Infrastructuri integrate pentru procesare distribuită
Teză de doctorat – Rezumat

pentru obținerea titlului științific de doctor la
Universitatea Politehnica Timișoara
în domeniul de doctorat _CALCULATOARE ŞI TEHNOLOGIA INFORMATIEI_

autor ing. _Răzvan-Mihai ACIU_
conducător științific Prof.univ.dr.ing. _ Horia CIOCÂRLIE _

luna_09_ anul_2017_

1. **Introducere**

În acest capitol începem prin a descrie diferite domenii de aplicații care sunt în mod special adecvate pentru procesarea distribuită. Pentru fiecare domeniu sunt date exemple și se poate observa că puterea de computație necesară este cu mult mai mare decât ceea ce un calculator obișnuit poate oferi, deci este necesară folosirea unui sistem distribuit [1]. De exemplu, în domeniul randării 3D, filmul de desene animate ”The Croods” a necesitat în jur de 80 de milioane de ore de computație și 250TB capacitate de stocare. Pentru acest proiect, producătorul DreamWorks a folosit serviciile unui furnizor specializat de resurse computaționale, Cerelink, care a furnizat un cloud cu o viteză teoretică de vârf (configurația din 2010) de 172 teraflopi și 133 teraflopi de operare susținută. Acest cloud a folosit 1792 noduri computaționale cu un total de 14336 nuclee de procesoare Xeon 3.0 GHz, asamblate în 28 de rack-uri.

Pentru proiecte de interes public, s-a dezvoltat o nouă metodă de a obține putere computațională: rețelele de voluntari. În acest caz, persoanele interesate de un proiect pun la dispoziția acestuia resursele lor computaționale, creând astfel un rețea computațională de tip grid. Unul dintre cele mai cunoscute proiecte de acest tip [2] este folding@home, care în septembrie 2013 a folosit o rețea de voluntari de peste 266000 de calculatoare.

După aceste exemple teza continuă cu o clasificare a infrastructurilor distribuite, făcută din punctul de vedere al vitezei de acces la componentele sale și al facilităților de management. Se pot distinge 3 cazuri [3]: **sisteme de procesare masiv paralele** (interconexiuni fizice foarte puternice, care în multe cazuri transformă în mod efectiv acest sistem într-un supercalculator), **clustere, clouduri, ferme de calculatoare/servere** (calculatoare independente, de obicei în aceeași zonă geografică, comunicând prin rețea de înaltă viteză, care pot fi gestionate de la distanță și de asemenea pot avea software de virtualizare), **griduri** (calculatoare independente situate în arii geografice diferite, cu componente hardware și sisteme software foarte diferite, de obicei fără virtualizare sau management de la distanță).

O altă clasificare a resurselor computaționale ține cont de posibilitățile lor de computație. Putem avea **sisteme cu microprocesoare independente** (cum sunt nucleele CPU, capabile de a procesa orice algoritm și de a accesare orice parte a sistemului) și **sisteme cu unități computaționale dependente**, adaptate unor sarcini specifice (cum sunt GPU-urile, care pot fi foarte puternice în aplicații specializate dar care au mai multe restricții decât CPU-urile).

De asemenea am considerat cazurile posibile de implementare a unei aplicații: **secvențial** (toate sarcinile sunt procesate una după cealaltă), **cu fire de execuție** (unele sarcini

pot fi realizate simultan, folosind microprocesoarele locale), **distribuite** (unele sarcini pot fi realizate simultan, atât pe calculatorul local cât și pe cele de la distanță) [4]. Pentru fiecare caz s-au discutat necesitățile sale din perspectiva implementării. Se poate observa că aplicațiile distribuite sunt cele mai complexe, deoarece ele necesită implementarea unor aspecte cum ar fi descoperirea resurselor, serializare/deserializare, instalare de cod pe resursele disponibile, apel la distanță și managementul resurselor computaționale. Considerând complexitatea aplicațiilor distribuite, ne vom concentra pe propunerea unor noi modele, infrastructuri și unelte, cu scopul de a ușura dezvoltarea acestui tip de aplicații. Unele criterii precum **fiabilitate**, **abstractizare** și **ușurință în utilizare** vor trebui îndeplinite.

2. Motivația tezei, obiective și structură

Pentru început este discutat modelul **MapReduce** [5], preluat din limbajele funcționale cum ar fi Lisp și sunt subliniate unele dintre deficiențele sale atunci când este folosit pentru aplicații distribuite. Multe dintre aceste deficiențe provin din faptul că implementarea originală a acestui model computațional a fost una secvențială. De exemplu, în general el nu are un suport la nivel de limbaj ci este implementat folosind biblioteci, ceea ce face mai greu la compilare sau la execuție să se automatizeze anumite funcționalități, cum ar fi inițializarea managerului sau sincronizarea rezultatelor. MapReduce trebuie să fie extins cu concepte specifice procesării distribuite, cum ar fi managementul accesului concurent la resurse și apeluri asincrone. De asemenea trebuie optimizat pentru a ține cont de timpul de transfer al datelor între diverse resurse, ceea ce în anumite situații poate fi semnificativ. Din cauza acestor deficiențe, noi modele și concepte trebuie cercetate și dezvoltate pentru a fi cât mai adaptate domeniului procesării distribuite. Aceste modele trebuie să fie generale, simple, ele trebuie să asigure o înaltă performanță computațională și conceptele folosite trebuie să fie familiare domeniului aplicației.

În această teză propunem un nou model computațional, bazat pe MapReduce, în mod special creat pentru aplicațiile de procesare distribuită, care își propune să îndeplinească într-o mare măsură cât mai multe dintre aceste necesități. În capitolul 4 detaliem noul nostru model.

Bazată pe modelul propus, este dezvoltată o infrastructură care furnizează mijloacele de a-l utiliza. Infrastructura propusă trebuie să rezolve anumite cerințe, precum:trebuie să automatizeze pe cât posibil funcționalități standard (descoperirea resurselor, apel la distanță, serializare/deserializare, balansarea încărcării și recuperarea erorilor), toate resursele computaționale trebuie să fie abstractizate și folosite într-un mod unitar, interfața trebuie să fie simplă și de asemenea trebuie furnizate toate componentele necesare, precum serverul de la distanță care preia apelurile la distanță pentru resursele din rețea. Infrastructura propusă este descrisă în capitolul 5.

GPU-urile sunt din ce în ce mai mult o resursă computațională valoroasă. Pentru algoritmi masivi paraleli, un GPU poate oferi creșteri de viteză apreciabile, în multe cazuri reducând timpul de execuție de mai multe ori. În bioinformatică, folosind biblioteci puternic optimizate, algoritmi care folosesc la maxim GPU-ul au reușit să producă accelerări de până la 1000x [6]. GPU-uri precum AMD Radeon Fury X sau NVIDIA GeForce GTX Titan X sunt capabile de 8.6 TFLOPS, respectiv 7 TFLOPS FP32. Un Intel Xeon X7560 CPU este capabil de 72.51 GFLOPS FP64. Din aceste date rezultă că dacă algoritmul este masiv paralel și operațiile în FP32 sunt suficiente, un GPU poate furniza o performanță comparabilă cu mai multe CPU-uri de tip desktop.

Pentru a putea obține impresionantul lor număr de nuclee, GPU-urile trebuie să impună anumite limitări pentru unele funcționalități. Câteva dintre compromisurile GPU-urilor sunt listate mai jos:

- Un GPU nu poate accesa funcțiile sistemului de operare, precum cele pentru sistemul

de fișiere, rețea, etc.

- GPU-urile nu au o stivă. Compilatorului de OpenCL [7] îi este impus să expandeze inline tot codul funcțiilor în kernelul (codul care va fi rulat pe GPU) rezultat, deci el elimină toate apelurile de funcții. Aceasta limitează aplicabilitatea folosirii GPU-urilor doar la algoritmii nerecursivi. Acesta este unul dintre motivele pentru care mulți cercetători încearcă să găsească variante optimizat nerecursive ale algoritmilor recursivi [8].
- Până la OpenCL 2.0, GPU-urile au avut un domeniu diferit de memorie decât cel al CPU-ului. Domeniile diferite de memorie fac dificilă partajarea structurilor de date complexe care folosesc pointeri. Aceste structuri trebuie serializate cu pointerii convertiți la alte reprezentări, precum indecși sau identificatori, transferate la GPU și deserializate.
- Un nucleu de GPU nu are propria sa unitate de aducere și decodare a instrucțiunilor ci mai multe nuclee sunt grupate în grupuri de lucru (workgroups) care rulează folosind același pointer de instrucțiuni dar date diferite. Dacă un salt condiționat face ca nuclee diferite din cadrul aceluiași grup de lucru să aleagă căi de execuție diferite, unele dintre ele vor fi puse în stare de așteptare până când execuția va ajunge din nou la aceeași instrucțiune, uneori doar după ce alte nuclee și-au terminat deja sarcina. Acest fenomen se numește branch divergence.

Din cauza limitărilor de mai sus, nu este ușor să se integreze GPU-ul pentru sarcini de programare generale, în special când sunt necesari algoritmi complecși. O direcție de cercetare fundamentală a acestei teze este integrarea GPU-urilor în computațiile complexe, pe cât posibil abstractizându-le în aceeași manieră ca pe CPU-uri, astfel încât programatorul să nu trebuiască să le trateze în mod separat. În această direcție capitolele 6 și 7 propun doi noi algoritmi de a folosire a GPU-urilor pentru cazuri diferite.

3. Concepte și necesități ale procesării distribuite

În această secțiune discutăm unele aspecte de procesare distribuită, relevante pentru cercetarea noastră. Deoarece aceste aspecte sunt comune multor aplicații distribuite, o direcție în cercetarea noastră a fost identificarea în cadrul lor a unor tipare comune, care în majoritatea cazurilor pot fi automatizate. În această idee, modelul nostru și algoritmii propuși urmăresc să automatizeze cât mai mult posibil sarcinile necesare pentru aceste concepte și necesități.

Managementul rețelei se ocupă cu aspecte precum descoperirea resurselor [9], comunicarea și securitatea [10]. Pentru descoperirea resurselor discutăm folosirea mesajelor difuzate în toată rețeaua și folosirea unor servere care indexează resursele. În special pentru rețelele de tip grid, descoperirea resurselor trebuie să fie un proces continuu, pentru că în acest caz calculatoarele de la distanță pot oricând să intre sau să iasă din rețea. Pentru comunicare se poate folosi o paletă largă de metode și formate, începând cu codificări binare trimise ce folosesc protocoale specifice prin TCP/IP, pentru a asigura un cât mai mare flux de date și terminând cu formate text generice precum XML sau JSON transmise prin protocoale standardizate gen HTTP. Securitatea trebuie să acopere atât comunicațiile cât și accesul neautorizat la resurse.

Instalarea de cod la distanță poate fi realizată în mai multe feluri, începând cu instalarea manuală a aplicației pe toate resursele computaționale disponibile. Deoarece în cercetarea noastră dorim să automatizăm cât mai multe dintre sarcini, urmărim să dezvoltăm o metodă de instalare automată a codului la distanță. Pentru a face aceasta folosim un **server la distanță** [11], care primește codul aplicațiilor diverșilor clienți și îl instalează pe calculatorul propriu. Serverul de la distanță trebuie de asemenea să trateze aspecte precum dezinstalarea

codului, stocarea sa și actualizări, pentru a putea refolosi un cod deja instalat, dacă aplicație îl retrimite. În cazul unui cod de aplicație cu probleme, serverul de la distanță trebuie să fie capabil să termine o computație posibil infinită. Pentru rețelele de voluntari, serverul de la distanță trebuie de asemenea să asigure o politică specifică de folosire a resurselor, pentru a nu fi disturbată activitatea obișnuită a acelui calculator.

Codul instalat pe diverse resurse este accesat prin intermediul **apelurilor la distanță** [12]. Acestea sunt trimise de aplicație și fiecare invocație necesită trimitea propriilor parametri la resursa computațională, computația propriu-zisă, recepționarea rezultatelor și posibil recuperarea din erorile externe care pot să apară din folosirea rețelei. Ca o optimizare importantă, noi facem diferența între **datele globale**, comune tuturor invocațiilor, care trebuie trimise doar o singură dată la o resursă și **datele specifice unei invocații**, care se transmit separat pentru fiecare invocație.

Erorile externe își pot avea originea în rețea (de exemplu o problemă de infrastructură) sau dacă un calculator părăsește rețeaua. În asemenea cazuri este important în primul rând să detectăm asemenea erori, de exemplu folosind contoare de timp sau teste periodice ale resurselor și în al doilea rând este important să se facă recuperarea lor, retrimițând invocația eșuată la o altă resursă computațională.

Pe partea de aplicație, **managerul de invocații** [13] este răspunzător pentru descoperirea resurselor, colectarea tuturor invocațiilor de procesat într-o listă de așteptare, trimiterea lor asincronă [14] la resursele disponibile, recepționarea rezultatelor și de asemenea recuperarea din erorile externe. Fiecare rezultat al invocațiilor este ulterior procesat și, dacă este necesar, asamblat în rezultatul final.

4. Modelul de execuție la nivel de aplicație

În acest capitol prezentăm noul nostru model de procesare distribuită. Deoarece el se bazează pe modelul MapReduce, pentru început analizăm acest model din perspectiva procesării distribuite. Fiindcă MapReduce a fost folosit prima oară în computații secvențiale, unele presupuneri au fost (în mod implicit) făcute. Enumerăm mai jos aceste presupuneri și cum se schimbă ele în cazul procesării distribuite:

1. Toate invocațiile sunt sincrone, deci rezultatele sunt disponibile la revenirea din funcție - în procesarea distribuită managerul adaugă invocația în mod asincron într-o listă de așteptare și revine. Rezultatele invocațiilor pot să fie disponibile uneori doar mult mai târziu după ce s-a revenit din apelul către manager [15].
2. Toate datele rezidă în aceeași memorie (ținând cont de memoria cache și de swap-ul pe disc), deci ea poate fi accesată în aproximativ același timp - o aplicație de procesare distribuită trebuie să trimită/recepționeze date prin rețea, un proces mult mai lent decât accesul la memorie și care de asemenea necesită serializare.
3. Pe lângă problemele hardware, nu sunt alte surse de erori externe - într-o computație distribuită, chiar și pentru invocații legitime și un cod lipsit de probleme pot să apară în mod uzual erori de rețea sau schimbări ale statusului resurselor și toate acestea pot determina eșuarea unei computații.
4. Nu există acces concurent la resurse (data racing) - într-un mediu de procesare distribuită nu există o ordine deterministică în care rezultatele devin disponibile, deci trebuie să existe o metodă de a ordona/identifica rezultatul fiecărei invocații, pentru ca acesta si fie în mod corect procesat.

Din considerațiile de mai sus, aceste presupuneri care sunt adevărate într-un model secvențial, nu mai sunt valabile într-un mediu distribuit. Modelul nostru urmărește să definească semantici clare pentru toate aspectele implicate și să creeze un algoritm generic care să poată fi folosit eficient și în siguranță pentru aplicații de procesare distribuită.

Pentru a introduce modelul nostru, am implementat computația unui interval specific al setului Mandelbrot, un binecunoscut algoritm, foarte potrivit pentru procesare paralelă. Pentru acest exemplu am conceput un limbaj derivat din C++ care conține unele construcții de limbaj ce implementează modelul nostru. Acest exemplu este discutat și se dă o descriere funcțională conceptelor modelului.

Primul concept din modelul nostru este denumit **unit**. El este oarecum similar unei clase din limbajele orientate pe obiecte, dar cu unele diferențe importante:

- Nu poate avea atribute statice și nu poate accesa direct sau indirect atribute statice ale altor clase sau date globale - aceasta în mod efectiv izolează fiecare sarcină, eliminând nevoile de sincronizare între sarcini.
- Fiecare **unit** are atribuit automat un **Identificator Global Unic (IGU)**, folosit pentru a se verifica dacă el există deja pe calculatorul de la distanță.
- Constructorul unui **unit** primește datele constante globale, care sunt trimise doar o singură dată pentru fiecare calculator, indiferent de numărul de sarcini care sunt rulate pe acel calculator.
- Metoda *run* a unui **unit** este punctul de intrare în computație. Ea este apelată cu datele specifice fiecărei sarcini. Rezultatul metodei *run* este rezultatul computației și este serializat înapoi către aplicație.

Al doilea concept al modelului nostru se numește **with** și el implementează managerul de invocații. Acesta primește doi parametri, o **destinație** și un constructor de **unit** cu argumentele sale, la fel ca și construcția **new**. O destinație este un concept abstract pentru orice procesare care implică rezultatele sarcinilor. Ea poate fi simplu un vector în care sunt depuse toate rezultatele sau un obiect care implementează o interfață standard care permite apelul său de către manager pentru fiecare rezultat sosit. Constructorul de **unit** folosit ca al doilea argument a lui **with** are două funcții: definește o sarcină specifică și grupează datele globale pentru toate sarcinile.

La începutul construcției **with** este creat un nou manager. Managerul are mai multe funcții, printre care și descoperirea resurselor, instalarea codului la distanță, adăugarea invocațiilor în lista de așteptare, balansarea încărcării, apelurile la distanță, recuperarea din erori și aducerea rezultatelor.

Al treilea concept al modelului nostru este numit **run** și este folosit pentru a adăuga o nouă sarcină în lista de așteptare. El are două seturi de argumente: un identificator unic pentru fiecare sarcină, care este folosit pentru a identifica fiecare sarcină în interiorul destinației și datele specifice fiecărei sarcini. Apelul **run** este asincron și se revine imediat. Apelurile **run** au loc în interiorul corpului lui **with** (incluzând funcțiile apelate de din acesta) și ele pot să apară de mai multe ori.

La sfârșitul corpului lui **run** este inserat automat un punct de sincronizare. În acest punct se așteaptă pentru terminarea computării tuturor sarcinilor și a procesării lor în destinație. În acest fel, după ce corpul lui **run** se încheie, toate apelurile **run** sunt procesate.

Modelul propus a fost analizat din punct de vedere teoretic și s-au evidențiat situațiile optime în care el poate fi folosit. De asemenea am implementat modelul nostru ca o mașină virtuală, pentru a putea astfel implementa toate necesitățile la nivel de limbaj și l-am testat atât într-o rețea de calculatoare cât și nucleele unui CPU dintr-un singur calculator. În cazul rețelei de calculatoare am început cu un singur calculator și am adăugat pe rând noi calculatoare, unul câte unul, măsurând accelerarea. Pentru un număr mic de calculatoare, atunci când volumul de procesare pe fiecare calculator este mare, am obținut o accelerare apropiată de cazul optim. Pentru un număr mai mare de calculatoare, viteza de trafic și managementul rețelei, care sunt constante, au început să contribuie într-un procent mai semnificativ și deci accelerarea totală a fost mai mică. Acest rezultat este consistent cu comportamentul prezis de modelul nostru teoretic. Pentru testele folosind un singur CPU,

accelerarea a fost aproape de cazul optim. În privința distribuției încărcării, în această situație diferența maximă în procente față optimum a fost de 0.6%.

5. Infrastructură pentru procesarea distribuită a componentelor unei aplicații

Pentru a fi de importanță practică, modelul nostru trebuie să aibă o implementare sub forma unor construcții de limbaj, o infrastructură sau o bibliotecă. Pentru construcțiile de limbaj un limbaj de programare trebuie extins cu instrucțiunile necesare (unit, with, for). Dacă modelul este implementat ca o infrastructură, această extindere nu este necesară și limbajul poate rămâne neschimbat. În acest caz este posibil, datorită lipsei de expresivitate a limbajului țintă sau a unor capacități specifici, să fie implementată doar o infrastructură parțială sau mai greu de folosit, deoarece construcțiile specifice modelului sunt implementate folosind doar facilități standard de limbaj precum clasele și apelurile de metode.

Am ales să implementăm modelul nostru în Java, din cauza mai multor factori:

- Este un limbaj larg răspândit și deci infrastructura poate fi folosită de mai mulți programatori
- Codul binar Java este independent de sistemul de operare și de CPU, deci instalarea aplicației pe arhitecturile eterogene este mai ușoară.
- Java oferă capacități puternice de reflecție, ceea ce permite aplicației introspecția propriului cod. Această introspecție este folosită la computarea de către infrastructură a tuturor dependențelor unui unit care trebuie instalat la distanță și de asemenea este folosită pentru a verifica consistența codului aplicației din punctul de vedere al cerințelor modelului (de exemplu pentru a ne asigura că nu dintr-un unit nu se accesează date statice).

Infrastructura constă într-o bibliotecă pentru partea de aplicație și un server. Serverul este folosit pe calculatoarele de la distanță ca un mijloc de a recepționa, executa și returna rezultatelor codului aplicației instalat pe acesta. Biblioteca furnizează tot codul și structurile de date necesare pentru a implementa modelul în aplicație. Fiecare calculator din procesarea distribuită rulează un server dedicat. Numărul de conexiuni simultane pentru fiecare server este cel mult egal cu numărul resurselor sale computaționale. Un fir de execuție executor pentru procesarea sarcinilor este creat de manager doar daca un server cu conexiuni libere este disponibil. După ce o conexiune este făcută între server și executor, conexiunea este menținută deschisă până la completarea tuturor sarcinilor sau până la apariția unei excepții.

Pe partea de aplicație, infrastructura constă în unele definiții de clase și interfețe care implementează modelul nostru. Cele mai importante sunt:

- *public interface* ***Distributed***<*InitialData,TaskIdx,RunData,ReturnType*> - implementează conceptul **unit**. Codul client care implementează această interfață și toate dependențele sale vor fi instalate pe calculatoarele disponibile de către manager. **Distributed** are două metode:
  - *boolean* ***dInit****(final InitialData initialData)* - primește datele globale și este apelată o singură dată la crearea unei noi instanțe.
  - *ReturnType* ***dRun****(final TaskIdx taskIdx,RunData rData)* - este apelată pentru fiecare sarcină cu IGU pentru acea sarcină și datele sale specifice.
- *public interface* ***Destination***<*TaskIndex,ReturnType*> - definește o destinație pentru rezultatele sarcinii. Are o singură metodă:
  - *void* ***set****(TaskIndex index,ReturnType retData)* - când un nou rezultat sosește, este trimis de către manager împreună cu IGU său la această metodă. Implementarea sa poate de exemplu să pună rezultatul într-un vector, să-l scrie într-un fișier, etc.
- *public class* ***Scheduler***<*InitialData,TaskIndex,RunData,ReturnType*> -

implementează conceptul **with**. Pentru inițializările automate (precum descoperirea resurselor), **Scheduler** are de asemenea un constructor static. Câteva dintre metodele sale sunt:

- *public **Scheduler**(final Class<?> distrClass, final InitialData initialData, Destination<TaskIndex,ReturnType> dest)* - un constructor care primește o referință la o implementare a interfeței **Distributed**, care este **unit**-ul care va fi instalat la distanță, datele globale și o destinație care va primi rezultatele computațiilor.
- *public void **addJob**(TaskIndex index,RunData rData)* - adaugă în mod asincron o nouă sarcină în lista de așteptare. Sarcina este definită de IGU său și de datele sale specifice.
- *public void **waitForAll()*** - implementează punctul de sincronizare de la sfârșitul managerului, după ce toate sarcinile au fost adăugate. Această metodă așteaptă computarea tuturor rezultatelor și procesarea lor de către destinație.

Am testat infrastructura noastră în același fel în care am testat mașina virtuală pentru modelul propus, pentru a verifica astfel de asemenea consistența implementării modelului. Au fost două tipuri de teste, unul într-o rețea de calculatoare și altul folosind nucleele unui singur CPU. Pentru testele în rețea, în cazul unui număr mic de calculatoare, accelerarea obținută a fost aproape de optimum. Când am crescut numărul de calculatoare am obținut o accelerare procentuală mai mică, deoarece algoritmul se termină relativ repede. În acest caz factorii invarianți precum managementul firelor de execuție și a conexiunilor de rețea contează pentru o mai mare parte a timpului de execuție. Pentru testele folosind nucleele unui singur CPU, accelerarea la adăugarea unor noi nuclee a fost aproape de optimum, cu o diferență de maxim 0.09% față de optimum. Distribuția încărcării pe fiecare nucleu a avut o diferență procentuală de maxim 0.6% față de optimum.

6. Algoritm pentru execuția hibridă pe CPU și pe GPU

Acest nou algoritm permite e împărțire eficientă a segmentelor de cod între CPU și GPU. El colectează de la mai multe fire de execuție fără a stopa nucleele CPU datele pentru sarcinile ce vor fi executate pe GPU și computează datele colectate într-un singur pachet, pentru a utiliza la maxim GPU-ul. Algoritmul este în mod special eficient pentru aplicațiile cu un mare număr de fire de execuție, dar la care un fir de execuție individual nu furnizează suficiente date pentru folosirea eficientă a GPU-ului. Din cauza acestei împărțiri eficiente CPU/GPU, orice cod compatibil cu execuția pe GPU poate beneficia de accelerarea oferită de GPU-uri, chiar dacă acest cod este o componentă a unui algoritm care nu poate rula în întregime pe GPU.

Algoritmul propus rulează pentru aplicațiile paralele acumulând de la mai multe fire de execuție ale CPU-ului argumentele invocațiilor destinate a fi rulate pe GPU, execută toate aceste apeluri pe GPU într-un singur pachet și reia firele de execuție CPU pasându-le rezultatele computațiilor pe GPU. Când codul ajunge la o astfel de invocație, argumentele sale sunt colectate și acel fir de execuție este pus în așteptare până când rezultatele vor fi primite. Între timp nucleele CPU sunt folosite pentru a rula alte fire de execuție, deci CPU-ul este folosit în întregime.

Pentru a reduce încărcarea suplimentară ce apare la schimbarea de context între firele de execuție [16], când un nucleu CPU este folosit pentru mai multe fire de execuție (în multitasking preemptiv), am menținut numărul de fire de execuție la minimum și în interiorul fiecărui fir de execuție am folosit fibre/corutine (multitasking cooperativ). În acest fel putem rula aplicații masiv paralele, cu un paralelism simultan mult mai mare decât dacă am fi folosit doar fire de execuție și de asemenea cu o mult mai mică încărcare suplimentară la schimbarea

de context. În același timp, folosind schimbare de context explicită, multe nevoi de sincronizare sunt eliminate, deoarece programatorul cunoaște în orice moment care instrucțiuni se execută și într-un model cooperativ doar o singură fibră este activă la un moment dat. Termenul ”fir de execuție” va fi folosit pentru fire de execuție preemptive; pentru fire de execuție cooperative vom folosi termenul ”fibră”.

Fiecare kernel GPU are propriul său manager. Acest manager este responsabil să acumuleze invocațiile din mai multe fibre, să le ruleze într-un singur pachet pe GPU și să rezume fibrele originale folosind rezultatele computate. Corpul principal a algoritmului este prezentat mai jos:

- Pentru fiecare kernel GPU inițializează managerul său
- Pentru fiecare nucleu CPU creează un nou fir de execuție și îl adaugă într-o listă
- Folosește un iterator circular pentru a itera lista de fire de execuție într-o manieră circulară și ia în considerație firul de execuție curent ca fiind cel dat de această iterație
- Pentru fiecare sarcină paralelă:
    - Creează o nouă fibră în firul de execuție curent și o adaugă la lista de fibre a acestui fir de execuție
    - Avansează iteratorul de fire de execuție
- Când în interiorul unei fibre se ajunge la un apel al unei funcții destinate a fi rulată pe GPU, se colectează argumentele apelului, se suspendă fibra curentă și se continuă execuția cu următoarea fibră a acelui fir de execuție
- Când s-a terminat o iterare circulară completă în lista de fire de execuție, se execută pe GPU toate apelurile de funcții colectate, folosind kernel-urile lor specifice si apoi se reiau fibrele cu rezultatele primite de la GPU
- Se repetă până când nu mai sunt fibre rămase în nici un fir de execuție

După fiecare iterație este posibil ca unele dintre fibre să se termine. Acestea sunt șterse din lista de fibre. Fibrele rămase sunt acelea puse pe pauză în așteptarea rezultatelor de la GPU. După o iterație a firelor de execuție toate listele de apeluri sunt șterse, deci la următoarea iterație ele vor fi goale. Cu acest algoritm, la fiecare iterație toate apelurile către GPU sunt colectate și aceasta asigură folosirea eficientă a nucleelor GPU-ului, rulând simultan pe ele cât de mult posibile seturi de date. Procesul este repetat până când toate fibrele sunt șterse din liste. Dacă o listă de fibre devine goală, firul ei de execuție este de asemenea oprit. În algoritmul propus doar managerul are nevoie de sincronizare, deoarece mai multe fire de execuție îl pot accesa în mod concurent. În interiorul unui fir de execuție, fibrele sale nu necesită sincronizare deoarece ele rulează cooperativ. Cu acest algoritm singura încărcare suplimentară în interiorul unui singur fir de execuție este aceea necesară schimbării de context între fibre.

Performanța algoritmului a fost analizată și de asemenea s-au luat în considerare factori care contribuie la încărcarea suplimentară care apare în cazul execuției pe GPU, precum transferurile de memorie, diferențele între nucleele GPU și CPU [17] și numărul de fire de execuție. Pentru a testa algoritmul nostru l-am implementat în C++ și am făcut o serie de teste în care am modificat diverși factori precum volumul computațional necesitat de fiecare fir de execuție. De asemenea am modificat numărul de fire de execuție. Rezultatele testelor au arătat că atunci când sarcina a crescut de ~11.8 ori, timpul de execuție al aplicației în cazul în care am folosit GPU-ului a crescut doar cu ~20.1%, în timp ce același timp a crescut cu ~774% în cazul în care pentru computație s-a folosit doar CPU-ul. Pentru timpul de execuție al aplicației acesta este un real progres, obținut când un GPU a fost folosit pentru procesarea unor segmente de cod.

7. Traducerea la execuție a codului binar Java în OpenCL și execuția sa pe GPU

În această secțiune prezentăm un nou algoritm și bibliotecă pentru translatarea automată la execuției a codului binar al propriei aplicații în OpenCL și execuția codului rezultat pe GPU. Algoritmul este util în cazurile în care anumite module ale aplicației pot rula în întregime pe GPU. Toți pașii necesari pentru folosirea GPU-ului sunt automatizați: translatarea codului binar Java, serializarea structurilor de date ale aplicației într-un format corespunzător pentru GPU, managementul și comunicarea cu GPU-ul, preluarea rezultatelor și conversia lor înapoi la formatul aplicației. Biblioteca urmează modelul propus de noi și abstractizează folosirea CPU-ului și a GPU-ului. Atât CPU-urile cât și GPU-urile sunt folosite în mod automat atunci când sunt disponibile, fără nicio setare specifică în codul sursă.

Algoritmul folosește reflexia codului la execuție pentru a accesa codul binar al propriei aplicații, urmată de dezasamblare, analiză și generare de cod, pentru a genera OpenCL din codul relevant. În faza de execuție datele necesare sunt serializate în mod automat și transferate la GPU. La preluarea rezultatelor, algoritmul nostru convertește datele primite înapoi în structurile de date ale aplicației. Execuția pe GPU este abstractizată pe cât posibil. Același cod poate fi rulat pe CPU și pe GPU și aceasta permite o depanare ușoară și de asemenea permite folosirea CPU-ului ca rezervă atunci când nu este nici un GPU disponibil. Biblioteca generează cod OpenCL pentru un cod binar de complexitate medie, care poate conține structuri cu tipuri referință (instanțe de clase), excepții și managementul dinamic al memoriei. Cu aceste facilități noi creăm noi posibilități de a rula cod pe GPU, prin comparație cu biblioteci gen Aparapi [18]. Aceste biblioteci implementează în general doar un nivel mic de abstractizare peste OpenCL și unele apeluri API OpenCL trebuie făcute în mod explicit de către programator. Ca un dezavantaj în cazul nostru, deoarece programatorul nu trebuie să codeze într-un stil specific OpenCL, translatarea la OpenCL poate cauza o anumită pierdere de performanță.

Managerul de sarcini preia sarcinile furnizate de aplicație, le execută folosind GPU-ul și recepționează rezultatele. Când nou sarcini sunt preluate, managerul verifică prima oară dacă nu cumva s-a generat deja cod OpenCL pentru acestea. Când există deja codul OpenCL corespunzător, acesta va fi folosit, altfel codul aplicației pentru acea sarcină este încărcat prin mecanismele Java de reflecție. Codul binar este dezasamblat și codul OpenCL corespunzător este produs. Datele specifice sarcinilor sunt serializate. Pentru a realiza aceasta, câmpurile lor sunt analizate prin reflexie. Conținutul lor este copiat într-un buffer. Acesta va fi folosit ca o memorie dinamică globală în codul OpenCL. Serializarea și dezasamblarea atât pentru cod cât și pentru date sunt realizate într-o manieră recursivă. Ele continuă până când toate dependențele au fost procesate. Când GPU-ul termină de executat codul, datele (modificate) din memoria dinamică globală sunt trimise înapoi aplicației și deserializate, actualizând datele aplicației. În acest fel modificările făcute la execuția kernelului sunt transmise înapoi aplicației. De asemenea devin disponibile aplicației structurile alocate de către kernel.

Spațiul folosit pentru memoria dinamică globală, datele serializate și memoria alocată dinamic sunt furnizate într-un singur buffer numit memoria globală OpenCL. Referințele la date sunt translatate în deplasamente în această memorie globală. Un kernel folosește indexarea relativă la această memorie pentru a accesa datele, în același fel ca accesele într-un vector. Algoritmul nostru de serializare copiază automat tipurile primitive Java. Tipurile referință sunt procesate conform cu două situații: vectori și instanțe de clase. În ambele cazuri primul membru este un identificator unic pentru acel tip de vector sau de clasă. În cazul instanțelor de clase, valorile lor de tip primitiv sau referință sunt adăugate în ordinea dată de mecanismul de reflecție. Toate clasele sunt implementate folosind structuri OpenCL. Pentru vectori, lungimea urmează după identificatorul unic și după aceasta urmează elementele vectorului. Vectorii de tipuri primitive sunt de tipuri separate.

Există unele facilități JVM specifice care nu există în OpenCL, de exemplu generarea/interceptarea excepțiilor, alocare dinamică, recursivitate, apeluri de metode virtuale (metode non-final). Aceste facilități trebuie făcute disponibile printr-un nivel de abstractizare deasupra funcțiilor suportate de OpenCL. Facilități precum apelul din OpenCL al funcțiilor din calculatorul gazdă deocamdată nu există și nu pot fi implementate în niciun fel, cu excepția stopării execuției kernelului, apelul funcției din interiorul aplicației și execuția kernelului din punctul în care a fost stopat. Aceasta interzice folosirea funcțiilor sistemului de operare, efectiv blocând accesul la API-uri gen rețea sau sistem de fișiere.

Pentru generarea de cod, în primul pas este simulată execuția codului binar în mod linear (fără salturi sau bucle), folosind o stivă simbolică. În această stivă o celulă reprezintă un nod AST. Codul binar care operează pe stivă combină nodurile operand într-un nou nod rezultat, care de asemenea va fi depus pe stivă, simulând acțiunea operatorului. Diferite noduri sunt create și folosite de instrucțiuni care nu folosesc stiva, cum ar fi instrucțiunea *goto*. Când acest pas este finalizat, vom avea câte un AST distinct construit pentru fiecare metodă. Traversând aceste AST-uri generăm codul OpenCL. Pentru funcțiile supraîncărcate, pentru a putea genera identificatori C multipli pornind de la același identificator Java, am creat nou mecanism de denumire, deoarece mecanismul de denumire original JVM face uz de caractere care nu pot să apară în identificatorii valizi C. Acest algoritm de numire este de asemenea necesar pentru a face diferența între metode cu același nume din clase diferite. Deocamdată generatorul nostru nu implementează apel dinamic (metode virtuale) și de aceea putem folosi doar clase Java finale, în care compilatorul poate deduce metoda specifică apelată.

Pentru a implementa alocarea de memorie, care într-un sistem masiv paralel poate crea gâtuiri dacă nu este implementată corespunzător [19], am creat un alocator de memorie simplificat și rapid, pe aceleași principii ca în [20], care deocamdată este capabil doar să aloce memorie. Excepțiile au fost implementate folosind valorile returnate de funcții. Fiecare funcție returnează un întreg care este un deplasament în memoria globală. Pentru valoarea 0 (asociată cu pointerii Java null), se consideră că nu s-a generat excepție. Dacă se generează excepții, obiectele excepțiilor sunt alocate și se returnează indexul la excepția alocată. Deoarece valorile returnate de metode au fost folosite pentru excepții, metodele Java non-void își returnează valoarea printr-un parametru suplimentar, transmis prin adresă. Unele clase standard Java ca Object, Math și Integer sunt tratate ca intrinseci. Aceasta permite o mai bună optimizare a codului generat, care folosește funcții OpenCL predefinite. Mai multe metode din Math pot fi translatate direct in funcții native, pentru că ele deja sunt definite în OpenCL. Am implementat de asemenea o bibliotecă Java cu funcții OpenCL precum produsele vectoriale și scalare. Toate apelurile la această bibliotecă sunt translatate direct în corespondentele native OpenCL.

Am testat algoritmul nostru folosind diverse metrici, precum performanța și domeniul de cod JVM care poate fi translatat. Accelerarea maximă când a fost folosit un GPU a fost de 12.65x. De asemenea am comparat rezultatele noastre cu rezultatele obținute folosind biblioteca Aparapi cu un program similar de test. Versiunea Aparapi a obținut un timp mai bun comparat cu propria noastră bibliotecă, dar a fost capabilă să proceseze doar un domeniu mai limitat de date. De asemenea pentru Aparapi a fost necesar să se codeze aplicația de test într-o manieră care nu este specifică Java (fără clase) și a necesitat cod de legătură pentru a putea serializa și deserializa datele aplicației într-o formă convenabilă pentru OpenCL.

8. Concluzii

Considerăm următoarele aspecte ca fiind contribuțiile majore pe care le-am cercetat și le-am dezvoltat în această teză de doctorat.

Un model original de procesare distribuită care este dezvoltat pornind de la binecunoscutul model MapReduce și care îl extinde cu noi concepte necesare pentru procesarea paralelă și cea distribuită. Modelul nostru folosește doar trei concepte și este simple de învățat, în special pentru programatorii care au experiență în domeniul programării pe obiecte. Modelul abstractizează într-o manieră uniformă resurse computaționale diferite, precum CPU-uri, GPU-uri și rețele de calculatoare, ceea ce face posibil pentru aplicație folosirea arhitecturilor eterogene fără a fi nevoie să se scrie cod specific pentru fiecare tip de resursă. În modelul nostru multe aspecte care sunt necesare în procesarea distribuită sunt în mod implicit tratate automat și astfel aspecte precum descoperirea resurselor, instalarea de cod la distanță, serializarea datelor, distribuția sarcinilor, apelurile la distanță, recuperarea erorilor și sincronizarea sunt tratate automat, cu bune rezultate în cele mai multe cazuri. Modelul include de asemenea optimizări specifice, de exemplu tratează datele constante ca fiind cazuri speciale și acestea sunt trimise doar o singură dată la resursele computaționale. În special pentru calculatoarele din rețea sau pentru comunicarea pe magistrale lente, această optimizare poate aduce îmbunătățiri considerabile timpului de computare. Am implementat modelul nostru sub forma unei mașini virtuale și rezultatele de test arată că acesta este scalabil și în același timp este capabil să distribuie sarcinile în mod egal pe resursele computaționale disponibile.

O infrastructură Java care implementează modelul nostru, pentru a putea fi folosit în aplicațiile Java. Am reușit să implementăm cele mai multe dintre semanticile și constrângerile necesare doar folosind structuri de date și apeluri de metode, fără a face uz de alți pași de compilare, cum ar fi o fază de preprocesare. Am testat infrastructura noastră și aceasta a fost capabilă să abstractizeze atât calculatoarele din rețea cât și nucleele CPU ca resurse computaționale și de asemenea a automatizat unele aspecte din procesarea distribuită precum descoperirea resurselor, instalarea de cod la distanță și apelurile la distanță. Rezultatele testelor pentru infrastructura noastră au arătat că ea este scalabilă atât sub aspectul calculatoarele din rețea cât și al nucleelor CPU și că managerul a reușit să distribuie sarcinile în mod uniform pe toate aceste resurse.

Un algoritm de execuție cooperativă CPU-GPU, potrivit pentru sarcini complexe care nu se pot computa doar pe GPU, de exemplu dacă ele conțin apeluri de intrare/ieșire. Algoritmul nostru colectează din mai multe fire de execuție CPU părțile din computație destinate a fi executate pe GPU și le rulează pe GPU într-un singur pachet. Pentru o performanță optimală, acest algoritm folosește o combinație de fire de execuție și fibre, ceea ce reduce timpul de comutație între firele de execuție și de asemenea permite crearea de mii de sarcini simultane, implementate ca fibre, ceea ce ne permite să folosim la maxim toate nucleele GPU. Algoritmul a fost implementat ca o bibliotecă C++ și aceasta este aplicabilă într-un mod facil la aplicațiile paralele existente. Rezultatele practice obținute cu biblioteca noastră au arătat o accelerare semnificativă față de cazul în care s-a folosit doar CPU-ul.

Un algoritm și bibliotecă pentru translatarea codului binar Java în OpenCL și execuția acestuia pe GPU. Biblioteca de asemenea tratează în mod automat sarcini precum serializarea/deserializarea datelor, comunicarea și sincronizarea cu GPU-ul. Algoritmul poate translata cod care folosește clase, tipuri referință, alocare dinamică de memorie și tratarea excepțiilor (dar deocamdată fără apeluri virtuale și recursivitate), ceea ce este o îmbunătățire considerabilă față de abordările existente. Am integrat biblioteca noastră cu modelul propus și am furnizat un nivel de abstractizare peste OpenCL, ceea ce permite folosirea într-o manieră abstractă atât a GPU-urilor cât și a CPU-urilor, fără nicio modificare în codul sursă. Rezultatele practice au arătat îmbunătățiri de peste zece ori a accelerării vitezei de execuție, atunci când a fost folosit un GPU.

Bibliografie

[1] A. S. Tanenbaum, M. Van Steen, “Distributed Systems – Principles And Paradigms”, 2002
[2] Berkeley Open Infrastructure for Network Computing (BOINC) Project, Berkeley University, http://boinc.berkeley.edu/
[3] J. Dongarra, T. Sterling, H. Simon, E. Strohmaier, "High performance computing: Clusters, constellations, MPPs, and future directions", “Lawrence Berkeley National Laboratory”, 2003
[4] G. R. Andrews, "Foundations of multithreaded, parallel, and distributed programming", Pearson, 1999
[5] J. Dean, S. Ghemawat, “MapReduce: simplified data processing on large clusters”, Commun. ACM 51, 2008
[6] L. Dematté, D. Prandi, “GPU computing for systems biology”, Briefings in Bioinformatics, Vol. 11, No. 3, pp. 323-333, 2010
[7] Khronos Group, “The open standard for parallel programming of heterogeneous systems”, https://www.khronos.org/opencl/, access time: 14.07.2015
[8] Y. Ke, H. Bingsheng, L. Qiong, V.S. Pedro, S. Jiaoying, “Stack-based parallel recursion on graphics processors”, 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2009
[9] B.S. Murugan, D. Lopez, “A Survey of Resource Discovery Approaches in Distributed Computing Environment”, International Journal of Computer Applications Volume 22, No. 9, 2011
[10] K. Popović, Z. Hocenski, “Cloud computing security issues and challenges”, Proceedings of the 33rd International Convention, Croatia 2010
[11] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, V. S. Pande, "Folding@home: Lessons From Eight Years of Volunteer Distributed Computing", IEEE International Symposium on Parallel & Distributed Processing, 2009
[12] V. Narvaez, “Distributed Objects and Remote Invocation”, Addison Wesley 2010
[13] J. Celaya, U. Arronategui, “Distributed Scheduler of Workflows with Deadlines in a P2P Desktop Grid”, Conference on Parallel, Distributed and Network-Based Processing (PDP), Pisa 2010
[14] M. Malawski, T. Bartyński, M. Bubak, “Invocation of operations from script-based Grid applications”, Future Generation Computer Systems, Volume 26, Issue 1, Pages 138–146, 2010
[15] E. B. Johnsen, O. Owe, "An Asynchronous Communication Model for Distributed Concurrent Objects", Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM 2004), IEEE press, Sept. 2004
[16] T. Usui, R. Behrends, J. Evans, Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency", Journal of Parallel and Distributed Computing, February 2010
[17] K. P. Raphael, “Multi-core architectures and their software landscape”, Computing Science Handbook, Vol. 1, Chap. 35, 2013
[18] AMD, “Aparapi”, https://github.com/aparapi/aparapi, access time: 16.07.2015
[19] M. Steinberger, M. Kenzel, B. Kainz, D. Schmalstieg, “ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU, Innovative Parallel Computing” (InPar), San Jose, California, U.S., 2012
[20] C. Hong, D. Chen, W. Chen, W. Zheng, H. Lin, “MapCG: Writing Parallel Program Portable between CPU and GPU”, Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, Vienna, Austria, 2010