Algoritmul Roy-Warshall-Floyd

grupa CR 3.2B, an 3, al 2-lea semestru, Calculatoare Română

Niculescu Marius-Andrei

April 11, 2022

Contents

1 Introducere

Scopul acestui proiect este de a implementa algoritmul Roy-Warshall-Floyd care determină calea de cost minim între toate nodurile dintr-un graf. Algoritmul va fi implementat atât folosind limbajul Java cât şi C++, folosind diferite tehnici. Prin urmare, voi implementa o variantă secvenţială şi una paralelă folosind thread-uri în Java şi două variante paralele în C++ cu ajutorul MPI-ului şi a STL-ului. La final vor fi comparate rezultatele obţinute în fiecare caz, folosind acelaşi set de date ca şi input.

- Java secvenţial
- Java paralel cu Thread-uri
- C++ paralel cu MPI
- C++ paralel cu STL

2 Date experimentale

Pentru a genera graful în care voi determina APSP (all pairs shortest paths), voi citi dintr-un fişier de input, numărul de noduri şi de muchii din graf. Aceste date sunt folosite de către generatorul de graf care îl construieşte în felul următor: Sunt generaţi 2 indecşi în mod aleatoriu, care reprezintă 2 noduri din graf. Dacă există deja o muchie între aceste 2 noduri, atunci se generează o altă pereche de indecşi şi se verifică iar. Atunci când se găseşte o pereche de noduri între care nu există o muchie, se ataşează una de cost random în intervalul [50,500], cost generat cu ajutorul funcţiei getRandomIntegerBetweenRange(double *min*, double *max*) din RandomGraphGenerator.java. Acest proces se repetă până când sunt adăugate atâtea muchii în graf, cât indică numărul citit din fişier.

3 Proiectarea aplicaţiei experimentale in Java

3.1 Structura de nivel înalt a aplicaţiei

- Node.java
- Edge.java
- RandomGraphGenerator.java
- RoyFloydSequential.java
- RoyFloydParallelMonitor.java
- RoyFloydParallel.java
- Application.java

3.2 Descrierea mulţimii datelor de intrare

Se citeşte dintr-un fişier numărul de noduri şi numărul de muchii din graful orientat, iar apoi iniţializez variabilele $numberOfNodes$ şi $numberOfEdges$, din clasa $RandomGraphGenerator$, cu valorile citite din fişier şi generez graful.

3.3 Descrierea ieşirilor / rezultatelor

Datele de ieşire din fişier menţionează numărul de noduri şi de muchii din graf, distanţa de cost minim între toate nodurile între care s-a găsit o cale şi timpii de execuţie ai algoritmului Roy-Floyd şi a întregului program.

3.4 Modulele aplicaţiei

Aplicaţia are următoarele module:

- 10 fişiere de tip text din care se citesc numărul de noduri şi de muchii care intervin în construirea în mod aleatoriu a grafului. Fiecare fişier contribuie la obţinerea unui test unic.
- 20 fişiere de tip text (10 pentru varianta secventiala, 10 pentru varianta paralela), în care sunt scrise datele de iesire menţionate în subsecţiunea de mai sus.
- Node.java în care se definesc atributele unui nod din graf. Fiecare nod conţine o listă de muchii (cu tot cu costul lor) către nodurile vecine cu care se conectează (aceste muchii se adaugă cu ajutorul funcţiei $addNeighbour(Node\ neighbour, int\ cost)$ care construieşte un drum între nodul curent şi cel dat ca parametru funcţiei).
- Edge.java este clasa care defineşte atributele unei muchii din graf. Aceasta are un cost, şi o variabilă $target$, care reprezintă al doilea nod cu care se leagă primul atunci când se apelează funcţia $addNeighbour$.
- RandomGraphGenerator.java este clasa pe care am descris-o în secţiunea 3 "Date experimentale".
- RoyFloydSequential.java este clasa care implementează varianta secvenţială a algoritmului Roy-Floyd
- RoyFloydParallelMonitor.java reprezintă o clasă suport, care conţine graful generat de clasa RandomGraphGenerator şi un zăvor utilizat pentru a actualiza costurile muchiilor. Această clasă este folosită în clasa RoyFloydParallel pentru a executa varianta paralelă a lui Roy-Floyd
- RoyFloydParallel.java este clasa care implementează varianta paralelă a algoritmului Roy-Floyd

- Application.java este clasa în care testez toate celelalte funcţii. Aici realizez citirea din fişierele de input, generez în mod aleatoriu graful cu ajutorul clasei RandomGraphGenerator, apelez funcţia corespunzătoare algoritmului Roy-Floyd (atât cea secvenţială cât şi cea paralelă) şi calculez şi timpul în care se execută această funcţie şi timpul în care se termină tot programul. Liniile 16-75 din funcţia main a acestei clase rulează varianta secvenţială a lui Roy-Floyd, iar liniile 79-177 rulează varianta paralelă.
- Funcţiile main din Application.java, addNeighbour din Node.java şi graphGenerator din RandomGraphGenerator.java au fost discutate anterior.

3.5 Funcţiile aplicaţiei

Aplicaţia are următoarele funcţii:

- royFloydSequential($List < Node > nodeList$, $Bufferedwriter\ writer$) este funcţia care implementează algoritmul secvenţial. Deoarece costurile ataşate muchiilor se consideră la iniţializare în intervalul [50,500], am considerat valoarea 1001, ca fiind un reper ce înseamnă că nu există o muchie între 2 noduri. În continuare am folosit 3 instrucţiuni repetitive cu număr cunoscut de paşi (for) pentru a încerca să găsesc un drum minim între toate perechile de 2 noduri, prin diferite noduri intermediare. Astfel, calculez distanţa de la nodul i la nodul k şi distanţa de la nodul k la nodul j. Dacă nu există o muchie de la i la k, respectiv de la k la j, costul va fi considerat 1001, cum am explicat la început. Dacă suma acestor două costuri este mai mică decât costul muchiei de la nodul i la j, atunci acesta va fi actualizat. În situaţia în care nu există o muchie de la i la j, dar costul muchiei de la i la k plus costul muchiei de la k la j este un număr mai mic decât 1001, atunci adăugăm o muchie între nodurile i şi j, cu acest cost. La final parcurgem lista de adiacenţă de la fiecare nod din graf şi scriem în fişier distanţele minime pe care le-a aflat algoritmul.
- run() este funcţia pe care o execută fiecare thread RoyFloydParallel. Am considerat că numărul de thread-uri care lucrează în paralel este egal cu $numberOfNodes/10$. Modul de lucru este asemănător celui secvenţial, dar aici fiecare thread consideră câte 10 noduri intermediare prin care încearcă să găsească drumul minim între toate perechile de câte 2 noduri. Atunci când algoritmul găseşte o muchie pe care să o relaxeze, acesta încearcă să achiziţioneze zăvorul din clasa suport denumită RoyFloydParallelMonitor, iar atunci când reuşeşte, îi actualizează costul.

4 Proiectarea aplicaţiei experimentale in C++ cu MPI

4.1 Structura de nivel înalt a aplicaţiei

- Roy_Floyd_MPI.cpp

4.2 Descrierea mulţimii datelor de intrare

Se citeşte dintr-un fişier numărul de noduri şi numărul de muchii din graful orientat, iar apoi iniţializez variabilele n şi e, cu valorile citite din fişier şi generez graful.

4.3 Descrierea ieşirilor / rezultatelor

Datele de ieşire din fişier menţionează numărul de noduri şi de muchii din graf, numărul de procese care lucrează în paralel, distanţa de cost minim între toate nodurile între care s-a găsit o cale şi timpii de execuţie ai algoritmului Roy-Floyd şi a întregului program.

4.4 Modulele aplicaţiei

Aplicaţia are următoarele module:

- 10 fişiere de tip text din care se citesc numărul de noduri şi de muchii care intervin în construirea în mod aleatoriu a grafului. Fiecare fişier contribuie la obţinerea unui test unic.
- 10 fişiere de tip text, în care sunt scrise datele de ieşire menţionate în subsecţiunea de mai sus.
- Roy_Floyd_MPI.cpp în care sunt definite toate funcţiile pe care le voi prezenta în continuare.

4.5 Funcţiile aplicaţiei

Aplicaţia are următoarele funcţii:

- Read_matrix(int local_mat[], int n, int e, int my_rank, int p, MPI_Comm comm) este funcţia care se ocupă cu iniţializarea grafului (mai exact procesul cu rank-ul 0). Prima dată procesul cu rank-ul 0 adaugă 0 pe acele poziţii în care $i = j$ şi $INFINITY\_VALUE$ (valoarea 1001) pe celelalte poziţii. După aceea, generează 2 indecşi în mod aleatoriu şi verifică dacă pe poziţia respectivă se află valoarea $INFINITY\_VALUE$. În caz afirmativ, ataşează un cost din intervalul [50,500], iar în caz contrar se reia procesul de generare a indecşilor până se găseşte o poziţie pe care se află $INFINITY\_VALUE$. La final procesele execută MPI_Scatter astfel încât fiecare proces să primească o parte din matricea generată (mai exact n^2/p, unde n este numărul de noduri, iar p numărul de procese).
- Print_matrix(int local_mat[], int n, int e, double durationRoyFloyd, double durationProgram, int my_rank, int p, MPI_Comm comm) reprezintă funcţia care se ocupă cu scrierea în fişier a datelor detaliate la subsecţiunea 4.3 "Descrierea iesirilor / rezultatelor". Procesul cu rank 0 realizează un MPI_Gather pentru a colecta matricele locale de la fiecare proces, iar apoi

parcurge matricea şi scrie în fişier, distanţele dintre noduri, acolo unde valoarea nu este $INFINITY\_VALUE$.

- Owner(int k, int p, int n) este funcţia care returnează rank-ul procesului care deţine linia indicată de iteratorul global k
- Copy_row(int local_mat[], int n, int p, int row_k[], int k) este funcţia care copiază în array-ul row_k[], linia din matricea locală a procesului, cu indicele obţinut din contorul global k, în felul următor: local_k = k % (n / p)
- Floyd(int local_mat[], int n, int my_rank, int p, MPI_Comm comm) reprezintă funcţia care implementează algoritmul lui Roy-Floyd. Fiecare proces va încerca să găsească un drum minim între nodurile aflate în matricea locală a procesului şi toate celelalte noduri, având în vedere toate nodurile intermediare. În această situaţie, dacă rank-ul procesului care apelează funcţia Roy-Floyd, este egal cu numărul oferit de funcţia Owner discutată anterior, atunci acest proces va trebui să apeleze Copy_row pentru a putea da MPI_Bcast la celelalte procese care nu au acces la matricea locală a acestui proces. Apoi se verifică de fiecare dată, dacă se poate relaxa muchia de la nodul i la j prin nodul intermediar k.

5 Proiectarea aplicaţiei experimentale in C++ cu STL

5.1 Structura de nivel înalt a aplicaţiei

- RoyFloydSTL.cpp

5.2 Descrierea mulţimii datelor de intrare

Se citeşte dintr-un fişier numărul de noduri şi numărul de muchii din graful orientat, iar apoi iniţializez variabilele $numberOfNodes$ şi $numberOfEdges$, cu valorile citite din fişier şi generez graful.

5.3 Descrierea ieşirilor / rezultatelor

Datele de ieşire din fişier menţionează numărul de noduri şi de muchii din graf, numărul de procese care lucrează în paralel, distanţa de cost minim între toate nodurile între care s-a găsit o cale şi timpii de execuţie ai algoritmului Roy-Floyd şi a întregului program.

5.4 Modulele aplicaţiei

Aplicaţia are următoarele module:

- 10 fişiere de tip text din care se citesc numărul de noduri şi de muchii care intervin în construirea în mod aleatoriu a grafului. Fiecare fişier contribuie la obţinerea unui test unic.
- 10 fişiere de tip text, în care sunt scrise datele de iesire menţionate în subsecţiunea de mai sus.
- RoyFloydSTL.cpp în care sunt definite toate funcţiile pe care le voi prezenta în continuare.

5.5 Funcţiile aplicaţiei

Aplicaţia are următoarele funcţii:

- readMatrix($std :: vector < std :: vector < int >>\&graph$, int numberOfNodes, int numberOfEdges) este funcţia care se ocupă cu iniţializarea grafului (mai exact procesul cu rank-ul 0). Prima dată procesul cu rank-ul 0 adaugă 0 pe acele poziţii în care $i = j$ şi $INFINITY\_VALUE$ (valoarea 1001) pe celelalte poziţii. După aceea, generează 2 indecşi în mod aleatoriu şi verifică dacă pe poziţia respectivă se află valoarea $INFINITY\_VALUE$. În caz afirmativ, ataşează un cost din intervalul [50,500], iar în caz contrar se reia procesul de generare a indecşilor până se găseşte o poziţie pe care se află $INFINITY\_VALUE$. La final procesele execută MPI_Scatter astfel încât fiecare proces să primească o parte din matricea generată (mai exact n^2/p, unde n este numărul de noduri, iar p numărul de procese).
- printMatrix($std :: vector < std :: vector < int >>\&graph$, int numberOfNodes, int numberOfEdges, milliseconds durationRoyFloyd, milliseconds durationProgram, int testNumber) reprezintă funcţia care se ocupă cu scrierea în fişier a datelor detaliate la subsecţiunea 4.3 "Descrierea iesirilor / rezultatelor". Procesul cu rank 0 realizează un MPI_Gather pentru a colecta matricele locale de la fiecare proces, iar apoi parcurge matricea şi scrie în fişier, distanţele dintre noduri, acolo unde valoarea nu este $INFINITY\_VALUE$.
- royFloyd($std :: vector < std :: vector < int >>\&graph$, int numberOfNodes) reprezintă funcţia care implementează algoritmul lui Roy-Floyd. Pentru a executa în paralel bucla i, am folosit un vector auxiliar care conţine indecşii liniilor din matrice. Acest vector conţine toate valorile pe care poate să le ia i, o singură dată. Folosesc for_each executat în paralel, unde fiecare partiţie a vectorului se referă la o partiţie din matrice (la o linie). În acest fel, distanţele de la nodurile i la nodurile j prin nodurile intermediare k se realizează în paralel.

6 Rezultate şi concluzii

6.1 Java secvenţial

Nr	Numărul de noduri	Numărul de muchii	Roy-Floyd	Întreg programul
1	50	250	0.195 s	0.197 s
2	125	700	0.308 s	0.309 s
3	250	1200	5.984 s	5.987 s
4	400	1750	25.643 s	25.645 s
5	500	2000	46.435 s	46.436 s
6	650	2500	113.887 s	113.890 s
7	800	3000	86.406 s	86.409 s
8	1000	5000	325.533 s	325.538 s
9	1500	6500	929.333 s	929.335 s
10	2000	8000	2306.873 s	2306.876 s

6.2 Java paralel cu Thread-uri

Nr	Numărul de noduri	Numărul de muchii	Roy-Floyd	Întreg programul
1	50	250	0.083 s	0.087 s
2	125	700	0.368 s	0.369 s
3	250	1200	0.851 s	0.852 s
4	400	1750	3.463 s	3.465 s
5	500	2000	5.882 s	5.884 s
6	650	2500	11.363 s	11.365 s
7	800	3000	20.862 s	20.866 s
8	1000	5000	89.499 s	89.504 s
9	1500	6500	229.608 s	229.610 s
10	2000	8000	432.792 s	432.793 s

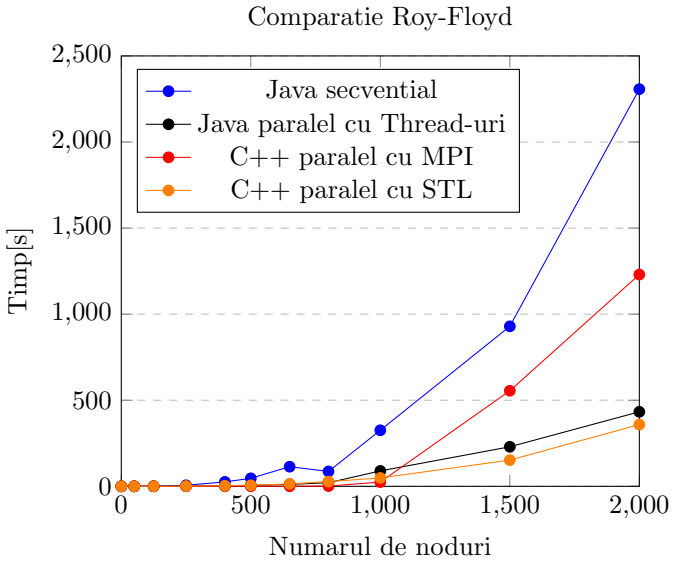
6.3 C++ paralel cu MPI

Nr	Numărul de noduri	Numărul de muchii	Roy-Floyd	Întreg programul
1	50	250	0.00093 s	0.00157 s
2	125	700	0.00236 s	0.00310 s
3	250	1200	0.07303 s	0.07446 s
4	400	1750	0.267 s	0.270 s
5	500	2000	0.424 s	0.429 s
6	650	2500	0.503 s	0.509 s
7	800	3000	1.429 s	1.450 s
8	1000	5000	24.496 s	24.516 s
9	1500	6500	554.914 s	554.979 s
10	2000	8000	1230.272 s	1230.522 s

6.4 C++ paralel cu STL

Nr	Numărul de noduri	Numărul de muchii	Roy-Floyd	Întreg programul
1	50	250	0.012 s	0.013 s
2	125	700	0.104 s	0.111 s
3	250	1200	1.071 s	1.097 s
4	400	1750	3.462 s	3.525 s
5	500	2000	6.047 s	6.143 s
6	650	2500	13.920 s	14.080 s
7	800	3000	27.422 s	27.669 s
8	1000	5000	48.605 s	48.997 s
9	1500	6500	152.186 s	153.035 s
10	2000	8000	358.633 s	360.142 s

6.5 Grafic final



Testele paralele pentru Java paralel cu Thread-uri şi C++ paralel cu MPI au considerat că numărul de procese care lucrează în paralel să fie egal cu numărul de noduri împărţit la 10. Pentru MPI timpul de execuţie a crescut brusc după 1000 de noduri, deoarece se pierde foarte mult timp ca procesele să comunice între ele prin intermediul funcţiei MPI_Scatter. Astfel, cu cât numărul de noduri creşte, cu atât creşte şi numărul de procese care lucrează în paralel, ceea ce conduce la un timp de execuţie relativ mai mare decât situaţia în care am avea câteva procese care lucrează în paralel. De exemplu, în situaţia în care am rulat un test pentru C++ paralel cu MPI pentru 2000 de noduri cu 5 procese care lucrează în paralel, a trebuit să aştept doar 17 secunde pentru ca programul să îşi termine execuţia.