Proiect Programare paralela si concurenta

Gasirea celor mai scurte drumuri intr-un hipercub

Tudor Alexandru

-grupa 406-

Un hipercub este corespondentul intr-un spatiu n-dimensional al patratului din spatiul bidimensional. Acesta este un obiect cu fetele plane, compact, convex, al carui schelet este format din segmente de aceeasi lungime, paralele, opuse, aliniate si perpendiculare unele pe altele. Lungimea celor mai lungi diagonale ale unui hipercub de dimensiune n este $\sqrt{n}$.

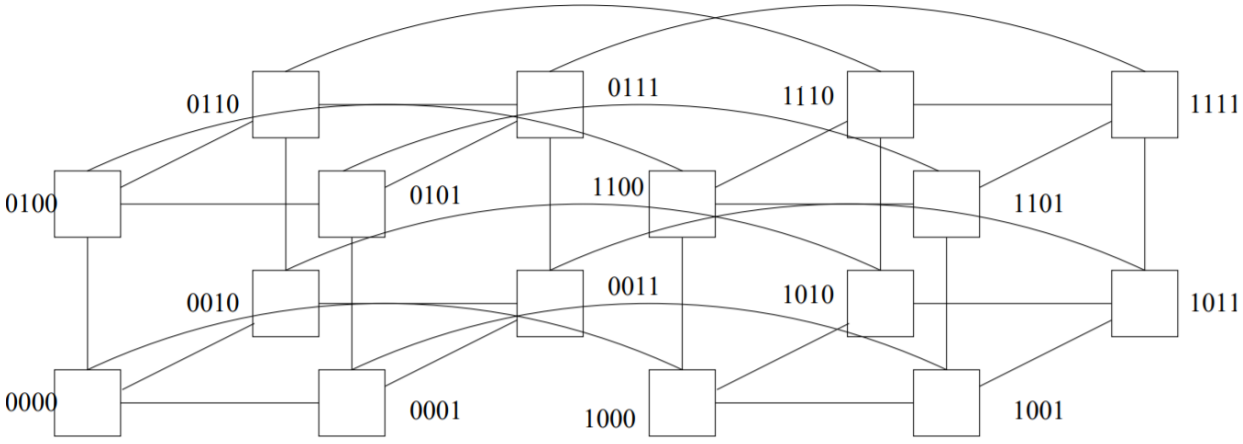

Figura 1 Hipercub in 4 dimensiuni (Tesseract)

Un hipercub poate fi definit prin creșterea numărului de dimensiuni:

0 – Un punct este un hipercub în zero dimensiuni.

1 – Deplasând punctul cu o unitate de lungime de-a lungul unei dimensiuni se va obține un segment, care este un hipercub cu o dimensiune.

2 – Deplasând acest segment într-o direcție perpendiculară pe direcția segmentului, cu o unitate de lungime, se va obține un pătrat, care este un hipercub în două dimensiuni.

3 – Deplasând acest segment într-o direcție perpendiculară pe planul pătratului, cu o unitate de lungime, se va obține un cub, care este un hipercub în trei dimensiuni.

4 – Deplasând acest segment într-o direcție perpendiculară pe celelalte trei dimensiuni, cu o unitate de lungime, se va obține un 4-cub, care este un hipercub în patru dimensiuni (numit și tesseract).

Procedeul se poate generaliza pentru orice numar de dimensiuni.

Algoritmul lui Dijkstra implementat in proiect este un algoritm pentru cautarea celor mai scurte drumuri de la un varf la toate celelalte varfuri din hipercub.

1. Se creează o listă cu distanțe, si o listă cu nodurile vizitate și doua variabile: nod curent si nod anterior.
2. Toate valorile din lista cu distanțe sunt inițializate cu o valoare infinită, cu excepția nodului de start, care este setat cu 0.
3. Toate valorile din lista cu nodurile vizitate sunt setate cu fals.
4. Toate valorile din lista cu nodurile anterioare sunt inițializate cu -1.
5. Nodul de start este setat ca nodul curent.
6. Se marchează ca vizitat nodul curent.
7. Se actualizează distanțele, pe baza nodurilor care pot fi vizitate imediat din nodul curent.
8. Se actualizează nodul curent la nodul nevizitat care poate fi vizitat prin calea cea mai scurtă de la nodul de start.
9. Se repetă (de la punctul 6) până când toate nodurile sunt vizitate.

Clasele proiectului:

- clasa *DijkstraAlgorithmBackend* gestioneaza starea de executie a algoritmului si executa in starea respectiva operatiile de baza prevazute de algoritm precum adaugarea unui varf la un set de varfuri procesate, sau selectarea varfului cel mai apropiat.
- clasa *DijkstraMPI* include o implementare paralela a algoritmului Dijkstra. Foloseste API-ul oferit de clasa *DijkstraAlgorithmBackend*. Aceasta clasa ofera o metoda de rulare care executa algoritmul si returneaza rezultatul acestuia.
- clasa *DijkstraMPISetup* este clasa responsabila pentru prepararea datelor legate de impartirea matricei de adiacenta (fiecare proces va gestiona o parte din aceasta).
- clasa *ResultPrinter* este responsabila de printarea rezultatelor algoritmului.

Implementarea MPI

Algoritmul consta in 3 parti principale:

1. intializare – impartirea datelor, validare si trimiterea datelor catre procese individuale.
2. calculul distantelor (algoritmul lui Dijkstra)
3. salvarea rezultatelor intr-un fisier si terminarea programului.

In continuare voi detalia aceste 3 parti principale:

1.Initializare

Prima parte a programului consta in mai multi pasi mai mici. La inceput are loc initializarea standard a protocolului MPI si atribuirea unui rank fiecarui proces. Urmatorul pas este de a crea o clasa responsabila pentru afisarea informatiilor despre fluxul programului la iesire.

Matricea de adiacenta este gestionata prin procesul principal (root). Acest proces imparte matricea in functie de numarul de procese implicate in executia algoritmului. Fiecare dintre procese primeste k coloane ale matricei originale sub forma unei zone de memorie continua.

Avand numarul de coloane din matrice (n) si numarul p de procese, fiecarui proces *i* i-se atribuie: a= [n/p] coloane. In cazul in care numarul de procese este mai mare decat numarul de coloane, fiecare proces primeste in mod implicit o coloana. Numarul de coloane ramas este dat de relatia : k= n mod p.

Dupa procesul de divizare, sunt trimise 3 informatii fiecaruia dintre procese:

- numarul total de noduri din hipercubul procesat cu *MPI_Bcast*.
- numarul de coloane pe care fiecare proces va trebui sa le gestioneze. Aceste informatii sunt necesare in acest moment, deoarece fiecare dintre procese trebuie sa pregateasca un tampon suficient de mare pentru datele din matricea lor. Aceste informatii sunt trimise folosind operatia *MPI_Scatter*, deoarece fiecare proces primeste informatiile sale individuale.
- continutul coloanelor matricei de adiacenta care vor fi procesate de proces. Aceste informatii sunt trimise prin intermediul functiei *MPI_Scatterv*, deoarece este capabil sa gestioneze trimiterea unei cantitati diferite de date catre fiecare dintre procese.

Dupa finalizarea trimiterii de date, procesele sunt impartite in doua grupe: procese care au primit cel putin o coloana si cele care nu au primit niciuna (au fost alocate mai multe procese decat exista varfuri). Pentru acesta, folosind operatia *MPI_Comm_split* , comunicatorul global este impartit in doi comunicatori. Criteriul de divizare aici este verificarea faptului ca procesului i s-au alocat coloane pentru a suporta operatiile (*nrColoaneDeManipulat > 0)*. Acest lucru permite excluderea proceselor redundante din cursul ulterior al algoritmului.

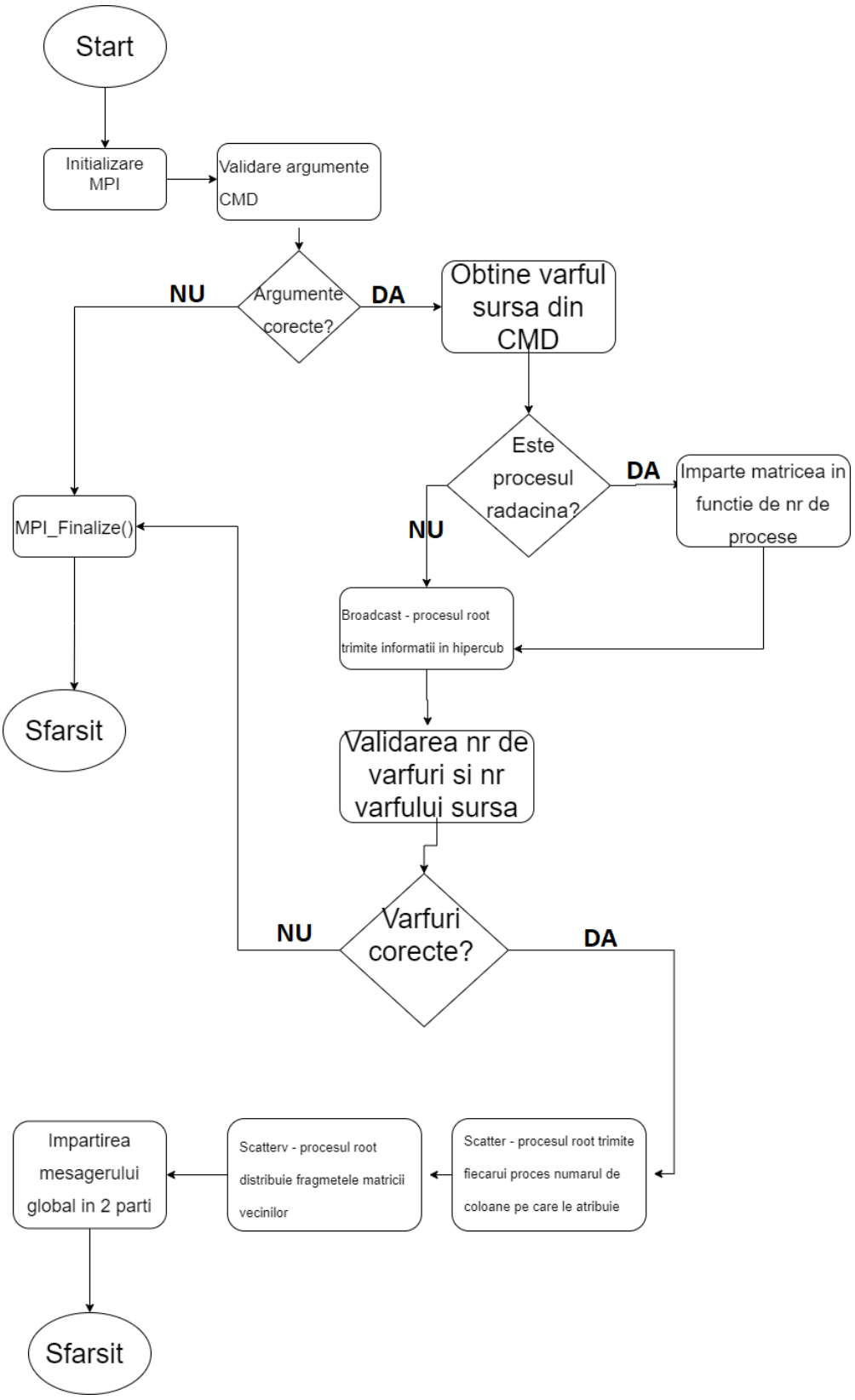

Fig. 2 Diagrama procesului de initializare

Implementarea algoritmului

Se va verifica daca procesului dat i-au fost alocate varfurile hipercubului (coloanele matricei de adiacenta) pentru procesare. Daca nu, atunci un astfel de proces nu este implicat in executarea ulterioara a algoritmului – comunicatorul creat este alocat si operatiunea este incheiata. In caz contrar, procesul continua pentru implementarea algoritmului.

Pentru implementarea algoritmului Dijkstra am folosit doua matrici: a distantelor (de la varful sursa pana la fiecare varf din hipercub) si a predecesorilor fiecarui varf (pentru a recrea cele mai scurte cai intre varfuri).

In procesul de gestionare al varfului sursa, intrarea sa in tabelul distantelor este setata la 0. Apoi, pana cand toate varfurile au fost procesate, algoritmul este parcurs intr-o bucla:

- fiecare proces selecteaza unul dintre nodurile atribuite acestuia (unul care nu a fost inca adaugat la setul de noduri procesate).
- prin operatia *MPI_Allreduce* este selectat varful cu cea mai mica valoare a distantei global (adica cel mai bun dintre cele mai bune din fiecare proces).
- daca un astfel de varf nu a fost determinat, bucla iese si algoritmul se termina.
- in caz contrar, varful este adaugat la setul de varfuri procesate (in fiecare proces acest set este alocat separat, in intregime)
- pentru fiecare vecin direct al varfului desemnat, se face o actualizare a tabelului distantelor si predecesorilor (varful selectat devine predecesorul vecinilor sai) daca noul cost se dovedeste a fi mai mic decat cel vechi.

Dupa finalizarea algoritmului, fragmentele tabelelor de distante si ale predecesorilor sunt combinate printr-o operatie de MPI_Gatherv. Procesul root urmareste caile de la varful sursa la fiecare varf din grafic si scrie rezultatele intr-un fisier. Apoi este afisat timpul de functionare al algoritmului. Ultimul pas este eliberarea comunicatorului creat anterior si bineinteles operatia de *MPI_Finalize*.

O variabila ce contine ora curenta este creata in mai multe locuri din program. La sfarsitul procesului de rulare, root-ul colecteaza aceste informati si afiseaza timpul de executie pentru fiecare din cele 3 parti ale programului , alaturi de timpul total, astfel:

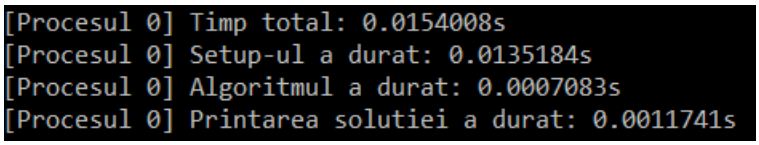

Figura 3 Afisarea rezultatelor

Masurarea se efectueaza folosind clasa *high_resolution_clock* din biblioteca *crono*. Acest ceas asigura ca timpul masurat este in timp real si nu timpul procesorului.

Figura 4 prezinta o diagrama ce ilustreaza fluxul algoritmului dupa initializare. Blocul asociat cu bucla principala a algoritmului lui Dijkstra este cel marcat cu rosu. Functionarea acestui algoritm este detaliata in figura 6.

Figura 5 este o diagrama ce ilustreaza operatiile recursive principale ale algoritmului Dijkstra. Aceasta diagrama este o extensie a blocului rosu din figura 5

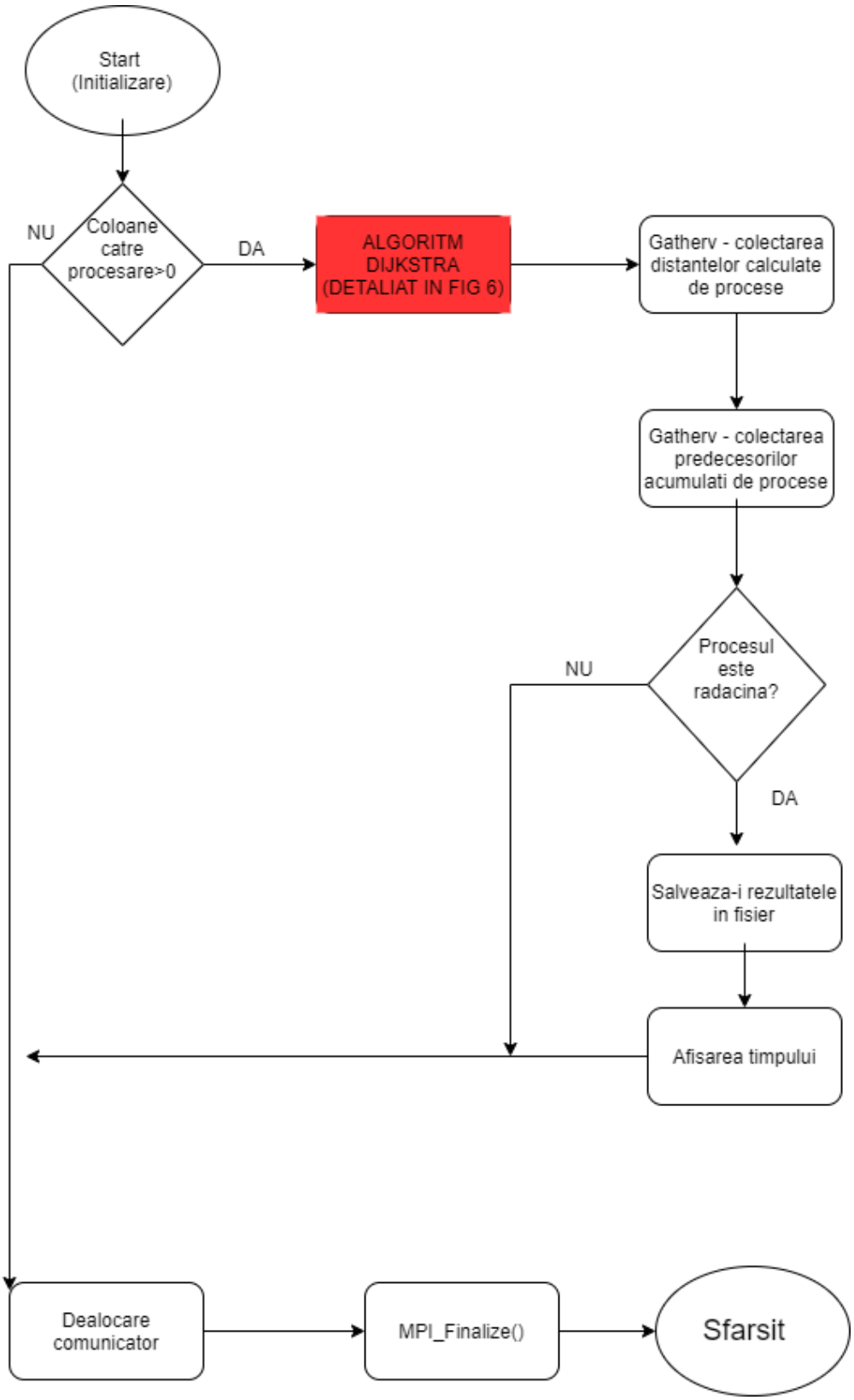

Figura 4 Fluxul programului dupa initializare

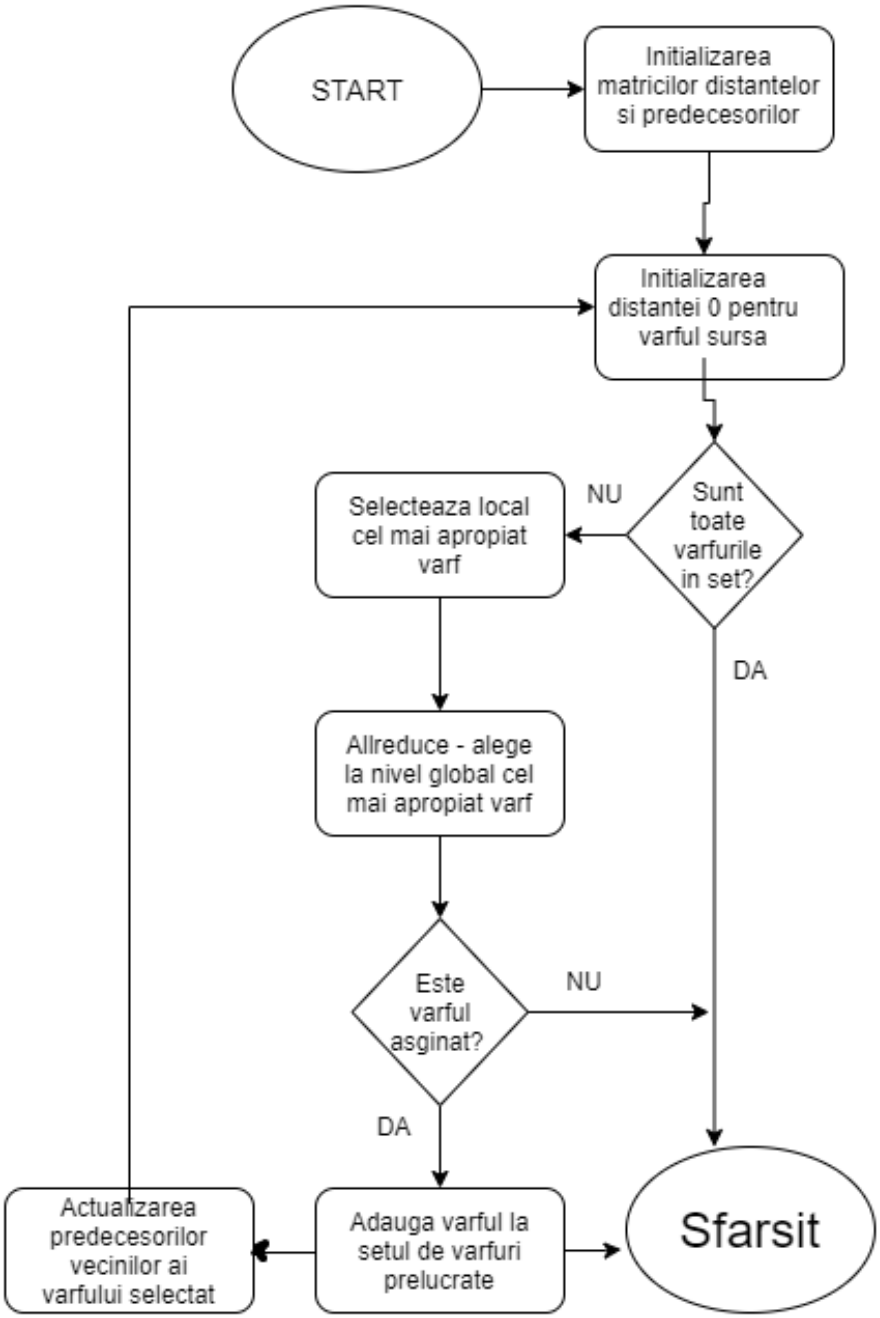

Figura 5 Algoritmul Dijkstra

Functionalitati MPI utilizate

Urmatoarele functionalitati MPI au fost utilizate ca parte a proiectului:

- MPI_Bcast – pentru a trimite catre toate procesele informatii despre numarul total de varfuri din hipercubul procesat.
- MPI_Scatter – pentru a trimite fiecaruia dintre procese numarul de varfuri pe care fiecare proces trebuie sa le gestioneze.
- MPI_Scatterv – pentru a trimite fiecaruia dintre procese datele din matricea de vecini. In acest caz, este necesar sa puteti trimite fiecaruia dintre procese date de lungime diferita.
- MPI_Allreduce – pentru a determina varful cu cea mai mica distanta.
- MPI_Gatherv – pentru a colecta rezultatele produse de fiecare dintre procese. In acest caz, este necesara posibilitatea de a primi date cu o lungime diferita de la fiecare proces.
- MPI_Comm_split si MPI_Comm_free – pentru a crea si lansa un nou communicator. Acest lucru permite excluderea proceselor pasive (redundante) de la participarea la executia algoritmului.

Teste de proiectare

Pentru teste, s-au folosit hipercuburi compuse din 8,16 ,32, 64 ,128 ,256 ,512 , 1024 varfuri si au fost utilizate urmatoarele numere de procese: 1,2,3,4,5, 6,7,8. Pentru fiecare combinatie a datelor de intrare si a numarului de procese, au fost efectuate 3 studii, din care a fost apoi extrasa media.

Figurile 6,7,8 si Tabelul 1 prezinta rezultatele obtinute in studiul timpului de executie al intregului program / numar de procese.

Nr. Proc	8 VARFURI	16 VARFURI	32 VARFURI	64 VARFURI	128 VARFURI	256 VARFURI	512 VARFURI	1024 VARFURI
1	0.013	0.013	0.028	0.032	0.095	0.035	1.5	6.29
2	0.03	0.03	0.025	0.033	0.08	0.02	1.12	4.47
3	0.018	0.017	0.025	0.03	0.079	0.22	1.01	3.61
4	0.031	0.031	0.04	0.029	0.077	0.2	0.92	3.39
5	0.025	0.025	0.02	0.03	0.09	0.21	0.98	3.25
6	0.02	0.021	0.02	0.032	0.08	0.2	0.96	3.21
7	0.015	0.017	0.023	0.032	0.072	0.2	0.91	3.2
8	0.015	0.016	0.024	0.03	0.08	0.19	0.91	3.19

Tabelul 1 Rezultatele Timpului total de executie / Numarul de varfuri ale hipercubului

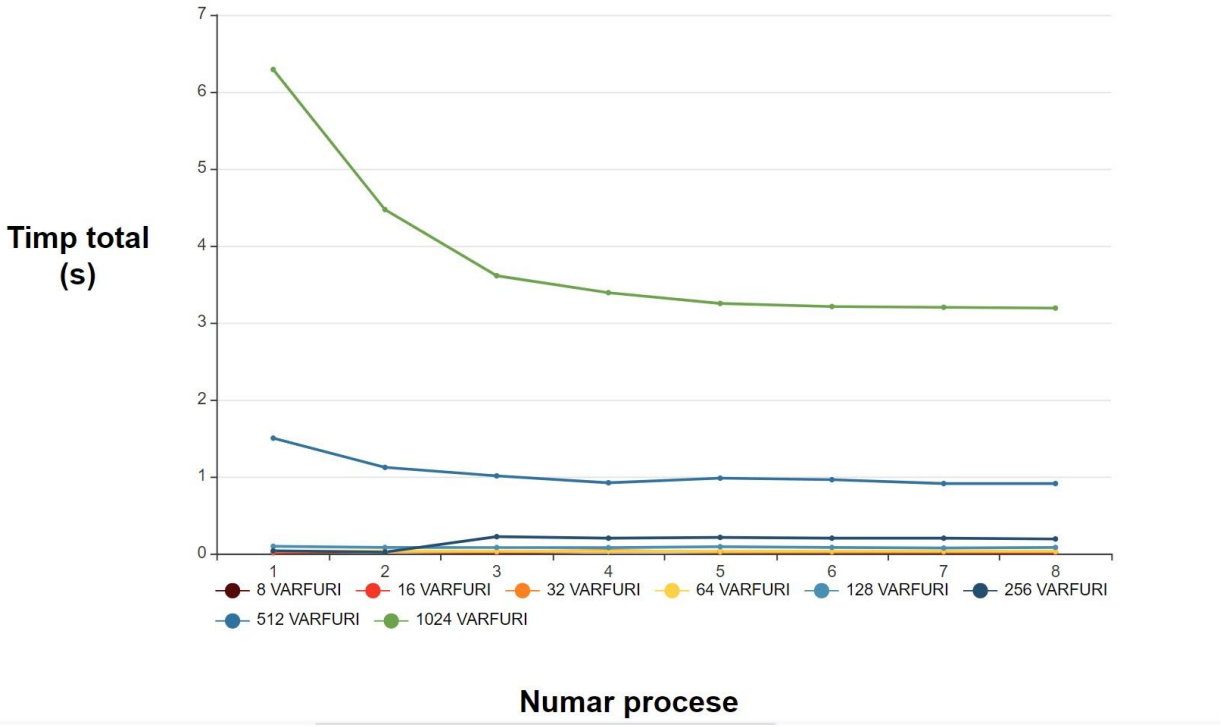

Figura 6 Reprezentarea pe chart a rezultatelor Timp Total/ Nr. Procese

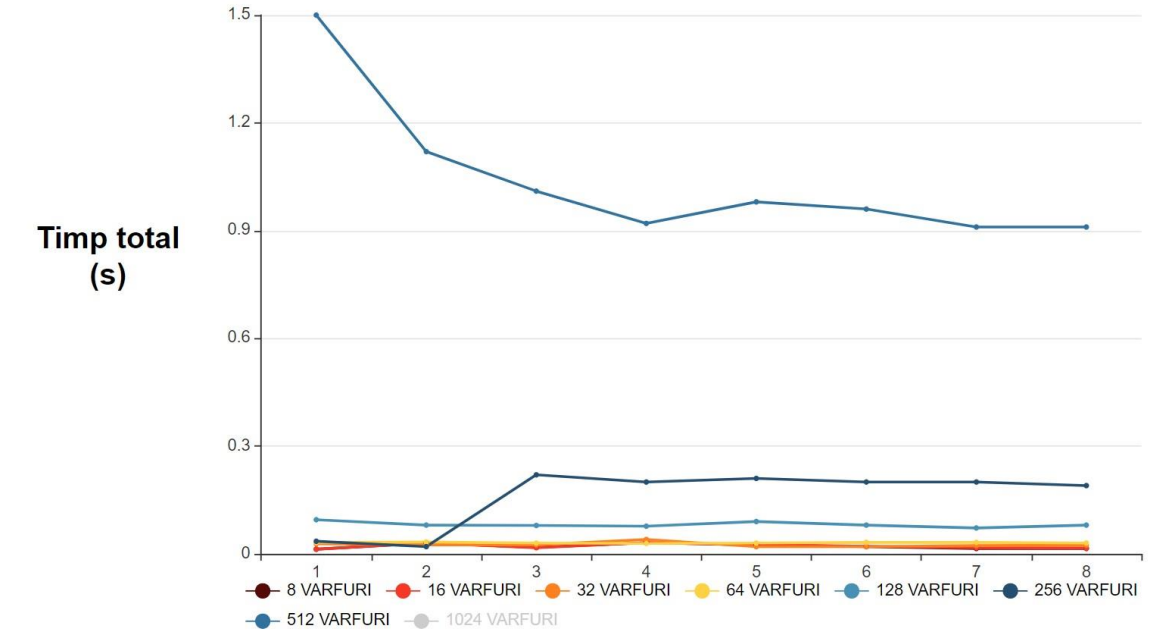

Figura 7 Chart-ul fara linia 1024 VARFURI

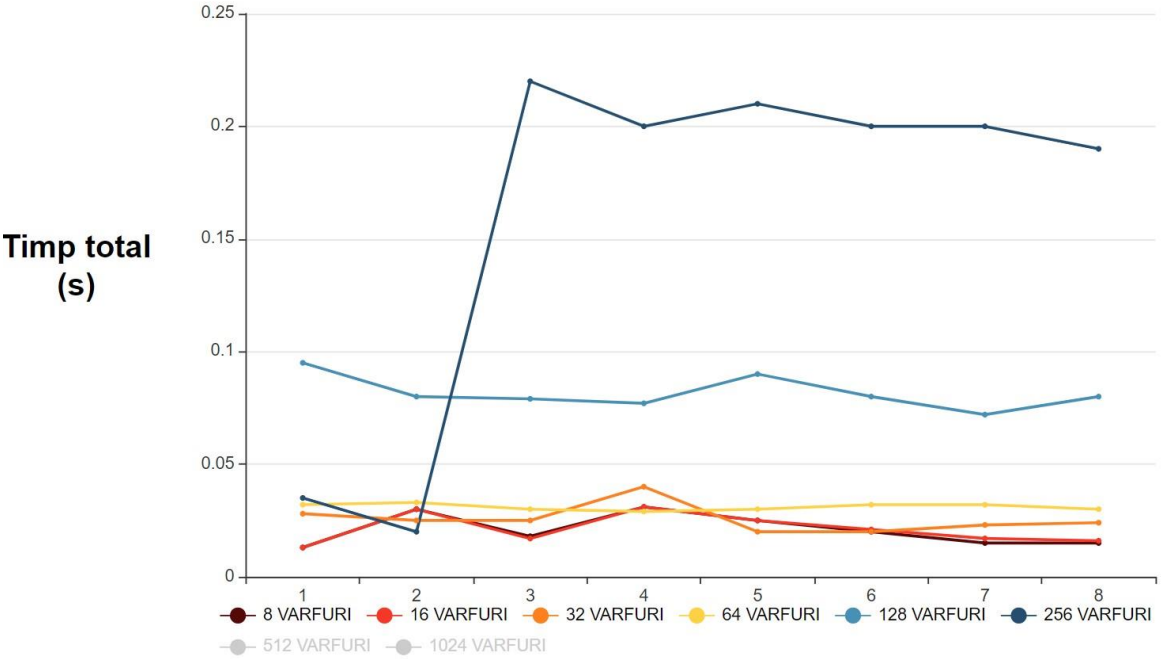

Figura 8 Chart-ul fara liniile 512 VARFURI si 1024 VARFURI

Din analiza rezultatelor putem observa ca la un numar mic de noduri, diferentele nu sunt foarte mari.Timpul total de executie a variat in general de la un numar de procese folosit la altul, iar in unele cazuri (la 16 varfuri), utilizarea MPI nu a fost eficienta pana ce nu s-au folosit 7 procese pentru paralizare.

Un rezultat surprinzator a fost eficienta a fix 2 procese pentru paralizare in cazul hipercubului cu 128 de noduri (timpul urmand sa creasca exponential de la 3 procese in sus).

La hipercuburile cu 512 si 1024 noduri, se observa eficientizarea timpului total de procesare printr-o curba asemanatoare in ambele cazuri. Prin urmare, utilizand mai multe procese, putem obtine o imbunatatire semnificativa a performantei atat timp cat dimensiunile datelor sunt suficient de mari.

De asemenea am mai realizat o comparatie asupra timpului de executie al algoritmului Dijkstra, reprezentat in figura 9 si in tabelul 2.

	8 VARFURI	16 VARFURI	32 VARFURI	64 VARFURI	128 VARFURI	256 VARFURI	512 VARFURI	1024 VARFURI
1	0.00021	0.0006	0.01	0.001	0.04	0.2	0.84	3.67
2	0.0007	0.001	0.007	0.007	0.02	0.1	0.46	1.95
3	0.0007	0.0015	0.005	0.004	0.02	0.08	0.31	1.247
4	0.0007	0.001	0.004	0.004	0.01	0.06	0.27	0.96
5	0.0009	0.001	0.004	0.005	0.01	0.06	0.28	1.3
6	0.0014	0.001	0.005	0.005	0.01	0.06	0.29	1.1
7	0.0015	0.0009	0.004	0.004	0.01	0.05	0.25	0.99
8	0.0015	0.0009	0.005	0.003	0.01	0.04	0.24	0.9

Tabelul 2 Rezultatele pentru timpul de executie al algoritmului per proces

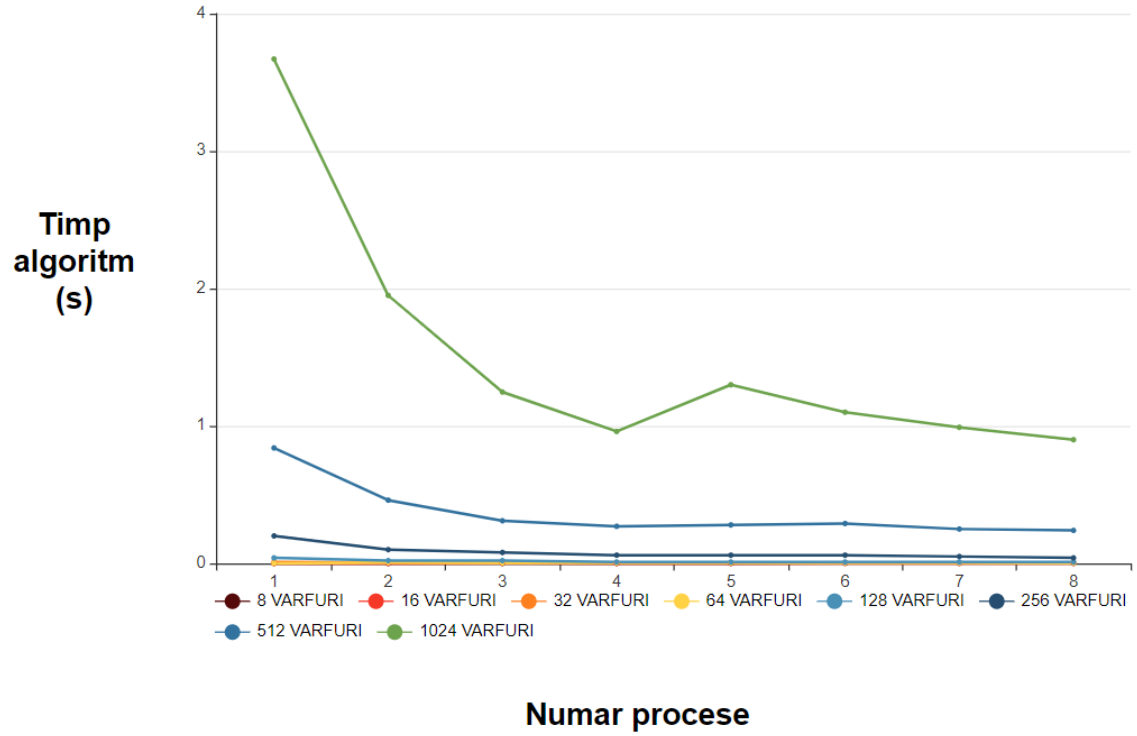

Figura 9 Analiza pe timp a algoritmului in functie de procese

Putem distinge din graficul alaturat, ca in aproape orice situatie, marirea numarului de procese folosite, a dus la un timp mai bun pentru aplicarea algoritmului Dijkstra.

Algoritmul pentru un hipercub incomplet

Un hipercub incomplet reprezinta un hibercub cu unul sau mai multe noduri lipsa.

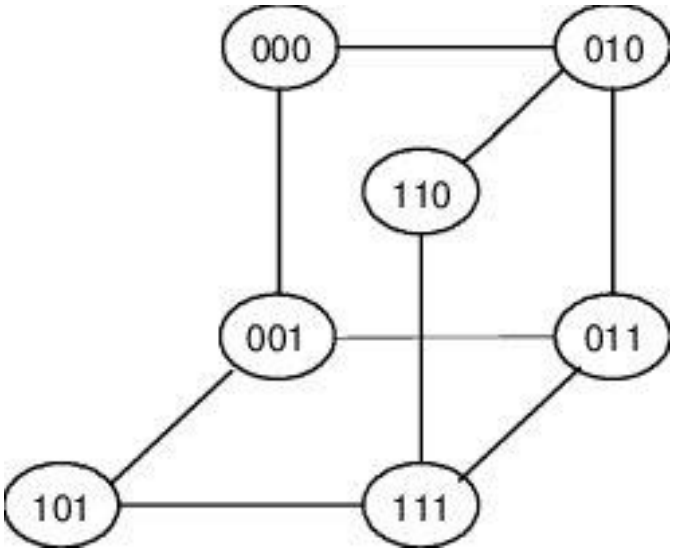

Figura 10 Hipercub incomplet

In continuare am realizat o comparatie a timpului total de executie pentru cateva hipercuburi incomplete cu 7, 63, 511, 1023 noduri. Comparatiile au fost realizate pentru timpul de executie in cazul a 8 procese, acestea se gasesc in figura 12 si tabelul 3.

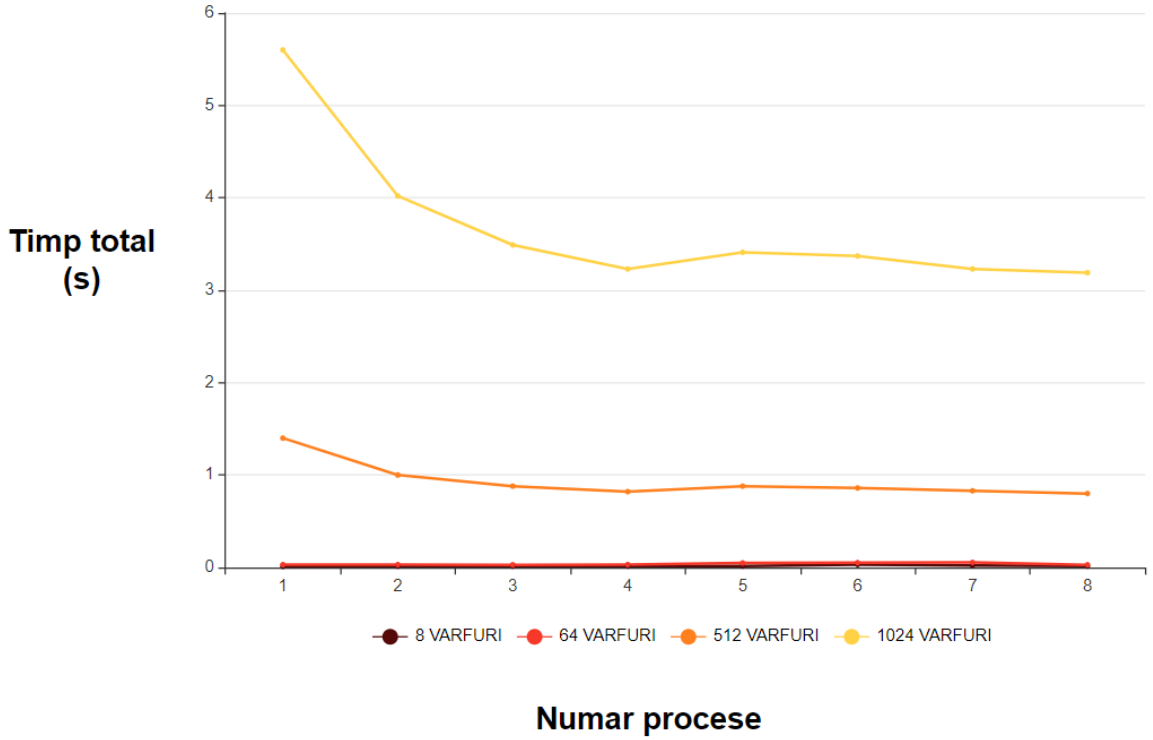

Figura 11 Timpul total pe nr. de procese pentru hipercubul incomplet

	8 VARFURI	64 VARFURI	512 VARFURI	1024 VARFURI
1	0.011	0.031	1.400	5.600
2	0.014	0.030	1.000	4.020
3	0.015	0.029	0.880	3.49
4	0.016	0.031	0.820	3.230
5	0.016	0.048	0.88	3.41
6	0.032	0.051	0.86	3.370
7	0.022	0.054	0.830	3.230
8	0.016	0.029	0.800	3.19

Tabelul 3 Timpul de executie pentru hipercubul incomplet/proces

Putem distinge din analiza rezultatelor ca si in cazul hipercubului incomplet, daca datele sunt consistente, se obtine o performanta mai buna prin paralelizare.