

# Modele paralele și distribuite în calculul evolutiv

decembrie 2003

## 1 Eficiența calculului evolutiv

Algoritmii evolutivi sunt consumatori atât de resurse spațiale (datorită lucrului cu populații) cât și temporale (datorită caracterului iterativ). Eficientizarea unui algoritm evolutiv se poate realiza prin:

- optimizarea algoritmului prin reducerea complexității prelucrărilor și/sau accelerarea convergenței;
- optimizarea implementării prin distribuirea și/sau efectuarea în paralel a calculelor.

La optimizarea unui algoritm evolutiv trebuie să se stabilească care dintre prelucrări este cea mai costisitoare [4] pentru a interveni adecvat. Din punct de vedere al influenței asupra costului pot fi identificate două categorii prelucrări:

- *Evaluarea elementelor populației.* Este o prelucrare costisitoare în special în cazul problemelor pentru care funcția obiectiv are o formă complexă sau necesită efectuarea unei simulări costisitoare în timp. Eficientizarea presupune reducerea numărului de evaluări ceea ce se poate realiza prin: (i) determinarea volumului minim al populației sau prin (ii) modificarea operatorilor genetici astfel încât soluția să fie determinată într-un număr mai mic de generații.
- *Aplicarea operatorilor genetici.* Este costisitoare în cazul unor transformări genetice complicate care înglobează și elemente de adaptare sau auto-adaptare. De exemplu o selecție bazată pe ordonarea rangurilor este mai costisitoare decât o selecție de tip turneu.

Cele două tipuri de prelucrări sunt corelate în sensul că prin utilizarea unor operatori sofisticati (de cost ridicat) se poate reduce numărul de generații necesar până la găsirea soluției, reducându-se astfel costul evaluării. Problema dificilă este de a stabili echilibrul între cele două prelucrări consumatoare de timp. În absența unor rezultate teoretice general valabile se pot folosi sugestiile experimentale care indică [4]:

- se estimează costul procentual al evaluării în raport cu celelalte prelucrări;
- se justifică modificarea operatorilor genetici în vederea reducerii costului evaluării doar dacă procentul acestuia din urmă este cel puțin 60%.

## 2 Modele paralele ale calculului evolutiv

Paralelizarea algoritmilor evolutivi este posibilă datorită caracterului intrinsec paralel al prelucrărilor (se operează cu populații de indivizi între care există comunicare doar în anumite etape ale prelucrării). Pe de altă parte, se consideră [12] că AE posedă regularitate atât spațială (determinată de structura populației) cât și temporală (determinată de caracterul repetitiv al prelucrărilor) ceea ce îi face adecvați pentru paralelizare și distribuire.

Paralelizarea se poate realiza la nivelul [3]:

- algoritmului - modelul naiv;
- evaluării funcției de adecvare (obiectiv) - modelul master-slave;
- populației - modelul insular;
- indivizilor - modelul celular.

### 2.1 Modelul naiv

Algoritmul evolutiv este rulat în paralel pe mai multe procesoare fără a exista comunicare între ele. Rulările pot să difere între ele prin: populația inițială și /sau parametrii de control. La sfârșitul execuției tuturor algoritmilor se colectează rezultatele și se întocmește statistica finală. În ciuda simplității abordării, permite reducerea timpului de execuție atunci când se urmărește obținerea unor rezultate cu caracter statistic prin experimente numerice independente. Acest model însă nu aduce nimic nou în abordarea calculului evolutiv întrucât nu realizează o paralelizare la nivelul componentelor algoritmului. Specificul acestui model este ilustrat în fig. 1.

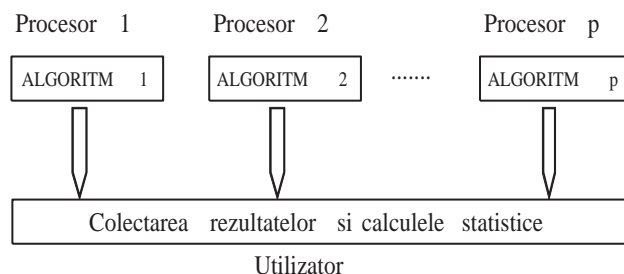


Figura 1: Modelul naiv de paralelizare

### 2.2 Modelul master-slave

Pornește de la ideea că prelucrarea cea mai costisitoare este cea a evaluării funcției de adecvare/obiectiv. Presupune existența unui proces master care execută algoritmul evolutiv propriu-zis stocând întreaga populație și efectuând prelucrările evolutive specifice (mutație, recombinare și selecție). Celelalte procese au doar rolul de a evalua funcția de adecvare pentru câte un element al populației, transmis de către procesul master. Comunicarea între procesul master și cele sclav este ilustrată în fig. 2.

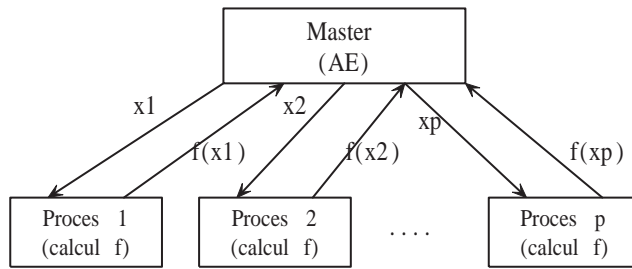


Figura 2: Modelul master-slave de paralelizare

Atunci când sunt mai mulți indivizi ( $m$ ) decât procese sclav ( $p$ ), procesul master trebuie să asigure repartizarea echilibrată a acestora. Acesta nu este un lucru dificil dacă costul evaluării este aproape același pentru fiecare individ (aspect valabil în cazul AG și SE dar mai puțin plauzibil în cazul PG).

Din acest motiv abordarea clasică, generațională (în care se așteaptă evaluarea tuturor indivizilor pentru a iniția procesul de selecție și pentru a trece la o nouă generație), nu este cea mai adecvată pentru că necesită o sincronizare a proceselor de evaluare. O variantă mai bună este cea asincronă (corespunde algoritmilor de tip "steady-state") în care după evaluarea fiecărui element acesta este introdus în populație înlocuindu-l pe cel mai slab.

Acest model poate fi implementat atât pe sisteme multiprocesor cu memorie partajată cât și pe sisteme cu memorie distribuită, inclusiv rețele de calculatoare pentru care trebuie însă asigurată comunicarea între procese.

Principalul avantaj îl reprezintă simplitatea (nu e necesar să se intervină în structura algoritmului) iar principalul dezavantaj este dat de faptul că nu întotdeauna evaluarea funcției obiectiv este prelucrarea cea mai costisitoare [4].

### 2.3 Modelul insular

Se bazează pe ideea descompunerii populației în subpopulații (numite "deme"). Fiecare subpopulație este supusă evoluției în maniera clasică un anumit număr de generații. Algoritmul corespunzător fiecărei subpopulații este executat pe câte un procesor. Până aici nu este nici o diferență față de modelul naiv. Particularitatea modelului insular constă în faptul că după un anumit număr de generații se inițiază un proces de *migrare* a unor indivizi dintr-o populație către alte populații *învecinate*. Rolul migrației este de a injecta diversitate în subpopulații în vederea creșterii capacității de explorare a acestora.

Proiectarea procesului de migrare necesită:

- Stabilirea pentru fiecare subpopulație a subpopulațiilor cu care comunică. Aceasta presupune existența unei topologii de migrare ce poate fi: de tip inel, grilă 2(3)-dimensională, graf arbitrar (ilustrată în fig. 3).
- Stabilirea intervalului de timp (numărul de generații) după care este inițiat procesul de migrare.
- Stabilirea numărului de emigranți (indivizi ce părăsesc subpopulația) și imigranți (indivizi ce pătrund în populație), a modului de selecție a acestora și a strategiei de înlocuire. Migrația

poate fi implementată prin mutare propriu-zisă (individul care emigrează nu mai rămâne în subpopulația din care a plecat) sau prin copiere (în subpopulație rămâne o copie a emigranului). Strategia de selecție a emigranților și asimilare a imigranților se poate baza pe ideea: emigrează cei mai buni indivizi iar imigranții îi vor înlocui pe cei mai slabi indivizi ai subpopulației. Se poate folosi însă și o strategie în care atât emigranții cât și elementele din populația destinație pe care le vor înlocui sunt selectate în mod aleator.

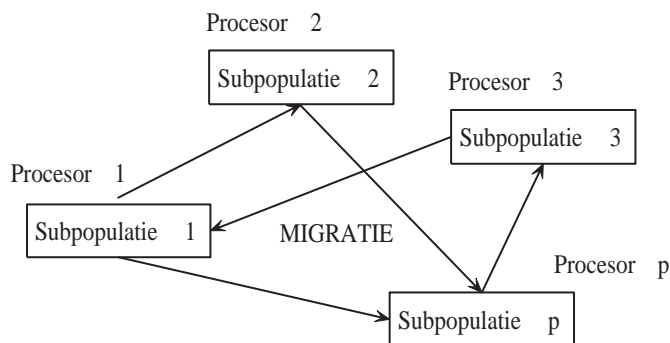


Figura 3: Model insular de paralelizare

Principalul avantaj îl reprezintă faptul că nu necesită comunicare intensivă între procese iar prin procesul de migrare se asigură conservarea diversității subpopulațiilor (din acest punct de vedere modelul insular este util și pentru evitarea convergenței premature în implementările seriale). Dezavantajul este dat de faptul că procesul de migrare necesită stabilirea unor noi strategii și parametri (frecvența de migrare și numărul migratorilor) pentru care nu se cunosc reguli de alegere fundamentate teoretic ci doar criterii euristice.

Poate fi implementat atât pe mașini paralele cu memorie partajată cât și pe sisteme distribuite (clustere (vezi GALOPPS) sau chiar grid (vezi DREAM)) datorită faptului că raportul dintre costul comunicării și cel al calculelor specifice fiecărei subpopulații este mic.

## 2.4 Modelul celular

Fiecare dintre elementele populației este amplasat într-un nod al unei grile (bi sau tri-dimensionale) și i se poate asocia un set de vecini (într-o manieră similară celei întâlnită la automatele celulare și la rețelele neuronale celulare). Variante de grile și vecinătăți sunt ilustrate în fig. 4.

Mecanismele de mutație, recombinare și selecție se aplică la nivelul vecinătății: pentru fiecare individ se selectează părinți pentru recombinare din cadrul vecinătății, se determină urmașul, acesta este supus mutației iar selecția este de tip turneu tot în cadrul vecinătății.

Fiecărui individ i se asociază un proces care va efectua prelucrările evolutive specifice. Procesele comunică local (doar cele vecine) însă comunicarea este intensivă fiind necesară la aplicarea fiecărui operator genetic.

Datorită ponderii relativ mari a comunicării între procese implementarea eficientă este cea pe mașini paralele de tip SIMD (MasPar sau Connection Machine) întrucât acestea implementează eficient în hard comunicările locale.

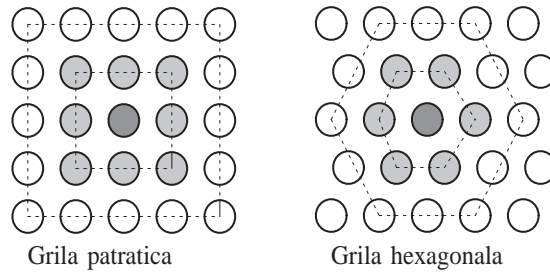


Figura 4: Tipuri de grile pentru modelul celular de paralelizare

## 2.5 Modele hibride

Există și modele ce nu se încadrează în nici unul dintre cele de mai sus îmbinând caracteristici ale mai multor modele de bază. Un exemplu îl reprezintă o structură *ierarhizată* în care la nivelul superior se folosește un model insular iar fiecare subpopulație este prelucrată prin intermediul unui model celular (suportul de implementare ar putea fi o rețea de mașini paralele). O altă variantă ierarhizată este aceea în care nivelul superior se bazează pe modelul insular iar cel inferior pe modelul master-slave (pentru fiecare subpopulație se calculează în paralel doar funcția de adecvare).

Tot o variantă mixtă o constituie modelele paralele *neuniforme* caracterizate prin faptul că fiecare subpopulație are reprezentare specifică, parametrii de control specifici și operatori evolutivi specifici. În acest caz este necesară și o strategie adecvată de migrare.

## 2.6 Ce model alegem ?

În alegerea unui model de paralelizare trebuie să se țină cont de:

- *Specificul problemei de rezolvat și a algoritmului evolutiv.* De exemplu dacă evaluarea funcției obiectiv domină clar din punct de vedere a costului, modelul master-slave ar putea fi cel indicat; dacă există diferențe mari între dimensiunile indivizilor - cum e cazul programării genetice - atunci modelul celular este mai dificil de implementat (dar nu imposibil după cum se arată în [6]).
- *Suportul de implementare avut la dispoziție.* Dacă se dispune doar de o rețea de calculatoare atunci se poate opta pentru modelul master-slave sau cel mult pentru cel insular.

## 3 Implementări orientate obiect ale algoritmilor evolutivi

Proiectarea de la zero a unei aplicații bazată pe calcul evolutiv este o sarcină care cere efort datorită faptului că nu se cunoaște apriori care sunt componentele (operatorii evolutivi) cele mai adecvate problemei concrete. De cele mai multe ori este necesară tatonarea și testarea mai multor variante, ceea ce poate presupune un efort mare de documentare în domeniul algoritmilor genetici și de programare. Astfel că a apărut necesitatea dezvoltării unor instrumente de proiectare facilă a algoritmilor evolutivi:

- Biblioteci de obiecte și metode evolutive (de exemplu GALIB și TEA) ce pot fi apelate din aplicațiile utilizator. Utilizatorului îi revine sarcina de a descrie problema de rezolvat (funcția obiectiv), de a stabili tipul de obiecte cu care operează și de a apela metodele specifice.
- Medii integrate de proiectare a unui algoritm evolutiv (vezi nivelul GUIDE de la DREAM) ce permite construirea prin intermediul unei interfețe grafice a algoritmului.

### 3.1 GALIB - Genetic Algorithms LIBrary

Este o bibliotecă C++ (<http://www.mit.edu/people/moriken/doc/galib>) de componente ale unui algoritm genetic dezvoltată la MIT . Folosește două clase de bază: una asociată individului (cromozom în cazul AG) și una asociată algoritmului genetic în întregime.

Clasei cromozom (**GAGenome**) îi sunt asociate metode corespunzătoare operatorilor genetici de mutație și recombinare, o metodă de inițializare și o funcție de evaluare. Clasei algoritm genetic îi sunt asociate metode ce asigură selecția și desfășurarea procesului iterativ de evoluție, inclusiv inițializarea populației.

Sunt implementate patru tipuri de reprezentări ale indivizilor: lista (**GAListGenome**), arbore (**GATreeGenome**), tablou (**GAArrayGenome**) și șir de biți (**GABinaryStringGenome**). Aceste clase sunt derivate din clasa **GAGenome** și din clasele asociate structurilor de date corespunzătoare (**GAList**, **GATree** etc.). Fiecare tip de reprezentare are asociate metode evolutive (mutație și recombinare) specifice. Pornind de la indivizi se construiește obiectul populație care are asociate metode de inițializare (ce folosesc inițializarea indivizilor), de evaluare (ce folosesc funcțiile de evaluare de la nivelul indivizilor) și de selecție. Obiectul populație are asociat un obiect care realizează scalarea funcției obiectiv în vederea determinării gradului de adecvare a fiecărui individ.

Sunt implementate patru variante de algoritmi genetici care diferă între ele prin modul în care se asigură înlocuirea indivizilor în cursul evoluției: *simplu* (variante generatională), *staționar* (strategia asincronă de înnoire a populației), *incremental* și cu *multi-populații* (corespunde modelului insular de paralelizare).

GALib permite derivarea propriilor clase și definirea unor noi funcții membru. Astfel, oferă suport pentru controlul gradului de aglomerare și formarea speciilor (utile în cazul optimizării multicriteriale). Deasemenea permite modificarea cu ușurință a unor metode (de exemplu cele corespunzătoare mutației și recombinării). Permite implementarea paralelă folosind PVM și modelul master-slave.

### 3.2 TEA - Toolbox for Evolutionary Algorithms

Este o bibliotecă C++ care permite proiectarea în manieră orientată obiect a AE standard și a celor particulari (non-standard). Caracteristicile bibliotecii sunt [5]: (i) proiectarea algoritmilor astfel încât să fie cât mai independenți față de reprezentare (astfel un același algoritm poate fi utilizat cu modificări minime pentru diferite probleme); (ii) suport pentru reprezentări nestandard (de exemplu, mixarea valorilor întregi, cu cele reale); (iii) posibilitatea schimbării interactive a operatorilor de variație (fără a necesita recompilarea); (iv) existența unor reprezentări predefinite; (v) conține și instrumente pentru proiectarea unor interfețe grafice simple.

Operatorii evolutivi (mutație, recombinare și inițializare) sunt definiți ca obiecte ce interacționează cu obiectele purtătoare de informație (indivizii și populațiile) pe care le modifică.

Operează la trei nivele de abstractizare: cromozom (**teaChromosome**), individ (**teaIndividual**) și populație (**teaPopulation**). De remarcat diferența dintre cromozom (genotip) și individ (fenotip)

- unui individ i se pot asocia mai mulți cromozomi). Fiecareia dintre clase îi sunt asociați operatori specifici, după cum urmează:

- **teaChromosome:** operatori de mutație, recombinare și inițializare (la nivelul codificării).
- **teaIndividual:** operatori de mutație, recombinare, evaluare a gradului de adecvare și verificare a restricțiilor.
- **teaPopulation:** operatori de evoluție a populației un anumit număr de generații, de selecție a părinților, de construire a populației de urmași, de selecție a supraviețuitorilor și de verificare a condiției de oprire a evoluției.

Implementează atât strategii evolutive cât și algoritmi genetici precum și modelul insular de paralelizare.

## 4 Implementări paralele și distribuite ale algoritmilor evolutivi

Implementările paralele/distribuite ale algoritmilor evolutivi pot fi clasificate în două categorii:

- *Orientate pe aplicație.* Sunt implementări ale unui algoritm evolutiv proiectat pentru a rezolva o problemă concretă.
- *Orientate pe algoritm.* Au ca scop furnizarea unor biblioteci sau medii de proiectare în manieră paralelă/distribuită a algoritmilor evolutivi. Exemple de astfel de sisteme sunt GALOPPS, DREAM și EVOLVE/G, descrise în cadrul acestei secțiuni.

### 4.1 Implementări orientate pe aplicație

În momentul aplicării unui algoritm evolutiv pentru o problemă concretă (optimizarea structurilor mari, determinarea configurațiilor clusterilor de energie minimă etc.) o problemă importantă este cea a costului implicat, atât din punct de vedere a spațiului (cu cât populația este mai mare cu atât puterea de explorare a algoritmului este mai mare) cât și al timpului de execuție (evaluarea indivizilor poate fi complicată iar numărul de generații până la convergență este mare). În aceste condiții o implementare paralelă sau una în care se folosesc resurse distribuite poate face diferența între un algoritm evolutiv ineficient și unul eficient. Câteva tipuri de aplicații în care calculul evolutiv paralel/distribuit a condus la obținerea de rezultate sunt prezentate în continuare.

**Rezolvarea problemelor de optimizare ce apar în proiectare.** În [2] este prezentat un algoritm evolutiv paralel de tip master-slave în variantă asincronă (după evaluarea fiecărui element nou generat acesta este inclus în procesul de selecție) în care procesele sclav au rolul de a evalua funcția obiectiv. Aplicația este realizată utilizând o rețea de 10 mașini SUN SPARC cu PVM și folosită pentru proiectarea reincărcării cu combustibil a unui reactor astfel încât să se optimizeze ciclul următor de ardere.

În [9], deși populația este împărțită în subpopulații, modelul paralel este tot cel de tip master-slave folosit în evaluarea elementelor nou generate. Implementarea este făcută pe o mașină paralelă cu 16 procesoare și aplicată la o problemă de optimizare multicriterială (abordare de tip Pareto) ce apare în proiectarea aripilor avioanelor supersonice.

**Prelucrarea imaginilor.** În [10] este proiectat un algoritm evolutiv bazat pe modelul celular destinat alinierii imaginilor digitale. Fiecare element al populației (în acest caz o imagine) este

afectat unui procesor. Structura celulară folosește o vecinătate de tip Moore, iar modificarea imaginii curente este bazată pe diferența dintre două imagini aparținând unor locații vecine.

**Regresie simbolică.** În [8] este prezentată o implementare bazată pe modelul master-slave a unei aplicații de programare genetică destinată regresiei simbolice (problema concretă este modelarea unei serii temporale). Sistemul pe care a fost testată implementarea este constituit din 14 procesoare iar transmiterea mesajelor între procese este realizată prin PVM.

## 4.2 GALOPPS - Genetic ALgorithm Optimized for Portability and Parallelism System

A fost dezvoltat de o echipă condusă de D.E. Goodman [7](<http://garage.cse.msu.edu/>) în cadrul grupului GARAGE. Este un ansamblu de funcții C care implementează componentele funcționale ale unui algoritm genetic având ca punct de plecare algoritmul genetic simplu. Permite operarea atât cu o singură populație cât și cu mai multe subpopulații. În varianta cu mai multe subpopulații permite atât simularea folosind un singur procesor cât și simularea folosind mai multe procesoare prin intermediul PVM-ului. Nu posedă interfață grafică, transmiterea parametrilor de intrare (ce specifică caracteristicile algoritmului) și preluarea rezultatelor realizându-se prin intermediul unor fișiere.

Gestiunea mai multor populații (modelul insular de paralelizare) se poate face în una dintre următoarele variante:

- *Simulare serială a modelului insular.* Fiecare dintre subpopulații este supusă evoluției un anumit număr de generații (care poate diferi de la o subpopulație la alta) după care starea ei este reținută într-un fișier. După ce toate subpopulațiile au evoluat numărul specificat de generații se efectuează migrația. Aceasta este controlată printr-un tabel "master" care stabilește pentru fiecare subpopulație (identificată printr-un număr de ordine) care sunt subpopulațiile vecine și câți indivizi vor fi transferați. Execuția algoritmului se bazează pe un singur proces. Lucrul cu subpopulații este motivat de asigurarea diversității populației.
- *Simularea paralelismului pe un singur procesor.* Pe o singură stație Unix (sau alt sistem ce permite multitasking) se rulează câte o copie a programului (fiecareia îi corespunde un proces) care prelucrează o populație (identificată printr-un număr de ordine). Toate procesele folosesc un fișier "master" comun care conține informații privind comunicarea între populații. Gestiunea acestui fișier comun necesită utilizarea unui protocol de blocare a accesului care permite evitarea situațiilor în care informația prelucrată de un proces este alterată de către altul. În cazul în care un proces nu are acces în mod repetat la fișierul "master" poate continua fără a efectua migrația (reținându-se însă această informație). Efectuarea prelucrărilor globale (cum este determinarea celui mai bun element din întreaga populație) se realizează pe baza unui fișier de stare cu informații statistice care este actualizat de către toate procesele activate. La începutul fiecărui ciclu (generațiile cuprinse între două migrații succesive) fiecare proces citește informațiile globale din fișierul de stare și le folosește în cadrul ciclului. La sfârșitul acestuia recitește informațiile statistice (ar putea fi deja modificate de către alte procese) și le actualizează. Astfel în decursul unui ciclu un proces nu "știe" ceea ce se întâmplă în cadrul celorlalte procese. Procesul corespunzător populației 0 (lansat primul) este cel care inițializează fișierul cu statistici. Prelucrarea populațiilor nu mai este sincronă (ca în cazul anterior) în sensul că nu se mai așteaptă parcurgerea numărului specific de generații de către fiecare populație înainte de a se trece la procesul de migrație. Acest mod de lucru poate fi util în faza de testare a implementării paralele.

- *Implementare paralelă propriu-zisă.* O primă variantă o reprezintă cea în care programul este rulat pe mai multe calculatoare conectate între ele. Pe fiecare dintre ele se simulează una sau mai multe subpopulații (ca în cazul simulării paralelismului pe un singur procesor). Procesoarele care cooperează în procesul de migrație partajează un fișier comun de tip "master" (ce conține informații privind subpopulațiile vecine și parametrii migrației).

O altă variantă o reprezintă PVM-GALOPPS, în care comunicarea între procese (inclusiv migrarea) este realizată prin mesaje prin intermediul PVM-ului. În acest caz se lucrează cu trei tipuri de procese: de tip master (citește fișierul de intrare și colectează statisticile globale), de tip monitor (asigură legătura dintre procesele de tip sclav și cel master și între procesele sclav vecine în timpul migrației) și de tip sclav (execută algoritmul genetic propriu-zis). Unei aplicații i se asociază un proces master și un anumit număr de procese monitor și sclav. Fiecare proces monitor creează un proces sclav și gestionează mesaje primite de la: procesul sclav (mesaj de terminare a unui ciclu), procese sclave vecine (mesaje de solicitare a emigranților) și de la procesul master (mesaj de terminare). Pe lângă aceste procese, pe fiecare mașină se creează un proces "verificator" care are rolul de a verifica dacă mașina este liberă.

Distribuirea proceselor se realizează după cum urmează. Procesele monitor vor fi distribuite uniform pe toate mașinile disponibile cu excepția celei de pe care a fost lansată aplicația (și pe care rulează procesul master). Fiecare proces monitor va crea procesul sclav asociat pe aceeași mașină pe care rulează.

### 4.3 DREAM - Distributed Resource Evolutionary Algorithm Machine

DREAM este un mediu destinat proiectării algoritmilor evolutivi și a sistemelor de agenți evolutivi într-o manieră distribuită ce are ca suport comunicarea peer-to-peer și Internet-ul [1]. Sistemul permite partajarea securizată a resurselor disponibile aparținând unui număr mare de calculatoare. Scalabilitatea sistemului permite abordarea unor probleme ce necesită putere mare de calcul sau un număr mare de agenți ce concurează și cooperează în vederea găsirii soluției unei probleme complexe. El a fost elaborat în cadrul unui proiect european demarat în 1999 (<http://www.dcs.napier.ac.uk/benp/dream/dream.htm>) al cărui scop era dezvoltarea unei infrastructuri tehnologice și software cu caracter deschis și scalabil care să ofere suport pentru distribuirea automată a algoritmilor evolutivi.

**Arhitectura sistemului DREAM.** Sistemul este constituit din 5 module, fiecare dintre ele având o interfață, un nivel de interacțiune și de abstractizare specifice.

Sistemul oferă puncte de intrare pentru cinci tipuri de utilizatori:

- *Utilizator de tip A.* Nu va efectua experimente proprii ci doar va oferi resurse disponibile sau va monitoriza experimentele altora. Utilizatorii de acest tip interacționează cu sistemul numai prin intermediul consolei.
- *Utilizator de tip B.* Este destinat utilizatorilor ce vor să proiecteze rapid un sistem fără să scrie linii de cod. Interacțiunea cu sistemul se face prin intermediul nivelului GUIDE ce permite definirea algoritmilor evolutivi distribuiți folosind o interfață grafică. Interfața dintre nivelele GUIDE și EASEA se realizează prin intermediul limbajului EASEA. EASEA (EAsy Specification of Evolutionary Algorithms - <http://www-rocq.inria.fr/EASEA/>) este un limbaj dedicat utilizării bibliotecilor de algoritmi evolutivi construit în scopul facilitării construirii unei aplicații evolutive fără a fi necesară cunoașterea unor detalii privind operatorii genetici sau programarea orientată obiect.

- *Utilizator de tip C.* Implementează un algoritm evolutiv distribuit folosind EASEA. Descrierea în EASEA este tradusă în Java prin intermediul unui compilator asociat sistemului. Codul produs utilizează obiectele și metodele din biblioteca JEO (Java Evolutionary Object).
- *Utilizator de tip D.* Programează direct în Java folosind biblioteca JEO care furnizează nu numai obiecte și metode pentru calculul evolutiv ci și interfață (API) la nucleul DRM (Distributed Resource Machine).
- *Utilizator de tip E.* Programează utilizând direct DRM API ceea ce îi permite efectuarea și a altor prelucrări distribuite în afara celor evolutive.

**Nivelul GUIDE.** Permite "construirea" unui algoritm evolutiv distribuit prin selecția componentelor sale din cadrul unor meniuri:

- *Meniul de specificare a problemei.* Permite alegerea componentelor ce depind de problemă: codificarea datelor (structura genotipului), inițializarea populației, operatorii de variație (mutație și recombinare) și evaluarea indivizilor (calculul gradului de adecvare). Elementele specifice problemei sunt descrise în limbajul EASEA având o sintaxă similară limbajelor C/C++/Java care însă ascunde structura complicată a claselor/obiectelor ce permit gestiunea populațiilor.
- *Meniul asociat modulului evolutiv.* Modulul evolutiv implementează structura generală repetitivă a algoritmilor evolutivi și procesele de selecție (a părinților și a supraviețuitorilor). Oferă un set generic de primitive pe baza cărora se poate construi o varietate de algoritmi evolutivi incluzând variantele: algoritmi genetici de tip generațional, algoritmi genetici de tip staționar, strategii evolutive de tip  $(\mu, \lambda)$  (supraviețuitorii se selectează doar dintre urmași), strategii evolutive de tip  $(\mu + \lambda)$  (supraviețuitorii se selectează atât dintre urmași cât și dintre părinți), programare evolutivă.
- *Meniul de control a distribuirii calculului.* Permite stabilirea modului în care se desfășoară procesul de migrație a indivizilor între subpopulații.
- *Meniul de monitorizare a rulării.* Prin intermediul acestuia utilizatorii lansează în execuție algoritmul și vizualizează rezultatele intermediare și finale.

**Nivelul JEO.** JEO (Java Evolutionary Objects) este un cadru pentru construirea algoritmilor evolutivi. Furnizează un punct de intrare în DREAM utilizatorilor de tip D. JEO produce o specificare a unui experiment evolutiv ce poate fi rulat în manieră distribuită utilizând modulul DRM. Poate fi văzut ca o punte de legătură între EASEA și modulele DRM care ascunde detaliile fizice ale DRM cum ar fi aspecte legate de protocoalele de comunicație și de firele de execuție, permițând utilizatorului să se concentreze asupra conceptelor calculului evolutiv. Utilizatorul acestui nivel va lucra direct cu clasele Java pentru a avea control direct asupra experimentului. În procedura de construire a experimentului se efectuează:

- utilizatorul crează o clasă Java pentru a crea obiectele ce vor fi plasate în fiecare subpopulație;
- pentru fiecare task ce va fi efectuat asupra unei subpopulații utilizatorul trebuie să creeze un obiect specific; fiecare dintre aceste obiecte trebuie să folosească interfața JEO corespunzătoare.

JEO oferă o clasă generală pentru *subpopulații*. Fiecărei subpopulații îi este asociat un mediu care conține pe lângă indivizii corespunzători și obiectele necesare procesului de evoluție. Mediile interacționează între ele prin intermediul unor obiecte de tip *evaluator* care evaluează simultan mediile.

După inițializarea tuturor subpopulațiilor (prin intermediul obiectelor de tip *inițializator*) se repetă următoarele prelucrări:

- Se verifică criteriul de oprire și calculează mărimile statistice asociate algoritmului (de către obiecte de tip *terminator*). Sunt implementate mai multe criterii de oprire precum și diferite statistici on-line și cumulate. Rezultatele calculelor statistice sunt transmise consolei DRM, meniului de monitorizare a experimentului din GUIDE sau pur și simplu afișate pe ecran (atunci când JEO este utilizat independent).
- Fiecare subpopulație este supusă transformărilor genetice (cele de variație și selecție).
- Se apelează evaluatorul pentru fiecare sub-populație. Acest pas poate fi o simplă evaluare a funcției obiectiv (în cazul unei probleme de optimizare) sau efectuarea unei simulări în cazul sistemelor de tip "artificial life". La sfârșitul acestui pas toți indivizii au asociat un grad de adecvare.
- Se efectuează migrația între subpopulații prin intermediul a două obiecte: *imigrator* (preia dintr-o zonă tampon indivizii "sosiți" de la alte subpopulații și decide cum vor fi incluși în subpopulația curentă) și *emigrator* (acesta selectează indivizii ce vor emigra și destinațiile acestora).
- Se elimină cei mai puțin adecvați indivizi, cei care rămân formând generația supraviețuitoare.

Experimentul poate fi transmis modulului DRM folosind consola DREAM sau linia de comanda DRM după care modulul DRM distribuie experimentul prin intermediul mașinii DREAM. Mecanismul de distribuire este complet transparent utilizatorului. Utilizatorul identifică experimentul printr-un nume și în cadrul acestuia identifică fiecare subpopulație printr-un nume. Evoluția în paralel a subpopulațiilor este efectuată în manieră asincronă prin utilizarea unor zone tampon în care se amplasează indivizii care migrează. Există un fir de execuție care primește imigranții și îi plasează în zona tampon de unde sunt preluați de către firul de execuție corespunzător algoritmului.

Proiectul DREAM și JEO au fost dezvoltate utilizând jdk 1.3. Clasele și interfețele sunt organizate în pachete. Cel principal, `dream.evolution`, include clasele și interfețele asociate subpopulațiilor. Alte pachete sunt: `dream.evolution.genomes` (include clasele și interfețele necesare construirii indivizilor având structură liniară, de arbore sau de graf) și `dream.evolution.operators` (include operatorii de variație și de selecție).

**Nivelul DRM.**(Distributed Resource Machine) Este reprezentat de un ansamblu de mașini conectate prin Internet care formează un mediu pentru aplicații distribuite. La acest nivel de abstractizare nu mai are importanță faptul că aplicația este din domeniul calculului evolutiv.

Într-un mediu bazat pe o singură mașină o aplicație este constituită din unul sau mai multe fire de execuție controlate de un sistem de operare, care asignează resursele și rezolvă problemele de securitate.

Extinderea la un mediu distribuit bazat pe un număr mare de mașini nu mai permite implementarea în aceeași manieră a aspectelor de control și securitate. O caracteristică cheie a modelului DREAM este faptul că aplicația este un set de agenți autonomi care cooperează. În cazul algoritmilor evolutivi fiecare agent corespunde unei subpopulații. Un agent este similar unui fir de execuție:

execuția aplicației este asigurată prin activarea unuia sau mai multor agenți care comunică între ei. DRM controlează agenții, resursele la care au acces, aspectele de securitate etc. Agenții au libertate mare putând să-și schimbe locația. Una dintre problemele sistemelor distribuite este posibilitatea apariției unor disfuncționalități ale canalelor de comunicație dintre noduri sau chiar ale nodurilor. Aceste probleme restrâng aria de aplicabilitate a sistemelor distribuite la *probleme robuste* (ce nu sunt sensibile la pierderea unei submulțimi de agenți) și care nu necesită comunicare masivă între agenți. Calculul evolutiv satisface aceste cerințe.

Implementarea DRM se caracterizează prin faptul că nu există servere centrale. În felul acesta se maximizează scalabilitatea și robustețea însă chiar și menținerea conectivității rețelei devine o problemă dificilă. Această problemă precum și cea a distribuirii informației prin DRM este rezolvată utilizând *protocoale epidemice*. În DREAM un astfel de protocol acționează după cum urmează: fiecare nod în DRM are asociată o bază de date incompletă care conține intrări în alte noduri. Aceste intrări conțin informații privind resursele disponibile și agenții activi în fiecare nod. Fiecare nod alege aleator un nod din baza sa de date cu care schimbă informații. Dacă dimensiunea bazei de date depășește o limită se selectează aleator câteva noduri care se elimină.

**Consola DREAM.** Este instrumentul ce permite gestiunea unui calculator conectat la DRM. Permite vizualizarea componentelor pornind de la baza de date gestionată de către DRM. Componentele vizualizate sunt: noduri, experimente, utilizatori, subpopulații și agenți. Consola ajută utilizatorul pe parcursul execuției unui experiment permițând: inspecția componentelor, startarea și controlul experimentelor, vizualizarea și analiza rezultatelor.

#### 4.4 EVOLVE/G

Este un sistem destinat construirii aplicațiilor de calcul evolutiv pe grid putându-se astfel beneficia de un ansamblu de resurse răspândite într-o zonă largă [11]. Necesitatea unui astfel de sistem derivă din faptul că multe aplicații de optimizare ce pot fi abordate prin calcul evolutiv necesită multe resurse (atât ca spațiu cât și ca timp). Exemple de astfel de aplicații sunt: probleme de optimizare a structurilor, de planificare a activităților, de determinare a structurii proteinelor.

În proiectarea sistemului EVOLVE/G s-a ținut cont că o aplicație orientată pe grid folosește eficient resursele acestuia dacă satisface următoarele caracteristici:

- Prelucrarea poate fi descompusă în mai subprelucrări (populația se descompune în mai multe subpopulații).
- Subprelucrările sunt slab corelate între ele și pot fi executate în paralel (subpopulațiile evoluează în paralel și sunt corelate doar prin elementele ce migrează).
- Comunicarea între subprelucrări este rară (migrarea are loc doar la anumite intervale de timp, după ce fiecare dintre subpopulații a evoluat un anumit număr de generații).
- Prelucrarea poate continua (eventualul cost al continuării este redus) dacă este eliminat un nod (pierderea unei subpopulații nu alterează drastic funcționalitatea aplicației).
- Prelucrarea poate asimila beneficiile obținute prin introducerea unor noi noduri în sistem (se pot adăuga noi subpopulații care permit creșterea eficacității algoritmului).

EVOLVE/G utilizează Globus 1.1.4 și este realizat în C și Java. O aplicație evolutivă proiectată cu EVOLVE/G constă din *agenți* (agents) și *efectori* (workers) a căror comportare o definește proiectantul aplicației. Rolul agenților este de a verifica efectorii la momente fixate de timp, colectează

informații privind progresul prelucrării și starea nodurilor și transmite comenzi efectorilor. Fiecare efector realizează prelucrări specifice algoritmului evolutiv fără să fie influențat de acțiunile altui efector. Structura generală a unei aplicații EVOLVE/G este ilustrată în fig. 5.

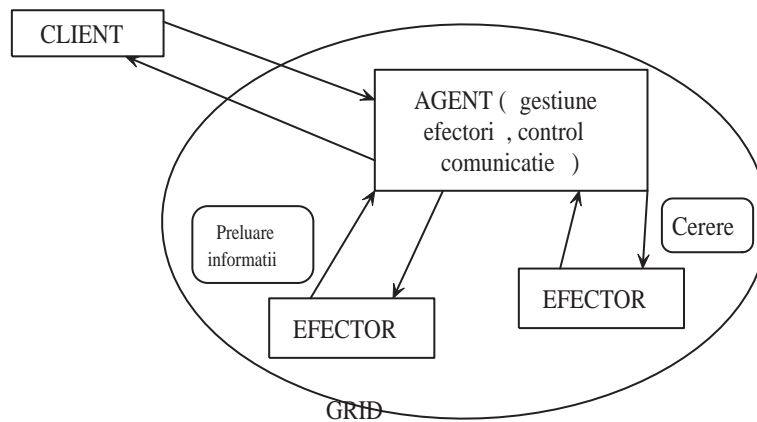


Figura 5: Structura generală a unei aplicații EVOLVE/G

Dacă în noduri se află clustere atunci structura este ierarhică: există un super-agent iar în fiecare nod există câte un agent (cu rol de efector din perspectiva super-agentului) și mai mulți efectori (corespunzători mașinilor care constituie clusterul). Aceasta corespunde modelului hibrid de paralelizare. În [11] este utilizat EVOLVE/G pentru dezvoltarea unei aplicații evolutive ce constă în evoluția în paralel a mai multor algoritmi de tip "simulated annealing" (executați de către efectori) care comunică între ei doi câte doi prin recombinarea soluțiilor la care ajung după un anumit număr de iterații (recombinarea este efectuată de către agenți).

## 5 Concluzii

Creșterea eficienței calculului evolutiv se poate realiza prin distribuirea și efectuarea în paralel a prelucrărilor. Acest lucru este posibil datorită faptului că algoritmi evolutivi prezintă o regularitate spațială și temporală. Până la ora actuală au fost dezvoltate mai multe modele de paralelizare a algoritmilor evolutivi (master-slave, insular, celular) precum și variante hibride (ierarhice) ale acestora.

În funcție de specificul modelului implementarea se poate realiza și în manieră distribuită folosind: rețele de calculatoare conectate în manieră clasică (modelul master-slave, modelul insular), clustere de calculatoare (modelul master-slave, modelul insular), structuri de tip grid (modelul insular, modele ierarhice).

Eficiențizarea activității de proiectare a algoritmilor evolutivi (în manieră clasică, paralelă sau distribuită) se poate asigura printr-o abordare orientată obiect și folosirea unor biblioteci de clase corespunzătoare componentelor algoritmilor evolutivi. Abordarea OOP în calculul evolutiv este facilitată de particularitățile algoritmilor evolutivi ce permit extrapolarea unei prelucrări de la individ la populație. Se pot identifica relativ ușor clasele de bază, cele derivate și metodele asociate. Până recent bibliotecile aferente calculului evolutiv erau orientate înspre una dintre cele patru direcții: AG, SE, PE sau PG. Tendința actuală este de a dezvolta biblioteci cu caracter general care să includă cât mai multe modalități de reprezentare și operatori evolutivi.

## Referințe

- [1] M.G. Arenas, P. Collet, A.E. Eiben, M. Jelasity, J.J. Merelo, B. Paechter, M. Preuß, M. Schonauer; A Framework for Distributed Evolutionary Algorithms, PPSN VII, Granada, 2002 (document online <http://www.dcs.napier.ac.uk/benp/dream/dreampaper6.pdf>).
- [2] T.Bäck, T. Beielstein, B. Naujoks, J. Heistermann; Evolutionary Algorithms for the Optimization of Simulation Models using PVM, in J. Dongarra et al. (eds), EuroPVM 95, 277-282.
- [3] E. Cantú-Paz; A Survey of Parallel Genetic Algorithms, IlliGAL Report, no. 97003, 1997.
- [4] P.Collet, J. Louchet, E. Lutton; Issues on the Optimisation of Evolutionary Algorithms Code, Congress on Evolutionary Computation, Honolulu, 2002 (document online <http://www.dcs.napier.ac.uk/benp/dream/dreampaper3.ps>).
- [5] M. Emmerich, R. Hosenberg; TEA - A C++ Library for the Design of Evolutionary Algorithms, (document online <http://ls11-www.cs.uni-dortmund.de/tea/teaDoc.html>).
- [6] G. Folino, C. Pizzuti, G. Spezzano; CAGE: A Tool for Parallel Genetic Programming Applications, J. Miller et al. (eds.): EuroGP 2001, Springer Verlag, LNCS 2038, 64-73, 2001.
- [7] E.D. Goodman; An Introduction to GALOPPS, Technical Report 95-06-01, Genetic Algorithms Research and Applications Group Michigan State University.
- [8] M. Oussaidène, B. Chopard, M. Tomassini; Programmation évolutionniste parallèle, RenPar'7, Actes de Tes Rencontres Francophones du Parallélisme, 1995.
- [9] A. Quagliarella, A. Vicini; Sub-population policies for a parallel multiobjective genetic algorithm with application to wing design, Proc. of Intern. Conf. on Systems, Man & Cybern., 3142-3147, 1998.
- [10] M. Salomon; Parallélisation de l'volution différentielle pour le recalage rigide d'images médicales volumiques, RenPar000, Besançon, june 2000.
- [11] Y. Tanimura, T. Hiroyasu, M. Miki, K. Aoi; The System for Evolutionary Computing on the Computational Grid, Fourteenth IASTED International Conference Parallel and Distributed Computing and Systems PDCS 2002 MIT, Cambridge, 2002
- [12] M. Tomassini; Parallel and Distributed Evolutionary Algorithms: A Review, In Evolutionary Algorithms in Engineering and Computer Science, J. Wiley and Sons, Chichester, 1999, K. Miettinen, M. Mkel, P. Neittaanmki and J. Periaux (editors), 113-133, 1999 (document online <http://lslwww.epfl.ch/marco/parea.ps.gz>).
- [13] M. Wall, GALib: A C++ Library of Genetic Algorithm Components, <http://www.mit.edu/people/moriken/doc/galib>.