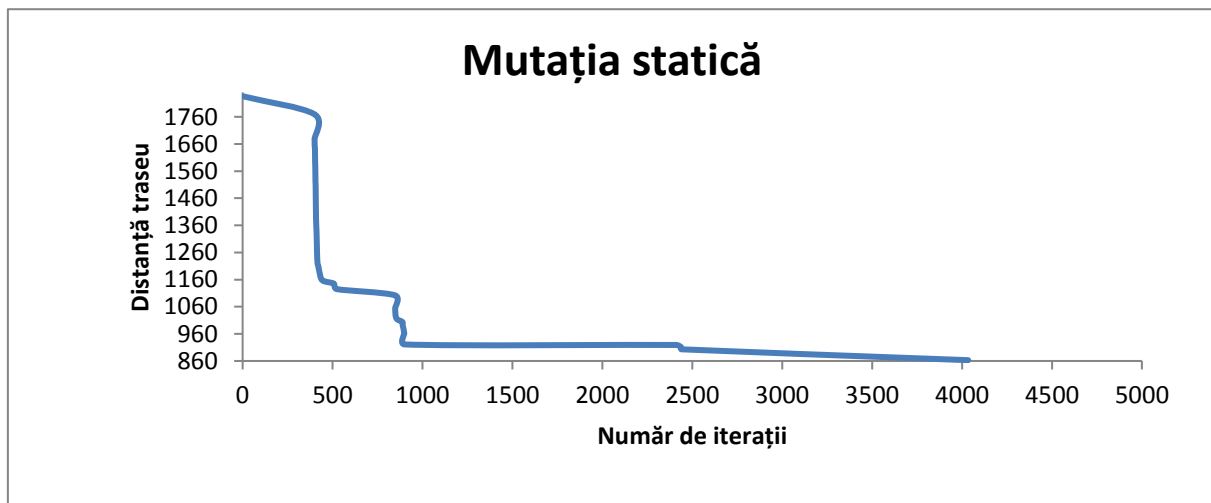
După cum se observă în graficul de mai sus, atunci când populația are dimensiunea în jur de 45 de indivizi se ajunge la soluția considerată aproape de optim, dar necesită mai mult timp deoarece la o iterație nu se creează suficienți indivizi cât să acopere o parte optimă din spațiul de soluții. Pe intervalul (50, 57) se poate vedea că algoritmul converge destul de rapid, de aici putem să concluzionăm, că atunci când alegem dimensiunea din acest interval vom avea rezultate bune. Mai putem să evidențiem faptul că în jurul valorii 65 o iterație de algoritm necesită mai mult timp. Această necesitate apare datorită faptului că trebuie să se determine mai mulți indivizi. În concluzie, selectarea corectă a acestui parametru este absolut necesară. Impactul pe care îl are asupra complexității timpului fiind destul de semnificativ.

4.1.2 Rata de mutație

Rata de mutație reprezintă numărul de indivizi din populația actuală ce vor fi modificați. Mutația are rolul de a insera diversitate în populație. Are aceeași semnificație ca mutația din biologie. Utilizând această tehnică se asigură faptul că vor apărea indivizi noi. În cazul prezentat în această lucrare mutația va acționa asupra unui traseu interschimbând două orașe. Totuși acest lucru trebuie să se realizeze cu o probabilitate decentă, altfel apare riscul ca un individ să fie distorsionat într-un mod negativ. Selectarea corectă a acestui parametru face diferența dintre succes și eșec. Acest parametru are scopul de a evita pătrunderea într-un maxim local. Fără acest factor populația va semăna din ce în ce mai mult, uneori se poate ajunge chiar la identitate. De multe ori pentru a obține o valoare cât mai bună pentru acest element se apelează la cunoscuta metodă trial-and-error (Sadiq, 2012). Totuși este de preferat construirea unui algoritm care să adapteze această valoare pe parcurs. Acest lucru a fost testat de câțiva cercetători acum ceva timp.



Pentru a crea acest grafic s-a utilizat metoda de ajustare automată a mutației. Pentru început acest parametru are o valoare mai mare deoarece se dorește să se diversifice cât mai mult populația, iar pe moment ce ne apropiem de un bestindivind această valoare scade. Dacă acest parametru ar rămâne constant, fiecare individ ar suferi modificări grave. Odată ce probabilitatea cu care se va executa mutația scade, vor apărea modificări nesemnificative dar importante. Aceste modificări sunt importante deoarece se pot interschimba două orașe ce inițial nu erau în poziția corectă. În continuare vom prezenta un caz în care parametru de mutație este static și are o valoare aleasă parțial arbitrar. Este evident că rezultatul nu va fi identic.













Din graficul „Mutația Statică” putem să deducem faptul că valoarea parametrului este una destul de adecvată. Inițial populația este diversificată mai greu (iterația 0 - 500), iar apoi, după mai multe iterații, se găsește o soluție bună (iterația 1000 - 2500). Determinarea unei soluții mai bune întâlnește un moment de stagnare deoarece în populație apare prea multă schimbare.

Totuși la un moment dat se ajunge la soluția aproape optimă. Am testat și cazul în care factorul de mutație lipsește total, dar rezultatele obținute sunt mult prea dezastruoase pentru a putea fi luate în calcul.

În concluzie, am putea să afirmăm următorul lucru: dacă rata de mutație este prea mare algoritmul se va îndrepta rapid către o soluție bună dar la un moment dat va stagna deoarece va modifica prea mult indivizii. Dacă rata este prea mică algoritmul va necesita foarte multe iterații până va ajunge la soluția optimă. Motivul fiind faptul că factorul random intervine prea rar. Așadar, utilizarea ajustării automate a parametrului de mutație rămâne cea mai bună modalitate.

4.1.3 Campionatul

Campionatul reprezintă o tehnică de selecție a indivizilor care vor participa la încrucișare. Din populația actual se selectează aleator un număr de indivizi. Numărul de indivizi care vor participa la campionat este destul de important. Dacă sunt mulți participanți probabilitatea ca un individ mai slab să câștige este mică, iar acest lucru nu se dorește. După ce indivizii au fost selectați se preia cel care are potențialul mai mare. Dacă numărul de participanți este prea crescut se ajunge la cazul în care doar cei care se află în topul indivizilor vor participa la crearea noii populații. Acest fapt este pus în evidență în diagrama de mai jos.

Populație										
Index	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
Individ										
Potențial	7	4	1	8	3	10	5	2	9	6

Tabel 1: Exemplu de populație

Tur	Campionat 5 participanți					Câștigător	Potențial
1.	9	1	4	7	2	9.	9
2.	3	6	1	7	1	6.	10
3.	8	1	5	2	4	4.	8
4.	2	1	5	3	8	1.	7

Tabel 2: Exemplu de selecție a indivizilor

În Tabelul 1 este prezentat un model de populație, iar Tabelul 2 reprezintă selecția a patru indivizi. Numărul de candidați care participă la campionat fiind 5. Se observă ușor că cei 4 indivizi care au potențialul cel mai mic nu vor fi selectați niciodată. Acesta este principalul dezavantaj atunci când se creează un turneu de dimensiuni mari. Dacă se merge pe un asemenea sistem, în final, cel mai probabil populația va arăta identic. Mai exact, se ajunge la un maxim local din care nu se mai poate ”evada”. Pentru a oferi o șansă și celor cu potențial redus, campionatul trebuie să fie format din puțini indivizi. Odată ce numărul de participanți la turneu scade, probabilitatea ca cei slabi să câștige crește. Aceasta este o tehnică de a căuta un maxim global. Un alt dezavantaj care apare atunci când numărul de participanți este mare îl reprezintă complexitatea de timp. Se necesită mai multe instrucțiuni pentru a obține individul cel mai bun. În acest caz diferența nu ar fi semnificativă, dar în situații reale când populația care urmează a fi creată este de dimensiuni considerabile, orice instrucțiune în minus este bine venită. Așadar, dimensiunea campionatului trebuie selectată în corelație cu dimensiunea populației, dar trebuie să tindă spre o valoare cât mai mică.

4.1.4 Elitismul

Prin elitism se oferă posibilitatea ca individul cel mai bun să persiste în generația următoare. Astfel se asigură faptul că soluția optimă detectată la o anumită generație nu va fi pierdută în timp. Această metodă are un impact foarte mare în a detecta soluția optimă. Dacă o soluție aproape de optim a fost detectată la o anumită iterație dar nu a fost salvată este foarte greu de preconizat faptul că aceasta va apărea iarăși peste câteva generații. Un mic dezavantaj ar fi faptul că prin această metodă se perturb gradul de diversitate al populației ce urmează a fi generată. În unele versiuni de algoritm genetic există un parametru care specifică, câți dintre indivizii cei mai buni să persiste în noua generație. Aici poate să apară iarăși o dilemă legată de numărul de indivizi permiși să fie transferați din populația prezentă către următoarea populație. Totuși, pentru a evita ca populația să devină din ce în ce mai identică ne-am limitat la a prelua doar cel mai semnificativ individ în noua generație.

4.1.5 Epoci

Numărul de generații(epoci) reprezintă câte iterații se vor executa până când va fi afișată soluția. În general, acest parametru este setat în concordanță cu dimensiunea datelor de intrare. Dacă pe parcursul mai multor testări se observă faptul că algoritmul nu ajunge la soluția așteptată se va

ajusta parametrul prin adăugare. Dacă algoritmul converge la soluția dorită, dar acesta încă mai rulează se va reseta parametrul cu o valoare mai mică față de valoarea inițială. Este important că acest parametru să fie setat corespunzător. Prin setarea normală se evită realizarea de iterații în plus.

4.2 Căutarea soluției

După ce toți parametrii au fost setați corespunzător, următorul pas este apelarea funcției de determinare a soluției. Această funcție are rolul de a construi noua generație. În următoarele rânduri se va descrie tehnicile utilizate pentru a determina următoarea populație:

- Mai întâi se verifică dacă parametru ce indică persistența individului cu potențialul maxim este setat. Dacă acesta este setat favorabil se caută individul respectiv în populația curentă și se adaugă la noua generație.
- Pentru a introduce mai multă diversitate se pot genera câțiva indivizi noi. Acești indivizi se vor crea random, fără a fi influențați într-un anumit mod.
- Restul indivizilor se vor crea prin încrucișare. Pentru a putea apela această metodă este nevoie ca mai întâi să se selecteze doi participanți.

4.3 Condiția de oprire

Prin oprirea algoritmului se înțelege faptul că nu se va mai genera o populație nouă. Cel mai bun individ din generația finală reprezintă soluția la care s-a ajuns. Pentru soluția pe care dorim să o obținem putem să setăm o anumită caracteristică. De exemplu, în cazul nostru dorim ca drumul care va fi determinat să aibă lungimea mai mică decât 50 km. Când bestindividul a îndeplinit această condiție algoritmul va trece în starea de repaus. Totuși această abordare nu este completă. Pot exista cazuri în care setul de intrare nu permite determinarea unei soluții cu dimensiunea cerută. În această situație algoritmul va rula la infinit. Pentru a evita acest caz s-a mai introdus un parametru prin care se specifică câte populații se vor genera. Dacă acest număr este depășit și soluția dorită nu a fost obținută algoritmul va fi stopat. Algoritmul mai poate fi oprit prin contorizarea populațiilor consecutive care nu aduc niciun câștig. Dacă nu se găsește o soluție mai bună timp de câteva generații este greu de crezut că în viitor se va modifica ceva în mod folositor. Așadar, dacă numărul de populații consecutive ce nu aduc schimbări la soluție este egal cu un n anterior selectat algoritmul poate fi oprit.

5 Tehnologii utilizate

5.1 C++

Este foarte important ca înainte de a demara un proiect să se realizeze anumite studii care să determine ce limbaj de programare este mai util de utilizat. Alegerea limbajului de programare care se îmbină perfect cu problema în cauză este un prim pas în optimizare, dar nu trebuie să uităm că eficiența aplicației finale ține de cât de bine a fost implementată. Un limbaj de programare de nivel înalt este des folosit în realizare de programe cu o structură bine definită pe când limbajele low-level sunt potrivite pentru eficientizarea vitezei de execuție. Limbajele compilate sunt mai rapide spre deosebire de cele interpretate. Acest lucru se datorează faptului că programul este transformat direct în cod mașină. Totuși interpretorul oferă portabilitate. Trebuie să conștientizăm că nu putem să obținem eficiență, viteză de dezvoltare și portabilitate în același timp. Uneori trebuie să știm să facem o alegere în detrimentul unui aspect astfel încât să ne atingem țelul final. Totuși alegerea trebuie realizată cu simț de răspundere.

Se va utiliza C++ pentru implementarea soluției care rezolvă problema. Acest limbaj este orientat obiect. Datorită acestui fapt se pot reutiliza anumite părți din prima aplicație pentru a construi celelalte variante mai performante. Având în vedere că aplicația finală va fi destul de mare, utilizând clase se va crea o structură ușoară de înțeles. În plus, acest limbaj permite crearea de aplicații performante de tip server-client. Se va utiliza acest avantaj în implementarea versiunii finale. Versiunea finală va rezolva problema utilizând mai multe calculatoare care vor comunica între ele. În plus, acest limbaj include anumite avantaje din C, astfel avem acces la optimizări de tip low-level. Este cunoscut faptul că C++ se află în top atunci când se dorește realizarea de aplicații optimizate. Prin intermediul avantajelor oferite de C++ se va crea o aplicație puternică și flexibilă ce va permite adăugarea de noi funcționalități.

5.2 Java

La fel ca C++, Java este orientat-obiect, ceea ce înseamnă că putem crea aplicații modulare. Se cunoaște că Java este un limbaj multithreaded. Multithreaded este capacitatea unui program de a executa anumite instrucțiuni în același timp. Un alt beneficiu al acestui îl reprezintă securitatea. Se va folosi Java pentru a crea partea de server-client.

5.3 JavaFx

JavaFx este o librărie pentru Java ce permite crearea de interfețe grafice ce pot rula pe diferite tipuri de dispozitive. Această librărie este cea mai nouă oferită până acum, predecesorul ei fiind

Swing. Se va utiliza această librărie pentru a crea o interfață grafică pe partea de client. Astfel, utilizatorul va putea comunica mai ușor cu aplicația.

5.4 SSLSocket

Acest tip de socket, pe lângă funcționalitățile de bază, oferă securitate utilizând protocoale cum ar fi SSL (Secure Socket Layer) și TLS (Transport Layer Security). TLS reprezintă o versiune mai nouă pentru SSL. Prin TLS s-a încercat să se construiască un standard. Acest tip de socket se va utiliza la comunicare dintre client și server deoarece în acest fel se evită atacurile de tipul Man in the Middle și multe altele.

SSLSocket oferă:

- Integritate. Se asigură faptul că mesajul nu va fi modificat.
- Autentificare. Se știe exact că mesajul vine de la o persoană de încredere.
- Confidențialitate. Datele care urmează a fi transferate sunt criptate.

5.5 TCP

Pentru transmiterea de informații între dispozitive se va utiliza TCP (Transmission Control Protocol). TCP este un serviciu bazat pe conexiuni, însemnând că mașinile care trimit informații cât și cele care primesc au o conexiune anterior stabilită. Acest protocol este cel mai utilizat datorită disponibilității și flexibilității lui având cel mai mare grad de corecție al erorilor.

5.6 JSON

JSON (JavaScript Object Notation) este o forma de a stoca informațiile într-un mod organizat pentru a putea fi accesate cu ușurință mai târziu. Acest tip de formatare al informației este independent de limbajele de programare. Se va utiliza JSON pentru a se realiza comunicarea dintre dispozitive. Astfel se va îmbina cu ușurință limbajul de programare Java cu C++.

5.7 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud este un serviciu oferit de cei de la Amazon. Serviciul este dedicat în mare parte companiilor care nu au suficient capital să își asigure partea hardware de care au nevoie pentru a-și rula anumite aplicații. EC2 reduce costul de investiție în hardware. Acest serviciu permite utilizatorului să își creeze câte mașini virtuale dorește. În plus, oferă posibilitatea de a configura partea de networking, storage manager și security. Utilizatorul are posibilitatea să își aleagă ce sistem de operare să ruleze pe computerul virtual: Linux, Windows, FreeBSD (Amazon, 2017). Odată ce nu mai este nevoie de o asemenea mașină aceasta se poate opri.

6 Framework-ul OpenMP

Acest framework² este utilizat pentru a construi programe concurente în limbajele de programare C și C++. OpenMP utilizează un model puternic, scalabil și portabil. (Barney, n.d.) Toate acestea conduc la crearea unei aplicații destul de stabilă. Procesoarele care sunt fabricate în zilele noastre au mai multe nuclee, ce utilizează tehnologia Hyper-Threading. Prin această tehnologie se utilizează resursele mult mai eficient. (Akhter & Roberts, 2006) Mai exact, se permite rularea mai multor fire de execuție pe fiecare nucleu. Folosind OpenMP se va crea o aplicație ce va utiliza procesoarele de acest tip la capacitate maximă. Așadar, prin această tehnică, se va crea o aplicație mult mai performantă, iar timpul de execuție se va reduce considerabil.

OpenMP este un model ce permite partajarea memoriei. Majoritatea variabilelor sunt vizibile de către toate firele de execuție. Pot exista și variabile private care sunt vizibile la nivel de thread. Nu toate programele pot fi paralelizate utilizând acest framework. Aplicația ce urmează a fi paralelizată trebuie să respecte anumite condiții.

Totuși, OpenMP are și un mic dezavantaj. Există posibilitatea să apară anumite bug-uri care sunt mai greu de depistat. Utilizarea debuggerului este mai greu de realizat. Având în vedere avantajele pe care le aduce acest framework, lucrurile mai puțin bune sunt nesemnificative.

Pentru a evidenția acest lucru se va crea un program în C++ care va afișa toate numerele prime din intervalul 1- 1000 utilizând framework-ul OpenMP.

```
#include <iostream>
#include <stdio.h>
#include <chrono>

bool prim(int nr){
    int c = 0;
    for(int i = 1; i <= nr; i++)
        if( nr % i == 0)
            c++;
    if(c == 2)
```

² Un framework este o platforma folosită pentru dezvoltarea aplicațiilor software. Acesta oferă o baza pe care dezvoltatorii software pot construi programe.

```

    return 1;
    return 0;
}
int main(){
    auto start = std::chrono::high_resolution_clock::now();
    #pragma omp parallel for
    for(int i = 0; i < 10000; i++)
        if(prim(i) == 1)
            printf("%d\n", i);

    auto elapsed = std::chrono::high_resolution_clock::now() - start;
    double microseconds =
std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
    printf("\nTotal time:%f", microseconds / 1000000);
    return 0;
}

```

```

C:\Windows\system32\cmd.exe
4969
4973
4987
4993
4999
Total time:0.973055

```

Figură 4:Procesare multicore (timpul de execuție)

```

C:\Windows\system32\cmd.exe
9931
9941
9949
9967
9973
Total time:1.713098

```

Figură 5:Procesare normală (timpul de execuție)

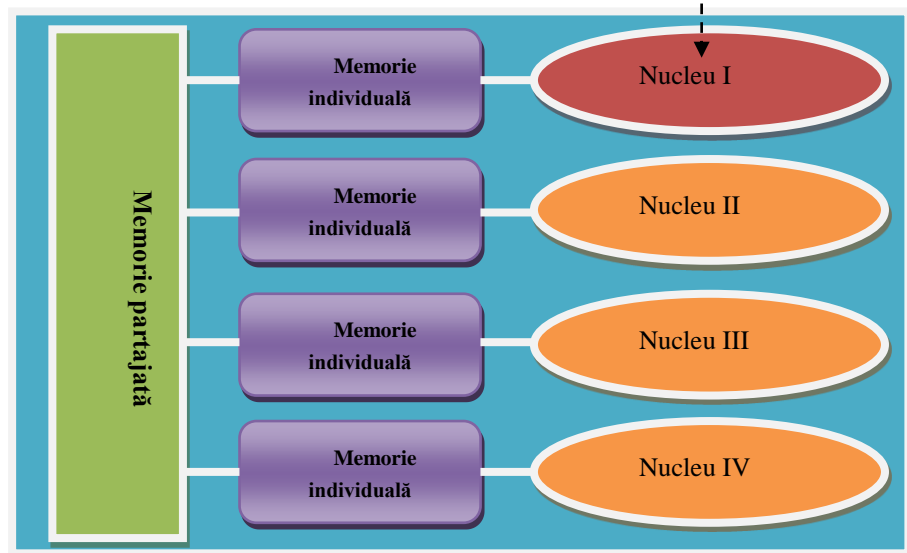
Diferența este realizată de linia colorată cu roșu. Prin acea linie se specifică faptul că instrucțiunea repetitivă for se va executa în paralel. În Figura 5 linia `#pragma omp parallel for` a fost comentată. Se observă faptul că timpul total de execuție este aproape dublu față de timpul obținut în Figura 4. Pentru a realiza Figura 3 s-a utilizat tehnica de procesare paralelă. Acest

lucru se poate observa și prin faptul că ultimele numere prime afișate nu sunt chiar ultimele din intervalul 1 – 10000 deoarece firul de execuție care se ocupă cu partea finală din for a finalizat task-ul mai rapid. Laptopul pe care s-a realizat testul are un procesor cu 4 nuclee. Pentru o înțelegere mai bună a faptului cum funcționează framework-ul OpenMP se va crea 2 diagrame.

6.1 Procesarea simplă a programului

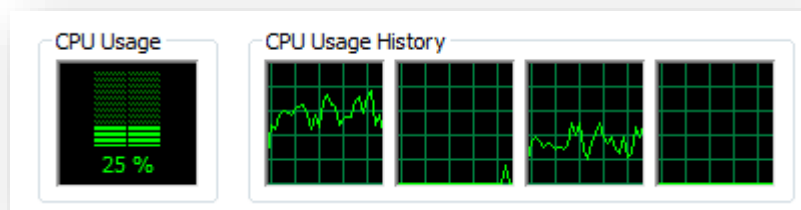
```
//#pragma omp parallel for
```

```
for(int i = 0; i < 10000; i++)  
    if(prim(i) == 1)  
        printf("%d\n", i);
```



Atunci când linia esențială este comentată codul este executat pe un singur nucleu³, restul nucleeleor fiind în mare parte libere. Astfel se observă faptul că avem putere de calcul dar nu este utilizată la capacitate înaltă. Dacă tragem o privire în task manager vom observa că procesorul lucrează doar la 25% din toată puterea disponibilă.

³ Un nucleu este o unitate de procesare care citește anumite instrucțiuni. În funcție de instrucțiunea citită se realizează o acțiune specifică. Un procesor deține mai multe nuclee. În procesarea paralelă fiecare nucleu realizează o acțiune simultan, iar apoi, dacă este necesar, acțiunile sunt combinate pentru a se ajunge la rezultatul final.

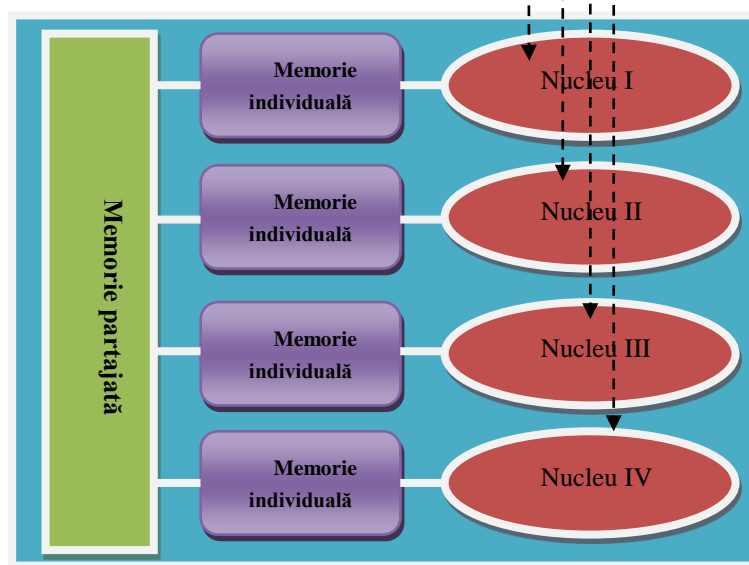


Figură 6: Reprezentarea nucleelor în task manager 1

În secțiunea CPU Usage History sunt prezentate cele 4 nuclee. Se observă că pentru a rezolva problema au lucrat doar 2 nuclee. Nucleul 1 și nucleul 3, puterea depusă fiind egală cu puterea unui singur core.

6.2 Procesarea paralelă a programului

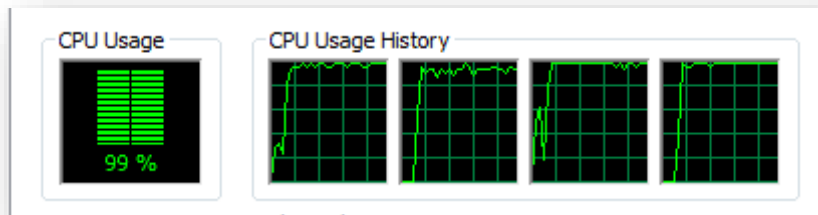
```
#pragma omp parallel for
for(int i = 0; i < 10000; i++)
    if(prim(i) == 1)
        printf("%d\n", i);
```



În cazul în care procesorul lucrează în paralel pentru a rezolva instrucțiunea repetitivă se procedează în felul următor:

- Se împarte n -ul la numărul de nuclee (în cazul nostru se împarte 10000 la 4)

- Se distribuie fiecărui nucleu o anumită parte din for
 - Nucleul I [0 – 2500)
 - Nucleul II [2500 – 5000)
 - Nucleul III [5000 – 7500)
 - Nucleul IV [7500 – 10000)
- Fiecare nucleu rezolvă partea primită



Figură 7:Reprezentarea nucleelor în task manager 2

Se observă că, atunci când folosim această metodă se obține o complexitate a timpului mult mai bună. Figura 7 ne indică faptul că toate nucleele lucrează la capacitate maximă pentru a rezolva problema cât mai rapid.

7 Arhitectura aplicației

7.1 Introducere

Pentru a realiza aplicația am utilizat un model de procesare și dezvoltare. Un model de procesare și dezvoltare este format dintr-un set de reguli care sunt respectate atunci când se creează un produs software. Scopul modelului este de a crea o aplicație fiabilă și de calitate, având în considerare toate etapele necesare.

Exemple de modele:

- Modelul cască
- Modelul V
- Modelul spirală
- Modelul incremental
- Modelul Agile

Modelul folosit este cel incremental. Acest model împarte cerințele inițiale în subseturi. Se pleacă de la un nucleu funcțional al aplicației. Odată ce un subset de cerințe a fost realizat și testat se adaugă la nucleu. Tot acest proces continuă până când nucleul îndeplinește toate cerințele. Acest model poate fi comparat cu un puzzle. Piesa de la care se pleacă reprezintă nucleul. Apoi se adaugă câte o nouă componentă. Trebuie asigurat faptul că noua componentă este cea corectă. Componenta reprezintă un modul ce a fost creat după un subset de cerințe. Astfel clientul are mereu acces la o aplicație funcțională.

7.2 Crearea aplicației

Aplicația va fi creată respectând modelul prezentat mai sus. Se va începe cu o versiune de bază, apoi se vor realiza optimizări până când rezultatul obținut este satisfăcător.

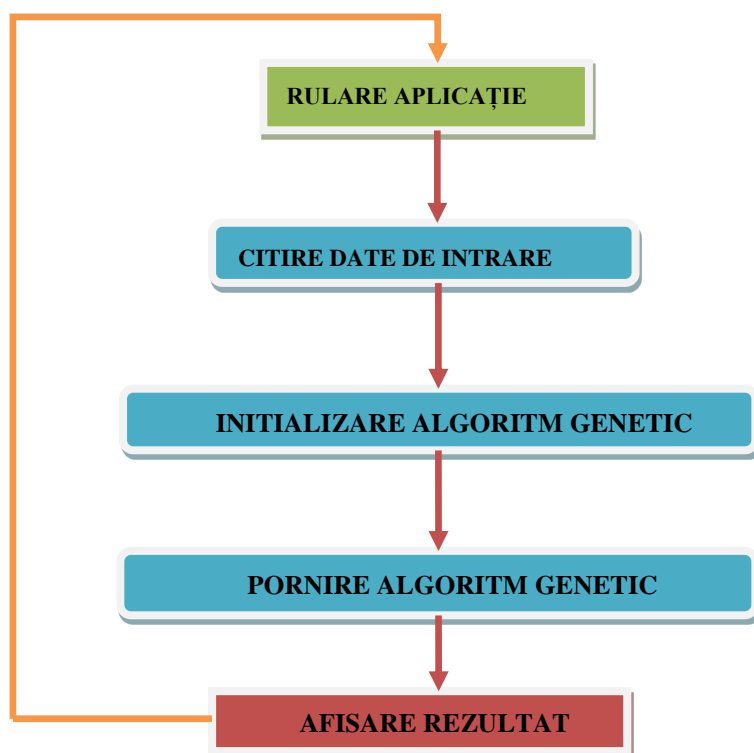
Evoluția aplicației:

- Realizarea nucleului aplicație
- Optimizare prin procesare multicore
- Optimizare prin rularea problemei pe un cluster de calculatoare
- Utilizarea Cloud Computing

7.2.1 Nucleul aplicației

Nucleul aplicației reprezintă prima versiune funcțională. Această versiune este formată din implementarea algoritmului genetic. Algoritmul genetic este adaptat astfel încât să funcționeze

în mod corespunzător pentru problema TSP. În continuare apare o diagramă care reprezintă funcționarea primei versiuni.



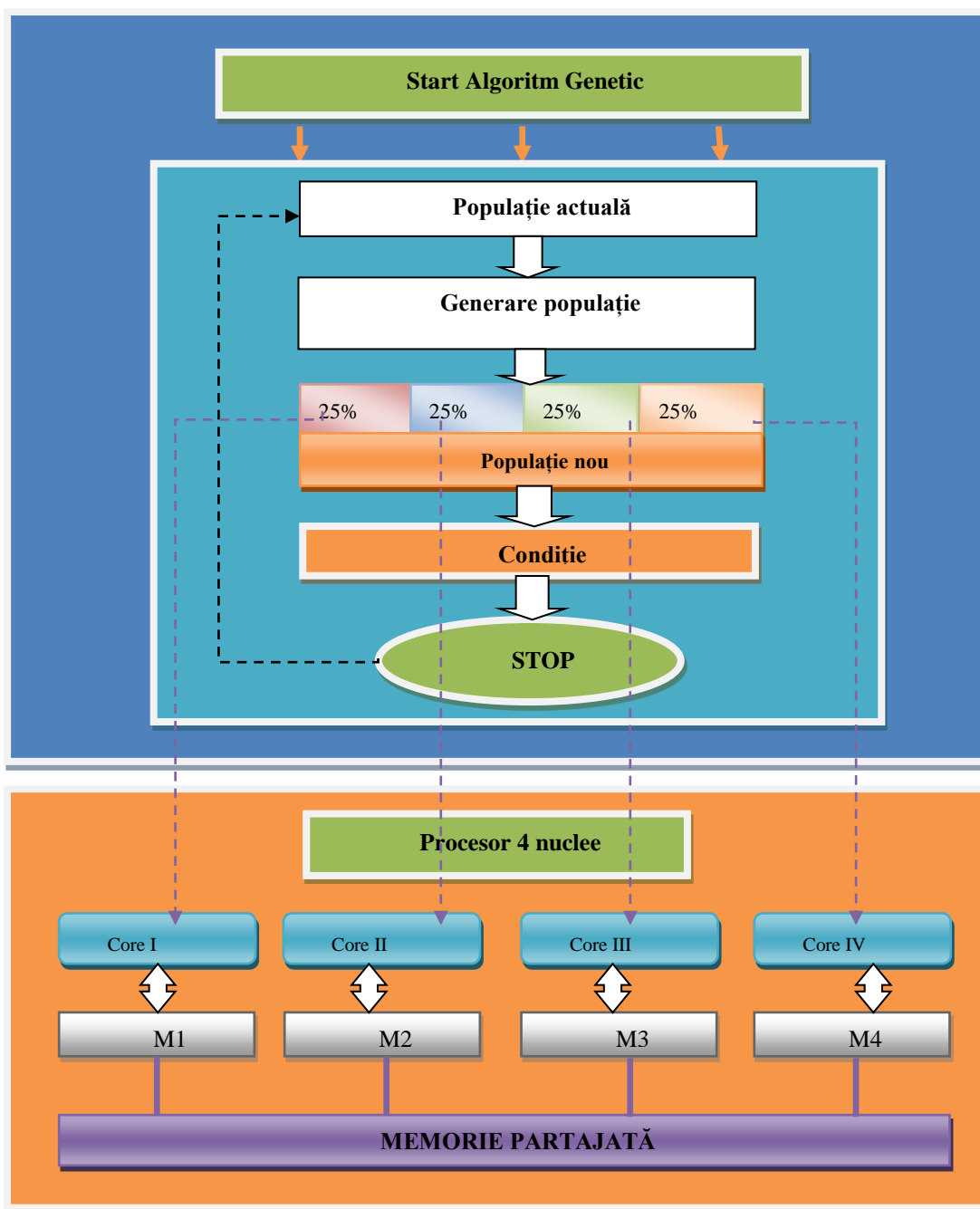
Figură 8: Arhitectura aplicației nucleu

La nivelul **citire date de intrare** se preia dintr-un fișier coordonatele diferitelor orașe și se încarcă în memorie. În continuare la **inițializarea algoritmului genetic** se setează parametri sus prezentați. Acești parametri vor fi adaptați în funcție de rezultatul obținut. Dacă setul de date este destul de mare se pot aplica niște tehnici special pentru estimarea parametrilor. Astfel se evită pierderea de timp pe rulări de test. La pasul unde se apelează **algoritmul genetic** se calculează soluția, iar apoi se trimit datele obținute către consolă.

7.2.2 Procesarea pe nuclee

Procesarea multicore reprezintă una dintre cele mai importante optimizări aduse acestei aplicații. La aplicația anterior prezentată se adaugă noua funcționalitate. Această procesare vine în ajutorul algoritmului genetic. Având în vedere că datele de intrare vor avea dimensiuni destul de considerabile algoritmul genetic necesită o procesare mai intensă pentru a genera noua populație. Așadar, această tehnică va fi utilizată în procesul de evoluție. Procesul de evoluție fiind cea mai intensă parte din algoritm. Realizând o analogie cu viața reală, generarea unei noi populații reprezintă trecerea de la generația în vârstă la o generație mai tânără. Cu cât această

parte se execută mai rapid cu atât se va ajunge într-un timp mai scurt la rezultat. Diagrama de mai jos prezintă noul mod de operare al algoritmului genetic asupra populației.



Figură 9: Procesarea multicore a algoritmului genetic

La o primă analiză se observă că pentru a genera noua populație cele 4 nuclee lucrează simultan. Fiecare nucleu generează 25% din noua populație. La final populațiile generate de fiecare nucleu sunt combinate într-o unică generație. Dacă procesorul ar fi mai performant (cu mai

multe nuclee) noua generație se va crea și mai rapid. Pentru a putea beneficia de această procesare multicore părțile care sunt realizate pe nuclee diferite trebuie să fie independente. Altfel, procesarea nu va fi posibilă. În acest caz, fiecare individ se generează într-un mod individual. Fără procesarea multicore populația era generată de un singur nucleu. Așadar, putem concluziona ușor faptul că prin această tehnică se obține un timp de rulare mai bun.

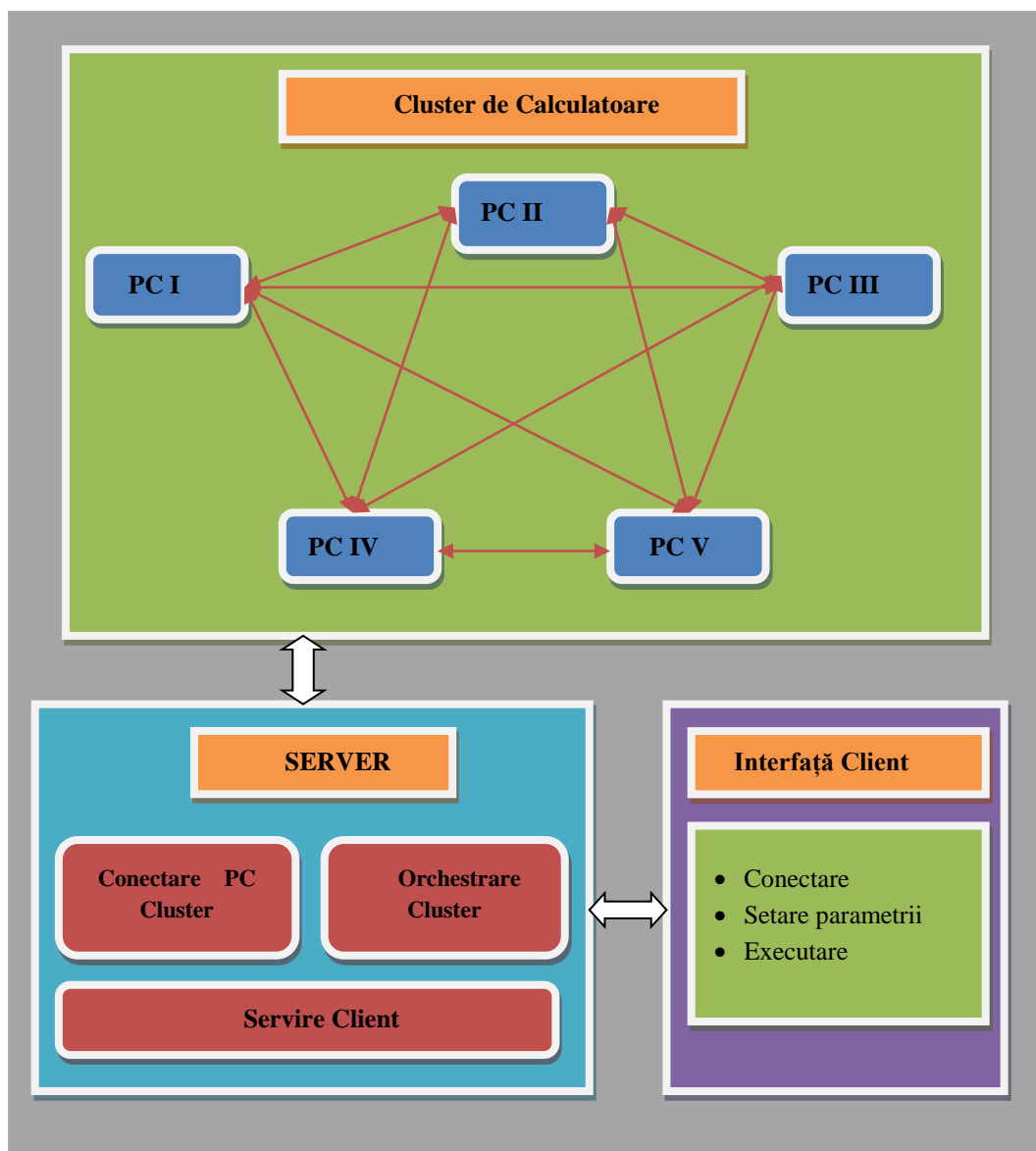
7.2.3 Rularea pe un cluster de calculatoare

Un cluster de calculatoare reprezintă mai multe unități de procesare interconectate printr-o rețea locală rapidă. Aceste calculatoare interconectate sunt privite ca o singură mașină dar cu o putere mai mare. Un cluster oferă următoarele avantaje:

- Viteză de procesare
- Capacitate de stocare
- Disponibilitate de resurse
- Fiabilitate

În general, clusterelor de calculatoare sunt utilizate pentru a mări puterea de procesare. Astfel, aplicații complexe pot să ruleze într-un ritm normal.

Pentru a mări performanța aplicației se va utiliza această tehnică. Mai exact, prin intermediul acestei metode se va duce conceptul de algoritm genetic la un nivel superior. Fiecare calculator din cluster va rula câte o instanță a algoritmului genetic. Datorită rețelei locale algoritmi genetici de pe diferite unități de procesare vor interschimba indivizi între ei. Astfel, se simulează migrația din viața reală. Inițial fiecare algoritm genetic va fi instanțiat cu o populație generată aleatoriu, după care la fiecare epocă se va genera o populație nouă. Rolul acestui cluster fiind evitarea deplasării algoritmului spre un maxim local și găsirea unei soluții cât mai aproape de optim.



Figură 10: Arhitectura aplicație ce implică procesarea pe un cluster de calculatoare

Cea mai importantă parte a acestei arhitecturi este reprezentată de server. Aceasta coordonează toată activitatea aplicației. În continuare se va descrie fiecare component din server.

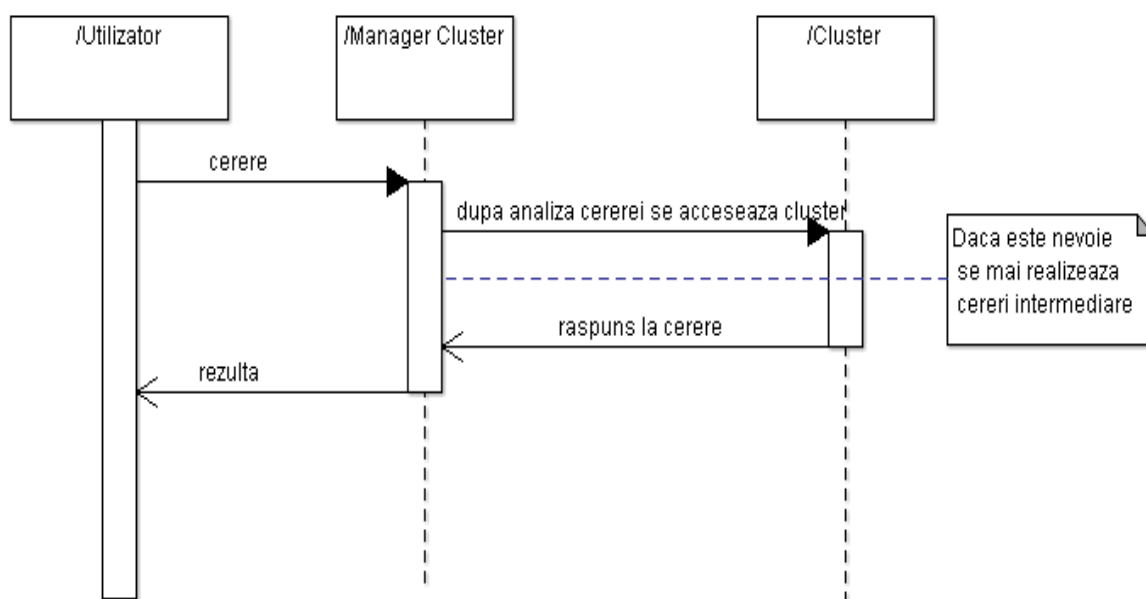
7.2.4 Conectare PC cluster

Prin intermediul serverului se permite adăugarea unei noi unități de procesare la cluster. Această nouă unitate, odată ce aderă la cluster, trebuie să aibă cunoștință și de celelalte componente. În plus, pe lângă acest fapt, este necesar să dețină și adresele de rețea ale acestora. Orice calculator trebuie să fie capabil să comunice cu un alt calculator din cluster, random ales, la un moment dat. Acest lucru este necesar pentru a realiza migrarea indivizilor. Toate datele necesare

comunicări între nodurile din cluster sunt oferite de către server. Având în vedere faptul că toate calculatoarele se alătură la cluster prin intermediul serverului acesta deține datele necesare realizării conexiunilor dintre noduri. Atunci când se conectează un utilizator toate nodurile sunt înștiințate cu cine trebuie să comunice.

7.2.5 Orchestrare cluster

Acest modul este foarte important. Are rolul de mediator între cluster și client. Practic, prin această cale se transmite calculatoarelor ce task trebuie să realizeze. Având în vedere că arhitectura clusterului este de tip master-slave⁴, serverul recepționează anumite cereri pe care le procesează. În urma procesării serverul își dă seama ce trebuie să ceară de la unitățile de procesare. Odată ce clientul stabilește o conexiune cu serverul, acesta are posibilitatea să realizeze anumite acțiuni. Acțiunile trimise către server sunt analizate, filtrate, iar mai apoi în funcție de decizia obținută se apelează clusterul, cerându-se un anumit rezultat. În timp ce nodurile rezolvă problema cerută serverul notifică utilizatorul în ce stadiu se află procesarea. Astfel se menține o legătură continuă cu userul.



Figură 11: Diagramă de secvență manager cluster

⁴ Arhitectura master/slave este formată din mai multe unități de procesare. Dintre toate aceste unități de procesare doar una are rol de master. Scopul master-ului este de a coordona activitatea celorlalte componente (slave). Cu alte cuvinte, există cineva care comandă și cineva care execută.

Diagrama de mai sus prezintă rolul de funcționare a managerului de cluster. Un client poate să acceseze clusterul doar dacă cererea trimisă este recunoscută de către server. Astfel se evită perturbarea componentelor din grup.

7.2.6 Comunicarea client – aplicație

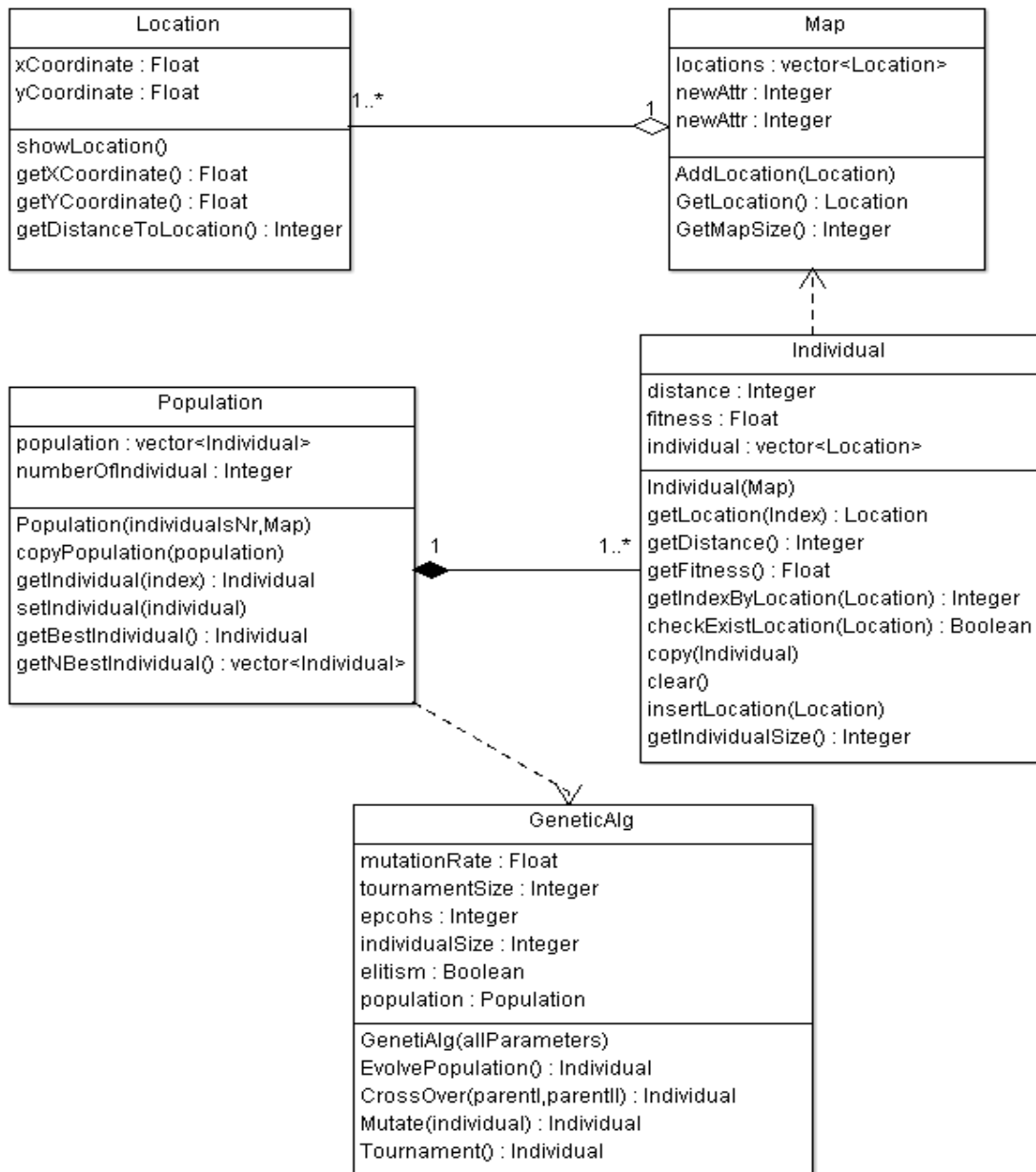
Comunicarea dintre client și aplicație se realizează prin intermediu unei interfețe desktop. Pentru ca această legătură să aibă loc este necesar ca ambele părți să dețină o conexiune de rețea exterioară. Înainte ca utilizatorul să aibă acces la funcționalitățile oferite de server trebuie să se asigure că există o conexiune între cele două părți. Pentru protejarea datelor ce trebuie să ajungă la server și invers se va utiliza protocolul de securitate SSL(Secure Sockets Layer). Scopul principal pentru care se va folosi acest protocol este de a oferi integritate datelor. Integritatea este proprietatea care asigură că datele ajung sub forma în care au fost trimise. Datele nu pot fi alterate de o persoană neautorizată. Interfața va fi user-friendly⁵. Odată ce conexiunea a fost stabilită, clientul are acces la diferite comenzi. Clientul nu are posibilitatea să execute comenzi necunoscute de server. Chiar dacă se reușește acest lucru, serverul va deține anumite tehnici de contracarare.

⁵ Prin interfață user-friendly se înțelege faptul că aceasta este ușor de utilizat și de înțeles. Utilizatorul trebuie să folosească aplicația cu plăcere.

8 Detalii de implementare

8.1 Implementarea aplicației nucleu

Pentru implementarea aplicației de bază s-a utilizat limbajul de programare C++. C++ fiind cel mai potrivit în această situație. Acest limbaj este orientat obiect ceea ce permite construirea unei aplicații complexe. În plus, suportă framework-ul OpenMp pentru procesare multicore.



Figură 12: Diagrama de clase a aplicației nucleu

Prin intermediul diagramei din Figura 12 se realizează trecerea de la descrierea aplicației la nivel architectural la o descriere mai explicită. Astfel se ușurează efortul depus la implementare. Problema tratată presupune detectarea celui mai scurt traseu dintr-o mulțime de orașe. Prin clasa **Location** se descrie modul de reprezentare al unei zone. O metodă importantă este **getDistanceToLocation()**. Rolul acestei metode este de a detecta distanța dintre două puncte $A(x_1, y_1)$ și $B(x_2, y_2)$ diferite. Distanța se calculează conform formulei: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ (distanța euclidiană). Clasa **Map** poate fi asociată cu o hartă din viața reală. Are ca scop păstrarea informațiilor despre fiecare locație. Una dintre cele mai importante clase este **Individual**. Un individ, în contextul algoritmilor genetici, este văzut ca fiind o potențială soluție. Acesta este format dintr-un vector cu locații. Fiecare locație este unică, iar poziția trebuie să fie random. Prima locație din vector reprezintă orașul de start, următorul fiind cel în care urmează să se ajungă.

Metode importante ale clasei **Individual**:

- **getDistance()** Această metodă returnează întreaga lungime a traseului. Pentru a afla distanța trebuie parcurs vectorul de locații, iar pentru fiecare două orașe consecutive se aplică metoda care returnează distanța dintre locații.
- **getFitness()** Are scopul de a returna cât de bună este o soluție. Astfel putem compara două rezultate

Clasa **Populație** reprezintă o mulțime de potențiale soluții. Numărul de potențiale soluții este stabilită prin intermediul unui parametru care a fost descris într-o secțiune anterioară. În primă fază populația este inițializată cu indivizi generați random. Cea mai importantă metodă existentă este reprezentată de **getBestIndividual()** care returnează soluția optimă existent în populație. Astfel știm oricând în ce stadiu se află rezolvarea problemei.

Ultima clasă, dar și cea mai importantă este **GenetiAlg**. Aici se află „inima” algoritmului genetic. În funcție de numărul de generații selectat se creează noi populații. Noua populație se creează utilizând-o pe cea veche peste care se aplică metoda de crossover, mutație, campionat etc. Pentru ca metoda crossover să poată fi accesată este necesar ca anterior să fi fost selectați doi indivizi din populație. Funcția realizează o combinație între cei doi indivizi ținând cont de anumite tehnici. Este foarte important ca această metodă să fie construită corect. Prin încrucișare, traseul rezultat nu trebuie să obțină două gene identice. Acest lucru ar însemna că

ar exista un traseu care trece de două ori printr-un oraș ceea ce nu dorim. Ar fi o abatere de la problema pe care trebuie să o rezolvăm.

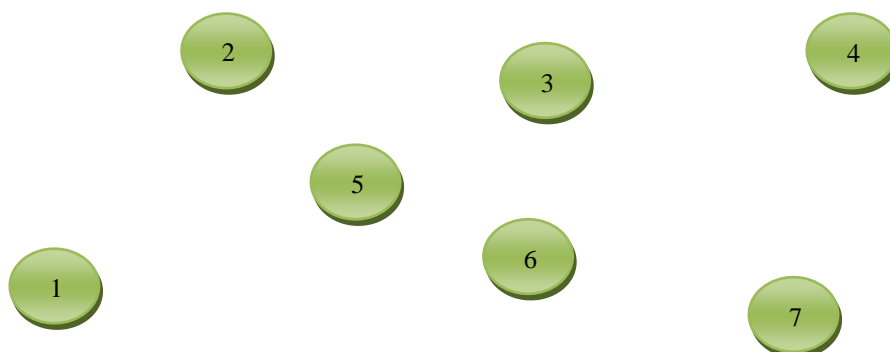
TRASEU I	Suceava	Bacau	Neamț	Vaslui	Iași	Botoșani	Vrancea
TRASEU II	Bacau	Neamț	Vaslui	Vrancea	Suceava	Iași	Botoșani
Rezultat	Suceava	Bacau	Neamț	Vrancea	Vaslui	Iași	Botoșani

Tabel 3: Exemplu de încrucișare

În exemplu de mai sus s-a procedat în următorul mod: Mai întâi s-a stabilit un prag de împărțire. Pragul selectat a fost trei. După care s-a trecut la formarea efectivă a traseului rezultat. S-a copiat primele trei (pragul selectat) orașe din traseul I, apoi, pentru că s-a îndeplinit condiția de împărțire s-a preluat restul orașelor din traseul II începând de pe poziția 4 (prag selectat + 1). În final s-a iterat din nou traseul I și pentru fiecare oraș s-a verificat dacă există în traseul nou creat. Dacă acesta nu există înseamnă că în noul traseu apare un duplicat, iar duplicatul va fi înlocuit de orașul care nu a fost găsit. Astfel ne asigurăm că traseul corespunde cerințelor.

Tehnica descrisă anterior este una generală care se aplică la majoritatea problemelor rezolvate utilizând algoritmi genetici. Pentru ca algoritmul să fie mai eficient am decis să utilizez o metodă de crossover specifică pentru problema Comisul Voiajor.

Metoda de reproducere prin recombinarea muchiilor. Aceasta abordare a fost propusă de către Darrell Whitley et al după doi ani de analiză. Ei au presupus că doar valoarea muchiilor sunt importante nu și direcția lor. Dacă privim problema după această presupunere observăm ca muchiile dintr-un traseu pot fi văzute ca informație ereditară ce poate fi transmisă. Operația de recombinare a muchiilor încearcă să preia de la părinți acele laturi care aduc câștig maxim prin combinare. Astfel se obține o nouă soluție mai bună. În continuare voi prezenta cum funcționează această metodă printr-un exemplu.



Presupunem că avem punctele anterior selectate și două trasee random generate.

— 5 <-> 4 <-> 2 <-> 1 <-> 7 <-> 6 <-> 3 <-> 5

— 7 <-> 3 <-> 5 <-> 1 <-> 2 <-> 6 <-> 4 <-> 7

Am notat cu $x \leftrightarrow y$ muchia de la nodul x la nodul y .

Modul de funcționare al metodei de reproducere prin recombinarea muchiilor:

- Pentru fiecare traseu se creează o listă a vecinilor fiecărui nod.

Traseu I	1	2	3	4	5	6	7
Vecin I	2	4	6	5	3	7	1
VecinII	7	1	5	2	4	3	6
Traseu 2	1	2	3	4	5	6	7
Vecin I	5	1	7	6	3	2	4
Vecin II	2	6	5	7	1	4	3

- Următorul pas este realizarea reuniuni celor două tabele.

Reuniune	1	2	3	4	5	6	7
Vecin I	2	1	5	2	1	2	1
Vecin II	5	4	6	5	3	3	3
Vecin III	7	6	7	6	4	4	4
Vecin IV				7		7	6

În acest tabel apar toate legăturile prezente în traseul I și traseul II. Pentru crearea acestui tabel se pot folosi mai multe trasee, dar nu este recomandat.

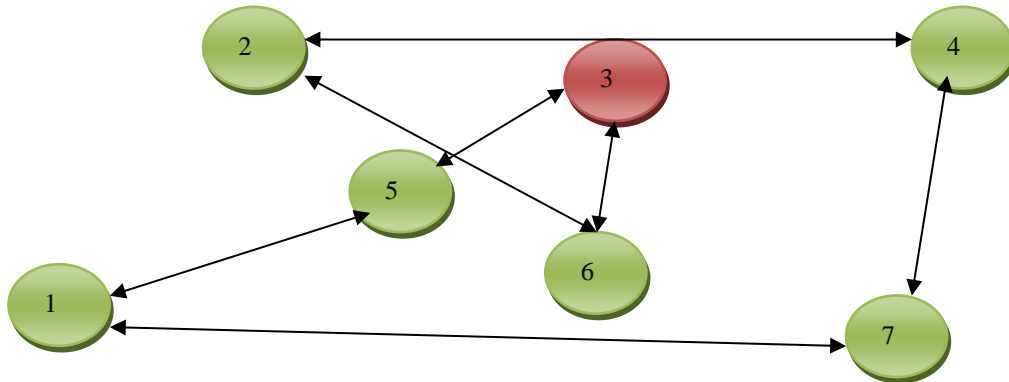
În continuare se selectează un nod random, se elimină acest nod din toate subseturile de vecini, iar apoi se selectează nodul cu cei mai puțini vecini din subsetul specific pentru numărul aleator ales.

Presupunem că numărul random selectat este 3. Ștergem acest nod din toate listele de vecini, iar apoi selectăm din lista de vecini corespunzătoare pentru 3 acel nod care are cei mai puțini vecini. Dacă două noduri au același număr de vecini se selectează aleator unu. Se repetă

acești pași până obținem un nou traseu cu dimensiunea corespunzătoare, doar că de această dată plecăm de la ultimul nod determinat.

Aplicând aceste reguli am obținut următorul traseu:

3 <-> 5 <-> 1 <-> 7 <-> 4 <-> 2 <-> 6



8.2 Implementare cluster de calculatoare

Obiectivul clusterului este de a îmbunătăți performanța aplicației prin creșterea numărului de candidați și prin variația lor. Cu cât există mai mulți candidați și cu cât aceștia sunt mai diferiți cu atât șansa ca soluția optimă să se regăsească printre ei este mai mare. Mai sus am prezentat că un individ are posibilitatea să migreze de pe un calculator pe altul. Pentru a realiza acest scop, pe lângă aplicația nucleu, mai apar două clase:

- Clasa **Migration** are ca scop crearea legături cu calculatorul pe care un anumit individ urmează să ajungă. Această legătură se creează la apariția semnalului că un candidat dorește să părăsească populația natală. După ce legătura a fost realizată urmează transferul propriu zis. În cele din urmă conexiunea este închisă.
- Clasa **Boundary** care are ca scop interceptarea cererilor de migrare. Instanța acestei clase trebuie să lucreze în paralel cu algoritmul genetic. Odată ce un individ a fost recepționat algoritmul genetic trebuie anunțat pentru al integra în populația existentă. Pentru a realiza această paralelizare se vor utiliza fire de execuție.

```
DWORD WINAPI Thread_no_1( LPVOID lpParam)
{
    Granita::connectionBetweenNodes();
    return 0;
}

int main() {
    //adresă ip server
    char ip[] = {"192.168.0.102"};
```

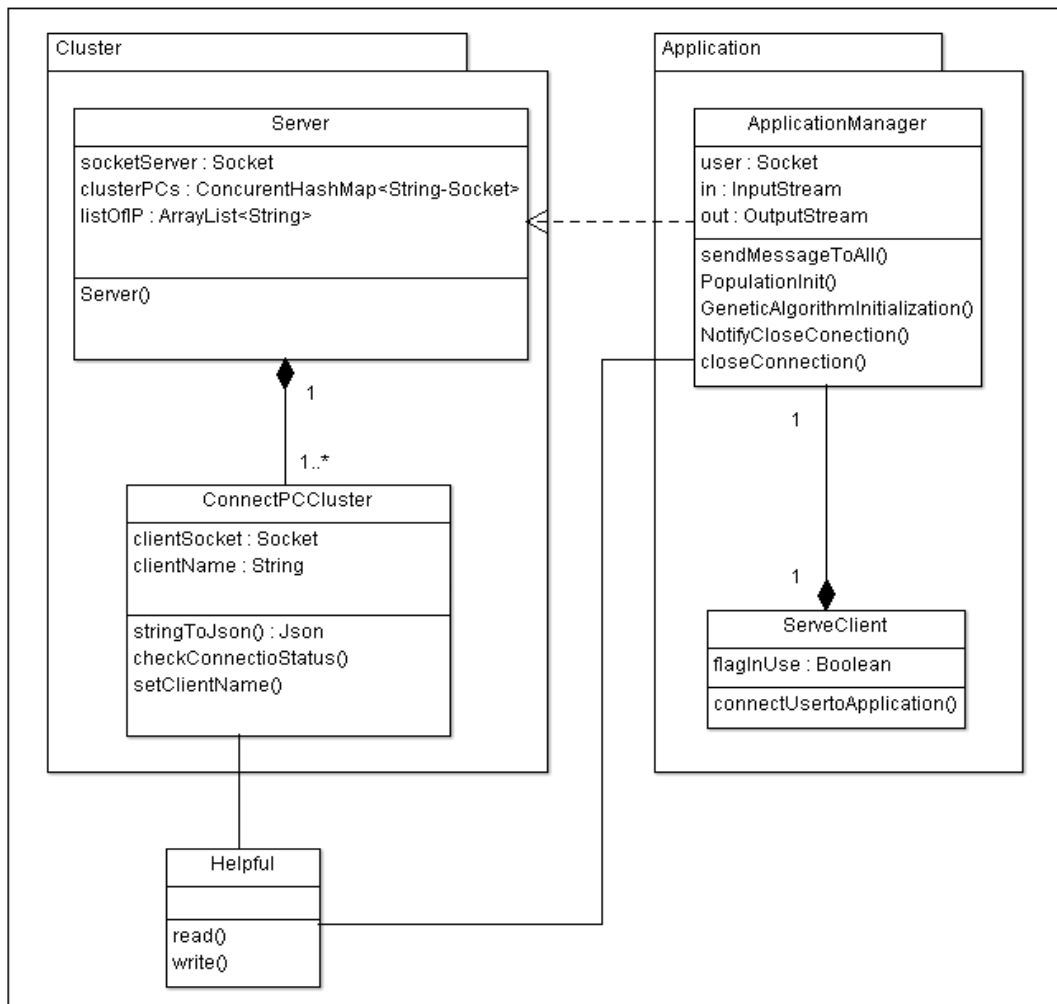
```

Node *pcCluster = new Node(7777,ip);
//stabilirea conexiunii cu serverul
if(pcCluster->InitConnection() == 1){
    printf("Crearea conexiunii a esuat!!!");
    exit(-1);
}
/*Dacă conexiunea a reușit se creează un thread care va aștepta
conexiuni de la celelelalte calculatoare din cluster*/
CreateThread( NULL, 0, Thread_no_1, reinterpret_cast
<void*>(client), 0, NULL);
pcCluster->Communication();
}

```

Codul de mai sus indică faptul că odată cu stabilirea comunicării pc – server se creează un thread care are rolul de a aștepta migranți.

8.3 Server

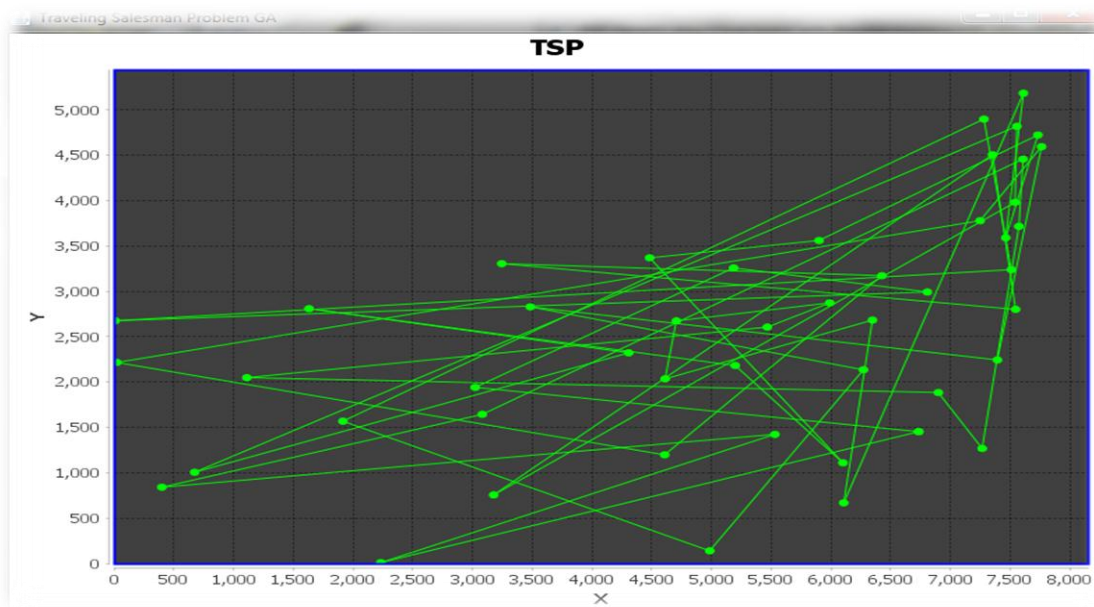


Figură 13: Diagrama de clase pentru server:

Instanța clasei **Server** și instanța clasei **ServeClient** rulează în paralel, astfel se permite adăugarea de noi calculatoare la cluster în caz că este nevoie de mai multă putere computațională. Atributul **clusterPCs** din clasa **Server** reține conexiunile cu calculatoarele din cluster. Instanța clasei **ServeClient** creează un thread care așteaptă conexiuni din partea clienților. Odată ce un utilizator accesează aplicația primește acces la instanța **ApplicationManager**. Prin intermediul metodelor puse la dispoziție de **AppManager** se pot transmite diverse acțiuni către cluster. Pentru a evita apariția unei erori în program, înainte ca algoritmul să ruleze se verifică starea nodurilor din cluster. Dacă un anumit nod nu mai răspunde acesta este eliminat. Uneori pot să apară situații speciale prin care conexiunile dintre noduri și server sunt întrerupte într-un mod nefiresc. Conexiunile sunt reconstruite la un anumit interval. Acest lucru se realizează doar în cazul în care algoritmul de rezolvare al problemelor nu este deja în funcțiune.

9 Performanțe obținute

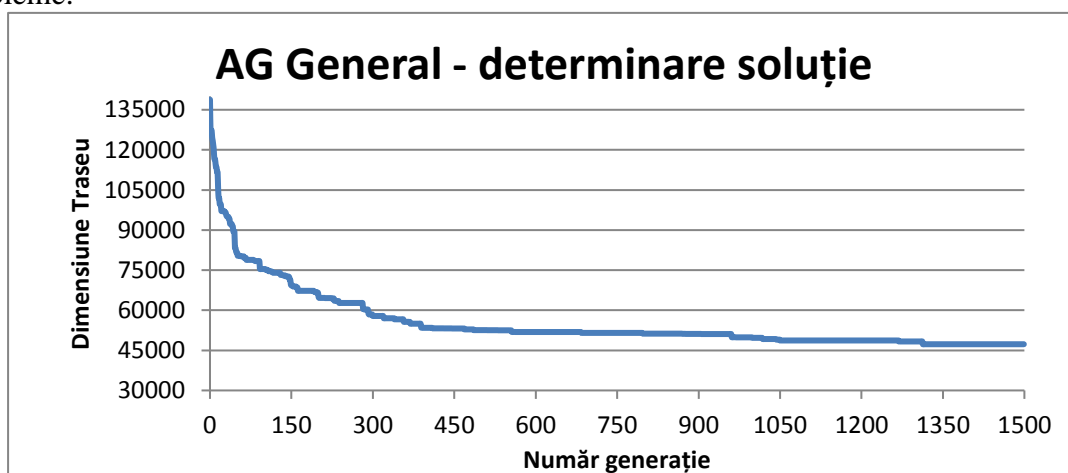
În această secțiune voi prezenta performanțele obținute plecând de la versiunea de bază până la versiunea cea mai avansată a aplicației. Pentru fiecare variantă se va realiza câte o statistică prin care se va evidenția timpul în care algoritmul ajunge la soluție. Pentru statistici am folosit un set de date format din coordonatele capitalelor statelor din SUA. Dimensiunea populației fiind de o sută de indivizi, iar numărul de generații 1500.



Figură 14: Exemplu de traseu inițial

9.1 Rezolvarea problemei cu un algoritm genetic general

Prima versiune pe care am realizat conține un algoritm genetic care se pretează pe mai multe probleme.

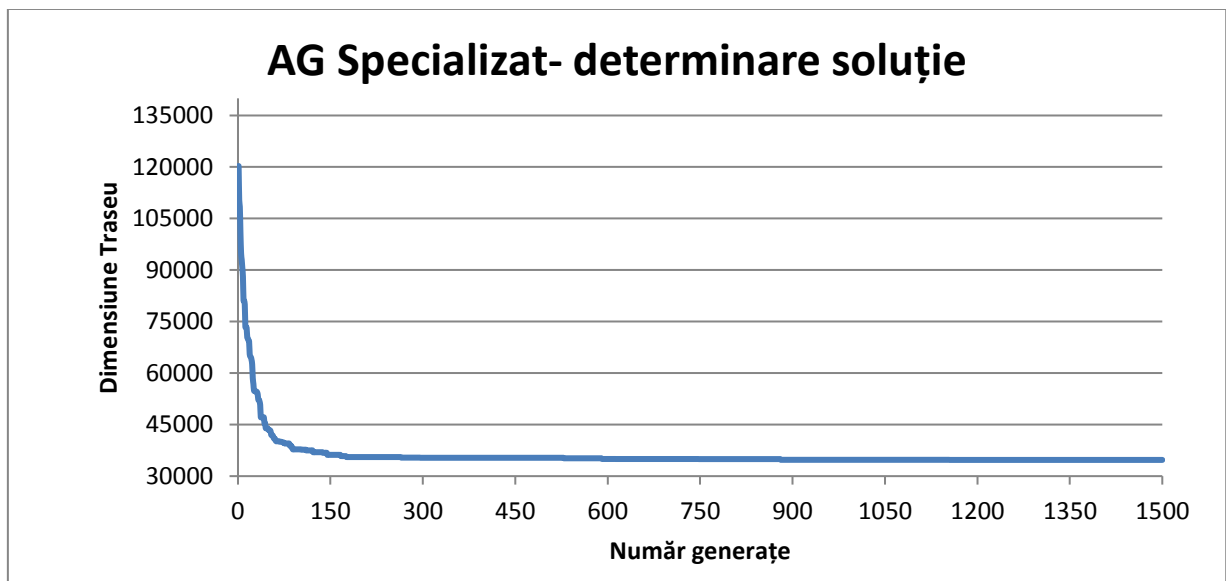


```
Timpul total în secunde:150.844  
Cel mai bun traseu(distanța):47282  
Numar total de iteratii:1500
```

După cum se observă din grafic algoritmul se stabilizează cam pe la iterația **1100** ceea ce înseamnă că timpul total de execuție ar fi în jur de **100 de secunde**.

9.2 Rezolvarea problemei utilizând un algoritm genetic specializat

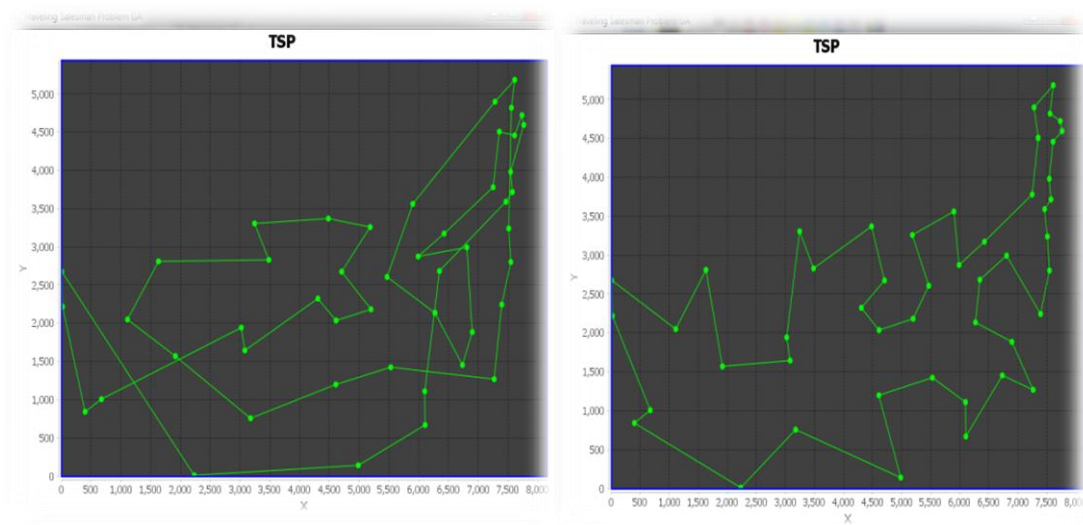
Următorul pas spre îmbunătățire a fost adaptarea algoritmului genetic pentru problema Comisul Voiajor. Pentru aceasta s-a adăugat funcții noi de încrucișare și mutație. Aceste metode au fost descrise în secțiunile de mai sus. În plus, s-a optimizat structura aplicației astfel încât algoritmul să nu realizeze operații în plus. Orice operație care nu își are rostul înseamnă pierdere de timp. După aplicarea acestor tehnici de optimizare am obținut următoarele rezultate:



```
Timpul total în secunde:118.584  
Cel mai bun traseu(distanța):34731  
Numar total de iteratii:1500
```

Se observă că am păstrat aceiași parametrii ca mai sus, dar de această dată algoritmul se stabilizează pe la **iterația 200**. Aplicând un calcul simplu deducem că se ajunge la o soluție optimă în jur de **15 secunde**. În plus, față de versiunea anterioară se observă că soluția determinată este mult mai bună. Diferența dintre traseul determinat anterior și traseul determinat acum fiind de 12551. Dacă am dori să ajungem la o soluție apropiată de cea returnată de prima variantă această nouă abordare ne-ar oferi-o într-un timp de doar **4 secunde**. Așadar

putem deduce că prin tehnicile aplicate am reușit să **economisim 96 de secunde**, un rezultat destul de bun. Totuși pentru a obține o soluție mai performantă algoritmul are nevoie de **15 secunde**.



Figură 15: Diferența dintre algoritmul genetic specializat și cel general

Figura 15 pune în evidență calitatea algoritmului genetic specializat. Se observă, cu ușurință, că acesta este superior celui prezentat în secțiunea 9.1.

9.3 Algoritm genetic plus procesare pe nuclee

Chiar dacă am obținut un rezultat destul de bun am continuat optimizarea prin adăugarea tehnicilor de procesare pe nuclee la versiunea anterioară. Pentru a testa această eficientizare am folosit serviciul de computing oferit de Windows Azure. Am configurat o mașină virtuală cu patru nuclee pe care am rulat partea de rezolvare a problemei. Graficul obținut în urma procesării arată ca cel de la punctul 9.2 ceea ce este normal deoarece nu s-a realizat nici o modificare la algoritmul. În plus, timpul de procesare scade considerabil.


```
Timpul total in secunde:52.591
Cel mai bun traseu(distanta):34349
Numar total de iteratii:1500
```

La versiunea fără procesare multicore timpul de procesare era de **118.584 secunde**, iar acum a scăzut până la **52.592 secunde**. Dacă luăm în calcul faptul că algoritmul s-a stabilizat la iterația 200 observăm că obținem soluția optimă în aproximativ **7 secunde**.

În concluzie, prin utilizarea procesării multicore am reușit să reducem timpul de procesare cu mai mult de 50%. Dacă înainte trebuia să pierdem **15 secunde** pentru a obține o soluție bună, acum **7 secunde** este mai mult decât suficient.

9.4 Implementarea pe un cluster de calculatoare

Implementarea soluției pe un cluster de calculatoare reprezintă versiunea finală de optimizare. Prin intermediul clusterului am încercat să trec algoritmul genetic la un alt nivel. Cu cât se utilizează mai multe calculatoare cu atât populația va fi mai diversificată. Realizând această implementare am vrut să simulez cât mai bine modul în care funcționează viața reală. Un individ are posibilitatea să migreze către altă populație. Acest lucru înseamnă că el poate să aducă informație nouă în populația în care va ajunge. Pentru testarea acestei versiuni am folosit serviciul de computing oferit de Amazon. Am configurat 6 mașini virtuale. Pe 5 dintre ele am instalat partea de noduri, iar pe una partea de server. În acest caz testarea performanței este mai greu de realizat deoarece mașinile virtuale la care am avut acces dețin doar un singur core. Procesarea pe nuclee fiind, așadar, inutilă.



<input type="checkbox"/>	Node_1	i-083dfbecc84e7287e	t2.micro	us-west-2c	running	2/2 checks ...
<input type="checkbox"/>	Node_2	i-0ebf4796d148eefcd	t2.micro	us-west-2c	running	2/2 checks ...
<input type="checkbox"/>	Node_3	i-0f8f5658295c1dac6	t2.micro	us-west-2c	running	2/2 checks ...
<input type="checkbox"/>	Node_4	i-0f97e5447979b1172	t2.micro	us-west-2c	running	2/2 checks ...
<input type="checkbox"/>	Node_5	i-0fd76b25cfa69627b	t2.micro	us-west-2c	running	2/2 checks ...
<input type="checkbox"/>	Server	i-0bc653f0805506842	t2.micro	us-west-2c	running	2/2 checks ...

Figură 16: Instanțele de EC2

În Figura 16 apar toate instanțele configurate corespunzător. După testele efectuate am ajuns la concluzia că procesarea pe cluster reprezintă un plus, dar nu suficient încât să depășească procesarea pe nuclee. Aici luăm în calcul și faptul că aplicația multicore poate fi utilizată pe un procesor ce deține mai multe nuclee.

9.5 Evidențierea diferențelor dintre versiuni

Pentru a evidenția cât mai bine diferențele dintre versiunile implementate cât și avantajele aduse de fiecare în parte am utilizat un set de date format din 14 tipuri de date de intrare. Am folosit atât inputuri de dimensiuni mici, dar și inputuri de dimensiuni mari. Astfel am testat modul de comportare al implementărilor în diferite situații. Pentru fiecare input am contorizat timpul de

rulare, numărul de iterații necesare pentru a se ajunge la soluția optimă, cât și distanța finală a traseului depistat. În graficele următoare am notat cu:

1. Implementarea algoritmului genetic general;
2. Implementarea algoritmului genetic care se pliază pe problema TSP;
3. Implementarea de la punctul doi plus procesare multicore;
4. Implementarea algoritmului genetic adaptat pe un cluster de calculatoare;

În tabelul de mai jos se poate vizualiza rezultatul pentru fiecare test în parte. Astfel se poate crea o viziune mai amplă asupra optimizărilor realizate. Pentru o analiză mai ușoară am preferat să împart tabelul în două părți. Așadar, avem un tabel în care apare algoritmul genetic general și algoritmul genetic adaptat și un tabel în care se prezintă rezultatele obținute pentru algoritmul genetic adaptat plus procesarea pe nuclee și algoritmul genetic adaptat plus procesarea pe cluster.

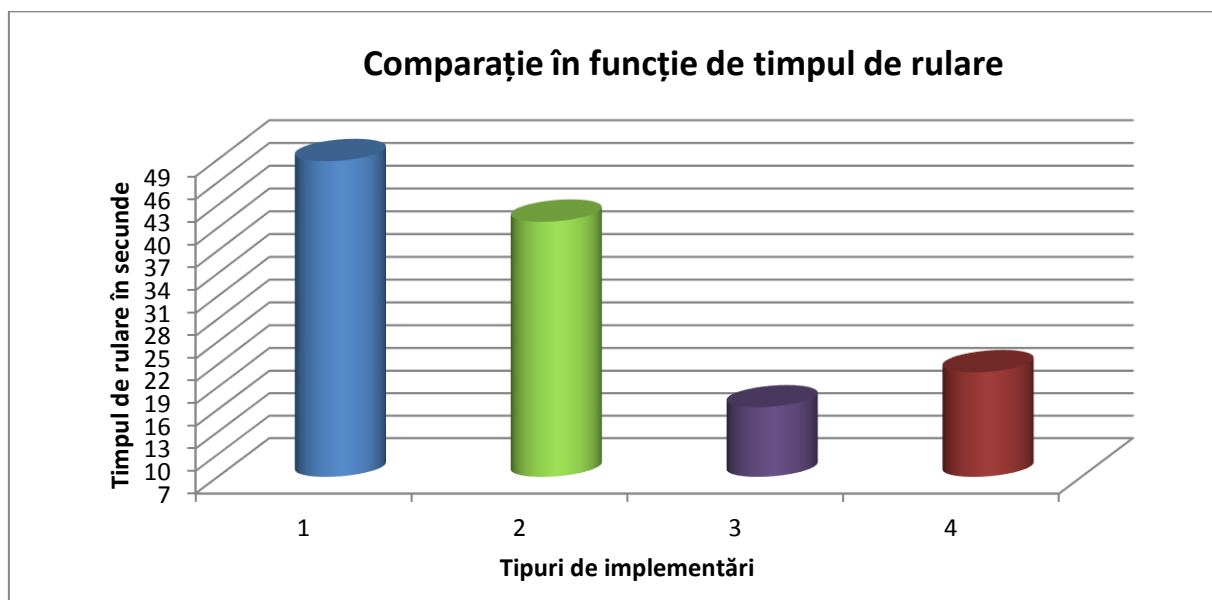
ID	Orașe / Set		AG General			AG Adaptat		
	Număr Orașe	Distanță Inițială	Timp rulare	Distanță Soluție	Număr Iterații	Timp rulare	Distanță Soluție	Număr Iterații
1	5	602.18	0.013	548.55	1	0.011	548.55	1
2	10	1482.28	0.275	965.65	16	0.107	965.65	6
3	15	1718.39	1.477	957.58	83	0.515	957.58	20
4	20	3273.58	12.4	1271.01	599	0.716	1132.14	21
5	25	3347.64	11.32	1664.21	474	6.75	1349.50	146
6	30	5190.252	15.21	1721.74	539	5.295	1342.61	90
7	35	5615.57	29.85	1878.21	888	9.566	1619.28	134
8	40	5901.33	36.28	2073.32	986	37.39	1615.03	441
9	45	7059.06	40.79	2252.78	995	23.27	1866.85	230
10	50	7985.34	81.14	2661.43	894	70.03	1787.02	599
11	55	7869.78	93.53	2871.12	853	72.47	2000.59	540
12	60	9422.61	101.27	3005.54	999	110.23	1977.84	916
13	65	9627.68	131.81	2948	987	135.11	1937.53	860
14	70	10231.09	127.55	3650	955	99.71	2248.00	445

Tabel 4:Rezultate obținute 1

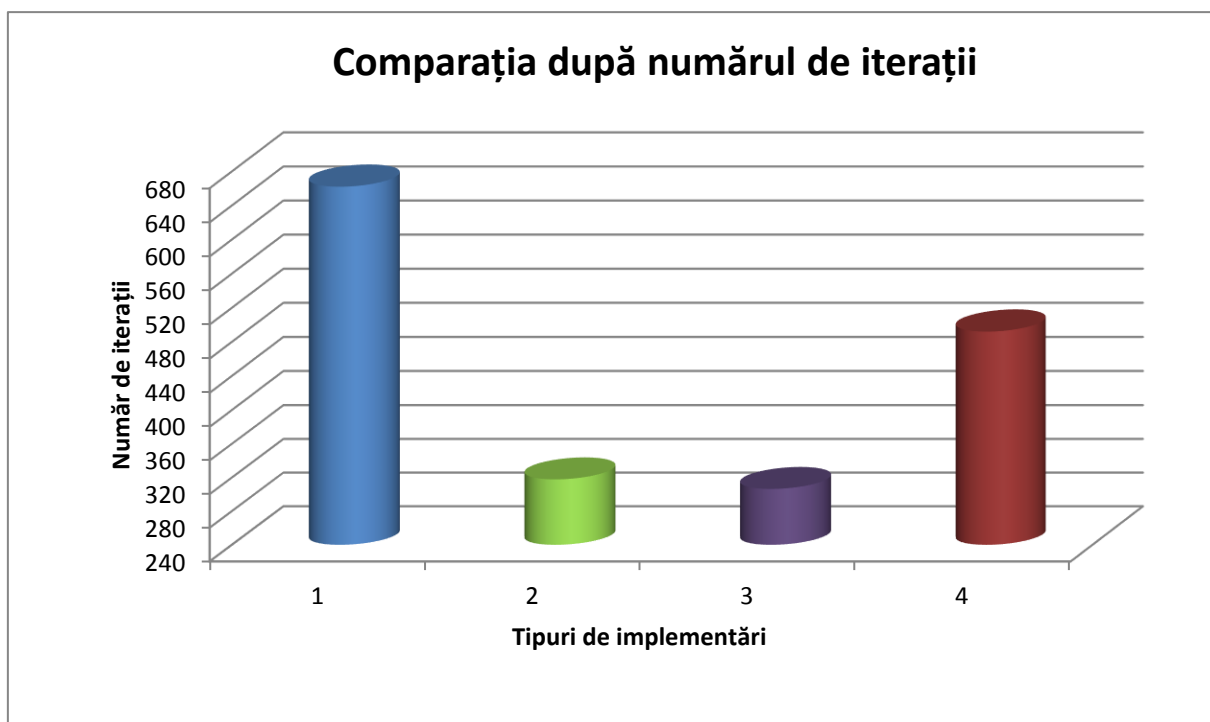
Table 1. Rezultate obținute II

ID	Orașe / Set		Orașe / Set			AGA + Procesare Cluster		
	Număr Orașe	Distanță Inițială	Timp rulare	Distanță Soluție	Număr Iterații	Timp rulare	Distanță Soluție	Număr Iterații
1	5	602.18	0.0049	548.55	1	1.713	548.55	1
2	10	1482.28	0.078	965.65	8	1.623	965.65	7
3	15	1718.39	0.230	957.58	18	2.932	957.58	142
4	20	3273.58	1.008	1132.14	71	4.138	1132.14	194
5	25	3347.64	2.093	1323.70	99	6.809	1297.33	361
6	30	5190.252	1.012	1332.77	39	11.10	1332.35	419
7	35	5615.57	6.52	1509.20	188	13.34	1521.75	576
8	40	5901.33	16.942	1592.22	479	19.04	1619.25	596
9	45	7059.06	9.40	1770.85	230	21.33	1775.84	642
10	50	7985.34	17.56	1796.07	374	29.25	1819.99	664
11	55	7869.78	43.62	1993.05	799	30.81	2072.18	735
12	60	9422.61	37.89	2119.76	721	41.17	2136.39	827
13	65	9627.68	49.84	2058.20	717	53.16	2158.59	792
14	70	10231.09	42.26	2371.11	546	57.20	2249	933

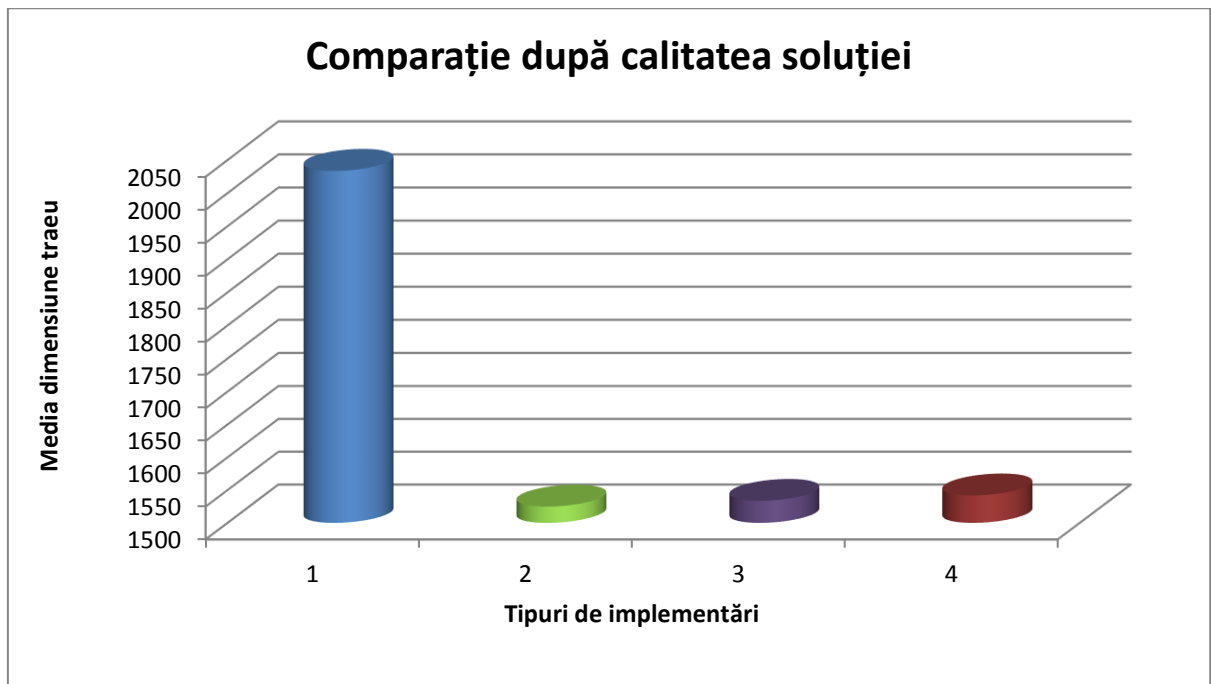
Graficele de mai jos au fost create peste informațiile de mai sus. Scopul graficelor este de a evidenția optimizările aduse.



Din graficul de mai sus se observă ușor faptul că algoritmul genetic specializat peste care s-a adăugat procesarea pe nuclee este net superior, în special, față de primele două variante. Doar varianta care rulează pe un cluster de calculatoare reușește să se apropie de performanțele maxime.



În cazul în care comparația se realizează după numărul de iterație până ce algoritmul converge situația se schimbă puțin. Este normal ca implementarea doi să se apropie de implementarea trei. În esență, algoritmi sunt aceeași, doar că cel de la trei beneficiază de procesare multicore, ceea ce înseamnă că iterațiile se execută mai rapid. În plus, observăm că aplicația patru pierde la acest capitol. Acest fapt se datorează împărțirii populației pe mai multe calculatoare. Pentru a compensa acest lucru se necesită mai multe iterații.



Din graficul de deasupra se observă clar că soluțiile oferite de algoritmul genetic general nu se ridică la nivelul soluțiilor oferite de restul implementărilor. În rest, celelalte versiuni determină cam aceeași calitate a soluției.

După analizarea amănunțită a tuturor acestor grafice concluzionăm că implementarea algoritmului genetic specializat pe problema Comisul Voiajor care utilizează procesare pe nuclee este alegerea perfectă. Acesta este capabil să determine o soluție bună într-un timp scurt. În plus, dacă această aplicație va rula pe o mașină puternică va conduce la descoperirea soluției într-un timp și mai rezonabil. Dacă avem acces la un cluster de calculatoare putem să alegem aplicația care a fost special creată să utilizeze mai multe unități de procesare. Aceasta va utiliza puțin mai mult timp pentru a determina rezultatul.

Concluzii

Consider că prin intermediul acestei lucrări am reușit să pun în evidență impactul imens al optimizării cât și importanța unei arhitecturi bine definite în construirea unei aplicații puternice, robuste, fiabile și performante. Prin tehnicile descrise în capitolele anterioare am reușit să obținem o optimizare a timpului cu peste **80 de procente**. Dacă în primă fază, pentru setul de intrare testat, aplicația a returnat soluția optimă în timp de **150 de secunde**, după optimizare soluția a fost determinat în mai puțin de **17 secunde**. Acest rezultat ar fi fost și mai satisfăcător dacă aplicația ar fi fost rulată pe o unitate de procesare care deține mai mult de patru nuclee. Totodată, au fost prezentate și anumite metode de optimizare care pot fi aplicate pe aproape orice tip de aplicație. Cea mai importantă fiind optimizarea prin procesare multicore. Acest tip de procesare a adus o îmbunătățire considerabilă.

În urma acestei lucrări am ajuns la concluzia că nu este suficient faptul că știm să programăm. Programarea poate fi văzută ca fiind partea cea mai ușoară din întreg procesul de construire al aplicației. Etapa cea mai notabilă fiind faza de proiectare a modelului. O aplicație cu o structură stabilă și bine definită conduce la un produs software de calitate. Este foarte important să se realizeze o analiză amplă asupra viziunii proiectului ce urmează a fi construit. Mai bine se pierde ceva timp la construirea arhitecturi decât să se depisteze ulterior că nu au fost cuprinse toate cerințele necesare.

În plus, am dobândit noi cunoștințe importante care cu siguranță le voi aplica în ceea ce urmează să realizez. Sunt de părere că faza de optimizare trebuie să aibă un loc bine definit în orice model de proiectare. Un produs software care nu a trecut prin faza de optimizare nu poate fi considerat ca fiind realizat profesionist.

Direcții de dezvoltare

Cu toate că rezultatul obținut în urma optimizării este destul de bun acesta încă mai poate fi îmbunătățit. Un următor pas spre eficientizare ar fi realizarea unei implementări utilizând CUDA. Compute Unified Device Architecture este o platformă pentru calcul paralel ce a fost inventată de către NVIDIA. Această platformă poate crește performanța aplicației foarte mult deoarece utilizează puterea oferită de placa video (GPU). Spre deosebire de CPU, unitatea de procesare GPU deține o arhitectură paralelă destul de puternică formată din mii de nuclee mai mici, eficiente, create special pentru a rezolva probleme simultan.

O altă cale spre îmbunătățire poate fi reprezentată de combinarea soluției actuale cu o altă abordare de rezolvare a problemei. De exemplu, algoritmul genetic actual construiește populația inițială folosind factorul random. Am putea să introducem tehnica greedy combinată cu o anumită probabilitate de inserare a unor gene.

O abordare mai profesionistă și mai interesantă, care se aseamănă dintr-un anumit punct de vedere cu algoritmul genetic, poate fi utilizarea rețelelor neuronale. O rețea neuronală artificială este o metodă de procesare a informației ce a fost creată prin încercarea de simulare a modului de funcționare a creierului uman. Acest model funcționează având la bază două tehnici: partea de antrenare și partea de testare. La partea de antrenare se asimilează modul de funcționare prin încercarea de rezolvare a unui set de date. Acest set de date conține un anumit număr de date de intrare și soluția corectă pentru fiecare input în parte. O rețea neuronală antrenată poate fi văzută ca un expert ce știe să rezolve probleme de tipul celor existente în setul de date.

Mai există o abordare interesantă ce poate fi folosit pentru îmbunătățirea acestei probleme, și anume optimizarea cu colonii de furnici. Această tehnică a fost descoperită prin analiza modului în care o familie de furnici își caută hrana. S-a realizat un experiment în care furnicile aveau acces la mâncare prin intermediul a două trasee. Un anumit traseu având distanța mai mică. În urma acestui experiment s-a concluzionat că furnicile tind să meargă pe calea mai redusă. Pentru a determina drumul minim furnicile comunică folosind o substanță denumită feromon. Mai întâi furnicile se deplasează aleatoriu și la fiecare pas lasă această substanță în urmă. În funcție de diferența de feromon produsă pe trasee diferite, furnicile decid care este cea mai bună rută.

Bibliografie

- Akhter, S., & Roberts, J. (2006). Multi-Core Programming: Increasing Performance through Software Multi-threading.
- Amazon. (2017). *AWS Documentation*. Preluat de pe <https://aws.amazon.com/documentation/>
- Barney, B. (fără an). *OpenMP Documentation*. Preluat de pe <https://computing.llnl.gov/tutorials/openMP/>.
- Breabăn, M. E. (2004). *Algoritmi Genetici*. Preluat de pe <https://profs.info.uaic.ro/~pmihaela/GA/curs.pdf>.
- Fog, A. (2015). *Optimizing software in C++*. Technical University of Denmark.
- Laporte, G. (1991). The Travelling Salesman Problem: An Overview of exact and approximate algorithms.
- Larranaga, P., Kuijpers, C. M., Murga, R. H., Inza, I., & Dizdaveric, S. (1999). Genetic Algorithm for the Travelling Salesman Problem: A Review of Representations and Operators.
- Lenuta, A. (2017). *Cloud Computing*. Preluat de pe <https://profs.info.uaic.ro/~adria/teach/courses/CloudComputing/coursepractical-works.html>.
- Melanie, M. (1999). An Introduction to Genetic Algorithm.
- Rabbah, R., & Amarasinghe, S. (2007, January). MIT Course: Multicore Programming Primer.
- Rai, K., Madan, L., & Anand, K. (2014). Research Paper on Travelling Salesman Problem and its Solution Using Genetic Algorithm.
- Sadiq, S. (2012). The Traveling Salesman Problem: Optimizing Delivery Routes Using Genetic Algorithms.
- Sub, M., & Leopold, C. (fără an). Common Mistake in OpenMP and How to Avoid Them. A Collection of Best Practices.
- Winston, P. H. (2010). MIT Opencourseware. Lecture 13 Learning Genetic Algorithms.