Detectia coliziunii dintre n obiecte sferice

Autor: Ilie Vasile-Vlad

Email: vasile.ilie4@student.usv.ro

Universitatea „Stefan cel Mare” din Suceava

Știința și Ingineria Calculatoarelor

Disciplina Algoritmi Paraleli Avansati

Abstract: Urmatoarea lucrare va prezenta un studiu de caz cu privire la proiectarea si rezultatele implementarii unui sistem de detectia a coliziunii dintre n obiecte sferice. Sistemul de detectie a coliziunii dintre n obiecte sferice este reprezentat printr-un mediu de comunicare paralel de tip cluster MPI in care fiecare proces va putea sa determine daca a avut loc o coliziune.

Cuvinte cheie: MPI, procesare paralela, coliziuni 3D.

1. Introducere

Implementarea unei solutii de detectie a coliziunilor intotdeauna a fost o provocare, in primul rand din cauza ca modul de solutionare a problemei detectiei coliziunilor depinde mult de formularea problemei dar mai ales de sistemul folosit pentru a studia cazul.

Prin definitie:

Detectia coliziunilor este o problema computationala de detectie a intersectiei dintre doua sau mai multe obiecte.[1]

Aceasta lucrare va efectua un studiu asupra unui sistem de coliziuni avand in cadrul sistemului n obiecte sferice care pot intra in coliziune. Dar de obicei in simularile fizice, experimente, cum ar fi jocul de biliard cu o formulare conceptuala referitoare la acesta, sunt folosite pentru a studia procesele fizice. In cadrul domeniilor fizicii, domeniul coliziunilor elastice si domeniul miscarilor corpurilor dure, fizica bilelor de biliard este bine inteleasa.

De exemplu in simularea complexa si aproape reala a unui joc de biliard, pentru calculul traiectoriilor, miscarilor precise si eventual a locurilor de asezare a bilelor dupa consumarea unui impuls primit de bilele jocului cu ajutorul unui tac, se va folosi un program care atunci cand va fi rulat pe un sistem de calcul va produce rezultate cu privire la aceste calcule anterior mentionate. Totusi pentru simulare cat mai reala este necesara o descriere

precisa a caracteristicilor fizice ale bilelor de biliard si a mesei de biliard, de asemenea si pozitiei initiale a bilelor.

Problema detectiei coliziunilor dintre n obiecte sferice este destul de recurenta in jocurile pe computer, unde exista astfel de cerinte pentru implementarea unor functionalitati de motor de fizica (physics engine), folosit pentru a simula interactiuni intre obiecte in diverse moduri, nu doar simple ciocniri cu ricosari e.g.: transformari ale obiectelor sau trecerea glontului prin peretele moale. Daca simularile pe calculator trebuie sa calculeze rezultate ale unor principii din fizica din lumea reala cat mai precis posibil, jocurile pe calculator, totusi, trebuie sa simuleze intr-un mod cat mai acceptabil aceste principii ale fizicii care determina ciocniri, ricosari si schimari in dinamica obiectelor: asezarea obiectelor, timpi pana la racirea impulsului avand in vedere ca rezultate sa fie obtinute in timp real si cat mai robust.

Unele sisteme de simulare a coliziunilor estimeaza momentul coliziunii prin interpolare liniara si pot sa vizualizeze pasii pana la coliziune si pot aplica legi naturale inspirate din fizica pentru a calcula coliziunile avand in vedere astfel si alte principii cum ar fi cat de moi sunt obiectele care vor intra in colizune sau cat de mare este forta de ricosare. Totusi trebuie avut in vedere ca astfel de calcule foarte granulare vor folosi procesorul de calcul intensiv creeand probleme de performanta sau de planificare a sarcinilor care simuleaza dinamica sistemului de coliziuni luand in considerare toate principiile de functionare ale acesteia.

Sunt frecvente studiile si chiar aplicatiile practice, implementari ale unori motoare de fizica, in care optimizarea computationala, cu metode de gasire a optimului cum ar fi metoda Gauss-Newton’s sau metoda Levenberg-Marquardt, este folosita pentru a determina cu precizie imbunatatita momente ale coliziunii cu perspectiva robusta a simularii si cu un model al sistemului bine definit matematic.

Ca urmare a unei coliziuni inelastice, stari speciale ale ricosarilor si ale asezarilor bilelor dupa consumarea impulsului pot sa apara, de exemplu, Open Dynamics Engine foloseste constrangeri pentru a simula acesta stari speciale.

Simulatorii de coliziuni fizice functioneaza de obicei in unul din doua moduri posibile:

- coliziunea este detectata a posteriori – dupa ce a avut loc, rezultand detectie discreta;
- coliziunea este detectata a priori – inainte ca aceasta sa aiba loc, rezultand detectie continua.

In cazul colizunii a posteriori, simularea fizica a coliziunilor este avansata cate un pas intr-un anumit interval de timp apoi se va verifica daca obiectele s-au intersectat sau se poate observat cat de aproape sunt acestea unul de celalalt astfel incat sa spune cat obiectele se vor intersecta. La fiecare pas de simulare, o lista de obiecte si pozitiile lor in cadrul sistemului este creata, dar pozitile acestora cat si traiectoriile sunt obtinute discret pana la coliziune. Aceasta metoda putem spune ca este a posteriori, pentru ca de obicei pierde perspectiva asupra momentului precis in care coliziunea a avut loc si prinde coliziune de abia dupa ce aceasta s-a intamplat.

Pe de alta parte in cazul metodei a priori de detectie a coliziunilor, algoritmul de detectie va putea prezice traiectoriile corpurilor fizice cu precizie. Instante de timp pana la coliziune vor fi calculate cu mare precizie iar corpurile fizice simulate nu se vor suprapune inainte de a obtine coliziunea, aceasta fiind vizualizata inainte sa se intample.

Exista avantaje si dezavantaje ale fiecarei dintre cele doua moduri de a formula un sistem de coliziuni. Se poate observa ca un sistem bazate pe o perspectiva a posteriori de a simula coliziunile dintre obiecte va putea sa ignore variabile ale sistemului care sunt prea apropiate de realitatea fizica, astfel incat o simpla lista cu obiecte introdusa intr-un algoritm iar acesta va produce o lista de obiecte care se intersecteaza. Elemente fizice cum ar fi forta de frecare, principii ale coliziunii elastice sau ale coliziunii plastice si a corpurilor care isi schimba forma nu trebuie avute in vedere atunci cand doar se determina instante ale sistemului in care obiectele sunt in starea de coliziune, acesta va fi si cazul acestei lucrari. In aceasta lucrare care are atasat un proiect cu MPI library se vor determina instante ale unui sistem avand n obiecte sferice, in care acestea sunt in situatia de coliziune intre ele. Pe langa aspectele domeniului fizicii, algoritmii a posteriori sunt cu o dimensiune mai simpli decat algortimii a priori, algoritmii a priori isi bazeaza simularea prin pasi in timp, luand in considerare momente temporale, pe cand algoritmii a posteriori genereaza instante ale sistemului determinand apoi stari ale obiectelor sistemului: in coliziune sau nu.

Dezavantajele unui sistem de detectie a coliziunilor care foloseste algoritmi a posteriori de determinare a coliziunilor vor avea probleme atunci cand trebuie decis cat de des trebui generata o noua instanta a starii sistemului, iar o alta problema este cea a obiectelor care se intersecteaza in cadrul instantei sistemului, detectia coliziunii fiind dupa ce acestea s-au atins deja. De asemenea, daca pasul discret de generare a coliziunii este prea mare, unele coliziuni pot ajunge sa nu fie detectate, rezultand astfel obiecte care trec unele prin altele atunci cand sistemul are si obiecte suficient de mici pentru ca asa ceva sa se intample.

Avantajele algoritmilor de detectie a priori a coliziunilor sunt fidelitatea traiectorie-coliziune mai buna si stabilitatea algoritmului. Totusi pentru implementarea unui astfel de algoritm se gaseste ca fiind dificila separarea dintre simularea dinamicii sistemului cu n obiecte sferice si algoritmul de detectie a coliziunilor in sine. Totusi pentru determinarea unei coliziuni inainte de a se intampla nu exista o formula clara, de obicei e necesar un model matematic si o regresie liniar sau chiar non-liniara.

Principala problema a sistemelor de detectie a coliziunii dintre n obiecte, sferice sau altfel de corpuri, o reprezinta problema ineficientei sistemului atunci cand aceasta detectie implica lucrul cu foarte multe obiecte si asta pentru ca nu sunt multe moduri de gasi o coliziune si din cauza calculelor foarte complicate care depind de forma obiectului.[3]

1 Algoritmi pentru detecția coliziunilor

Obiectivul algoritmilor de detectie a coliziunii il reprezinta gasirea acelor perechi dintr-un sistem care urmeaza sa se intersecteze. In motoarele de fizica (physics engines) utilizate in grafica generata pe computer aceste perechi sunt analizate mai departe astfel incat rezultatul algorimului de coliziune este procesat discret iar acesta poate fi trimis mai departe

printr-un pipeline de procesare pentru a obtine efectul corespunzator fizicii coliziunii motiv pentru care toti pasi de dinainte de coliziune cat si coliziunea in sine sunt analizati.

Memoization este o tehnica din algoritmica folosita si de algoritmii de detectie a coliziunii, i.e. rezultate calculelor anteriorare sunt pastrate si folosite in calculele care urmeaza pentru determinarea mai rapida a coliziunilor.[3]

2 Algoritmul Sweep and Prune

Acesta este un algoritm secvential, nu este gandit pentru lucrul in paralel datorita formularii matematice in care se opereaza pe multimi. Algoritmul lucreaza cu principiile impartirii in intervale a fiecarei axe prin care rezulta arii, volume sau hipervolume care cuprind perfect n corpuri dintr-o scena si principiile care presupun ca daca la un anumit moment doua corpuri care sunt cuprinse in intervale diferite pe axele sistemului se intersecteaza atunci este o probabilitate mare ca acestea sa se intersecteze si la urmatorul pas. De asemenea daca nu s-au intersectat intr-un pas anterior atunci nu o vor face nici la pasul urmator. Astfel problema este redusa la a urmari de la un cadru la altul ce intervale se intersecteaza. Dupa determinare a doua intervale care se intersecteaza se indentifica si corpurile care urmeaza sa-si depaseasca intervalul astfel desi initial problema este simplificat prin acest algoritm care implementeaza detectia grupand in cadre largi obiectele sistemului in care au loc coliziuni.[2]

3 Algoritmul MPI paralel

Acest algoritm este implementat in cadrul acestei lucrari. La baza algoritmului sta library-ul MPI care permite comunicare eficienta intre procese diferite care executa instructiunile pe nuclee procesor, procesoare sau masini de calcul diferite. Astfel fiecare obiect din sistemul dinamic in care este necesara detectia coliziunii este incapsulat intr-un proces MPI diferit care va putea detecta, folosind datele din domeniul de comunicare, o eventuala coliziune cu celelalte obiecte din sistem.

2. Materialele si metodele studiului

Pentru a crea un studiu pe sistemul de detecție a coliziunilor in primul rând a trebuit sa identific componentele acestui sistem si obiectivele apoi sa formulez algoritmic modul in care sunt determinate coliziunile dintre corpurile sistemului.

Problema tratata de acest studiu este cea a determinării coliziunilor dintre n obiecte sferice astfel componentele sistemului sunt:

- cele n obiecte sferice care au fost modelate folosind proprietățile:
    - Coordonatele carteziene ale centrului sferei pe axa x, axa y si axa z, fiind obiecte 3D;
    - Raza sferei.
- Spațiul 3D in care acestea vor avea o traiectorie sau in care se vor genera instanțe care conțin diferite pozitii ale acestor corpuri;
- Pozitiile acestor obiecte care vor fi folosite in calculul acestui algoritm.

Obiectivul unei solutii care simuleaza traiectoriile obiectelor sistemului este detectarea evenimentelor de coliziune in care doua obiecte se intersecteaza cazul discret sau urmeaza sa se atinga cazul continuu.

1. Message Passing Interface

MPI (Message Passing Interface) este un standard de comunicare prin mesaje proiectat de un grup de cercetători academici si din industria IT. Acesta standard rezolva problema comunicării in sisteme de calcul paralel, funcționează prin crearea unui comunicator, acesta reprezentând un context de comunicare care poate funcționa cu mai multe modele: unicast, multicast, broadcast. Standardul pune la dispoziția dezvoltatorilor si inginerilor o definiție a sintaxei si semanticii procedurilor library-ului central, acesta va putea fi folosit de o gama larga de utilizatori pentru a implementa programe in C, C++ sau Fortran care vor avea astfel funcționalitatea de comunicare interproces prin mesaje, In comparație cu comunicare printr-un pipe prin raw bytes, MPI are si semantica si sintaxa bine definite.

Prin urmare MPI va fi folosit ca si un protocol de comunicare pentru a implementa programe pentru calcul paralel. Printre scopurile MPI sunt performanta îmbunătățita, scalabilitate, portabilitate, pe lângă faptul ca poate fi folosit pentru comunicare point-to-point cat si comunicare distribuita si mai ales luând in considerare ca oferă un API cu protocol si semantica specificate prin standard. MPI încă este la acest moment modelul dominant utilizat in calculul de mare performanta. MPI a devenit un standard de facto pentru comunicarea intre procese care modelează împreuna un program paralel care este lansat in execuție pe un sistem cu memorie distribuita. API MPI se afla in layerele mai mari de 5 ale modelul de referința OSI, totuși implementarea utilizează concepte din toate layerele utilizând sockets si servicii ale Transmission Control Protocol.

Interfața MPI acoperă funcționalitățile de crearea a unei topologii virtuale prin conceptul de comunicator, sincronizare prin intermediul unui sistem de tipul token-ring si comunicarea intre procese care sunt asociate unor noduri/server/sistem de calcul diferite într-un mod independent de limbaj cu sintaxa specifica limbajului de programare.

Programele MPI lucreaza cu procese dar de obicei aceste procese sunt denumite procesoare datorita faptului ca exista o asociere foarte stransa intre proces si procesor, de obicei pentru performanta maxima fiecarui CPU ii va fi asociat un singur proces. Aceasta asignare a unui proces pe un CPU este efectuata de un executabil utilitar care este oferit impreuna cu cadrul de dezvoltare MPI (mpirun sau mpiexec).

MPI conceptualizeaza utilizarea protocolului de comunicare prin mesaje inter-proces creând context si aducand o mai buna intelegere a acelor caracteristici care ajuta un programator sa decida ce library sa foloseasca in implementarea sa de proiect MPI:

Domeniile de Comunicare (Comunicatorul)

Instanțele domeniilor de comunicare, *obiectele comunicator*, conectează grupuri de procese într-o sesiune MPI. Un comunicator agrega procesele aflate in execuție si atribuie fiecarui procesor un identificator si un rang astfel incat se creeaza o topologie ordonata. Domeniile de comunicare pot fi la randul lor partitionate utilizand comenzi MPI.

Comunicare intreproces de la un nod la altul

MPI in primul rand aduce un API care poate fi folosit pentru comunicare de la un nod procesor (proces) la alt nod procesor (proces). Pentru a realiza aceasta comunicare MPI implementeaza interfetele MPI_Send si MPI_Receive. Acest mod de operare, numit point-to-point este de obicei folosit pentru implemntarea unui antipattern de comunicare sau a unui

mod neplanificat de comunicare cum ar fi semnalizarea unor evenimente e.g.: o arhitectura de date paralela in care fiecare procesor muta regiuni de date specifice unor anumite procesoare de date intre pasii de calcul, sau o arhitectura master-slave in care master-ul trimite noi date la un slave ori de cate ori sarcina prioritare este completa.

Comunicare colectiva in cadrul domeniului

Functiile de comunicare colectiva implica comunicarea intre toate procesele dintr-un grup de procese. O functie caracteristica acestui mod de comunicare este apelul MPI_Bcast. Aceasta functie ia date de la un nod si le trimite la toate procesele din grupul de procese. Operatie inversa a functiei anterior mentionata este functia MPI_Reduce, care ia date de la toate procesele grupuilui is efectueaza o operatie de sumare care stocheaza toate rezultate intr-un singur nod de procesare. MPI_Reduce este o metoda folositoare la inceputul si la sfarsitul unui calcul distribuit, in care fiecare procesor opereaza pe o parte din date si apoi combina rezultatul.

Tipuri de date derivate (user defined data types)

Bibleoteca MPI ia în considerare tipul de date transferat acesta implica necesitatea specificarii tipului de date atât la nodul care trimite cât și la nodul care primește tipul de date. Pentru aceasta bibleoteca MPI are tipuri derivate și nu folosește tipuri fundamentale de date din limbajul în care se implementeaza procesorul. Motivația care sta la baza definirii unor noi tipuri de date este ca MPI intetioneaza sa suporte medii eterogene în care tipurile sunt reprezentate diferite de la un nod procesor la altul, de exemplu fiecare nod ar putea fi lansat în execuție pe sisteme de calcul cu arhitecturi diferite care au endianness diferit, în acest caz MPI poate efectua conversii de date prin propria implementare.

Implementari ale standardului MPI

Implementarea initiala a standardului a fost făcută de Argonne National Laboratory (ANL) si Mississippi State University și s-a numit **MPICH.** De asemenea IBM a avut o implementare, în trecut companiile care foloseau sistemele de calcul de tip supercomputer fie implementau propria lor versiune fie comparau versiune comerciala MPICH. Recent o implementare a standardului este oferita de **OPEN MPI,** acesta fiind un proiect care combina tehnologii și resurse din câteva alte proiecte FT-MPI, LA-MPI, LAM/MPI și PACX-MPI. OPEN MPI este folosit de primele 500 supercomputer, inclusiv de Raodrunner, acesta din urma fiind chiar cel mai rapid supercomputer intre anii 2008 și 2009 și apoi de K computer cel mai rapid supercomputer intre anii 2011-2012.[4]

3. Rezultate

Pentru a demonstra utilizarea bibliotecii MPI si pentru a alcătui un studiu de caz pe tema coliziunii celor n sfere s-a implementat un program care va fi executat cu mpirun, acesta va lansa un număr de procesoare specificat prin linia de comanda cu utilitarul mpirun.

Proiectul asociază fiecărui procesor o sferă cu dimensiunea și coordonatele poziției ei. Principiul după care funcționează implementarea este:

- se inițializează MPI – Message Passing Interface;
- la primul pas in algoritm se generează pozițiile inițiale ale sferelor de către doar unul dintre procesoare astfel încât generatorul de numere pseudo-random sa poate genera numere pseudo-aleatoare diferite intre ele, aceasta se poate doar secvential;

- programul care a generat pozițiile inițiale va trebui să împrăștie pozițiile astfel încât fiecare dintre celelalte procesoare sa primească o poziție, poziția inițiala a sferei reprezentate de acel procesor;
- fiecare procesor va modifica poziția sferei la fiecare pas al algoritmului astfel încât sfera va realiza o traiectorie in spațiul 3D:
    - având in vedere ca fiecare sfera are o traiectorie, sferele se pot intersecta realizând coliziuni, astfel la fiecare pas algoritmului va trebui sa verifice si daca sferele se intersectează;
- pentru a verifica daca sferele se intersectează la fiecare pas al algoritmului se va folosi un calcul matematic care va determina daca sferele doua cate doua sunt intersectate, astfel fiecare procesor va trebui sa aiba pozitia sa actualizata si pozitiile celorlalte sfere actualizate pentru aceasta se va folosi o procedura de tipul - aduna date -.
- la detecția unei coliziune de preferat ar fi sa se re-genereze aleator poziția sferei pentru a scoate sfera din intersectie si a reincepe o traiectorie;

Programul MPI

Pentru a putea folosi apelurile din biblioteca MPI un program trebuie sa initializeze mai întâi aceasta biblioteca folosind apelul:

```
int MPI_Init(int *argc, char ***argv);
```

Programul va trebui sa cunoască exact câți membri procesor are grupul de comunicare de aceea se va folosi subrutina MPI:

```
int MPI_comm_size(MPI_Comm comm, int* size);
```

pentru a determina mărimea grupului asociat cu comunicatorul. Numărul de membri procesor este și numărul de sfere din simularea sistemului de n sfere aflate în mișcare cu traiectorii în posibila intersectie. Numărul de procesoare este determinat de parametrul *-np* și poate fi de maxim numărul trimis prin parametrul de configurare *–host <hostname>:<number>* astfel comanda MPI cu care va fi lansata simularea poate fi de exemplu:

```
mpirun -np 100 –host ivyblue:200 run_collisions
```

Identificarea unui procesor în cadrul grupului comunicator lansat în execuție se va face în programul procesor cu ajutorul subrutinei:

```
int MPI_comm_rank(MPI_Comm comm, int* rank);
```

După ce datele despre mărimea grupului comunicator și rank-ul procesorului în care se afla contextul de execuție sunt luate de la API-ul MPI se va genera o lista cu dimensiunea fiecarei sfere și poziția initiala a acestora, pentru alcătuirea acestei liste se va genera aleator pentru fiecare variabile numere aleatoare folosind procedurile

pentru generare de numere aleatoare bazate pe time seed din limbajul C, biblioteca standard, introduse în proiect prin header-ul:

```
#include <cstdlib>
```

Toate apelurile către API-urile MPI – Message Passing Interface sunt introduse în cadrul implementarii programului prin header-ul

```
#include <mpi.h>
```

Pentru ca fiecare procesor sa primeasca poziția initiala a sferei și dimensiunea acestei pentru a simula traiectoria ei și pentru a detecta coliziune se va folosi apelul de API MPI:

```
/**
* \desc Sends data from one process to all other processes în a communicator
* \param[in] sendbuf address of send buffer (choice, significant only at root)
* \param[in] sendcount number of elements sent to each process (integer, significant only at root)
* \param[in] sendtype data type of send buffer elements (significant only at root) (handle)
* \param[in] recvcount number of elements în receive buffer (integer)
* \param[in] recvtype data type of receive buffer elements (handle)
* \param[in] root rank of sending process (integer)
* \param[in] comm communicator (handle)
* \param[out] recvbuf address of receive buffer (choice)
* \return error code like all the MPI subroutines
*/
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

acest apel de API realizeaza transferul de date de la un procesor la toate celalalte procesoare dintr-un grup de comunicare.

Fiecarei sfere i se va genera o traiectorie astfel:

- după ce fiecare procesor are poziția initiala a sferei corespunzatoare rank-ului sau, procesorul va intra într-o bucla de procesare;
- în cadrul bluclei de proceseare la fiecare pas de iteratie se va genera un nou pas în cadrul traiectoriei aleatoare a sferei, pasul ales de programul implementat pentru acest studiu de caz poate fi trimis ca parametru de intrare al comenzii *mpirun* astfel încât un pas mai mic va face simularea mai realistica iar un pas mai mare va creea instante ale dinamicii unui

sistem de n sfere cărora li se detecteaza coliziunile în timpul miscarii și în timp real. Exemplu de lansare în execuție a grupului de procesare asociat comunicatorului implicit MPI:

- *mpirun -np 10 --host ivyblue:20 run_collisions **12*** , unde 12.0 este marimea maxima a pasului algoritmului care va genera traiectoriile sferelor.

După ce sfera face un pas aleator urmând astfel o traiectorie în cadrul dinamicii sistemului de n sfere toate celelalte procesoare din cadrul grupului de comunicare vor trebui sa primeasca poziția actualizata a acesteia și implicit a celoralalte, astfel încât oricare procesor va trebui sa cunoască pozițiile tuturor celorlalte sfere în cadrul dinamicii sistemului. Pentru realizarea acestui transfer de date într-un mod cât mai modular și fără reimplementari se poate folosi un API pe care biblioteca MPI îl aduce:

/**

* \desc Gathers data from all tasks and distribute the combined data to all tasks

* \param[in] sendbuf starting address of send buffer (choice)

* \param[in] sendcount number of elements în send buffer (integer)

* \param[in] sendtype data type of send buffer elements (handle)

* \param[in] recvcount number of elements received from any process (integer)

* \param[in] recvtype data type of receive buffer elements (handle)

* \param[in] comm communicator (handle)

* \param[out] recvbuf address of receive buffer (choice)

* \return error value

*/

int **MPI_Allgather**(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);

Și asta pentru ca fiecare procesor va încerca sa determine dacă s-a întâmpla vreo coliziune între cele n sfere ale sistemului aflat în timpul unei simulari în timp real.

Astfel un procesor va efectua un calcul matematic folosind formula:

*Dacă **d1 <= r1 + r2** atunci cele doua obiecte se intersectează, unde d1 este distanța dintre originile a doua sfere luate pentru verificare O1(x1, y1) și O2(x2, y2) iar r1 este raza primei sfere, r2 raza celei de-a doua sfere.*

In cadrul sistemului de coliziuni traiectoria celor n sfere va fi generata aleator de fiecare procesor și fiecare procesor va primi noua poziție a sferelor, aceste date vor fi folosite de fiecare procesor pentru a determina dacă exista coliziuni. Programul se terimina după un numar de iteratii care va determina tot de un parametru de intrare pentru comanda mpirun.

4. Discuții

MPI trebui intializat și apoi dezinitializat

În cadrul unui procesor MPI apelurile MPI_Init și MPI_Finalize sunt pereche astfel încât în cazul în care unul dintre ele lipsește se va produce o eroare care va genera un mesaj de eroare ca cel de mai jos, am lăsat doar mesajul generic și textul specific pentru cazul în care MPI_Finalize lipsește:

mpirun has exited due to process rank 0 with PID 0 on

node ivyblue exiting improperly. There are three reasons this could occur:

2. this process called "init", but exited without calling "finalize".

By rule, all processes that call "init" MUST call "finalize" prior to

exiting or it will be considered an "abnormal termination"

Algoritmii MPI

Apelurile MPI_Scatter și MPI_Allgather pot fi implementate prin intermediul unor algoritmi care folosesc MPI_Bcast și MPI_Barrier, varianta cu aceste apeluri de API permite o particularizare a procesarii dar nu este niciodată mai eficiența poate doar în cazul în care se folosește în implementare chiar algoritmul care implementeaza MPI_Scatter sau MPI_Allgather. De exemplu aceeași situație se întâmpla și cu MPI_Bcast, acest API se poate implementa prin apeluri în bucla către API-ul MPI_Send și apoi API-ul MPI_Recv dar folosirea unei simple bucle care realizeaza trimiterea de la un singur nod la toate celelalte va fi sigur mai puțin eficiența decât implementarea pe care MPI_Bcast o folosește. MPI_Bcast folosește o structura arborescenta în care procesoarele colaboreaza pentru a realiza o difuzare astfel fiecare program preia o parte din sarcina.

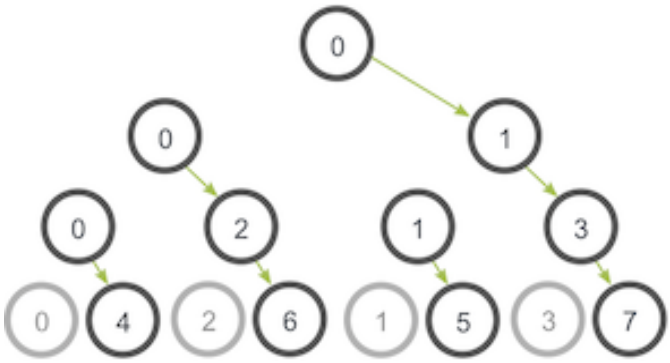

Figure 1: MPI_Bcast realizeaza o difuzare arborescenta

Prin urmare utlizarea completa a API-ului MPI aduce imbunatatiri in implementarea programelor procesor prin intermediul unor algoritmii eficienti de distribuire a informațiilor intre procesoare în cadrul grupului de comunicare.

Detectie a priori și detectie a posteriori

Calculul, care determina dacă o sfera s-a ciocnit cu una dintre celelalte sfere din sistemul dinamic de sfere ale caror traiectorii sunt simulate, este realizat după ce locatiile au fost colectate cu ajutorul API-ului MPI_Allgather de la celelalte procesoare din grupul asociat comunicatorului MPI. Astfel se poate spune ca detectia coliziunilor este realizata a posteriori după ce obiectele s-au intersectat în spațiul în care traiectoria este simulata. Pentru a realiza o detectie a priori a coliziunilor dintre aceste obiecte trebuie ca generarea urmatorului pas în cadrul traiectoriei unei sfere să se facă după ce s-a generat o alta anterioara cu verificarea la generarea pasului a posibilei coliziuni și astfel introducerea unui efect de coliziuni a priori înainte de intersectie. Cu ajutorul procesoarelor MPI se poate implementa un sistem a priori de detectie a coliziunilor dar acest sistem va lucra secvential:

- un procesor desemnat va începe generarea urmatorului pas în traiectorie cu verificarea posibilei sale coliziuni cu pozițiile anterioare ale sferelor;
- celelalte procesoare vor anunța ca un anumit obiect și-a găsit o noua poziție fără ca aceasta sa intre în coliziune cu pozitiile curente ale sferelor, apoi o alta sfera desemnata în ordine va alege o noua poziție fără sa intre în coliziune cu celelate sfere;
- fiecare procesor poate marca pozițiile care ar genera coliziune dar algoritmul ar detecta a priori coliziunile dintre cele n sfere ale sistemului.

Outputul algoritmului

@4 736 800us Sphere(0.02899, 10.88939, 11.83665, 10.95098) 2 colided with sphere(0.02793, 10.84630, 11.81369, 10.94095) 0.
@4 736 775us Sphere(0.02793, 10.84630, 11.81369, 10.94095) 0 colided with sphere(0.02899, 10.88939, 11.83665, 10.95098) 2.

Procesorul MPI care va detecta o coliziune intre doua sfere va printa un mesaj care va conține:

- Timestampul în microsecunde (us);
- Cele doua sfere care au intrat în coliziune
  - Dimensiunea (raza) și originea sferei în spațiul tridimensional;

- Rank-ul procesorului caruiai ii aparţine sfera cu primul procesor specificat primul în rând

Mesajul de coliziune apare de doua ori corespunzător celor doua procesoare care genereaza traiectoria celor doua sfere.

Exemplu lansare în execuţie:

mpirun -np 100 --host ivyblue:100 run_collisions 32 10

Codul sursa compilat cu comanda:

mpicxx collisions.c -o run_collisions

5. Concluzii

Concluzia studiului de caz prezentat în aceasta lucrare este programul care foloseşte bibilioteca MPI şi implementeaza detectia coliziunii dintre cele n sfere ale sistemului. Codul sursa al acestui program care demonstrează implementarea unui sistem de coliziune intre n sfere este disponibila la adresa: Collisions_dot_c

References:

[1] "Collision Detection - an overview | ScienceDirect Topics". www.sciencedirect.com. Retrieved 2020-06-12.

[2] Caldwell, Douglas R. (2005-08-29). Unlocking the Mysteries of the Bounding Box. US Army Engineer Research & Development Center, Topographic Engineering Center, Research Division, Information Generation and Management Branch. Archived from the original on 2012-07-28. Retrieved 2014-05-13.

[3] Anonymous. Collision_detection. https://en.wikipedia.org/wiki

[4] Anonymous. API Documentation static/docs. https://www.mpich.org