OPTIMIZAREA PERFORMANȚEI ȘI A CONSUMULUI DE ENERGIE PRIN LOCALIZARE ECHILIBRATĂ A DATELOR LA EXECUȚIA APLICAȚIILOR PARALELE PE SISTEME NUMA

Teză de doctorat – Rezumat

pentru obținerea titlului științific de doctor la
Universitatea Politehnica Timișoara
în domeniul de doctorat Calculatoare și Tehnologia Informației

autor ing. Știrb Iulia
conducător științific Prof.univ.dr.ing. Horia CIOCÂRLIE

Cuprins:

1. Introducere
2. Stadiul actual al cercetărilor în domeniile abordate în teză
3. Descrierea algoritmilor NUMA-BTLP și NUMA-BTDM
4. Rezultate experimentale
5. Concluzii și contribuții personale

Conținut rezumat:

1. Introducere

Motivația acestei teze a fost să realizez un algoritm numit *NUMA-BTLP* [1], care asignează la compilare câte un tip fiecărui fir de execuție din codul de intrare, clasificarea firelor fiind bazată pe criterii statice pe care le-am definit în teza, și un alt algoritm numit *NUMA-BTDM* [2] care mapează firele de execuție (prin mapare se stabilesc core-urile pe care firele de execuție vor rula) la compilare în funcție de tipul acestora, având ca scop îmbunătățirea localizării echilibrate a datelor pe sisteme NUMA. *NUMA-BTDM* [2] ține cont de comportamentul static al codului în realizarea mapării și elimină dezavantajele mapării dinamice (timp de execuție și consum de energie în plus la rulare) și câteva dezavantaje importante ale mapării statice: impredictibilitatea la compilare a numărului de fire de execuție și a latențelor operațiilor cu memoria.

În identificarea tipului unui fir de execuție m-am bazat pe mai multe considerente printre care și următoarele:

1. firele de execuție care nu au dependențe de date cu nici un fir creat pe același nivel în ierarhia de generare a firelor de execuție, se consideră a fii de tipul *autonom*. Firele care sunt create de firul principal execută de obicei procesări independente, caz în care se consideră a fii de tipul *autonom*;
2. firele de execuție între care există dependențe de date, cum pot fi, de exemplu, cele create într-o buclă, sunt considerate ca fiind de tipul *alăturat*;

3. firele care nu necesită execuție imediată sunt considerate a fi de tipul *amânat,* aceste fire regăsindu-se de obicei printre ultimele create într-o funcție. Acestea au proprietatea că au dependențe de date doar cu firul generator.

Scopul cercetării este optimizarea atât a timpului de execuție cât și a consumului de energie pentru aplicațiile paralele multi-threading C/C++ care utilizează paralelism la nivel de task obținut cu ajutorul bibliotecii ***PThreads*** [3] și pot utiliza și paralelism la nivel de buclă, prin optimizarea localizării echilibrate a datelor la rularea pe sisteme Non-Uniform Memory Access (NUMA) (relația între firele de execuție și task-uri este de unu-la-unu în cazul modelului de calcul paralel POSIX utilizat de ***PThreads*** [3]). Optimizarea aplicațiilor paralele este realizată prin aplicarea, la compilare, asupra codului de intrare în reprezentare intermediară, a celor doi algoritmi prezentați în lucrare care au fost implementați în compilatorul *Low-Level Virtual Machine* (LLVM) [4].

Deși o mapare la compilare prezintă în general anumite inexactități datorate schimbărilor comportamentului dinamic la execuție, aceste inexactități sunt în mare măsură eliminate în cazul celor doi algoritmi propuși prin elementul de noutate adus de aceștia și anume inserarea la compilare, după fiecare apel ***pthread_create***, de creare a unui fir de execuție, un apel ***pthread_setaffinity_np***, de setare a core-urilor pe care firul de execuție va rula. Astfel, indiferent dacă firul respectiv va fii sau nu creat în timpul execuției (aspect care ține de comportamentul dinamic al aplicației), este garantată maparea eficientă a acestuia la compilare, doarece această mapare ține cont doar de procesările efectuate de fire în funcția atașată fiecăruia și determinate prin analiză statică. O mapare eficientă a firelor de execuție optimizează localizarea echilibrată a datelor la rularea aplicațiilor paralele pe sisteme NUMA, ceea ce conduce la creșterea performanței execuției și la scăderea consumului de energie.

Analiza care a dus la decizia de a implementa algoritmul *NUMA-BTDM* [2] în LLVM [4] a avut ca punct de pornire utilitatea practică a acestui algoritm. Dacă algoritmul ar fi fost implementat in biblioteca de calcul paralel ***PThreads*** atunci utilizatorii acestei biblioteci ar fi putut apela algoritmul de mapare *NUMA-BTDM* [2] în codul pe care îl dezvoltă. Dar apelul acestui algoritm necesită în prealabil asignarea automată a câte unui tip fiecărui fir de execuție pe baza caracteristicilor statice ale codului și anume: (1) categoria firelor de execuție autonome, (2) categoria firelor de execuție alăturate ca *distanță NUMA* (adică datele se obțin din două surse – memorii – aflate în apropiere sau aceleași) față de firul de execuție generator sau față de alte fire și (3) categoria firelor de execuție amânate ca *timp NUMA* (timpul de access la date poate fi mai mare) raportat la firul generator. Fiecare categorie are la bază mai multe criterii care țin de caracteristicile statice ale codului. Analiza neautomată (de către utilizator) a îndeplinirii unui criteriu necesită cunoștințe avansate de management intern al memoriei în sistemele NUMA dar și de hardware și sisteme de operare în general, motiv pentru care este algoritmul *NUMA-BTDM* [2] nu a fost implementat în ***PThreads*** [3], ci în LLVM [4], unde a putut fi implementată și analiza de clasificare a firelor a algoritmului *NUMA-BTLP* [1].

2. Stadiul actual al cercetărilor în domeniile abordate în teză

Acest capitol începe cu descrierea caracteristicile sistemelor NUMA care determină dezvoltarea de algoritmi de mapare a firelor de execuție. Acestea sunt:

- rapiditatea de furnizare a datelor datorită structurării memoriei pe mai multe nivele de cache privat și cache comun precum și existenței mai multor controlere de memorie [5], care contribuie la minimizarea latenței operațiilor de aducere a datelor din memorie atunci când datele sunt plasate optim în memorie [6] (prin optim se întelege faptul că datele sunt gestionate de controlerul potrivit)

- timpii de acces la memorie sunt inegali [5], accesele locale (accesul datelor din același nod NUMA poartă numele de *acces local*), fiind mai rapide decât *accesele la distanță* (accesul datelor din memoria asociată unui alt nod NUMA este denumit *acces la distanță*) [7]

Prin avantajele de mai sus, sistemele NUMA oferă posibilitatea optimizării acceselor la memorie cu ajutorul algoritmilor de mapare care urmăresc îmbunătățirea localizării datelor.

Datorită faptului că algoritmul de mapare propus în această lucrare este static, am analizat și prezentat aspectele care îngreunează acuratețea predicțiilor la compilare ale algoritmilor de mapare asupra comportamentului dinamic al programelor paralele. Aceste aspecte sunt [8]:

- nu toate caracteristicile execuției sunt cunoscute (comportament dinamic)
- timpul de acces al datelor volatile și non-volatile este stocastic și nedeterministic
- existența altor programe care rulează în paralel pe alte core-uri sau pe aceleași core-uri ca programul paralel, utilizând resursele de memorie comune și accentuând caracterul nedeterministic al execuție programului paralel

De asemenea, au fost prezentate tiparele de mapare bazate pe *politica de alocare* în care fiecare fir de execuție rămâne asignat pe core-ul inițial până când își încheie execuția și *politica de migrare* în care firele de execuție sunt migrate de pe un core pe altul în timpul execuției în funcție de comunicarea dintre acestea la momentul anterior migrării, ambele tipare de mapare fiind aplicate mapării dinamice.

Toate aceste operații de mapare sau migrare sunt consumatoare de resurse, motiv pentru care maparea firelor de execuție nu ar trebui realizată la execuție deoarece o mapare în această fază ar induce considerabil mai mult consum de energie datorat *reasignării firelor de execuție* [9].

Pentru a determina *maparea/migrarea* optimă a firelor de execuție (fie aceasta statică sau dinamică) sunt necesare în prealabil două informații:

- modul în care firele de execuție accesează datele utilizate în comun și de alte fire, informație care este de cele mai multe ori reprezentată printr-o matrice de comunicare în care elementul (i,j) reprezintă cantitatea de comunicare (de date) dintre firele de execuție a căror identificatori sunt i și j
- informații legate de arhitectura hardware pe care rulează programul paralel, cum ar fi numărul de procesoare, core-uri și nivele de cache (aceste informații pot fi furnizate de utilitarul ***hwloc*** [10])

În acest capitol au mai fost prezentate optimizări software care eficientizează execuția programelor paralele pe sistemele NUMA, precum șioptimizări ale compilatorului LLVM [4] prin care se realizează acest lucru. Câteva din optimizările care îmbunătățesc paralelizarea automată prin reducerea timpului scurs cu gestiunea task-urilor sunt următoarele [11]: ***extinderea scalarilor*** (***scalar expansion***: cea mai eficientă tehnică în acest sens, implică convertirea datelor scalare în tablou uni sau bidimensional), ***reducerea prin substituție*** (a doua cea mai eficientă în acest sens, implicând substituție și apoi reducere), ***substituție recurentă***, ***eliminarea variabilelor de inducție*** (transformarea codului așa încât corpul buclei să nu depindă de variabila de inducție a acesteia), ***substituție succesivă*** (substituirea valorilor în aceeași manieră în care se rezolvă un sistem de ecuații liniare) și ***interschimbarea buclelor***. ***Fuziunea buclelor*** contribuie la optimizarea paralelismului la nivel de buclă prin diminuarea numărului total de iterații ale buclelor, iterații executate în paralel. Am detaliat în teză această optimizare, pe care am implementat-o în compilatorul ***LLVM*** [4] anterior programului de doctorat, deoarece aceasta poate fi utilizată împreună cu algoritmii propuși în teză la optimizarea programelor paralele care conțin atât paralelism la nivel de task (optimizat prin algoritmii propuși), cât și parallelism la nivel de buclă (optimizat prin algoritmul de fuziune a buclelor), cum ar fi aplicațiile cu sistem în timp real de tipul client-server.

Am descris și clasificat mai mulți algoritmi de mapare precum ***Limited Best Assignment*** (***LBA***) [8], ***Opportunistic Load Balancing*** (***OLB***) [8] și alți ***algoritmi Greedy*** descriși în [8], ***SCOTCH*** [12], ***METIS*** [13,14], algoritmul de mapare parte a setului de utilitare ***Zoltan*** [14], algoritmi care identifică tipare în graful de comunicarea între task-uri prin metoda descrisă în [15], ***Treematch*** [16,17], ***EagerMap*** [18] și *NUMA-BTDM* [2] (prezentat în teză) în funcție de criteriile: momentul mapării, metoda de mapare utilizată, considerarea sau nu a arhitecturii harware pe care rulează aplicația.

Am descris, de asemenea, factorii care influențează în mod direct eficiența mapării firelor de execuție pe sisteme NUMA. Aceştia sunt:

- *factorii ce țin de arhitectura hardware*: lățimea de bandă, care este indicat să fie cât mai mare [19] și, mai puțin important ca lățimea de bandă, numărul de hopuri traversate în cazul acceselor la distanță la alte noduri NUMA [19], precum și numărul numărul procesoarelor [20], care, cu cât este mai mare, cu atât devine mai importantă problema *mapării și a localizării datelor*;
- *factorul ce ține de execuția programelor paralele*: localizarea echilibrată a datelor pe sisteme NUMA [20].

Am arătat în acest capitol că alocarea memoriei în contextul migrării datelor sau a firelor de execuție, contribuie și aceasta la îmbunătățirea localizării datelor. Planificatoare firelor de execuție cum ar fi *Completely Fair Scheduler* (CFS) [21], utilizat în prezent de Linux au (alături de algortimii de mapare) un rol important, deoarece acestea se axează pe echilibrarea încărcării și utilizarea în mod justificat a resurselor [22,23] și nu dețin informații despre comportamentul dinamic al aplicației, acesta neinfluențând planificarea. Planificarea se referă în principal la ordinea de execuție, spre deosebire de mapare, care decide locul de execuție al firelor.

În încheierea capitolului am explicat câtiva factori care cresc consumul de energie al procesării paralele comparativ cu cea secvențială, cum ar fii salvarea și restaurarea contextului de execuție, planificarea firelor de execuție sau degradarea localizării datelor [24]. Algoritmii de mapare urmăresc îmbunătățirea acestor aspecte consumatoare de energie, precum și a performanței prin care, indirect, poate fi optimizat consumul de energie.

3. Descrierea algoritmilor NUMA-BTLP și NUMA-BTDM

Cercetarea vizează optimizarea aplicaţiilor paralele C/C++ care utilizează biblioteca ***PThreads*** [3] pentru gestiunea firelor de execuţie, prin doi algoritmi, de clasificare statică a firelor de execuție, respectiv de mapare statică a acestora. Algoritmii elimină unele din dezavantajele necunoaşterii comportamentului dinamic la compilare precum necunoașterea numărului de fire de execuţie. Algoritmii optimizează timpul de execuție și consumul de putere al aplicațiilor prin îmbunătățirea localizării echilibrate a datelor la rularea acestor aplicații pe sisteme NUMA.

Printr-o analiză statică care clasifică firele de execuție în trei tipuri: autonome, alăturate și amânate (algoritmul *NUMA-BTLP* [1]), maparea statică în funcție de tipul firului de execuție (algoritmul *NUMA-BTDM* [2]) asigură plasarea în execuţie, a firelor care utilizează date în comun, pe aceleaşi core-uri identificate optim şi a celor independente, pe core-uri diferite.

Scaderea consumului de putere se datorează scăderii puterii dinamice, scăderea puterii dinamice se datorează optimizării utilizării memoriei cache, iar optimizarea utilizării memoriei cache este realizată prin îmbunătăţirea localizării echilibrate a datelor pe sisteme NUMA, obținută prin cei doi algoritmi.

Creşterea performanţei este obținută prin optimizarea localizării echilibrate a datelor pe sisteme NUMA, optimizare datorată faptului că firele care comunică sunt mapate pe aceleși core-uri şi celelalte fire sunt distribuite uniform pe core-uri.

Cei doi algoritmi, *NUMA-BTLP* [25] și *NUMA-BTDM* [2], sunt implementați în compilatorul ***LLVM*** [4]. În implementarea acestora, algoritmii utilizează două structuri arborescente: arborele de generare al firelor de execuție și arborele de comunicare al firelor de execuție [25].

Arborele de generare al firelor de execuție este construit conform următoarelor reguli [25]:

1. firul principal (execută funcţia main) este rădăcina arborelui, reprezentând primul nivel din arbore;
2. firele create direct din funcţia main, prin apeluri ***pthread_create***, sunt nodurile fiu ale rădăcinii, formând al doilea nivel din arbore;
3. firele de execuţie create direct din functiile ataşate firelor create din funcţia main, reprezintă al treilea nivel din arbore ș.a.m.d.

După obținerea arborelui de generare al firelor de execuție, acesta este traversat în preordine și fiecărui fir de execuție din arbore îi este asignat câte un tip, acest tip fiind ieșirea unei analizei statice pentru firul respective [25].

Un fir de execuție *i* este autonom dacă între acesta şi fiecare alt fir, nu există nici o dependenţă de date. Am considerat că firul autonom nu are dependențe de date atunci când:

1. un alt fir nu scrie nici o dată pe care firul autonom o citeşte;
2. firul autonom nu scrie nici o dată citită de vreun alt fir;
3. firul automon poate citi datele citite doar de toate firele.

Firul de execuţie *i* este alăturat în raport cu firul de execuţie *j* dacă între acestea există cel puţin o dependenţă de date. Alăturarea nu este tranzitivă de obicei, adică dacă firul *i* este alăturat în raport cu firul *j* şi firul *j* este alăturat în raport cu firul *k, i* nu este alăturat în raport cu *k,* decât dacă cele două alăturări se bazează pe cel puţin o dată comună. Oricare două fire de execuţie pot fi alăturate între ele, indiferent de poziţia în ierarhia de generare a firelor de execuţie. Dacă un fir este alăturat cu cel puţin un alt fir, atunci acesta nu mai poate fi autonom.

Un fir de execuţie *i* este amânat dacă firul *i* are dependeţe de date doar cu firul *j,* unde *j* este firul de execuţie generator al firului *i.* Firul de execuţie generator al firului amânat execută cu prioritate celelalte fire fiu, cu care firul amânat nu are dependenţe de date. Firul amânat nu scrie date citite de firele de pe acelaşi nivel sau niveluri inferioare în arborele de generare a firelor de execuţie.

O a doua structură arborescentă utilizată de cei doi algoritmi este arborele de comunicare al firelor de execuție [25]. Acesta este construit pe baza următoarelor reguli [25]:

1. dacă firul de execuţie este autonom sau amânat, se adaugă ca nod fiu al firului generator;
2. dacă firul de execuţie este alăturat, se adaugă ca nod fiu al tuturor firelor în raport cu care este alăturat, deja adăugate în arbore.

Analiza statică pentru un fir de execuție presupune apelarea unui algoritm care primește ca parametru firul și părintele acestuia în arborele de generare [1]. Analiza identifică dependențele de date între fir și toate firele din subarborele cu rădăcina părintele firului, dacă aceste dependențe există [1]. Cu ajutorul claselor specifice din compilatorului ***LLVM*** [4], implementarea analizei statice ia în calcul și dependențele de tip alias. Termenul de alias se referă la aceeași zonă de memorie indicată de două variabile diferite.

Maparea firelor de execuție (algoritmul *NUMA-BTDM* [2]) presupune maparea tuturor firelor de execuție autonome, a firelor alăturate și respectiv a firelor amânate, în această ordine.

Maparea firelor de execuție autonome se realizează prin parcurgerea arborelui de comunicare în preordine şi adăugarea firele de execuţie autonome într-o listă. Apoi, se mapează nodurile din listă uniform, pe core-uri, astfel: se împarte numărul de fire *i* la numărul de core-uri *j* rezultând numărul real *k* şi se obţine core-ul pentru fiecare fir însumând succesiv *k* cu el însuşi pornind de la 0 şi aplicând, funcţiile *modulo* la *j* (număr core-uri) şi *floor,* fiecărui rezultat parţial.

Maparea firelor de execuție alăturate se realizează astfel:

1. Se parcurge arborele de comunicare în inordine şi se setează afinitatea CPU pentru fiecare nod de tip alăturat astfel: se face reuniunea dintre core-urile pe care se execută nodul şi core-urile pe care se execută nodul părinte iar rezultatul este noua afinitate CPU atât a firului alăturat.
2. Rădăcina şi firele alăturate de pe primul nivel sunt mapate, începând cu core-ul 0, exceptând cele alăturate în raport cu rădăcina (se aplică Punctul 1).

Maparea firelor de execuție amânate este realizată în felul următor: se parcurge arborele de comunicare în postordine şi se mapează pe rând fiecare fir amânat identificat pe cel mai puţin încărcat core la acel moment (din perspectiva aplicaţiei, nu a întregului sistem). Încărcarea core-ului este determinată static şi reprezintă numărul de fire de execuţie mapate pe acesta la un anumit moment de timp. Un fir de execuţie amânat care nu are alţi fraţi, este setat cu prioritate ca fiind alăturat.

În concluzie, am detaliat succesiunea de apeluri ale celor doi algoritmi:

1. Identifică numărul de core-uri logice şi numărul de core-uri logice per CPU la compilare (*NUMA-BTLP* [25]);
2. Creează arborele de generare a firelor de execuţie din reprezentarea intermediară LLVM IR a codului de intrare (*NUMA-BTLP* [25]);
3. Determină tipul fiecărui fir de execuţie pe baza unei analize statice şi îl adaugă în arborele de comunicare (*NUMA-BTLP* [25]);
4. Mapează firele de execuţie pe baza arborelui de comunicare obţinut în pasul anterior (*NUMA-BTDM* [2]);
5. Identifică în reprezentarea intermediară LLVM IR a codului de intrare, câte un apel ***pthread_create***, care creează firul de execuţie (*NUMA-BTLP* [25]);
6. Adaugă după fiecare ***pthread_create***, câte un apel ***pthread_setaffinity_np***, care setează afinitatea CPU a firului creat conform mapării obţinute în pasul anterior (*NUMA-BTLP* [1]).

4. Rezultate experimentale

Concluzii ale rezultatelor experimentale pentru aplicația de referință CPU-X

În urma aplicării algoritmului *NUMA-BTDM* [2], rezultatele experimentale indică o optimizare în medie cu 0.27 W/s, a consumului de putere al sistemului NUMA pe care se execută aplicația CPU-X [26] și respectiv, cu 0.32 W/s a sistemului UMA, pentru un număr de fire de execuție alăturate mai mic decât 12. Optimizarea se obține scăzând din consumul de putere mediul atunci când algoritmul *NUMA-BTDM* [2] nu este aplicat, consumul de putere mediu atunci când algoritmul este aplicat. Varianța optimizării se obține în aceeași manieră, scăzând din varianța medie atunci când algoritmul *NUMA-BTDM* [2] nu este aplicat, varianța medie atunci când algoritmul este aplicat și are valoarea ~0.06 pentru sistemul NUMA și 0.13 pentru sistemul UMA. Datorită faptului că varianța este mai mică decât optimizarea, se poate concluziona că există o optimizare. Aceasta se datorează creșterii ratei de obținere a datelor din memoria cache, mai exact a ratei de obținere a datelor din primul și al doilea nivel de cache folosit în comun de firele de execuție alăturate, mapate de către algoritmul

NUMA-BTDM [2] pe același core. Creșterea ratei de obținere a datelor din memoria cache conduce la creșterea numărului de accese locale în detrimetul celor la distanță (adică a acelor accese la un alt nod NUMA), rezultând optimizarea.

Conform aceluiași tip de raționament ca cel din paragraful anterior rezultă că, consumul CPU-ului la execuția aplicației CPU-X, nu este optimizat (optimizare 0.17 W cu o varianță a optimizării de 1.92). Acest lucru se datorează degradării de performanță produsă de numărul mare de tranziții activ-în așteptare și invers, firele fiind toate mapate pe același core. Totuși, optimizarea de consum de putere datorată eficientizării operațiilor cu memoria este mai mare decât degradarea datorată numărului mare de tranziții activ-îin așteptare și invers, rezultând astfel optimizarea consumului de putere al întregului sistem din paragraful anterior.

Rezultatele experimentale mai indică faptul că, cu cât numărul firelor alăturate care beneficiază de datele din nivelurile 1 și 2 de cache crește, cu atât optimizarea de consum de putere este mai mare, aceasta ajungând la valoarea maximă de 15%, rezultatul probat pentru un număr maxim de 12 fire de execuție alăturate.

Concluzii ale rezultatelor experimentale pentru aplicația de referință CPU

Timpul de execuție pe sistem NUMA pentru aplicația de referință CPU [27], compusă din aplicațiile CPU [27], Flops [27] și Iops [27], este optimizat cu o valoare foarte mică (0.03 s raportat la timpul total de execuție de 600 s al aplicațiilor Flops [27] și Iops [27] și 0.02 s raportat la timpul total de execuție de 0.59 s al aplicației CPU [27]), putându-se considera că timpul de execuție se păstrează odată cu optimizarea aplicațiilor prin cei doi algoritmi.

În cazul aplicațiilor Flops [27] și Iops [27], optimizarea produsă de algoritmul *NUMA-BTDM* [2] în urma experimentelor, nu este foarte mare (0.03 s), deoarece maparea unui număr mare de fire de execuție (2400) produce o degradare a timpului de execuție (datorită apelurilor ***pthread_setaffinity_np*** și a timpului de gestiune al firelor de execuție). Degradarea însă este acoperită de câștigul de timp de execuție rezultat în urma aplicării algoritmului *NUMA-BTDM* [2], obținându-se în final un câștig de performanță.

La CPU [27] se păstrează același timp de execuție (la fel ca în cazul aplicației de referință CPU-X [26]), dar se observă o reducere a consumului de putere al sistemului NUMA cu 0.9 W/s și al sistemului UMA cu 7.56 W/s. Varianța optimizării este și în acest caz calculată ca fiind diferența dintre varianța medie a măsurătorilor de consum de putere al sistemului pe care rulează aplicația neoptimizată și varianța medie a măsurătorilor de consum de putere al sistemului pe care rulează aplicația optimizată. Varianța optimizării este cu mult mai mare decât optimizarea în sine, deci aplicația CPU [27] nu se consideră optimizată. Optimizarea execuției aplicației Flops în urma aplicării algoritmului *NUMA-BTDM* [2] pe sistem NUMA este de 0.6 W/s și de 0.3 W/s pe sistem UMA. Aceasta se consideră a fi o optimizare având în vedere varianța optimizarii este mai mică decât optimizarea în sine (varianța optimizării are valoarea 0.14). Aplicația Iops [27] nu este optimizată de algoritmul *NUMA-BTDM* [2], rezultând un consum de putere mai mare în medie atunci când algoritmul este aplicat decât atunci când acesta nu este aplicat.

Concluzii ale rezultatelor experimentale pentru aplicația de referință Context Switch

Timpul de execuție al aplicației de referință Context Switch [28] pe sistem NUMA nu este îmbunătățit la aplicarea algoritmului *NUMA-BTDM* [2]. Datorită numărului mic de fire de execuție pe care aplicația le utilizează și anume două fire de execuție autonome, mapate pe core-uri diferite de către algoritm, timpul cu care este redus timpul total de execuție al aplicației, în urma aplicării algoritmului *NUMA-BTDM* [2], este mai mic decât timpul de

execuție consumat cu maparea efectivă a firelor de execuție pe core-uri prin apeluri ***pthread_setaffinity_np***, rezultând o degradare nesemnificativă de performanță (0.44 s, raportat la 41.9 s, timpul mediu de rulare al aplicației).

Rezultatele experimentale indică o optimizare în medie cu 0.32 W/s a consumului de energie pentru aplicația de referință Context Switch [28] care rulează pe sistem NUMA utilizând două fire de execuție autonome, adică o optimizare cu până la 5%, păstrându-se același timp de execuție. Reducerea consumului de energie este cauzată de reducerea datorită mapării a numărului de tranziții din stare activă în stare de așteptare și invers, a firelor de execuție.

5. Concluzii și contribuții personale

NUMA-BTDM [2] este un algoritm de mapare, aplicat la compilarea aplicațiilor paralele, care decide afinitatea CPU a fiecărui fir de execuție bazat pe tipul acestuia. Tipul firului de execuție este returnat pe baza caracteristicilor statice ale codului de către algoritmul *NUMA-BTLP* [1] care clasifică firele de execuție în autonome, alăturate și amânate în urma unei analize statice a dependențelor de date. *NUMA-BTLP* [1] și *NUMA-BTDM* [2] contribuie la obținerea localizării echilibrate a datelor, optimizând maparea firelor de execuție pe sisteme NUMA. După aplicarea celor doi algoritmi la compilare asupra programelor paralele care utilizează biblioteca ***PThreads*** [3] în vederea obținerii paralelismului la nivel de task, la rulare va fi îmbinat paralelismul la nivel de task cu localizarea echilibrată a datelor, ceea ce optimizează consumul de energie. *NUMA-BTDM* [2] folosește biblioteca ***PThreads*** [3] pentru setarea afinității CPU a fiecărui fir de execuție, permițând firelor de execuție să se execute cât mai aproape în termeni de timp și distanță NUMA față de datele care le utilizează.

Elementele de noutate ale celor doi algoritmi sunt:

1. Abilitatea de a permite aplicațiilor paralele C/C++ care utilizează ***PThreads*** [3] să particularizeze și să controleze maparea firelor de execuție pe core-uri în funcție de caracteristicile statice ale codului, în loc să permită sistemului de operare să realizeze această mapare aleator.
2. Eliminarea dezavantajului necunoașterii aspectului dinamic număr de fire de execuție, în faza de compilare, prin inserarea la compilare a câte unui apel ***pthread_setaffinity_np*** imediat după fiecare apel ***pthread_create***, apel prin care se mapează firul(e) creat(e) de apelul ***pthread_create***, conform algoritmului *NUMA-BTLP* [1], indiferent de numărul acestora și eliminarea dezvantajului impredictibilității latențelor acceselor la distanță în faza de compilare, prin favorizarea accelor locale în detrimentul celor la distanță ca urmare a mapării firelor de execuție alăturate care utilizează aceleași date, pe aceleași core-uri
3. Definirea unor criterii statice originale de clasificare a firelor de execuție în 3 categorii și definirea acestor categorii
4. Maparea firelor de execuție în funcție de tipul acestora
5. Integrarea algoritmilor de clasificare a firelor de execuție și de mapare a acestora într-un compilator modern
6. Maparea firelor de execuție utilizând 2 arbori: unul care descrie dependențele de date și un altul care descrie ierarhia de generare [25]
7. Asigurarea portabilității algoritmilor pe orice arhitectură NUMA pe care rulează sistemul de operare Linux prin inserarea în codul sursă, la compilare, a unor apeluri sistem care execută comenzi prin intermediul cărora se identifică, la execuția aplicației paralele, numărul de core-uri și numărul de core-uri logice per CPU ale arhitecturii hardware, necesare la rularea algoritmului de mapare *NUMA-BTDM* [2]

NUMA-BTDM [2] este una din puținele optimizări la compilare dedicate unui anumit tip de sistem, în acest caz dedicată sistemelor NUMA. Pe lângă acest fapt, acești algoritmi obțin localizarea echilibrată a datelor printr-o nouă manieră introdusă de această lucrare și anume: repartizarea echilibrată pe core-uri a firelor de execuție autonome, proximitatea în timp și distanță NUMA a firelor de execuție alăturate față de datele utilizate și existența firelor amânate care nu "fură" cache-ul de la firele de execuție critice care necesită execuție imediată, fiind mapate pe cel mai puțin încărcat core.

Cu toate că algoritmul *NUMA-BTLP* [1] inserează în cod, la compilare, apeluri adiționale de funcții care setează afinitatea CPU a fiecărui fir de execuție, algoritmul *NUMA-BTLP* [1] nu degradează nici timpul de execuție, nici consumul de putere pe sisteme NUMA sau UMA, în cazul aplicațiilor testate, ci îmbunătățește timpul de execuție și consumul de putere pentru număr mic de fire de execuție autonome și doar consumul de putere pentru număr mare de fire de execuție autonome și un număr mic de fire de execuție alăturate.

Bibliografie

[1] Iulia Știrb. „NUMA-BTLP: A static algorithm for thread classification”. În 2017 5th International Conference on Control, Decision and Information Technologies (CoDIT)”, p. 882–887. IEEE, 2017.

[2] Iulia Știrb. „NUMA-BTDM: A thread mapping algorithm for balanced data locality on NUMA systems”. În 2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), p. 317–320. IEEE, 2016.

[3] POSIX Threads. 2017.

[4] The LLVM Compiler Infrastructure Project. https://llvm.org/, 2018. Accesat: 9.10.2018.

[5] Nakul Manchanda și Karan Anand. Non-uniform memory access (NUMA). New York University, 4, 2010.

[6] Manu Awasthi, David Nellans, Kshitij Sudan, Rajeev Balasubramonian, și Al Davis. „Handling the problems and opportunities posed by multiple on-chip memory controllers”. În 2010 18th International Conference on Parallel Architectures and Compilation Techniques (PACT), p. 318–330. IEEE, 2010.

[7] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Philippe OA Navaux, și Israel Koren. „Affinity-based thread and data mapping in shared memory systems”. ACM Computing Surveys (CSUR), 49(4):64, 2017.

[8] Robert Armstrong, Debra Hensgen, și Taylor Kidd. „The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions”. În Proceedings Seventh Heterogeneous Computing Workshop (HCW 98), p. 79–87. IEEE, 1898.

[9] Haris Ribic și Yu David Liu. “Energy-efficient work-stealing language runtimes”. În ACM SIGARCH Computer Architecture News, vol. 42, p. 513-527. ACM, 2014.

[10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, și Raymond Namyst. „hwloc: A generic framework for managing hardware affinities in hpc applications”. În 2010 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p. 170–176. IEEE, 2010.

[11] Imen Fassi și Philippe Clauss. “Xfor: filling the gap between automatic loop optimization and peak performance”. În 2015 14th International Symposium on Parallel and Distributed Computing (ISPDC), p. 100–109. IEEE, 2015.

[12] François Pellegrini și Jean Roman. „Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs”. În International Conference on High-Performance Computing and Networking, p. 493–498. Springer, 1896.

[13] George Karypis și Vipin Kumar. „A fast and high quality multilevel scheme for partitioning irregular graphs”. SIAM Journal on Scientific Computing, 20(1):359–392, 1898.
[14] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, și Umit V Catalyurek. „Parallel hypergraph partitioning for scientific computing”. În Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, p. 10–pp. IEEE, 2006.
[15] Eduardo HM Cruz, Matthias Diener, și Philippe OA Navaux. „Using the translation lookaside buffer to map threads in parallel applications based on shared memory”. În 2012 IEEE 26th International Parallel and Distributed Processing Symposium, p. 532–543. IEEE, 2012.
[16] Hao Zhou și Jingling Xue. „A compiler approach for exploiting partial SIMD parallelism”. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):11, 2016.
[17] Emmanuel Jeannot și Guillaume Mercier. „Near-optimal placement of MPI processes on hierarchical NUMA architectures”. În Euro-Par 2010-Parallel Processing, p. 189–200, 2010.
[18] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, și Philippe OA Navaux. „An efficient algorithm for communication-based task mapping”. În 2015 22rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p. 197–204. IEEE, 2015.
[19] Baptiste Lepers, Vivien Quéma, și Alexandra Fedorova. „Thread and memory placement on NUMA systems: Asymmetry matters”. În USENIX Annual Technical Conference, p. 267–279, 2015.
[20] Timothy Brecht. „On the importance of parallel application placement in NUMA multiprocessors”. În Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), p. 1–17, 1893.
[21] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, și Fun Wey. „Towards achieving fairness in the Linux scheduler”. ACM SIGOPS Operating Systems Review, 42(5):34–43, 2008.
[22] David Tam, Reza Azimi, și Michael Stumm. „Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors”. În ACM SIGOPS Operating Systems Review, vol. 41, p. 47–58. ACM, 2007.
[23] Tong Li, Dan Baumberger, și Scott Hahn. „Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin”. În ACM Sigplan Notices, vol. 44, p. 65–74. ACM, 2009.
[24] Candy Pang, Abram Hindle, Bram Adams, și Ahmed E Hassan. „What do programmers know about software energy consumption?”. IEEE Software, 33(3):83–89, 2016.
[25] Iulia Știrb. „Extending NUMA-BTLP algorithm with thread mapping based on a communication tree”. Computers, 7(4):66, 2017.
[26] Cpu-benchmarking. https://github.com/pdpriyanka/CPU-Benchmarking, 2016. Accesat: 16.08.2018.
[27] Cpu-x. https://github.com/X0rg/CPU-X, 2018. Accesat: 16.08.2018.
[28] contextswitch. https://github.com/tsuna/contextswitch, 2016. Accesat: 16.08.2018.