ALGORITMI PARALELI ȘI DISTRIBUIȚI
Tema #1 Paralelizarea unui algoritm genetic

Responsabili: Radu-Ioan Ciobanu, Silviu-George Pantelimon, Florin Mihalache, Radu-Cătălin Nicolescu, Sandra Dascălu, Ioana Marin, Sergiu Toader

Termen de predare: 21-11-2021 23:59
Ultima modificare: 14-11-2021 18:40

Cuprins

Cerință

Pornind de la implementarea secvențială, se cere să se scrie un program paralel care rulează un algoritm genetic pentru a rezolva problema rucsacului. Programul va fi scris în C/C++ și va fi paralelizat utilizând PThreads. Implementarea temei trebuie să conducă la aceeași soluție ca implementarea secvențială și să scaleze cu numărul de fire de execuție.

Algoritmi genetici

În domeniul calculatoarelor, un algoritm genetic este o tehnică de căutare și optimizare inspirată de procesul selecției naturale și de teoria evoluționistă a lui Darwin. Un astfel de algoritm este folosit pentru a genera soluții pentru probleme de optimizare și de căutare prin intermediul unor operații inspirate din biologie, precum selecție, crossover sau mutație.

Într-un algoritm genetic, o mulțime de soluții potențiale ale unei probleme este reprezentată ca o **generație** formată din **indivizi**. La fiecare etapă, o generație curentă evoluează către o generație viitoare pe baza principiul evoluționist "survival of the fittest". În mod tradițional, un individ este reprezentat ca un șir de **cromozomi**, adică valori de 0 sau 1 (deși alte codificări sunt de asemenea posibile).

Evoluția pornește de obicei de la o populație de indivizi generați aleator și este un proces iterativ. La fiecare generație, calitatea fiecărui individ este evaluată prin intermediul unei **funcții de fitness**. Indivizii cu valorile cele mai mari ale funcției de fitness sunt selectați din populația curentă pentru a pune bazele următoarei generații. Indivizii din noua generație rezultă deci din indivizii cei mai buni ai generației curente, modificați prin crossover sau mutație.

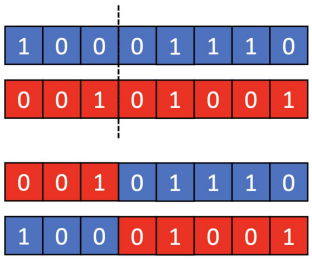
Un algoritm genetic are astfel nevoie de două elemente importante: o reprezentare a domeniului soluției și o funcție de fitness pentru evaluare. Odată ce aceste două elemente sunt stabilite, se generează o populație inițială de soluții, care se îmbunătățește succesiv prin aplicarea repetată de operatori genetici.

Selecția

Selecția celor mai bun indivizi dintr-o generație se realizează cu ajutorul unei funcții de fitness care reprezintă calitatea fiecărui individ (în alte cuvinte, calitatea fiecărei soluții candidate). Odată ce fiecare individ a fost evaluat, aceștia sunt sortați în funcție de valoarea funcției de fitness, o parte din ei fiind selecționați pentru a pune bazele următoarei generații. În unele cazuri, o parte din cei mai buni indivizi sunt păstrați direct și în următoarea generație, fără a fi modificați de către operatorii genetici.

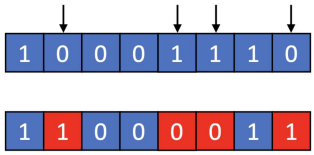
Crossover

Crossover (cunoscut de asemenea ca și recombinare) este un operator genetic folosit pentru a combina informația genetică a doi părinți din generația curentă pentru a genera noi copii pentru următoarea generație. Cel mai cunoscut mod de a realiza această operație este crossover într-un punct ("one-point crossover"). În acest caz, se alege un punct aleator la ambii părinți, și cromozomii din dreapta acestui punct sunt schimbați între părinți. Astfel, din doi părinți vor rezulta doi copii, așa cum se poate observa în imaginea de mai jos.



Mutația

Mutația este un operator genetic folosit pentru a oferi diversitate genetică de la o generație la alta într-un algoritm genetic. Prin acest operator, se modifică unul sau mai mulți cromozomi dintr-un individ. Cea mai simplă metodă de mutație este mutația de tip șir de biți ("bit string"), unde se aleg aleator cromozomi ai unui individ și se inversează valorile lor (presupunând o reprezentare binară a cromozomilor), așa cum se poate observa în imaginea de mai jos.



Terminarea

Un algoritm genetic se oprește, în general, atunci când se ajunge la un număr maxim de generații sau la o soluție suficient de bună a problemei care trebuie rezolvată.

Problema rucsacului

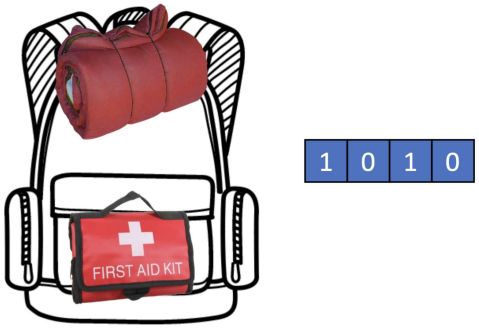
În cadrul acestei teme, vom încerca să rezolvăm problema rucsacului (varianta 0-1) folosind algoritmi genetici. Acesta este un exemplu clasic de problemă de optimizare care se poate reprezenta facil ca un algoritm genetic. În problema rucsacului, se dă o mulțime de obiecte definite printr-o greutate și un profit, precum și un rucsac care poate susține o greutate maximă dată. Scopul problemei este de a găsi combinația de obiecte care pot fi adăugate în rucsac fără a depăși greutatea maximă a acestuia, și care aduc profit maxim. În cazul variantei 0-1 a acestei probleme, se presupune că putem plasa maxim un singur obiect de un anumit tip în rucsac.

Problema rucsacului cu algoritmi genetici

În continuare, prezentăm un exemplu de rezolvare a problemei rucsacului (varianta 0-1) cu algoritmi genetici.

Reprezentarea unui individ și generația inițială

Cea mai simplă reprezentare a unui individ în cadrul problemei rucsacului este cea din imaginea de mai jos.



Astfel, se poate observa că un individ este format dintr-un număr de cromozomi egal cu numărul de obiecte care pot fi adăugate în rucsac, cromozomul i având valoarea 1 dacă obiectul cu indexul i se află în rucsac, sau 0 altfel.

Setul inițial de indivizi poate fi generat aleator, dar o variantă mai eficientă ar putea presupune că acesta este format din toți indivizii care conțin câte un singur obiect în rucsac. Astfel, dacă exista N obiecte care pot fi puse în rucsac, vor exista N indivizi în generația inițială, fiecare din ei având un singur cromozom cu valoarea 1, și restul 0.

Selecția

Funcția de fitness se poate calcula ca suma profiturilor obiectelor din rucsac, adică a obiectelor corespunzătoare cromozomilor cu valoarea 1 ai unui individ. Dacă greutatea obiectelor din sac depășește greutatea maximă permisă, funcția de fitness va avea valoarea 0.

În cadrul implementării din această temă, primii 30% din indivizii din generația curentă (din punct de vedere al funcției de fitness) vor fi păstrați și la generația următoare (această operație se numește selecția elitei).

Crossover

În cadrul rezolvării problemei rucsacului din această temă, vom folosi crossover într-un punct pentru a genera câte doi copii din câte doi părinți.

Într-o implementare ideală, punctul de crossover va fi ales aleator, putând fi astfel diferit de la o generație la alta (sau de la o rulare la alta). Pentru a avea totuși determinism în cadrul acestei teme, punctul de crossover p se alege folosind formula următoare:

$$p = 1 + index\_generatie \ \% \ dimensiune\_individ \tag{1}$$

Mutația

Pentru mutație, implementarea din această temă utilizează două variante de mutație bit string. Prima variantă se va aplica pe primii 20% din indivizii generației curente (ordonați după valorile funcției de fitness) și presupune inversarea primilor 40% din cromozomii unui individ cu un pas s pentru indivizii cu index par, respectiv inversarea ultimilor 80% din cromozomii unui individ pentru indivizii cu index impar, cu același pas. Pasul s este calculat astfel:

$$s = 1 + index\_generatie \ \% \ (dimensiune\_individ - 2) \tag{2}$$

A doua variantă de mutație bit string se aplică pe următorii 20% din indivizii generației curente și presupune inversarea tuturor cromozomilor unui individ, cu același pas s descris mai sus.

Terminarea

Pentru a păstra determinismul rulării algoritmului, programul se va termina după un număr fix de generații dat ca input la rulare.

Atenție! Algoritmii genetici sunt metaeuristici de optimizare, ceea ce înseamnă că nu oferă garanția că vor găsi soluția optimă. Elementele aleatoare din pașii de mutație și crossover au rolul de a lărgi orizontul de soluții posibile și de a oferi varietate într-o generație. Din acest motiv, este posibil ca eliminarea acestui nedeterminism din soluția propusă în temă să nu ofere întotdeauna cea mai bună soluție.

Implementarea secvențială

În repository-ul temei, găsiți fișierele sursă care conțin implementarea secvențială. Programul rezultat în urma compilării cu Makefile-ul aferent primește următorii parametri la rulare:

```
./tema1 <fisier_intrare> <numar_generatii>
```

Se poate deci observa că o rulare a programului secvențial va rula algoritmul genetic descris mai sus pe baza unui fișier de intrare cu următorul format:

```
nr_obiecte capacitate_rucsac
obiect1_profit obiect1_greutate
...
obiectN_profit obiectN_greutate
```

Prima linie specifică numărul de obiecte N care pot fi introduse în rucsac, precum și greutatea maximă permisă de către acesta. Mai departe, urmează N linii care conțin profitul și greutatea fiecărui obiect.

Atenție! Programul acceptă doar un număr de obiecte multiplu de 10.

Să presupunem că avem următorul fișier de intrare:

```
10 17
7 3
4 3
2 4
2 1
9 1
4 2
5 1
1 1
5 3
3 1
```

În acest caz, generația inițială va arăta ca în figura de mai jos.

1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

Pașii de funcționare ai implementării secvențiale pe care o găsiți în repository-ul temei sunt următorii:

1. se citesc datele de intrare ale programului (funcția `read_input` din fișierul *genetic_algorithm.c*)
2. se rulează algoritmul genetic (funcția `run_genetic_algorithm` din fișierul *genetic_algorithm.c*); întâi se creează generația inițială, apoi, pentru fiecare generație în parte, se fac următorii pași:

(a) se calculează fitness-ul fiecărui individ din generație (prin funcția `compute_fitness_function`) și apoi se sortează indivizii descrescător după valorile de fitness

(b) se păstrează primii 30% din indivizi pentru generația următoare (selecția elitei)

(c) se aplică prima variantă de mutație bit string pe primii 20% din indivizii din generație (prin funcția `mutate_bit_string_1`)

(d) se aplică a doua variantă de mutație bit string pe următorii 20% din indivizii din generație (prin funcția `mutate_bit_string_2`)

(e) se aplică crossover într-un punct pe primii 30% din indivizii din generație (prin intermediul funcției `crossover`); dacă numărul de părinți este impar, ultimul individ din generație este păstrat pentru următoarea generație (pentru variație în soluție)

(f) se suprascrie generația curentă cu noua generație (în alte cuvinte, are loc tranziția la următoarea generație)

3. se repetă acest proces până când s-a ajuns la ultima generație, conform datelor de intrare
4. se sortează (după fitness) indivizii din ultima generație, cel mai bun dintre ei fiind considerat cea mai bună soluție obținută.

Paralelizarea calculelor

Cerința principală a temei este paralelizarea calculelor pornind de la implementarea secvențială, astfel încât progamul **să scaleze** și **să obțină aceleași rezultate ca la implementarea secvențială**. Sursele voastre trebuie să poată fi compilate într-un binar numit *tema1_par* care are același comportament ca cel secvențial și care va fi rulat în felul următor (unde P este numărul de thread-uri):

```
./tema1_par <fisier_intrare> <numar_generatii> <P>
```

În implementare, nu se recomandă crearea și join-ul de thread-uri de mai multe ori (temele care pornesc și opresc thread-urile de mai multe ori vor primi o **depunctare de 30% din nota finală)**. Astfel, veți crea și porni cele P thread-uri la început, și le veți folosi pentru toate operațiile paralele, realizând sincronizare folosind primitivele învățate la laborator. Toate cele P thread-uri vor contribui la rularea algoritmului.

Pentru testarea corectitudinii implementării, vă recomandăm să folosiți implementarea secvențială ca etalon. Astfel, puteți rula programul secvențial pe un set de fișiere, iar apoi să rulați programul paralel, cu valori diferite pentru P, pe același set de fișiere. Puteți apoi să folosiți utilitarul *diff* pentru a verifica dacă rezultatul rulării paralele este același cu cel al rulării secvențiale.

Notare

Tema se va trimite și testa automat la această adresă și se va încărca de asemenea și pe Moodle. Se va încărca o arhivă Zip care, pe lângă fișierele sursă, va trebui să conțină următoarele două fișiere **în rădăcina arhivei**:

- *Makefile* - cu directiva *build* care compilează tema voastră (**fără flag-uri de optimizare**) și generează un executabil numit *tema1_par* aflat în rădăcina arhivei
- *README* - fișier text în care să se descrie pe scurt implementarea temei.

Punctajul este divizat după cum urmează:

- **50p** - scalabilitatea soluției

- **30p** - corectitudinea implementării[1]
- **20p** - claritatea codului și a explicațiilor din README.

Nerespectarea următoarelor cerințe va duce la depunctări:

- **-100p** - pseudo-sincronizarea firelor de execuție prin funcții cum ar fi `sleep`
- **-100p** - utilizarea altor implementări de thread-uri în afară de Pthreads (cum ar fi *std::thread* din C++11)
- **-30p** - crearea și oprirea de thread-uri în mod repetat (pentru a nu lua această depunctare, trebuie să creați un singur set de P thread-uri la început și să le folosiți pentru toate calculele)
- **-10p** - utilizarea variabilelor globale (soluția pentru a evita variabile globale este să trimiteți variabile și referințe la variabile prin argumentele funcției pe care o dați la crearea firelor de execuție).

Testare

Pentru a vă putea testa tema, găsiți în repository-ul temei un set de fișiere de intrare de test, precum și un script Bash (numit *test.sh*) pe care îl puteți rula pentru a vă verifica implementarea paralelă. Acest script va fi folosit și pentru testarea automată, singura diferență fiind că atunci se vor rula niște teste în plus pe fișiere de intrare și valori ale lui P adiționale[2].

Pentru a putea rula scriptul așa cum este, trebuie să aveți următoatoarea structură de fișiere:

```
$ tree
.
+-- skel
|   +-- Makefile
|   +-- genetic_algorithm.c
|   +-- genetic_algorithm.h
|   +-- individual.h
|   +-- inputs
|   |   +-- in0
|   |   +-- in1
|   |   +-- in2
|   |   +-- in3
|   |   +-- in4
|   +-- sack_object.h
|   +-- tema1.c
+-- sol
|   +-- Makefile
|   +-- README
|   +-- [...] (sursele voastre)
+-- test.sh
```

La rulare, scriptul execută următorii pași:

1. compilează și rulează implementarea secvențială pe patru fișiere de intrare
2. compilează și rulează implementarea paralelă pe cele patru fișiere de intrare pentru 2, 3 și 4 thread-uri
3. se compară rezultatele obținute în urma rulărilor paralele și cele secvențiale

[1] Acest punctaj este condiționat de scalabilitate. O soluție secvențială, deși funcționează corect și dă rezultate bune, nu se va puncta.

[2] Atenție! Nota obținută în urma rulării automate poate fi scăzută pe baza elementelor de depunctare descrise mai sus.

4. se calculează accelerația pentru primul set de fișiere de intrare (de la varianta secvențială la 2, respectiv 4 thread-uri, și de la 2 la 4 thread-uri)
5. se calculează punctajul final din cele 80 de puncte alocate testelor automate (20 de puncte fiind rezervate pentru claritatea codului și a explicațiilor, așa cum se specifică mai sus).

Scriptul necesită existența următoarelor utilitare: *awk*, *bc*, *diff*, *sed*, *time*, *timeout*.

Atenție! Dacă aveți un calculator cu două core-uri (sau patru cu hyper-threading), va trebui să modificați măsurarea accelerației să nu ia în considerare și testul de 4 thread-uri, pentru că implementarea paralelă nu va scala. Dacă programul nu trece nici măcar unul din testele de scalabilitate, punctajul final va fi 0.

Docker

Pentru a avea un mediu uniform de testare, sau pentru a putea rula scriptul de test mai ușor de pe Windows sau MacOS (și cu mai puține resurse consumate decât dintr-o mașină virtuală), vă sugerăm să folosiți Docker. În repository-ul temei, găsiți un director numit *Docker*. Dacă aveți Docker instalat și rulați scriptul *start.sh*, vi se va da acces de shell într-un container care are montat sistemul local de fișiere la calea */tema1/*, de unde puteți apoi rula scriptul de testare automată. De menționat că, dacă descărcați de pe Git pe Windows, va trebui să schimbați sfârșitul liniilor din "CRLF" în "LF" și să mai rulați următoarea comandă în container, pentru a da drepturi de execuție pe script: `chmod a+x /tema1/test.sh`.

Rulare pe cluster

Dacă doriți să vă testați scalabilitatea implementării pe un număr mai mare de core-uri și sunteți limitați de resursele hardware, vă recomandăm să încercați să rulați pe cluster-ul facultății. Pentru a realiza acest lucru, puteți urmări pașii din această secțiune.

Conectare la FEP

Pentru a putea rula aplicații pe cluster, primul pas necesită conectarea la FEP (front-end processor), care reprezintă punctul de acces prin intermediul căruia putem da comenzi în cluster. Pentru a ne conecta, trebuie să dăm următoarea comandă (unde *username* reprezintă numele de utilizator cu care vă logați pe Moodle):

```
$ ssh <username>@fep.grid.pub.ro
```

Dacă avem nevoie să copiem fișiere pe FEP, putem executa următoarea comandă pe mașina noastră locală:

```
$ scp <fisier> <username>@fep.grid.pub.ro:.
```

Pentru conectare, se folosește parola de pe Moodle.

Scriptul de rulare

Varianta cea mai facilă de a rula un job pe cluster este de a-l defini prin intermediul unui script Bash. Un exemplu de script care compilează și rulează această temă se poate observa mai jos:

```
#!/bin/bash
module load compilers/gnu-5.4.0
make build
./tema1_par in.txt 10000 4
make clean
```

Se poate observa că prima comandă din script încarcă modulul *compilers/gnu-5.4.0*, pentru a se putea compila tema. Dacă doriți să vedeți modulele disponibile pe cluster, puteți da următoarea comandă:

```
$ module avail
```

Pornirea și analiza unui job

Mașinile din cluster sunt grupate în cozi, care sunt mulțimi de calculatoare cu arhitecturi similare. Pentru a putea vedea cozile din cluster și disponibilitatea lor, se poate executa următoarea comandă:

```
$ qstat -g c
```

Pentru trimiterea unui job spre execuție pe o coadă din cluster, se poate folosi următoarea comandă:

```
$ qsub -cwd -pe openmpi 24 -q ibm-nehalem.q script.sh
```

Dacă doriți să vă testați această temă în cluster, vă recomandăm să folosiți coada *ibm-nehalem.q*, care conține mașini cu 24 de nuclee. Putem urmări evoluția unui job folosind comanda de mai jos, care ne arată starea sa (*qw* dacă este în așteptare, *r* dacă este în execuție, iar dacă a terminat de rulat nu va mai apărea în listă):

```
$ qstat
```

Atunci când un job se termină de executat, se vor crea în directorul curent de pe FEP două fișiere, de forma *script.sh.e.ID_JOB* și *script.sh.o.ID_JOB*, care vor conține ceea ce s-a scris la *stderr* și respectiv *stdout*.

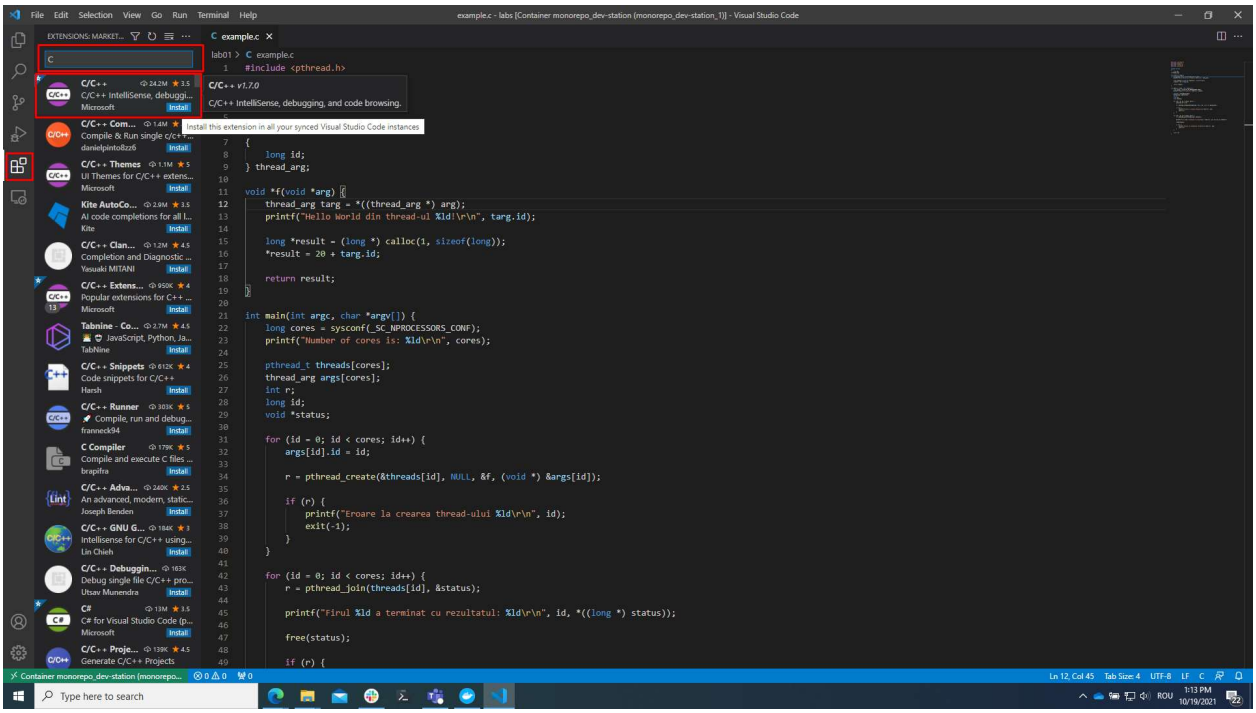
Atenție! Rulați doar folosind un script cum s-a explicat mai sus, nu direct pe FEP. La final, verificați că nu ați lăsat job-uri rulând în cluster.

Recomandări de implementare și testare

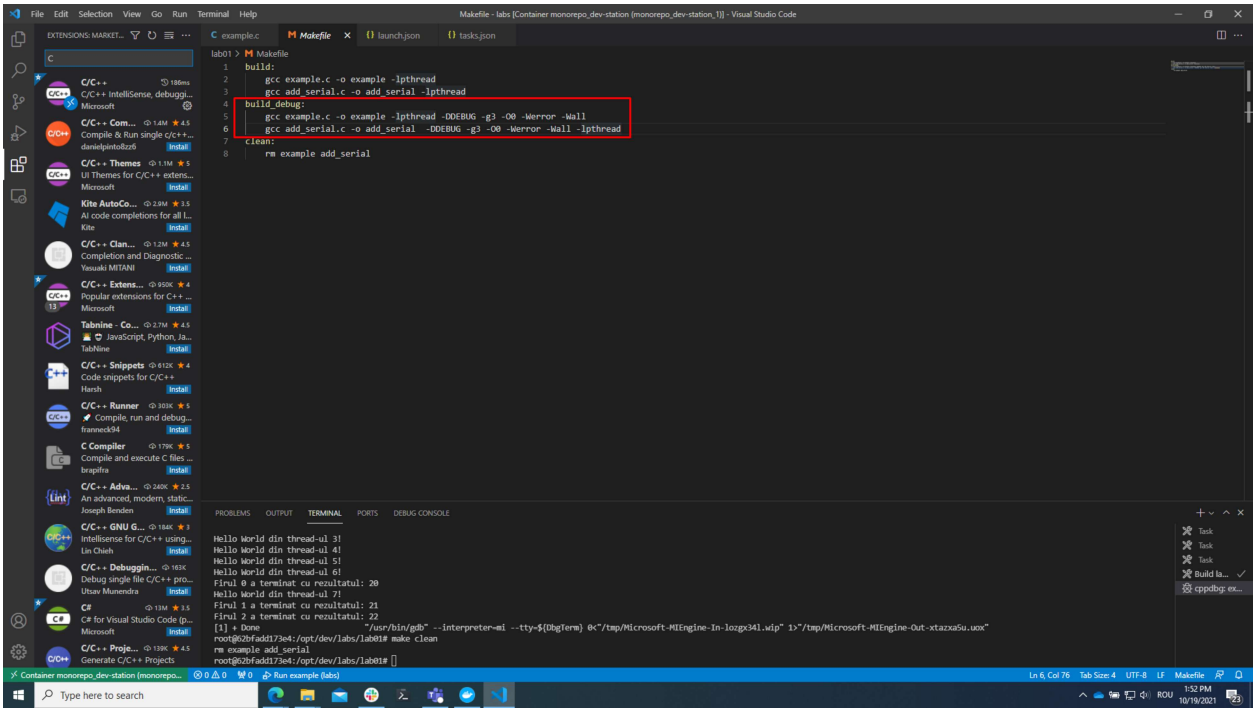
Găsiți aici o serie de recomandări legate de implementarea și testarea temei:

1. la compilare, folosiți flag-urile `-Wall` și `-Werror`; `-Wall` vă afișează toate avertismentele la compilare, iar `-Werror` vă tratează avertismentele ca erori; aceste flag-uri vă ajută să preîntâmpinați erori cum ar fi variabile neinițializate, care se pot propaga ușor
2. folosiți un repository **privat** de Git și dați commit-uri frecvent; vă ajută să vedeți diferențe între iterații de scris cod și vă asigură că aveți undeva sigur codul ca să-l puteți recupera în cazul în care l-ați șters din greșeală sau mașina pe care lucrați nu mai funcționează
3. citiți paginile de manual pentru funcțiile din biblioteca Pthreads și verificați valorile returnate de acestea; dacă aceste valoari corespund unor erori descrise acolo, vă puteți da seama ușor dacă folosiți greșit funcțiile sau structurile din bibliotecă
4. dacă lucrați pe Windows, vă recomandăm să folosiți WSL2 sau Docker (în loc de o mașină virtuală) pentru dezvoltare în Linux, deoarece sunt soluții care necesită mai puține resurse, iar performanța e mai aproape de cea a unui sistem de operare Linux care rulează direct pe mașinile voastre
5. folosiți un IDE precum CLion sau Visual Studio Code; pe lângă completarea automată a codului, aveți posibiliatea de a rula și a face debug pe cod.

În continure, aveți un exemplu de cum puteți să faceți debug în Visual Studio Code. În primul rând, trebuie să instalați plugin-ul pentru C/C++.

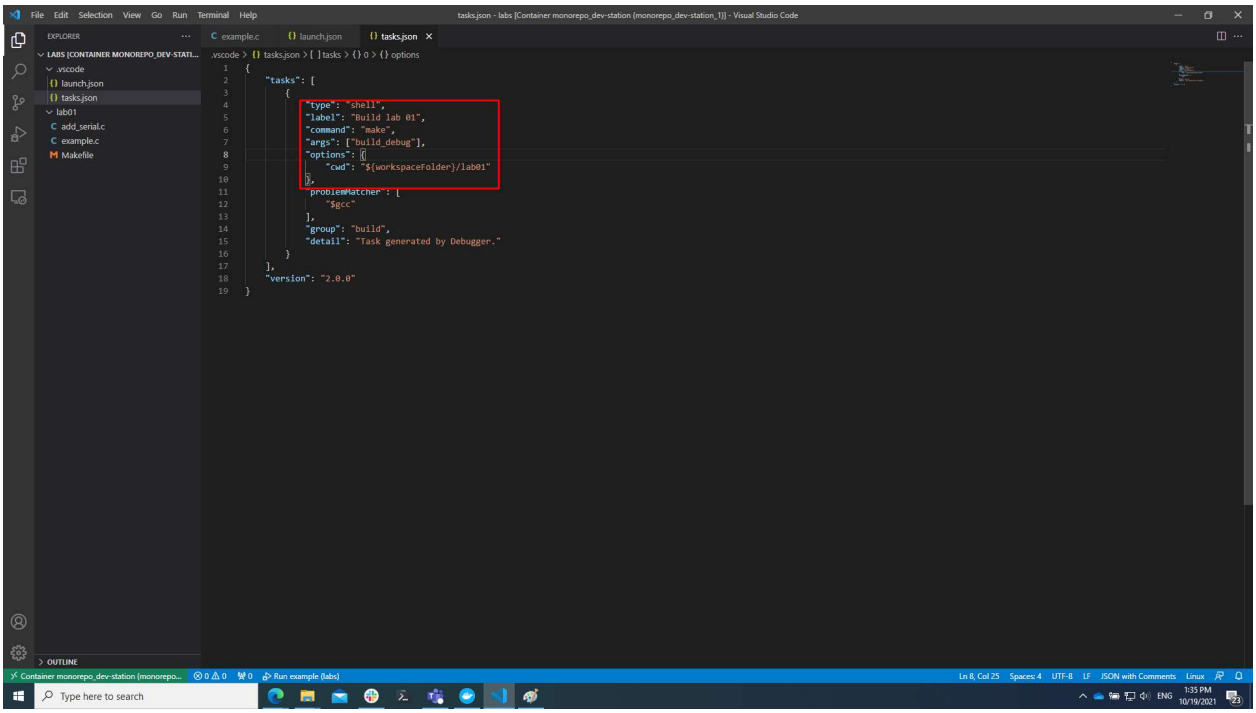


Apoi, trebuie să vă faceți în *Makefile* un build de debug care să aibă flag-urile `-O0 -g3 -DDEBUG`.

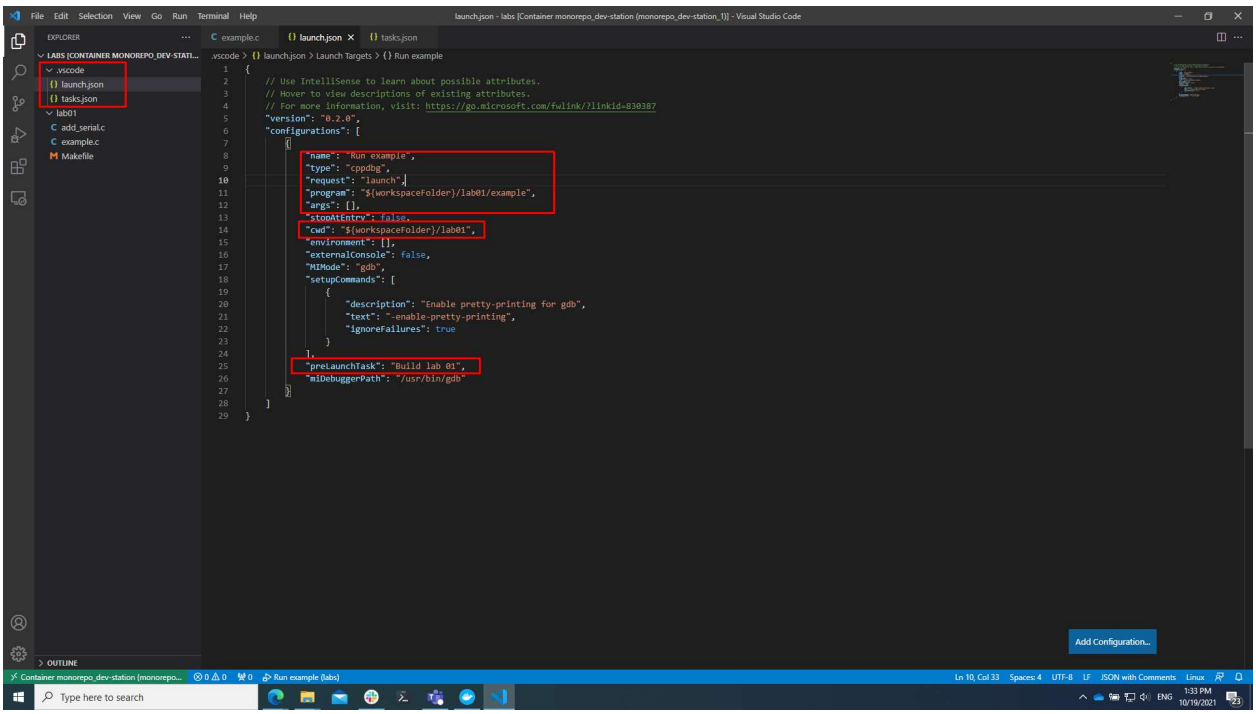


Ca să faceți build și să rulați codul, trebuie să aveți un director *.vscode* în care să aveți două fișiere, *launch.json* și *tasks.json*. În *tasks.json*, definiți ce acțiuni trebuie realizate pentru a se face build. În imaginea de mai jos, aveți un exemplu de cum trebuie să arate acest fișier. Trebuie să specificați unde se face build, care sunt

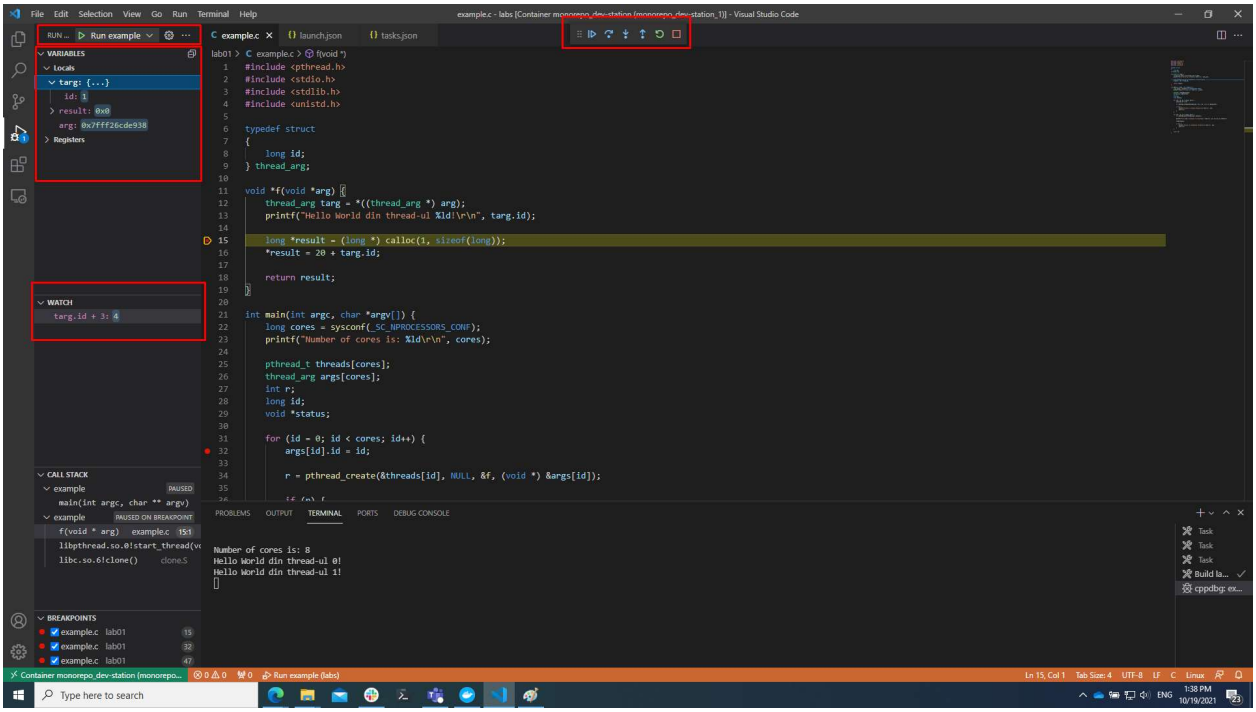
argumentele pentru *make*, precum și un label pentru a identifica acțiunea de build.



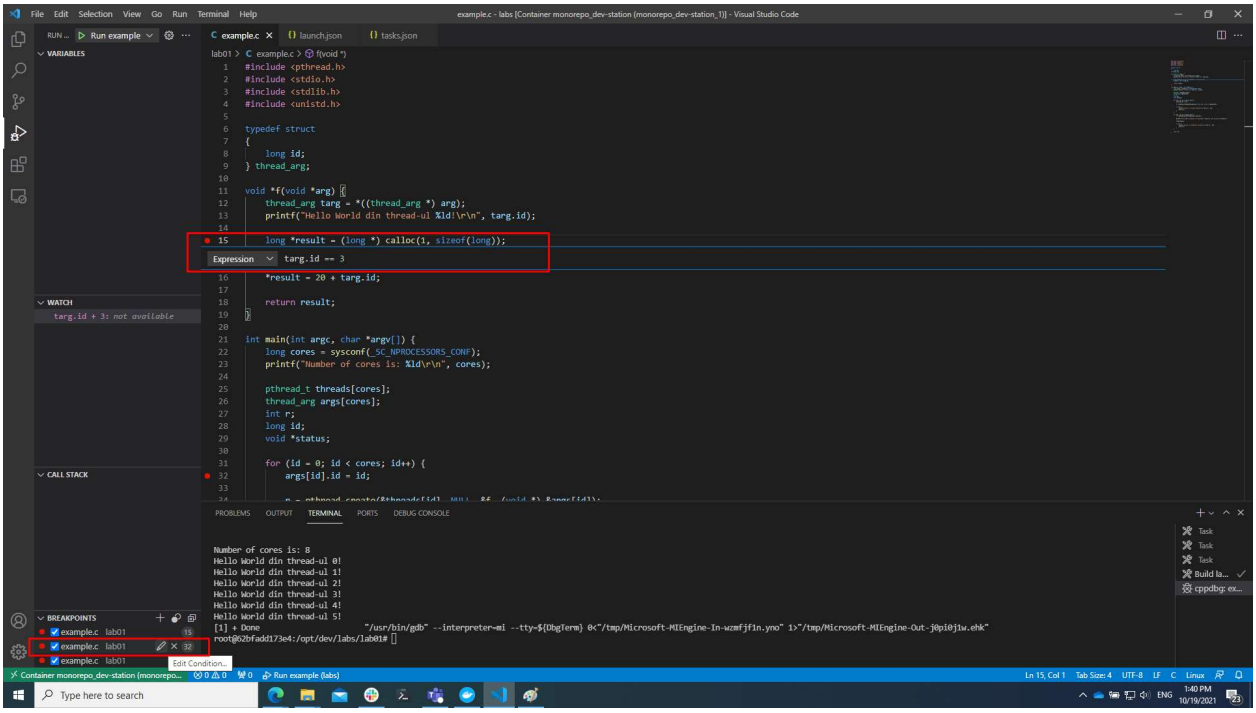
Pentru fișierul *launch.json*, trebuie să specificați programul de debug care trebuie rulat, directorul unde se rulează, precum și argumentele. De asemenea, este necesar să puneți label-ul pentru acțiunea de build definit precedent. Pentru rulare, trebuie să aveți instalat și utilitarul *gdb*.



După ce au fost create aceste fișiere, puteți să rulați executabilul de debug, să adăugați breakpoints și să navigați prin execuția programului. Puteți să mai vedeți variabilele locale și să evaluați expresii simple.



Un lucru foarte util pentru a face debug pe programe paralele este posibilitatea de a edita un breakpoint astfel încât să se declanșeze doar atunci când anumite condiții se îndeplinesc (de exemplu, dacă un fir de execuție are un anumit identificator).



Link-uri utile

1. Algoritmi genetici
2. Problema rucsacului
3. Debugging în Visual Studio Code
4. Docker
5. Docker Compose