REDUCEREA CONSUMULUI DE ENERGIE ȘI A TIMPULUI DE EXECUȚIE PRIN OPTIMIZAREA COMUNICĂRII ÎNTRE FIRELE DE EXECUȚIE ȘI PRIN LOCALIZAREA ECHILIBRATĂ A DATELOR LA EXECUȚIA PROGRAMELOR PARALELE, PE SISTEME NUMA

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea Politehnica Timişoara
în domeniul
CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI
de către

ing. Iulia Știrb

Conducător ştiinţific:	prof.univ.dr.ing. Horia Ciocârlie
Referenţi ştiinţifici:	prof.univ.dr.ing. Mircea Popa
	prof.univ.dr.ing. Dana Petcu
	prof.univ.dr.ing. Liviu Cristian Miclea

Ziua susţinerii tezei: 9 decembrie 2020

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică și Telecomunicații
8. Inginerie Industrială
9. Inginerie Mecanică
10. Știința Calculatoarelor
11. Știința și Ingineria Materialelor
12. Ingineria sistemelor
13. Inginerie energetică
14. Calculatoare și tehnologia informației
15. Ingineria materialelor
16. Inginerie și Management
17. Arhitectură
18. Inginerie civilă și instalații
19. Inginerie electronică, telecomunicații și tehnologii informaționale

Universitatea Politehnica Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul Şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300223 Timişoara, Bd. Vasile Pârvan nr.2B
Tel./fax 0256 404677
e-mail: editura@upt.ro

Cuvânt înainte

Teza de doctorat a fost elaborată pe parcursul activităţii mele în cadrul Departamentului de Calculatoare și Tehnologia Informației al Universităţii Politehnica Timişoara.

Mulţumiri deosebite se cuvin conducătorului de doctorat prof.univ.dr.ing. Horia Ciocârlie, pentru păşirea pe acest drum dificil, dar plin de satisfacții profesionale și sufletești, al doctoratului. Domnul profesor m-a îndrumat și în cariera profesională de cadru didactic asociat, în decursul doctoratului, carieră didactică pe care doresc să o urmez și în anii ce vor urma. Empatia, calmul și blândețea în exprimare și în gând sunt trasături caracteristice unui cadru didactic, d-nul profesor fiind un exemplu pentru mine în formarea mea profesională și ca om.

În egală măsură, cu aceeaşi recunoştinţă, le aduc mulţumirile mele şi membrilor comisiei de îndrumare, conf.dr.ing. Ciprian-Bogdan Chirilă, conf.dr.ing. Răzvan Bogdan și șef lucr.dr.ing. Codruţa-Mihaela Istin, care m-au direcţionat cu o înaltă rigurozitate, m-au ajutat să depăşesc momentele cele mai dificile, făcând diferite comentarii pertinente la documentele pe care le-am elaborat pe parcursul doctoratului şi mi-au acordat sprijinul de specialitate de care aveam nevoie. Un loc aparte în perioada de pregătire l-a avut prof.univ.dr.ing. Marius Marcu ajutându-mă într-un moment dificil al efectuării experimentelor practice prin punerea la dispoziție a echipamentelor de care aveam nevoie pentru a mări calitatea și acuratețea măsurătorilor. Le mulțumesc tuturor pentru răbdare și competență.

Le mulțumesc foarte mult membrilor comisiei de referenți, prof.univ.dr.ing. Dana Petcu de la Universitatea de Vest, prof.univ.dr.ing. Mircea Popa, și în facultate profesor fiindu-mi și prof.univ.dr.ing. Liviu Cristian Miclea de la Universitatea Tehnică din Cluj-Napoca pentru solicitudinea cu care au răspuns invitației de a face parte din Comisia de doctorat, acceptând să analizeze și să evalueze lucrarea pe care am elaborat-o.

Chiar dacă în ultimele rânduri, mulţumesc în mod cu totul special membrilor familiei mele care mi-au înţeles temerile şi nevoile, m-au sprijinit şi au crezut întotdeauna în munca mea.

Timişoara, septembrie 2020 Iulia Știrb

Destinatarii dedicaţiei.

Prof.univ.dr.ing. Horia Ciocârlie

Prof.univ.dr.ing. Mircea Popa

Prof.univ.dr.ing. Dana Petcu

Prof.univ.dr.ing. Liviu Cristian Miclea

Prof.univ.dr.ing. Marius Marcu

Conf.univ.dr.ing. Ciprian-Bogdan Chirilă

Conf.univ.dr.ing. Răzvan Bogdan

Ș.l.dr.ing. Codruța-Mihaela Istin

Știrb, Iulia

Reducerea consumului de energie și a timpului de execuție prin optimizarea comunicării între firele de execuție și prin localizarea echilibrată a datelor la execuția programelor paralele, pe sisteme NUMA

Teze de doctorat ale UPT, Seria 14, Nr. 43, Editura Politehnica, 2020, 212 pagini, 100 figuri, 23 tabele.

ISSN:2069-8216
ISSN-L: 2069-8216
ISBN:978-606-35-0396-2

Cuvinte cheie:maparea statică a firelor de execuție, localizare echilibrată a datelor, sisteme NUMA, Linux, ***Pthreads***, C, **LLVM**

Rezumat,
Algoritmii de clasificare statică a firelor de execuție NUMA-BTLP și de mapare statică NUMA-BTDM implementați în compilatorul **LLVM**, permit aplicațiilor paralele C/C++ care utilizează biblioteca ***Pthreads***, să particularizeze și să controleze maparea firelor în Linux, în funcție de caracteristicile statice, eliminând dezavantajul necunoașterii, la compilare, a numărului firelor. Prin analiză statică, algoritmul NUMA-BTLP clasifică firele în trei tipuri: autonome (nu au dependențe de date cu nici un alt fir), alăturate în raport cu alte fire (au dependențe de date între ele) și amânate (fire necritice, au dependențe de date doar cu firul generator). În funcție de tipul firelor, algoritmul NUMA-BTDM realizează maparea astfel: firele autonome sunt mapate uniform pe core-uri, cele alăturate sunt mapate pe aceleași core-uri și cele amânate sunt mapate pe cel mai puțin încărcat core, îmbunătățind localizarea echilibrată a datelor pe sisteme NUMA și astfel, timpul de execuție și consumul de energie, cu până la 15%.

CUPRINS

Listă de lucrări științifice publicate în decursul doctoratului....8
Notaţii, abrevieri, acronime....9
Listă de tabele....10
Listă de figuri....12
Listă de coduri....17
1. Introducere....18
1.1. Contextul și motivația cercetării....18
1.2. Scopul și obiectivele cercetării....19
1.3. Argumentarea necesității algoritmilor prezentați în teză....20
1.4. Argumentarea alegerii categoriilor de fire de execuție....21
1.5. Studiul alegerii produsului software pentru implementarea algoritmilor....23
1.6. Alegerea tipului de sistem hardware....23
1.7. Structurarea tezei pe capitole....24
2. Stadiul actual al cercetărilor în domeniile abordate de teză....26
2.1. Caracteristici hardware care au contribuit la dezvoltarea conceptului de mapare....26
2.1.1. Considerente generale ale sistemelor multiprocesor....26
2.1.2. Definirea conceptului de sistem NUMAI....26
2.1.3. Precondițiile configurării unui sistem ca NUMAI....27
2.1.4. Analiză SWOT: Utilizarea sistemelor NUMA în industrie, în detrimentul sistemelor UMA....28
2.1.5. Definirea conceptului de mapare și caracteristicile sistemelor NUMA care favorizează dezvoltarea de algoritmi de mapare....29
2.2. Eficientizarea execuției aplicațiilor paralele prin optimizări la compilare....31
2.2.1. Tipare de cod sursă pentru care se aplică optimizările la compilare....31
2.2.2. Validarea respectării standardelor limbajelor de programare în implementările optimizărilor la compilare....31
2.2.3. Influența ordinii de aplicare a optimizărilor la compilare asupra eficienței execuției....32
2.3. Infrastructura de compilare LLVM....32
2.3.1. Prezentare comparativă a particularităților infrastructurii de compilare LLVM în raport cu cele ale colecției de compilatoare GCC....32
2.3.2. Modelul poliedral....33
2.3.2.1. Caracteristicile modelului poliedral....33
2.3.2.2. Optimizări care se încadrează în modelul poliedral....34
2.4. Optimizările infrastructurii de compilare LLVM....35
2.4.1. Optimizări la compilare ale buclelor care îmbunătățesc performanța execuției codului....35
2.4.2. Descrierea și încadrarea în necesitate a optimizărilor la compilare ale buclelor și a altor optimizări care îmbunătățesc paralelizarea....36
2.5. Analiza impactului compilării asupra timpului de execuție și a consumului de energie....41
2.5.1. Acuratețea predicțiilor la compilare a timpului de execuție și a altor aspecte legate de execuția aplicațiilor paralele....41
2.5.2. Modul în care compilarea influențează consumul de energie la rulare....42
2.6. Maparea firelor de execuție....44
2.6.1. Tipare de mapare și reacția fiecăruia la schimbări în comportamentul dinamic al aplicațiilor paralele....45

2.6.2 Informații necesare în realizarea unei mapări eficiente 46
2.6.3 Clasificarea algoritmilor de mapare 46
2.6.4 Algoritmi de mapare 47
2.6.4.1 Algoritmi de mapare naivi 47
2.6.4.2 Algoritmi de mapare bazați pe metoda partiționării grafurilor 47
2.6.4.3 Algoritmi de mapare care țin cont de arhitectura hardware 48
2.6.4.4 Algoritmi de mapare bazați pe istoricul rulărilor precedente 51
2.6.5 Comparație între algoritmii de mapare 52
2.6.6 Aspecte care influențează direct eficiența algoritmilor de mapare pe sisteme NUMAI.....52
2.7 Biblioteci utilizate pentru mapare 53
2.7.1 Biblioteci actuale pentru maparea firelor de execuție.....53
2.7.2 Dezavantajele bibliotecilor actuale privind maparea 54
2.8 Aspecte conexe care influențează maparea firelor de execuție.....54
2.8.1 Politici de planificare a firelor de execuție în sistemul de operare Linux.....54
2.8.2 Impactul alocării memoriei asupra eficienței mapării firelor de execuție.....57
2.9 Reducerea consumului de energie prin maparea firelor de execuție 57
3 Concluzii 59
3. Prezentarea algoritmilor NUMA-BTLP și NUMA-BTDM 61
3.1 Descrierea algoritmului NUMA-BTLP de clasificare a firelor de execuție.....61
3.2 Analiza categoriilor de fire de execuție definite de algoritmul NUMA-BTLP.....62
3.2.1 Fire de execuție autonome.....62
3.2.2 Fire de execuție alăturate 63
3.2.3 Fire de execuție amânate 65
3.3 Reprezentarea sub formă de arbore a ierarhiei de generare a firelor de execuție 67
3.4 Reprezentarea sub formă de arbore a tiparelor de comunicare între firele de execuție 68
3.5 Implementarea algoritmului NUMA-BTLP în pseudocod.....70
3.6 Implementarea algoritmului NUMA-BTDM în pseudocod.....74
3.7 Concluzii 76
4. Rezultate experimentale 80
4.1 Obiective specifice propuse la implementarea algoritmilor și criterii de îndeplinire 80
4.2 Materiale și metode de obținere a rezultatelor experimentale.....80
4.2.1 Arhitectura hardware pe care au fost obținute rezultatele experimentale 80
4.2.2 Metodologia de obținere a rezultatelor experimentale de timp de execuție ..81
4.2.3 Metodologia de obținere a rezultatelor experimentale de consum de putere 81
4.3 Aplicarea algoritmului NUMA-BTLP.....82
4.4 Aplicația de referință CPU-X.....83
4.4.1 Descrierea aplicației.....83
4.4.2 Metodologia de obținere a rezultatelor experimentale de consum de putere specifică aplicației de referință CPU-X.....84
4.4.3 Rezultate experimentale pe sisteme UMA.....84
4.4.3.1 Timp de execuție 84
4.4.3.2 Consum de putere 84
4.4.4 Rezultate experimentale pe sisteme NUMAI 93
4.4.4.1 Timp de execuție 93

4.4.4.2 Consum de putere 93
4.4.5 Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMAI..... 102
4.4.5.1 Comparație pentru timp de execuție..... 102
4.4.5.2 Comparație pentru consum de putere..... 102
4.4.6 Concluzii ale rezultatelor experimentale pentru aplicația de referință CPU-X 103
4.5 Aplicația de referință CPU 107
4.5.1 Descrierea aplicației..... 107
4.5.2 Rezultate experimentale pe sisteme UMA..... 108
4.5.2.1 Timp de execuție 108
4.5.2.2 Consum de putere 109
4.5.3 Rezultate experimentale pe sisteme NUMAI 123
4.5.3.1 Timp de execuție 123
4.5.3.2 Consum de putere 123
4.5.4 Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMAI..... 137
4.5.4.1 Comparație pentru timp de execuție..... 137
4.5.4.2 Comparație pentru consum de putere..... 138
4.5.5 Concluzii ale rezultatelor experimentale pentru aplicația de referință CPU 146
4.6 Aplicația de referință Context Switch 147
4.6.1 Descrierea aplicației..... 147
4.6.2 Rezultate experimentale pe sisteme UMA..... 147
4.6.2.1 Timp de execuție 147
4.6.2.2 Consum de putere 149
4.6.3 Rezultate experimentale pe sisteme NUMAI 156
4.6.3.1 Timp de execuție 156
4.6.3.2 Consum de putere 158
4.6.4 Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMAI..... 166
4.6.4.1 Comparație pentru timp de execuție..... 166
4.6.4.2 Comparație pentru consum de putere..... 166
4.6.5 Concluzii ale rezultatelor experimentale pentru aplicația de referință Context Switch 171
4.7 Avantaje și limitări ale algoritmilor NUMA-BTLP și NUMA-BTDM 173
5 Concluzii şi contribuţii personale 174
.
.
Anexe..... 176
A1 Aplicarea algoritmilor NUMA-BTDM în mod automat și NUMA-BTLP în mod neautomat pe un exemplu de cod cu un anumit tipar de comunicare între firele de execuție 176
A2 Implementarea algoritmilor NUMA-BTLP și NUMA-BTDM în LLVM 183
A3 Metodica activităților de laborator..... 203

Bibliografie..... 206

Listă de lucrări științifice publicate în decursul doctoratului

Capitol indexat ISI publicat în ediția unei cărți indexate ISI

1. I. Știrb, "Comparison Between OpenMP and MPICH Optimized Parallel Implementations of a Cellular Automaton That Simulates the Skin Pigmentation Evolution" în "Emerging Trends in Applications and Infrastructures for Computational Biology, Bioinformatics, and Systems Biology. Systems and Applications", ed. 1, Morgan Kaufmann, 22nd May 2016, cap. 5.

Lucrări ştiinţifice publicate în volumele unor manifestări ştiinţifice (Proceedings) indexate ISI Proceedings

2. I. Știrb, H. Ciocârlie, "Improving performance and energy consumption with loop fusion and parallelization", în 2016 IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, Noiembrie 2016, pp. 000099-000104.
3. I. 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), Guangzhou, China, China, Decembrie 2016, pp. 317-320.
4. I. Știrb, "An Implementation of Loop Fusion for Improving Performance and Energy Consumption of Shared-Memory Parallel Codes", în 2017 IEEE 13th International Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, România, Septembrie 2017.
5. I. Ştirb, "NUMA-BTLP: A static algorithm for thread classification", în 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), Thessaloniki, Grecia, Aprilie 2018, pp. 882-887.

Lucrări ştiinţifice publicate în jurnale indexate ISI

1. I. Știrb, "Extending NUMA-BTLP algorithm with thread mapping based on a communication tree" în Computers, vol. 7(4), nr. art. 66
2. I. Știrb, "Improving runtime performance and energy consumption through balanced data locality with NUMA-BTLP and NUMA-BTDM static algorithms for thread classification and thread type-aware mapping" în International Journal of Computational Science and Engineering, vol. 22, nr. 2-3, pp 200-210, 2020

Notaţii, abrevieri, acronime

BIOS – Basic Input/Output System
CFS – Completely Fair Process
CPU – Central Processing Unit
DIO-E - Distributed Intensity Online-Energy
DRAM - Dynamic Random Access Memory
DVFS – Dynamic Voltage And Frequency Scaling
GCC – GNU Compiler Collection
Intel KNL – Intel Knights Landing
IT – Information Technology
LBA – Limited Best Assignment
LICM – Loop-Invariant Code Motion
LLVM – Low-Level Virtual Machine
LLVM IR – Low-Level Virtual Machine Intermediate Representation
MCDRAM – Multi-Channel Dynamic Random Access Memory
MPI – Message Passing Interface
NP-Hard – Non-Deterministic Polynomial-Time - Hard
NUMA – Non-Uniform Memory Access
NUMA-BTDM – Non-Uniform Memory Access – Balanced Thread and Data Mapping
NUMA-BTLP – Non-Uniform Memory Access - Balanced Task and Loop Parallelism
OLB - Opportunistic Load Balancing
OpenMP – Open Multi-Processing
OpenMPI – Open Message Passing Interface
Pthreads – POSIX Threads
SIMD – Single Instruction, Multiple Data
SSA – Static Single Assignment Form
SWOT – Strenghts Weaknesses Opportunities Threats
TBB - Intel Threading Blocks
UMA – Uniform Memory Access

Listă de tabele

Tabel 1. Analiză strategică SWOT: Utilizarea sistemelor NUMA în industrie, în detrimentul sistemelor UMA
Tabel 2. Cazuri tratate de algoritmul de fuziune a buclelor
Tabel 3. Categorii de politici de planificare a firelor de execuție în Linux și politicile aferente categoriilor
Tabel 4. Rezultate experimentale pe sistem UMA prezentând optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației **CPU-X**
Tabel 5. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației **CPU-X**
Tabel 6. Rezultate experimentale prezentând comparativ pentru sistem UMA și sistem NUMA, optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației **CPU-X**
Tabel 7. Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMA pentru aplicația **CPU-X**
Tabel 8. Rezultate experimentale de timp de execuție al aplicație **CPU** (formată din aplicațiile **CPU**, **Flops**, **Iops**) rulată pe sistem UMA, prezentate comparativ atunci când algoritmul NUMA-BTDM este aplicat și atunci când nu este aplicat
Tabel 9. Comparație între rezultatele experimentale pentru aplicația **CPU**, obținute din cele două surse, utilitarul turbostat, respectiv dispozitivul WattsUp, atât când aceasta nu este optimizată, cât și când este optimizată
Tabel 10.Rezultate experimentale pe sistem UMA prezentând optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației **CPU**
Tabel 11. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicațiilor **Flops** și **Iops**
Tabel 12. Rezultate experimentale de timp de execuție al aplicație **CPU** (formată din aplicațiile **CPU**, **Flops**, **Iops**) rulată pe sistem NUMA, prezentate comparativ atunci când algoritmul NUMA-BTDM este aplicat și atunci când nu este aplicat
Tabel 13. Comparație între rezultatele experimentale pentru aplicația **CPU**, obținute din cele două surse, utilitarul turbostat, respectiv dispozitivul WattsUp, atât când aceasta nu este optimizată, cât și când este optimizată
Tabel 14. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației **CPU**
Tabel 15. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicațiilor **Flops** și **Iops**
Tabel 16. Comparație între rezultatele experimentale de timp de execuție obținute pe sisteme UMA și pe sisteme NUMA pentru aplicația **CPU**
Tabel 17. Rezultate experimentale prezentând comparativ pentru sistem UMA și sistem NUMA, optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației de referință **CPU**
Tabel 18. Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMA pentru aplicația **CPU**
Tabel 19. Rezultate experimentale de timp de execuție al aplicației **Context Switch** rulată pe sistem UMA, prezentate comparativ atunci când algoritmul NUMA-BTDM este aplicat și atunci când nu este aplicat

Tabel 20. Rezultate experimentale de timp de execuție al aplicației **Context Switch** rulată pe sistem NUMA, prezentate comparativ atunci când algoritmul NUMA-BTDM este aplicat și atunci când nu este aplicat
Tabel 21. Comparație între rezultatele experimentale de timp de execuție obținute pe sisteme UMA și pe sisteme NUMA pentru aplicația **Context Switch**
Tabel 22. Rezultate experimentale prezentând comparativ pentru sistem UMA și sistem NUMA, optimizarea obținută la aplicarea algoritmului NUMA-BTDM asupra aplicației **Context Switch**
Tabel 23. Comparație între rezultatele experimentale de consum de putere obținute pe sisteme UMA și pe sistem NUMA pentru aplicația **Context Switch**

Listă de figuri

Figura 1. Sistem NUMA cu patru noduri de memorie fiecare având atașate două core-uri
Figura 2. Schema bloc a algoritmului de fuziune a buclelor
Figura 3. Reprezentarea ierarhiei de generare a firelor de execuție sub formă de arbore
Figura 4. Reprezentarea tiparului de comunicare a firelor de execuție sub formă de arbore
Figura 5. Reprezentarea grafică a zonelor de memorie din tabloul alocat dinamic args utilizate de firele de execuție
Figura 6. Consumul de putere (W) al sistemului UMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 7. Consumul de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 8. Consumul de putere (W) al CPU-ului și al întregului sistem UMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 9. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 10. Valorile medii de consum de putere (W) al CPU-ului din sistemul UMA și al sistemului pe care rulează aplicația **CPU-X** obținute în runda 1 pentru diferite valori ale numărului de fire de execuție
Figura 11. Valorile de consum de putere (W) al CPU-ului din sistemul UMA și al sistemului pe care rulează aplicația **CPU-X** obținute în runda 1
Figura 12. Valorile medii de consum de putere (W) al sistemului UMA în stare "în așteptare"
Figura 13. Valorile medii de consum de putere (W) ale CPU-ului din sistemul UMA în stare "în așteptare"
Figura 14. Consumul de putere (W) al CPU-ului din sistemul UMA și al întregului sistem în stare "în așteptare" corelate
Figura 15. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului în stare "în așteptare"
Figura 16. Consumul de putere (W) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție
Figura 17. Optimizarea de consum de putere (W) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție
Figura 18. Optimizarea de consum de putere (%) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție
Figura 19. Consumul de putere (W) al sistemului NUMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 20. Consumul de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 21. Consumul de putere (W) al CPU-ului și al întregului sistem NUMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție
Figura 22. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Figura 23. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **CPU-X** obținute în runda 1 pentru diferite valori ale numărului de fire de execuție
Figura 24. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **CPU-X** obținute în runda 1
Figura 25. Valorile medii de consum de putere (W) al sistemului NUMA în stare "în așteptare"
Figura 26. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA în stare "în așteptare"
Figura 27. Consumul de putere (W) al CPU-ului din sistemul NUMA și al întregului sistem în stare "în așteptare" corelate
Figura 28. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare"
Figura 29. Consumul de putere (W) al aplicației **CPU-X** care rulează pe sistem NUMA cu diferite valori, stabilite static, pentru numărul de fire de execuție
Figura 30. Optimizarea de consum de putere (W) al aplicației **CPU-X** care rulează pe sistem NUMA cu diferite valori, stabilite static, pentru numărul de fire de execuție
Figura 31. Optimizarea de consum de putere (%) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție
Figura 32. Optimizarea de consum de putere (%) al aplicației **CPU-X** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA
Figura 33. Consumul de putere (W) al sistemului UMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde
Figura 34. Consumul de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde
Figura 35. Consumul de putere (W) al CPU-ului și a întregului sistem UMA pe care rulează aplicația **Flops** în diferite runde
Figura 36. Consumul de putere (W) al CPU-ului și al întregului sistem UMA pe care rulează aplicația **Iops** în diferite runde
Figura 37. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului pe care rulează aplicațiile **Flops** și **Iops** în diferite runde
Figura 38. Valorile medii de consum de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **Flops** obținute pentru fiecare rundă în parte
Figura 39. Valorile de consum de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **Flops** obținute în runda 1
Figura 40. Valorile de consum de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **Iops** obținute în runda 1
Figura 41. Valorile medii de consum de putere (W) al sistemului UMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare
Figura 42. Valorile medii de consum de putere (W) al CPU-ului din sistemul UMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare
Figura 43. Consumul mediu de putere (W) al CPU-ului din sistemul UMA și al întregului sistem în stare "în așteptare", corelate, valori rezultate conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**
Figura 44. Diferența per rundă dintre consumul mediu de putere (W) al întregului sistem UMA și cel al CPU-ului în stare "în așteptare", rezultată conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**

Figura 45. Consumul de putere (W) al aplicației **CPU** care rulează pe sistem UMA în diferite runde
Figura 46. Optimizarea de consum de putere (W) al aplicației **CPU** care rulează pe sistem UMA în diverse runde
Figura 47. Optimizarea de consum de putere (%) al aplicației **CPU** care rulează pe sistem UMA în diverse runde
Figura 48. Consumul de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem UMA în diferite runde
Figura 49. Optimizarea de consum de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem UMA în diverse runde
Figura 50. Optimizarea de consum de putere (%) al aplicațiilor **Flops** și **Iops** care rulează pe sistem UMA în diverse runde
Figura 51. Consumul de putere (W) al sistemului NUMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde
Figura 52. Consumul de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde
Figura 53. Consumul de putere (W) al CPU-ului și al întregului sistem NUMA pe care rulează aplicația **Flops** în diferite runde
Figura 54. Consumul de putere (W) al CPU-ului și al întregului sistem NUMA pe care rulează aplicația **Iops** în diferite runde
Figura 55. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului pe care rulează aplicațiile **Flops** și **Iops** în diferite runde
Figura 56. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **Flops** obținute pentru fiecare rundă în parte
Figura 57. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **Flops** obținute în runda 1
Figura 58. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **Iops** obținute în runda 1
Figura 59. Valorile medii de consum de putere (W) al sistemului NUMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare
Figura 60. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare
Figura 61. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA și al întregului sistem în stare "în așteptare", corelate, valori rezultate conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**
Figura 62. Diferența per rundă dintre consumul mediu de putere (W) al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare", rezultată conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**
Figura 63. Consumul de putere (W) al aplicației **CPU** care rulează pe sistem NUMA în diferite runde
Figura 64. Optimizarea de consum de putere (W) al aplicației **CPU** care rulează pe sistem NUMA în diverse runde
Figura 65. Optimizarea de consum de putere (%) al aplicației **CPU** care rulează pe sistem NUMA în diverse runde
Figura 66. Consumul de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem NUMA în diferite runde
Figura 67. Optimizarea de consum de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem NUMA în diverse runde

Figura 68. Optimizarea de consum de putere (%) al aplicațiilor **Flops** și **Iops** care rulează pe sistem NUMA în diverse runde
Figura 69. Optimizarea de consum de putere în procente al aplicației **CPU** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA
Figura 70. Optimizarea de consum de putere în procente al aplicației **Flops** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA
Figura 71. Optimizarea de consum de putere în procente al aplicației **Iops** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA
Figura 72. Timpul de execuție al aplicației **Context Switch** pe sistem UMA, atât atunci când algoritmul NUMA-BTDM este aplicat, cât și când algoritmul nu este aplicat
Figura 73. Consumul mediu de putere (W) al sistemului UMA obținut în fiecare rundă pentru aplicația **Context Switch**
Figura 74. Consumul mediu de putere (W) al CPU-ului din sistemul UMA obținut în fiecare rundă la rularea aplicației **Context Switch**
Figura 75. Consumul mediu de putere (W) al CPU-ului și al întregului sistem UMA obținut în fiecare rundă la rularea aplicației **Context Switch**
Figura 76. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului obținută pentru fiecare rundă la rularea aplicației **Context Switch**
Figura 77. Consumul mediu de putere (W) al CPU-ului din sistemul UMA și al sistemului obținute în fiecare runda la rularea aplicației **Context Switch**
Figura 78.Valorile de consum de putere (W) al CPU-ului din sistemul UMA și al sistemului obținute la fiecare 1 s în runda 1 de rulare a aplicației **Context Switch**
Figura 79. Consumul mediu de putere (W) al sistemului UMA în stare "în așteptare" obținut pentru fiecare rundă
Figura 80. Consumul mediu de putere (W) al CPU-ului din sistemul UMA în stare "în așteptare" obținut pentru fiecare rundă
Figura 81. Consumul mediu de putere (W) al CPU-ului din sistemul UMA și al întregului sistem, ambele în stare "în așteptare", obținut pentru fiecare rundă
Figura 82. Diferența dintre consumul mediu de putere (W) al întregului sistem UMA și cel al CPU-ului, ambele în stare "în așteptare"
Figura 83. Consumul mediu de putere (W) al aplicației **Context Switch** obținut în fiecare rundă la rularea sistem UMA
Figura 84. Optimizarea de consum de putere (W) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem UMA
Figura 85. Optimizarea de consum de putere (%) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem UMA
Figura 86. Timpul de execuție al aplicației **Context Switch** pe sistem UMA, atât atunci când algoritmul NUMA-BTDM este aplicat, cât și când algoritmul nu este aplicat
Figura 87. Consumul mediu de putere (W) al sistemului NUMA obținut în fiecare rundă la rularea aplicației **Context Switch**
Figura 88. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA obținut în fiecare rundă la rularea aplicației **Context Switch**
Figura 89. Consumul mediu de putere (W) al CPU-ului și al întregului sistem NUMA obținut în fiecare rundă la rularea aplicației **Context Switch**
Figura 90. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului obținută pentru fiecare rundă la rularea aplicației **Context Switch**

Figura 91. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA și al sistemului obținute în fiecare runda la rularea aplicației **Context Switch**
Figura 92. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA și al sistemului obținute în runda 1 la rularea aplicației **Context Switch**
Figura 93. Consumul mediu de putere (W) al sistemului NUMA în stare "în așteptare" obținut pentru fiecare rundă
Figura 94. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA în stare "în așteptare" obținut pentru fiecare rundă
Figura 95. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA și al întregului sistem, ambele în stare "în așteptare", obținut pentru fiecare rundă
Figura 96. Diferența dintre consumul mediu de putere (W) al întregului sistem NUMA și cel al CPU-ului, ambele în stare "în așteptare"
Figura 97. Consumul de putere (W) pe sistem NUMA al aplicației **Context Switch** obținut în fiecare rundă
Figura 98. Optimizarea de consum de putere (W) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem NUMA
Figura 99. Optimizarea de consum de putere (%) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem NUMA
Figura 100. Optimizarea de consum de putere (%) al aplicației **Context Switch** obținută în fiecare rundă, prezentată comparativ pentru sistemele UMA și NUMA

Listă de coduri

Cod 1. Exemplu de cod sursă pentru analiza localizării datelor
Cod 2. Algoritmul de fuziune a buclelor
Cod 3. Exemplu de fir de execuție autonom în raport cu un alt fir autonom
Cod 4. Exemple de fir de execuție alăturat în raport cu firul generator respectiv un alt fir de pe același nivel în ierarhia de generare a firelor de execuție
Cod 5. Exemplu de fir de execuție amânat
Cod 6. Analiza algoritmului NUMA-BTLP apelată de fiecare dată atunci când este întâlnit un apel al funcției pthread_create
Cod 7. Aplicarea analizei algoritmului NUMA-BTLP pentru fiecare funcție în parte din codul de intrare
Cod 8. Analiza algoritmului NUMA-BTLP apelată de fiecare dată la întâlnirea în cod a unui apel al funcției pthread_create
Cod 9. Setarea afinității CPU a unui fir de execuție și maparea corespunzătoare afinității acestuia

1. Introducere

1.1. Contextul și motivația cercetării

Cercetarea vizează optimizarea aplicațiilor paralele C/C++ multi-threading cu număr mediu și mare de fire de execuție în special, doarece acestea consumă o cantitate considerabilă de energie [1]. În scopul optimizărilor de consum de energie și timp de execuție prin această teză, sunt incluse doar aplicațiile care utilizează biblioteca ***Pthreads*** [45] în crearea firelor de execuție, pentru obținerea paralelismului la nivel de task. În acest sens, am dezvoltat un algoritm de clasificare statică a firelor de execuție în trei categorii, denumit Non-Uniform Memory Access-Balanced Task and Loop Parallelism (***NUMA-BTLP***) [87], și un algoritm de mapare statică a firelor de execuție pe core-uri în funcție de tipul acestora, denumit Non-Uniform Memory Access-Balanced Thread and Data Mapping (***NUMA-BTDM***) [55]. Algoritmii urmăresc eliminarea dezavantajelor necunoașterii la compilare a aspectelor dinamice cum ar fi numărul de fire de execuție sau latențele operațiilor cu memoria.

Algoritmii de clasificare statică și de mapare statică a firelor de execuție optimizează atât timpul de execuție, cât și consumul de energie, ale aplicațiilor paralele mai sus menționate, la execuția acestora pe sisteme Non-Uniform Memory Access (NUMA), prin îmbunătățirea *localizării echilibrate a datelor* (conceptual este definit în cadrul sistemelor NUMA).

Majoritatea algoritmilor de mapare a firelor de execuție determină maparea acestora pe core-uri în mod dinamic, la rulare, ceea ce consumă timp de execuție și consum de energie, în plus, pe lângă timpul consumat cu apelul propriu-zis al funcției de mapare, pentru fiecare fir de execuție în parte. Apare necesitatea determinării mapării firelor de execuție pe core-uri la compilare, pentru a nu încărca procesul de rulare cu analiza care determină maparea. Dacă maparea este realizată la compilare, atunci aceasta depinde doar de caracteristicile statice ale codului.

Tipul unui fir de execuție este stabilit printr-un raționament original bazat pe următoarele considerente:

1. firele de execuție care nu au dependențe de date cu nici un fir se consideră a fi de tipul *autonom*. Fire autonome sunt de obicei firele care sunt create de firul principal și care execută procesări independente vreun alt fir;
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 au dependențe de date doar cu firul generator sunt considerate a fi de tipul *amânat*. Acestea au proprietatea că nu necesită execuție imediată, regăsindu-se de obicei printre ultimele fire create într-o funcție.

Principalele elementele de noutate ale tezei sunt definirea criteriilor de clasificare și clasificarea firelor de execuție în trei categorii (*autonome*, *alăturate* și *amânate*) și maparea statică bazată pe tipul firelor de execuție, mapare care elimină în mare măsură inexactitățile mapărilor statice în general, datorate schimbărilor în

comportamentul dinamic al aplicației paralele (acest aspect va fi detaliat în continuare în subcapitolul 1.4).

1.2 Scopul și obiectivele cercetării

Scopul cercetării este optimizarea atât a timpului de execuție cât și a consumului de energie pe sisteme NUMA, a aplicațiilor multi-threading C/C++, care utilizează paralelismul la nivel de task (și/sau la nivel de buclă), obținut prin folosirea bibliotecii Linux de calcul paralel ***Pthreads*** [45]. Relația între firele de execuție și task-uri este de unu-la-unu. Optimizarea provine de la îmbunătățirea localizării echilibrate a datelor la execuția acestor aplicații pe sisteme NUMA. Optimizarea acestora este realizată prin compilarea aplicațiilor utilizând compilatorul *Low-Level Virtual Machine* (**LLVM**) [2], și aplicarea, la compilare, a celor doi algoritmi prezentați în lucrare, care au fost implementați în etapa de optimizare a acestui compilator.

Obiectivele generale ale cercetării sunt următoarele:

A. Obiective generale legate de arhitectura hardware aleasă
   1. Descrierea caracteristicilor sistemelor multiprocesor și a sistemelor NUMA, în special, precum și prezentarea condițiilor care trebuie îndeplinite de un sistem pentru a putea fi configurat ca NUMA (Cap. 2.1).
   2. Realizarea unei analize comparative privind avantajele și dezavantaje sistemelor NUMA și UMA în utilizarea acestora în cadrul companiilor IT (Cap. 2.1).
   3. Prezentarea conceptului de mapare în contextul aplicării acesteia pe sisteme NUMA, cât și a caracteristicilor sistemelor NUMA care favorizează dezvoltarea de algoritmi de mapare a firelor de execuție (Cap. 2.1).

B. Obiective generale legate de compilatorul ales cu scopul obținerii de cod optimizat
   1. Alegerea unui compilator modern având optimizări care favorizează paralelismul la nivel de buclă pentru implementarea algoritmilor de mapare propuși în lucrare (Cap. 2.3).
   2. Descrierea optimizărilor din compilatorul modern **LLVM** [2] care îmbunătățesc paralelismul la nivel de buclă (Cap. 2.4).
   3. Descrierea unui algoritm de fuziune a buclelor pe care l-am implementat în **LLVM** [2] și care favorizează paralelismul la nivel de buclă (Cap. 2.4).

C. Obiective generale care deservesc scopul lucrării: optimizarea timpului de execuție și a consumului de energie la rulare prin optimizări statice
   1. Analiza modului în care faza de compilare îmbunătățește timpul de execuție și consumul de energie la rulare (Cap. 2.5).
   2. Analiza gradului de exactitate a predicțiilor la compilare asupra comportamentului dinamic al aplicației (Cap. 2.5).
   3. Implementarea unui algoritm de mapare a firelor de execuție la compilare ***NUMA-BTDM*** (*Balanced Thread and Data Mapping* - numele algoritmului indică rezultatul obținut de către acesta), în compilatorul **LLVM** [2], care împreună cu optimizările la compilare ale **LLVM** [2] să îmbunătățească și mai mult paralelismul (Cap. 3.6).

4. Analiza și prezentarea algoritmilor de mapare existenți în comparație cu algoritmul propus ***NUMA-BTDM*** (Cap. 2.6).
5. Analiza criteriilor statice care influențează utilizarea dinamică datelor și definirea unor tipuri de fire de execuție bazat pe aceste criterii (Cap. 3.1 și Cap. 2.8).
6. Implementarea unui algoritm de clasificare a firelor de execuție ***NUMA-BTLP*** [88] (*Balanced Task and Loop Parallelism* - numele algoritmului indică cele două tipuri de paralelism care pot fi prezente în programele multi-threading optimizate folosind algoritmii propuși), în compilatorul **LLVM** [2], care împreună cu optimizările la compilare ale **LLVM** [2] și algoritmul ***NUMA-BTDM*** [55], să automatizeze optimizarea paralelismul (Cap. 3.5).
7. Analiza modului în care politicile de planificare din Linux influențează efectul mapării firelor de execuție (Cap. 2.8).
8. Obținerea de rezultate experimentale pe aplicații de referință din industrie (Cap. 4.4-6).
9. Formularea concluziilor privind reducerea consumului de energie și a timpului de execuție, pe baza rezultatelor experimentale (Cap. 5 și Cap. 4.4-6)

1.3 Argumentarea necesității algoritmilor prezentați în teză

Algoritmii de mapare sunt necesari în special în sectorul economic comercial la optimizarea utilitarelor din departamentele de cercetare și dezvoltare din companiile IT cum ar fi aplicații de compilare a mai multor resurse în paralel sau la optimizarea serviciilor cloud de baze de date, de mail, de procesare de imagini sau de cartografiere.

Sistemul de operare pentru care se dezvoltă majoritatea algoritmilor de mapare este sistemul Linux, deoarece acesta suportă rularea unui număr foarte mare de fire de execuție în același timp și, este asigurată, în același timp, portabilitatea algoritmului de mapare pe toate distribuțiilor Linux.

Politicile de planificare a firelor de execuție decid ordinea în care sunt lansate în execuție firele. Sistemul de operare Linux are următoarele politici de planificare a firelor de execuție: ***SCHED_FIFO***, ***SCHED_RR***, ***SCHED_NORMAL, SCHED_BATCH***, ***SCHED_IDLE*** care acționează complementar cu maparea firelor de execuție în scopul îmbunătățirii performanței la execuția programelor paralele.

Adesea, aplicațiilor care sunt optimizate prin algoritmi de mapare sunt aplicații cu un număr mare și, în special, foarte mare de fire de execuție și/sau care procesează multe calcule distribuit sau în paralel.

Câteva exemple de aplicații paralele care pot fi optimizate prin algoritmi de mapare sunt aplicații de tipul sistem în timp real de tipul client-server, aplicații de monitorizare a parametrilor sistemului sau aplicații care realizează funcția de multi-tasking.

Un posibil obstacol în realizarea unui algoritm de mapare eficient este decizia asupra momentului mapării: la compilare sau la execuție. Fără a pune în calcul rularea algoritmilor de mapare sau migrare a firelor de execuție, doar execuția apelurilor de mapare ar crește timpul de execuție (cu 0.01 s pentru ~1000 de fire de execuție [1]) și consumul de energie. În cazul sistemelor în timp real, de exemplu, întârzierea răspunsului nu este de dorit.

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 elementele de noutate aduse de aceștia:

- Inserarea la compilare, în codul de intrare, după fiecare apel ***pthread_create***, de creare a unui fir de execuție, câte un apel ***pthread_setaffinity_np***, de setare a core-urilor pe care firul de execuție va rula. Astfel, indiferent dacă firul respectiv va fi 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.
- Maparea firele de execuție alăturate pe core-uri în aceeași manieră în interiorul fiecarui nod NUMA, ceea ce urmărește să prevină și să reducă posibilitatea ca celelalte aplicații care rulează în paralel cu aplicația optimizată de algoritmii prezentați în teză, să elimine din cache datele necesare mai multor fire de execuție alăturate. Astfel, este redus numărul de accese la distanță și, implicit, latențele datorate acestor accese.

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. Termenul de *performanță* va fi folosit în continuare în lucrare cu sensul de timp de execuție.

Scăderea consumului total de energie al aplicației optimizate printr-un algoritm de mapare provine de la scăderea puterii dinamice consumate de fiecare core, datorită optimizării utilizării memoriei cache prin îmbunătățirea localizarii echilibrate a datelor.

Performanța crește datorită optimizării localizării echilibrate a datelor pe sisteme NUMA, ca urmare a mapării pe aceleași core-uri a firelor care comunică și prin distribuirea uniformă a firelor de execuție pe core-uri de către algoritmul de mapare.

1.4 Argumentarea alegerii categoriilor de fire de execuție

Localizarea datelor poate fi *spațială* sau *temporală*. *Localizarea temporală* se referă la cazul în care datele sunt utilizate des într-o perioada scurtă de timp. În acest caz, optimizarea localizării temporale a datelor presupune lansarea imediată în execuție, pe core-ul deservit de memoria cache, a cât mai multor fire care utilizează datele din cache în respectiva perioadă scurtă de timp. În cazul în care un fir de execuție nu utilizează aceste date, execuția acestuia este întârziată. Întârzierea se datorează operației de aducere a datelor necesare execuției firului, din memoria principală, în memoria cache. Astfel, am definit firele a căror execuție este întârziată, ca făcând parte din categoria firelor de execuție "*amânate*". Pe de altă parte, dacă firele utilizează des într-o perioadă scurtă de timp aceleași date, am considerat că acestea fac parte din categoria firelor de execuție pe care le-am numit "*alăturate*".

Localizarea spațială se referă la cazul în care datele învecinate unor date, sunt utilizate si acestea mai mult, cu cât datele sunt utilizate mai mult. În acest context, optimizarea localizării spațiale a datelor presupune creșterea ratei de

utilizare a datelor din memoria cache. Este recomandat (si dovedit experimental în acest raport) ca firele de execuție care nu au dependențe de date cu alte fire să nu fie mapate pe aceleași core-uri ca acestea, ci pe un core separat, așa încât memoria cache să deservească în proporție cât mai mare execuția firului, crescând rata de obținere a datelor din memoria cache. Acest tip de fire le-am denumit fire de execuție "*autonome*".

În continuare voi prezenta o analiză a gestiunii memoriei cache pe un exemplu de cod, prin care doresc să evidențiez principiile de care am ținut cont în definirea criteriilor de încadrare ale unui fir de execuție în una dintre cele trei categorii: *autonom*, *alăturat* și *amânat*.

Primul nivel de memorie cache al unui sisteme NUMA (dar și orice cache al unui alt sistem) funcționează pe principiul etichetării începutului unei zone de memorie din cache, etichetă prin care este identificată și accesată acea zonă de memorie din interiorul cache-ului. De exemplu, printr-o *operație de aducere a unui anumit volum de date* din memoria principală în memoria cache și stocarea datelor respective în elementele unui vector, *operația de aducere a operanzilor din memorie* furnizează un anumit număr de elemente din totalul elementelor unui vector în primul nivel de cache al sistemelor NUMA.

În scopul realizării analizei, voi ilustra un exemplu de analiză a managementului intern al memoriei pe un cod sursă C++.

Cod 1. Exemplu de cod sursă pentru analiza localizării datelor

```
for(int i = 0; i < 1000000; i++)
{
        if(b[i] && check(i))
                b[i] = i;
        a[i] = b[i];
}
```

Considerând că ***funcția*** *check* apelată cu argumentul *i* operează asupra tuturor elementelor ***vectorului*** *b* din intervalul [primul element, al *i*-lea element], această funcție va umple primul nivel de cache cu elementele ***vectorului*** *b* atunci când acest nivel are dimensiunea de 2MB (dimensiune des întâlnită la sistemele NUMA pentru primul nivel de cache). Prin urmare, atunci când prima condiție de la if este adevărată, se va repeta următorul scenariu: presupunând că bucla ***for*** nu este la prima iterație, atunci când este apelată ***funcția*** *check*, *b* va suprascrie în primul nivel de cache elementele ***vectorului*** *a* care au fost aduse în iterația anterioară în primul nivel al cache-ului din memoria principală, ceea ce va determina ca aceleași elemente ale lui *a* să fie readuse în primul nivel de cache din memoria principală printr-o operație adițională de aducere a datelor din memorie. Astfel, fiecare iterație a buclei în care este executată ***funcția*** *check* va determina o operație de aducere a datelor din memorie în plus.

Extrapolând, se poate considera că fiecare din cei doi vectori sunt utilizați de câte un fir de execuție sau că ambele fire utilizează același vector, caz în care firele utilizează date în comun. Astfel, firele care utilizează date în comun și pe care le-am denumit *alăturate*, trebuie să acceseze aceste date din același loc și în aceeași perioadă scurtă de timp, așa încât aceste date să nu fie suprascrise de alte fire. Prin urmare, firele *alăturate* sunt mapate pe aceleași core-uri. Pe de altă parte, firele de execuție *autonome* nu utilizează date în comun cu nici un alt fir. Firele de execuție *amânate* sunt un hibrid între firele de execuție *alăturate* și cele *autonome*, acestea utilizând date în comun doar cu firul generator. Însă maparea firelor *amânate* pe

aceleași core-uri pe care este mapat și firul generator, ar conduce la suprascierea cache-ului cu datele necesare firului *amânat* și la eliminarea din cache a datelor utilizate de firele în raport cu care firul generator (al firului amânat) este alăturat. De aceea, firele *amânate* vor fi mapate pe un alt core, care este cel mai puțin încărcat până la acel moment (încărcarea este definită în teză ca fiind numărul de fire mapare pe core), scăzând astfel probabilitatea de a elimina din cache date utilizate de alte fire (numărul firelor mapate pe core fiind mai mic).

1.5. Studiul alegerii produsului software pentru implementarea algoritmilor

Analiza care a dus la decizia de a implementa algoritmul ***NUMA-BTDM*** în **LLVM** [2] a avut ca punct de pornire utilitatea practică a acestui algoritm. Dacă algoritmul ar fi fost implementat in biblioteca de calcul paralel ***Pthreads*** [45] atunci utilizatorii acestei biblioteci ar fi putut apela algoritmul ***NUMA-BTDM*** în programele pe care le dezvoltă și care fac apel la rutine din biblioteca ***Pthreads*** [45]. Apelul acestui algoritm necesită în prealabil etichetarea fiecărui fir de execuție ca făcând parte din una dintre cele trei categorii definite de algoritmul ***NUMA-BTDM*** pe baza caracteristicilor statice ale programului și anume: (1) fire de execuție autonome, (2) fire 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 și de alte fire și (3) fire de execuție amânate ca *timp NUMA* (timpul de acces 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 iar atunci când (cel puțin) unul din criteriile unei categorii este îndeplinit, firul de execuție face parte din acea categorie. 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.

Prin urmare, este necesară implementarea algoritmilor prezentați în teză într-un compilator, unde analiza dependențelor de date se poate realiza în mod automat asupra codului de intrare în reprezentare intermediară. Această reprezentare păstrează informații legate de structurarea codului, de succesiunea a instrucțiunilor, stochează instrucțiunile în sine, dar și informații legate de contextul de definire a variabilelor. Toate aceste informații sunt stocate într-o manieră intuitivă în compilatorul **LLVM** [2], în clase specifice precum Module, Function, Loop sau LoopInfo, care alcătuiesc o reprezentare intermediară specifică **LLVM** [2], denumită **LLVM IR** (Intermediate Representation) [2].

1.6. Alegerea tipului de sistem hardware

Am realizat o analiză cu scopul de a identifica pe ce sisteme se execută eficient un algoritm de mapare a firelor de execuție. În acest sens am identificat mai multe avantaje ale sistemelor Uniform Memory Access (UMA), precum și ale sistemelor NUMA.

Avantajele sistemelor UMA care favorizează dezvoltarea de algoritmi de mapare sunt următoarele:

- Consum de putere mai mic datorită numărului redus de unități de procesare. Sitemele UMA nu sunt scalabile la un număr mare de procesoare și consumă mai puțină energie datorită numărului redus de procesoare;
- Consum de putere mai redus în anumite cazuri datorită lipsei latențelor. Atunci când magistrala către memoria principală nu este congestionată, latențele nu există în cazul acestor sisteme, timpul de acces al datelor din memorie fiind considerat constant;
- Nu există acele latențe datorate acceselor la distanță, aceste accese fiind definite în contextual sistemelor NUMA. Ele reprezintă accese la datelor din memoriile altor noduri NUMA.

Avantajele sistemelor NUMA care contribuie la dezvoltarea de algoritmi de mapare sunt următoarele:

- Traficul dintre memorie și unitățile de procesare nu este congestionat, aceste sisteme având ierarhia de memorie multi-nivel. În cazul acestor sisteme, datele sunt obținute cu precădere din memoriile cache de pe nivelul 1 care deservesc fiecare câte un core sau din memoriile de pe nivelul 2 care pot fi utilizate în comun de mai multe core-uri sau care pot fi un nivel în plus de memorii cu aceleași proprietăți ca cele de pe nivelul 1, exceptând dimensiunea, care este, în acest caz, mai mare la nivelul 2. Astfel, traficul datelor în memoria principală și unitățile de procesare nu este congestionat;
- Creșterea vitezei de procesare la scalare. Sisteme NUMA sunt scalabile la un număr foarte mare de procesoare. Acest avantaj contribuie în special la dezvoltarea de algoritmi de mapare, deoarece aceștia optimizează viteza de procesare, optimizare necesară odată cu scalarea sistemului la un număr foarte mare de procesoare;
- Posibilitatea de creștere a ratei de obținere a datelor din memoria cache prin optimizarea localizării echilibrate a datelor;
- Latențe mai mici în cazul acceselor locale. Accesele locale reprezintă accese la memoriile din același nod NUMA și reprezintă un tip de accese la memorie definit în cadrul sistemelor NUMA. Acest avantaj al sistemelor NUMA contribuie, de asemenea, la răspândirea algoritmilor de mapare pentru aceste sisteme, deoarece algoritmii de mapare cresc numărul de accese locale, în detrimentul celor la distanță.

1.7. Structurarea tezei pe capitole

În Capitolul 2 sunt descrise caracteristicile sistemelor NUMA care favorizează dezvoltarea de algoritmi de mapare a firelor de execuție. În același capitol, sunt descrise aspecte ce țin de acuratețea predicțiilor la compilare ale algoritmilor de mapare asupra comportamentului dinamic al programelor paralele. De asemenea, sunt prezentate tiparele de mapare și reacția fiecăruia la schimbări în comportamentul dinamic al programelor paralele, precum și informațiile necesare în realizarea unei mapări eficiente. Tot în Capitolul 2 sunt prezentate optimizări software care eficientizează execuția programelor paralele pe aceste sisteme, precum și optimizări ale compilatorului **LLVM** [2] prin care se realizează acest lucru. În același capitol, am clasificat algoritmii de mapare, am prezentat comparativ câțiva algoritmi existenți de mapare a firelor de execuție și am descris factorii care influențează în mod direct eficiența mapării firelor de execuție pe sisteme NUMA, precum și alte aspecte conexe legate de alocarea memoriei în sistemele de operare actuale și politicile de planificare ale sistemului de operare Linux. În încheierea

Capitolului 2 este tratată reducerea consumului de energie prin maparea firelor de execuție. Capitolul 3 prezintă aspecte legate de algoritmul ***NUMA-BTLP*** [88]: reprezentarea sub formă de arbore a ierarhiei de generare a firelor de execuție, descrierea tipurilor de fire de execuție definite de algoritmul ***NUMA-BTLP*** [88], reprezentarea sub formă de arbore a tiparelor de comunicare între firele de execuție și implementările algoritmilor ***NUMA-BTLP*** [88] și ***NUMA-BTDM*** [55], prezentate în pseudocod. Capitolul 4 prezintă rezultatele experimentale și este structurat în mai multe subcapitole: un prim subcapitol care prezintă obiectivele specifice propuse la implementarea algoritmilor și criteriile de îndeplinire, două subcapitole care descriu metodologia de obținere a rezultatelor experimentale de timp de execuție, respectiv de consum de putere, urmat de 3 subcapitole, fiecare dintre ele tratând câte o aplicație de referință pentru care au fost obținute rezultate experimentale de timp de execuție și consum de putere, atât pe sistem NUMA cât și pe UMA. De asemenea, în cazul fiecăreia dintre cele 3 aplicații, rezultatele experimentale mai cuprind o comparație între rezultatele experimentale obținute pe sistem NUMA și cele obținute pe sistem UMA, precum și concluziile rezultatelor experimentale pentru acea aplicație. Capitolul 4 se încheie cu o secțiune de discuții legate de contribuțiile obținute și limitările algoritmilor prezentați în lucrare. Capitolul 5 include concluziile finale ale tezei.

2. Stadiul actual al cercetărilor în domeniile abordate în teză

2.1 Caracteristici hardware care au contribuit la dezvoltarea conceptului de mapare

2.1.1 Considerente generale ale sistemelor multiprocesor

Placa de bază a sistemelor multiprocesor conține socluri (socket-uri) descrise ca fiind adâncituri în placa de bază unde pot fi introduși pinii unei unități centrale de procesare (Central Processing Unit - CPU). Relația între un soclu și un CPU este de unu-la-unu sau unu-la-zero, însemnând că un soclu poate rămâne neumplut. Un CPU are mai multe core-uri fizice și mai multe core-uri logice. Core-urile logice sunt caracteristica unui core fizic de a procesa un număr de task-uri simultan egal cu numărul core-urilor logice suportate de acesta. CPU-urile pot fi configurate să folosească, sau nu, core-urile logice de care dispun prin setarea din BIOS referitoare la activarea sau dezactivarea *Hyper-Threading*. Core-urile logice au apărut datorită necesității de a utiliza în proporție cât mai mare resursele CPU-ului, astfel că, atunci când un core fizic execută anumite instrucțiuni și ar putea executa simultan și altele care nu interferează cu primele, acel core fizic suportă două sau mai multe core-uri logice (a căror număr maxim este stabilit prin fabricație). Core-urile logice, în cazul acestei lucrări, dar și în general, scad timpul de execuție prin multiplicarea numărului de unități de procesare cu același consum de energie statică deoarece suportul hardware este neschimbat.

2.1.2 Definirea conceptului de sistem NUMA

Pentru a face față tendinței impuse de dezvoltarea tehnologiei de a furniza cât mai rapid date, în cipurile moderne, a fost introdusă o structurare a memoriei pe mai multe nivele. Sistemele Non-Uniform Memory Access (NUMA) constau în mai multe nivele de cache privat și cache comun precum și mai multe controlere de memorie și se caracterizează prin faptul că timpii de acces la memorie sunt inegali [3]. Controlerele de memorie contribuie (alături de memoriile cache), la minimizarea latenței operațiilor de aducere a datelor din memorie, atunci când datele sunt plasate optim în memorie [4]. Prin optim se înțelege că acestea sunt gestionate de controlerul potrivit.

Fig. 1 ilustrează un exemplu de sistem de memorie NUMA cu patru noduri NUMA, fiecare nod NUMA conținând un controler de memorie și accesând o parte a memoriei. Fiecare nod NUMA conține două core-uri de procesare și fiecare core are câte un cache privat, plasat pe primul nivel de cache și un cache comun, plasat pe al doilea nivel de cache și care îl împarte cu celălalt core. Într-un astfel de sistem, un acces la memorie al unui core, poate fi deservit de către un cache local (din același nod NUMA, poartă numele de *acces local*), de către un controler de memorie prin

acces la memoria principală, sau de către un cache sau memorie asociate unui alt nod NUMA (denumit *acces la distanță*) [5]. La sistemele NUMA, latența și consumul de energie al acceselor la memorie depinde în mare măsură de cache-ul sau controlerul de memorie care deservește accesul realizat de core (accesele locale sunt de cele mai multe ori mai eficiente decât accesele la distanță) [5]. Este de asemenea important să nu se supraîncarce cache-urile sau controlerele de memorie, astfel reducându-se conflictele acceselor la resurse și îmbunătățindu-se utilizarea echilibrată a resurselor [5,7].

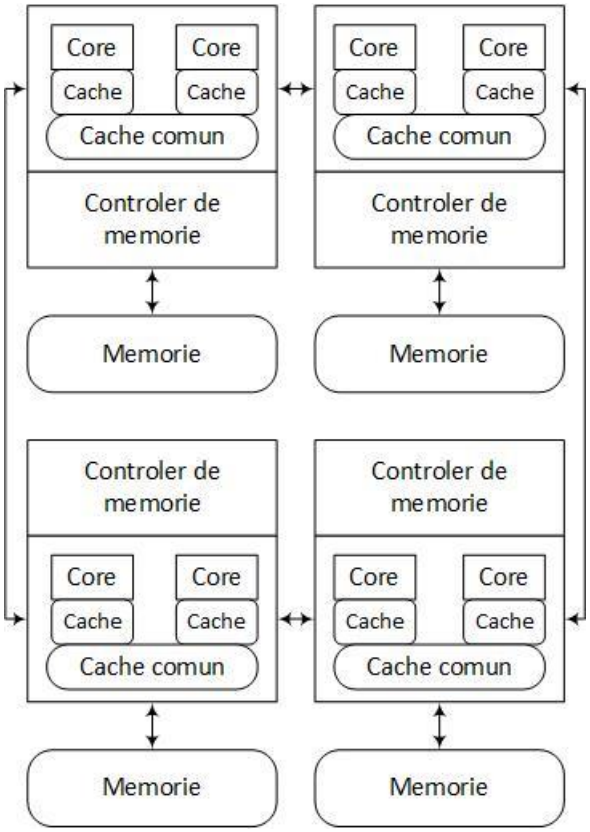

Figura 1. Sistem NUMA cu patru noduri de memorie fiecare având atașate două core-uri

2.1.3 Precondițiile configurării unui sistem ca NUMA

Relația între CPU-uri și nodurile NUMA este de unu-la-unu (cu câteva excepții, de exemplu procesorul Intel Knights Landing, abreviat Intel KNL). Pentru ca un sistem să fie NUMA trebuie să aibă mai multe noduri NUMA. adică să îndeplinească următoarele caracteristici:

1. Placa de bază trebuie să conțină mai multe socluri de CPU
2. Placa de bază trebuie să conțină mai multe subsisteme de memorie
3. Placa de bază trebuie să conțină mai multe CPU-uri

4. Doar subsistemele de memorie asignate CPU-urilor trebuie să fie utilizabile
   BIOS-ul trebuie să permită configurarea subsistemele de memorie pentru configurația NUMA sau cel puțin să ofere posibilitatea utilizării alternative a ambelor tipuri de acces la memorie, NUMA și UMA. Pentru a realiza setarea sistemului ca NUMA utilizând BIOS-ul, o cerință obligatorie este ca sistemul de operare să poată suporta activarea NUMA. De exemplu, pentru a verifica capabilitățile de activare a NUMA în cazul sistemului de operare Windows, poate fi consultat Task Manager, apoi selectat Performance unde sunt listate numărul de noduri NUMA sub denumire Sockets. Sub Linux comanda folosită pentru a afla dacă sistemul este NUMA este numactl.

Fiecare soclu poate avea slot-uri adiționale de memorie adiacente CPU-ului, în plus față de subsistemul de memorie al nodului NUMA aferent. De obicei sistemele cu un singur CPU au un singur sistem de memorie (excepție face procesorul Intel Xeon Phi 200 care poate fi opțional partiționat în 1, 2 sau 4 noduri NUMA prin configurarea MCDRAM pentru a funcționa ca și cache sau ca nod de memorie adițional). Există totuși și plăci de bază cu două socluri, prin urmare maxim două CPU-uri, care au un singur subsistem de memorie aferent unui CPU, acest CPU putând fi considerat doar el nod de memorie, sistemul per ansamblu nefiind NUMA. Subsistemele de memorie sunt asignate doar soclurilor populate cu câte un CPU. Astfel că, nodurile NUMA nu au nici o legătură cu virtualizarea memoriei, în sensul că un CPU are suport fizic pentru acces la propriul subsistem de memorie, cât și la subsistemele de memorie ale altor CPU-uri, accesele la subsistemele locale de memorie având latență mai mică decât accesele la alte subsisteme de memorie.

În continuare sunt menționate câteva exemple de procesoare dezvoltate de Intel și integrate în sisteme NUMA: procesoarele Intel Xeon în cazul cărora abrevierile din denumirile procesoarelor specifică numărul maxim de socluri de CPU care pot fi utilizate pentru plasarea acestor CPU-uri pe placa de bază. De pildă, procesorul Intel Xeon E5-2620 are două socluri de CPU iar procesorul Intel Xeon E5-8620 are opt socluri de CPU. Alte procesoare din aceeași familie sunt procesoarele Intel Xeon 6C E5-2609v3 (având ca și client pe producătorii de PC-uri/servere Lenovo și Dell) și Intel Xeon 6C E5-2620v3 (având ca și client pe Lenovo).

În cealaltă categorie de procesoare, care nu sunt NUMA, se încadrează procesorul Intel Core i7 care suportă un singur socket, deci un singur nod de memorie (care poate avea 1, 2 sau mai multe canale de memorie).

2.1.4 Analiză SWOT: Utilizarea sistemelor NUMA în industrie, în detrimentul sistemelor UMA

O analiză SWOT presupune identificarea următoarelor considerente: puncte tari, puncte slabe, oportunități oferite și amenințări. Primele două considerente reprezintă ceea ce există deja, iar ultimele două vizează viitorul.

În urma identificării celor patru puncte, se realizează un proces decizional care presupune următoarele:

1. construirea (cercetării) pe baza punctelor tari
2. eliminarea punctelor slabe (prin intermediul cercetării realizate)
3. exploatarea oportunităților (prin obținerea de rezultate experimentale favorabile)
4. îndepărtarea amenințărilor

Tabelul 1 reprezintă analiza strategică SWOT cu tema "Utilizarea sistemelor NUMA în industrie, în detrimentul sistemelor UMA", în care se prezintă comparativ sistemele NUMA și cele UMA din perspectiva cererii comerciale a acestora. Această analiză mi-a direcționat cercetarea și logica algoritmilor realizați către sistemele NUMA, în direcția îmbunătățirii timpului de execuție și a consumului de energie la execuția programelor paralele pe aceste sisteme, fără excluderea posibilității execuției algoritmilor pe sisteme UMA, cu păstrarea sau îmbunătățirea performanței. Cei doi algoritmi de optimizare la compilare din lucrare sunt meniți să-și, dovedească aplicabilitatea practică la execuția, nu numai pe sistemele performante NUMA, unde sunt așteptate îmbunătățiri considerabile, dar și pe alte sisteme NUMA, unde este posibilă o optimizare. În concluzie, prin algoritmii propuși în această lucrare voi urmări:

1. creșterea numărului de accese locale la memorie (în interiorul nodului NUMA) în detrimentul celor la distanță (la subsistemele de memorie ale unui alt nod NUMA)
2. creșterea ratei de obținere a datelor din memoria cache prin îmbunătățirea localizării echilibrate a datelor
3. îmbunătățirea timpului de execuție și a consumului de energie prin maparea eficientă a firelor de execuție pe core-uri astfel încât datele să fie accesate de nodul NUMA care le stochează, în majoritatea acceselor la aceste date
4. obținerea informațiilor hardware precum numărul de core-uri sau numărul de core-uri logice per CPU îmi va permite adaptarea dinamică a algoritmilor la orice tip de sistem NUMA făcând aceste optimizări portabile
5. îmbunătățirea aspectelor execuției pentru un număr mare de fire de execuție/core-uri
6. unele dintre cele mai noi și performante sisteme NUMA nu sunt comerciale, prin urmare algoritmii nu trebuie să degradeze timpul de execuție și consumul de energie la execuția programelor paralele pe sistemele comerciale, indiferent de tipul sau anul fabricației arhitecturii hardware

Aspectele de mai sus se regăsesc doar în cazul sistemelor NUMA, ca atare, pentru a le optimiza, am ales aceste sisteme în detrimentul sistemelor UMA.

2.1.5 Definirea conceptului de mapare și caracteristicile sistemelor NUMA care favorizează dezvoltarea de algoritmi de mapare

În contextul apariției sistemelor NUMA, care permit *accese locale* la cache, mai puțin costisitoare decât cele *la distanță* (din punct de vedere al performanței și al consumului de energie), s-au dezvoltat algoritmi de mapare a datelor și a firelor de execuție pentru aceste sisteme, care favorizează *accesele locale* în detrimentul celor *la distanță* [5].

Maparea se bazează pe conceptul de *afinitate*. Există două tipuri de *afinitate*: *afinitatea firelor de execuție între ele* și *afinitatea unui fir de execuție*.

Afinitatea firelor de execuție între ele. Maparea datelor este o mapare a paginilor de memorie pe controlere de memorie astfel încât firele de execuție deservite de controlerul de memorie, să utilizeze cu precădere față de alte fire, datele din paginile de memorie asignate controlerului. Consecința mapării optime a datelor este că mai multe fire de execuție accesează aceeași dată în mod optim [8]. Când două fire de execuție utilizează aceleași date, se spune că acestea au o afinitate între ele [5]. Legătura între firul de execuție și datele utilizate se realizează

Tabel 1. Analiză strategică SWOT: Utilizarea sistemelor NUMA în industrie, în detrimentul sistemelor UMA

Puncte tari (reprezintă ceea ce există)	**Puncte slabe (reprezintă ceea ce există)**
- latența mai mică a acceselor locale la memorie datorită proximității subsistemului de memorie față de CPU în cazul NUMA, comparativ cu accesul la memorie în timp constant la UMA - mai multe nivele de cache la NUMA ceea ce crește rata de obținere a datelor din cache - sistemele NUMA sunt sisteme multiprocesor scalabile datorită proximității datelor față de locul de procesare al acestora determinând creșterea vitezei de procesare - traficul datelor din memorie către unitățile de procesare nu este congestionat la NUMA spre deosebire de sistemele UMA cu număr mare de core-uri	- programele optimizate pentru un anumit sistem NUMA nu sunt portabile pe alte sisteme NUMA cu configurație hardware diferită (exemplu: număr diferit de core-uri) - nu se poate detecta la compilare dacă sistemul este NUMA, informație care ar fi utilă - prețul ridicat
Oportunități (reprezintă viitorul)	**Amenințări (reprezintă viitorul)**
- favorizează integrarea a cât mai multe core-uri în arhitecturile cu număr mare de core-uri: many-core (exemplu: Intel Many Integrated Core Architecture, abreviat MIC) - favorizează dezvoltarea de servere care sunt accesate de un număr în creștere de utilizatori păstrând aceeași performanță ridicată a serviciilor oferite (exemplu: servere de baze de date, de mail, de procesare de imagini, de cartografiere etc. necesită sisteme NUMA). Principalii clienți ai procesorului NUMA Intel Xeon sunt Facebook, Google și Amazon - sistemele NUMA vor avea o răspândire tot mai largă și de aici a pornit necesitatea reducerii consumului de energie devreme ce numărul de unități de procesare din arhitecturile cu număr mediu de core-uri (multi-core) și număr mare de core-uri (many-core) este în continuă creștere	- optimizările pentru anumite sisteme NUMA ar putea degrada performanța sau consumul de energie pe alte sisteme NUMA - înlocuirea sistemelor UMA cu sisteme NUMA cere dezvoltarea într-un ritm accelerat a cât mai multe optimizări software pentru sisteme NUMA astfel încât dezvoltarea software-ului să țină pasul cu suportul hardware performant

prin controlerul de memorie iar maparea datelor își dovedește eficiența ca o consecință a utilizării optime a controlerelor de memorie [5].

Afinitatea unui fir de execuție. Mai există și un al doilea tip de afinitate, între firele de execuție și datele pe care acestea le utilizează [5]. Maparea firelor de execuție este o mapare a unui firul de execuție pe core-urile pe care acesta se va executa. Asocierea fir de execuție-core poartă denumirea de afinitate CPU a firului de execuție. Scopul acestei asocieri este accesarea în mod optim a datelor utilizate în comun de mai multe fire de execuție. O mapare eficientă a firelor de execuție îmbunătățește performanța și consumul de energie datorită utilizării în mod optim a cache-ului și a interconexiunilor, ca urmare a mapării [5].

Ambele tipuri de mapări pot să îmbunătățească localizarea și echilibrul acceselor la memorie [6,8]. Utilizate împreună, maparea datelor și a firelor de execuție își pot influența reciproc efectul și pot furniza rezultate mai bune decât aplicate separat [9].

2.2 Eficientizarea execuției aplicațiilor paralele prin optimizări la compilare

Acest subcapitol este menit să argumenteze importanța optimizărilor la compilare în optimizarea aspectelor ce țin de execuția paralelă, a alegerii corecte a acestor optimizări în funcție de caracteristicile statice ale codului precum și a ordinii optime de aplicare.

2.2.1 Tipare de cod sursă pentru care se aplică optimizările la compilare

Optimizările la compilare se pretează a fi aplicate cu precădere codului iregular (prezintă dependențe de date și accese indirecte la elementele tablourilor) deoarece performanța acestui tip de cod este degradat datorită timpului adițional scurs până când datele ajung la unitățile de procesare, parcurgând ierarhia de memorie [10].

Performanța *codului stencil* (acest tip de cod folosește tablouri în corpul buclelor și pentru fiecare element din tablou, există un set de elemente vecine numit stencil și un număr fix de operații aplicate fiecărui stencil [11]) și performanța codului care folosește programare dinamică depind într-o măsură foarte mică de datele de intrare, ceea ce face ca aceste clase de programe să se preteze testelor de eficiență a optimizărilor la compilare [12].

MODESTO [11] este un optimizor pentru *codul stencil* care sugerează transformări pentru o anumită arhitectură țintă folosind tehnici bazate pe raționamente matematice cu scopul de a selecta transformările care împreună generează rezultatul optim.

Un produs software ca MODESTO [11] ar putea fi extins pentru a selecta acele transformări care aplicate împreună îmbunătățesc execuția pe sistemele NUMA.

2.2.2 Validarea respectării standardelor limbajelor de programare în implementările optimizărilor la compilare

În programele concurente, optimizările compilatorului pot să schimbe accesele la memorie într-un mod care nu este în concordanță cu standardul ***C11*** [13]. Pentru a obține cod optimizat în limbaj de nivel înalt din cod neoptimizat, în [13] se folosește un generator de programe concurente care determină lansarea optimizărilor din compilatorul **LLVM** [2] asupra codului ***C*** neoptimizat, rezultând cod ***C*** optimizat. Pentru a verifica concordanța cu ***C11***, codul ***C*** optimizat este testat prin intermediul unui validator [13].

Astfel de validatoare pot fi utilizate la testarea criteriului de independență a două bucle: două bucle sunt dependente dacă toate iterațiile primei bucle trebuie să fie parcurse și execuția acesteia să se finalizeze ca cea de-a doua să se poată lansa în execuție. De exemplu, dacă două bucle sunt dependente atunci testul de memorie folosind un astfel de validator ar trebui să eșueze atunci când optimizarea de fuziune

a buclelor este aplicată în mod eronat, schimbând logica programului. Astfel se poate valida corectitudinea algoritmului de fuziune a buclelor.

În categoria optimizărilor de memorie la compilare intră și optimizările care determină o gestiune optimă a memoriei cache [14].

2.2.3 Influența ordinii de aplicare a optimizărilor la compilare asupra eficienței execuției

Utilizarea optimizărilor la compilare într-o anumită ordine poate îmbunătăți și mai mult performanța. Identificarea secvenței optime de aplicare a optimizărilor este o problemă care se rezolvă utilizând programarea non-liniară prin implementarea unui model predictiv instruit off-line să genereze o serie de permutări de opțiuni la compilare [15].

Modelul predictiv din [15] selectează la fiecare pas următoarea optimizare care maximizează performanța până în acel moment. Selecția se realizează pe baza unui vector de caracteristici ale codului, colectate în decursul mai multor rulări [15].

Un alt model predictiv [15], bazat de asemenea pe inteligență artificială, determină pentru fiecare arhitectură cea mai potrivită optimizare pentru fiecare porțiune de cod, decizie luată pe baza caracteristicilor statice ale codului. În aceeași măsură, și algoritmii ***NUMA-BTDM*** [55] și ***NUMA-BTLP*** [87] determină maparea firelor de execuție pe core-uri la compilare pentru orice tip de arhitectura harware.

Tehnica de compilare *mapping deviation* furnizează informații utilizatorului sau compilatorului legate de semantica codului cu scopul restructurării acestuia [16]. Restructurarea se realizează în așa manieră încât optimizările să fie eficient aplicate mai apoi la compilare [16]. Predicția asistată de utilizator a celei mai optime secvențe de optimizări la compilare îmbunătățește performanța mai mult decât restructurarea manuală a codului de către utilizator sau aplicarea automată a optimizărilor la compilare [17].

2.3 Infrastructura de compilare LLVM

Acest subcapitol prezintă avantajele alegererii infrastructurii de compilare **LLVM** [2] pentru integrarea optimizărilor și a algoritmilor de mapare propuși în teză, care îmbunătățesc execuția programelor paralele. Unele dintre aceste avantaje sunt: existența în **LLVM** [2] a facilității de *vectorizare* care reduce numărul de operații cu memoria [18] și particularitățile *codului instrumentat* (adică structurile de date folosite de compilator în faza de optimizare și operațiile definite cu aceste date) care favorizează implementarea și integrarea optimizărilor la compilare.

2.3.1 Prezentare comparativă a particularităților infrastructurii de compilare LLVM în raport cu cele ale colecției de compilatoare GCC

Inițial o infrastructură de compilare pentru limbajele ***C/C++***, **LLVM** [2] își extinde în prezent partea de front-end pentru a suporta limbaje precum ***Ada*** sau ***Fortran***. **LLVM** [2] este conceput ca un sistem modular cuprinzând module ca

optimizorul, generatorul de cod, asamblorul, linker-ul și în trecut optimizorul Polly [19], în prezent un pas separat de optimizare în infrastructură.

Pentru a evita duplicarea analizei la compilare pentru fiecare limbaj de programare suportat, compilatoarele actuale precum **LLVM** [2] (sau **GCC** [20]), folosesc o reprezentare intermediară care se aplică pentru toate limbajele imperative. De exemplu, rezultatul traducerii buclelor în reprezentare intermediară este constituit din salturi între blocuri de bază, referințele la zone de memorie devin accesări prin intermediul pointerilor ale valorilor stocate în zone de memorie iar expresiile sunt traduse în cod cu trei adrese (cele trei adrese sunt formate din adresele celor doi operanzi asupra cărora se aplică un operator binar și adresa rezultatului) [21].

Ca și în cazul **GCC** [20] și a altor infrastructuri de compilare moderne, faza de optimizare a **LLVM** [2] este bazată pe tipul de reprezentare intermediară Static Single Assingment (SSA) [22] care presupune ca fiecare variabilă să fie asignată o singură dată și să nu se folosească o variabilă care nu este în prealabil inițializată. Această formă de reprezentare intermediară ajută la definirea în mod exhaustiv a relației definire-utilizare în cazul variabilelor, prin care se simplifică analizele făcute de mai multe optimizări precum eliminarea sub-expresiilor comune și mutarea codului invariant (execuția acestuia nu depinde de iterațiile buclelor) în afara buclelor (de obicei înainte) [23].

Fiind o infrastructură de compilare modernă, **LLVM** [2] dezactivează multe optimizări în mod implicit și permite utilizatorilor să le activeze pe cele care îmbunătățesc cel mai mult performanța codului compilat, prin specificarea unor nivele de optimizare care să fie aplicate codului (un nivel de optimizare produce cea mai mare îmbunătățire a performanței pentru majoritatea aplicațiilor compilate) [15].

LLVM [2] optimizează buclele parțial vectorizabile prin maximizarea utilizării *Single Instruction Multiple Data* (SIMD), adică prin creșterea dimensiunii *intrucțiunilor vector* (creșterea numărului de operații din *instrucțiunea vector*), și în același timp minimizează numărul de operații cu memoria fără a introduce erori de memorie sau excepții numerice [24].

2.3.2 Modelul poliedral

2.3.2.1 Caracteristicile modelului poliedral

Un optimizor conceput pe baza modelului poliedral se axează pe analiza dependențelor și pe transformări automate care au impact asupra generării de cod [25]. În principal, modelul poliedral are la bază raționamente matematice legate de corectitudinea transformărilor efectuate asupra buclelor (folosind algebra liniară, programarea liniară ș.a.) [26]. Un exemplu de astfel de optimizor poliedral este Polly [19], actualmente un pas de optimizare în **LLVM** [2].

În cadrul compilatoarelor ca **GCC** [20] și **LLVM** [2], care își structurează optimizările sub forma unor pași care se aplică reprezentării intermediară a codului, identificarea buclelor din reprezentarea intermediară, care pot fi optimizate utilizând modelul poliedral, conduce la timp de compilare adițional. Principalul avantajul al modelului poliedral este scăderea timpului de execuție al programelor. Deși sunt aduse beneficii considerabile la execuție, există totuși și un dezavantaj al analizei

poliedrale și anume că aceasta va crește timpul de compilare pentru codul care conține numeroase bucle, în vreme ce pentru codul care are un număr redus de bucle, este consumat un timp de compilare considerabil fără a obține nici un beneficiu la execuție.

Instanța algoritmului SCoP [27] care este implementată în optimizorul poliedral Polly [19], este dedicată compilatoarelor care suportă limbaje de programare imperative. Algoritmul extrage din arborele sintactic abstract (Abstract Syntax Tree) instrucțiuni sau operatori cum ar fi etichete, apeluri de funcții, operatori de egalitate și comparație asupra întregilor, atunci când acestea se află în corpul buclelor și le furnizează mai departe analizei poliedrale. Recent modelul poliedral a avansat cu raționamentele matematice ajungând cu studiile la reprezentarea matematică a buclelor încuibate [19].

Analiza poliedrală exclude, de cele mai multe ori, construcțiile care pot provoca schimbări în fluxul de control al programului cum ar fi expresii condiționale care depind de date din memorie sau apeluri de funcții care generează cazuri excepție [27].

Modelul poliedral poate favoriza atât parallelismul, cât și vectorizarea. Un exemplu în acest sens este chiar Polly [19,27].

2.3.2.2 Optimizări care se încadrează în modelul poliedral

Optimizările la compilare ale **LLVM** [2] care utilizează modelul poliedral prezentat în continuare, sunt implementate în optimizorul Polly [19], în prezent un pas de optimizare separat în compilatorul **LLVM** [2]. Unele dintre aceste optimizări pot îmbunătăți timpul de gestiune a firelor de execuție în cazul paralelismului la nivel de buclă (la acest tip de paralelism relația între o iterație a unei bucle și un fir de execuție este de unu-la-unu adică un fir execută o iterație a buclei), fie prin reducerea numărului acestora (exemplu: *fuziunea buclelor*), fie prin distribuirea a mai puține operații în virgulă flotantă fiecărui fir de execuție, crescând granularitatea paralelismului (exemplu: *divizarea buclelor*). Datorită faptului că marea majoritate a optimizărilor la compilare din **LLVM** [2] se bazează pe modelul poliedral (exemplu: *fuziunea buclelo*r - studiată și implementată în cadrul unor lucrări științifice proprii - care scade granularitatea paralelismului prin reducerea numărului de fire de execuție și îmbunătățește localizarea datelor), voi explica în continuare acest model prin exemplificarea a câteva cazuri particulare de optimizări din cadrul acestuia.

O *buclă afină*, care se mai numește și *regulară*, poate fi optimizată utilizând *modelul poliedral* chiar și atunci când este conținută în corpul unei alte bucle care este *iregulară*, deși modelul poliedral nu suportă optimizarea *buclelor iregulare*. O altă variantă posibilă este ca întreaga construcție de bucle încuibate să fie tratată ca iregulară [28] și să nu mai fie optimizată.

Modelul poliedral nu poate reprezenta operații aritmetice între o variabilă inductivă și o alta în corpul unei bucle. Un astfel de exemplu de operație aritmetică frecvent întâlnită este i*N din expresia A+i*N+j care este liniarizarea accesului la elementul (i,j) a unui tablou bidimensional de dimensiune NxN. Instanța algoritmului SCoP [27] implementat în Polly [19] deliniarizează astfel de accese la memorie și verifică dacă locația de memorie este aceeași ca cea a elementului (i,j) a tabloului bidimensional (această informație este pierdută în urma traducerii codului din front end-ul compilatorului către middle-end reprezentat de modulul de optimizare).

Un exemplu de optimizare la compilare care se încadrează în modelul

poliedral este algoritmul de fuziune a buclelor din [29], care realizează analize asupra dependențelor de date între bucle cu scopul fuziunii acestora în cazul în care nu există astfel de dependențe. Atunci când fiecare dintre cele două bucle candidate la fuziune sunt paralelizate la nivel de iterații, vor rezulta un număr de 2xN fire de execuție, unde N este numărul de iterații al ambelor bucle. Fuziunea acestora înjumătățește numărul de fire de execuție, contribuind la reducerea timpului alocat gestiunii firelor de execuție, deci la îmbunătățirea performanței paralelismul la nivel de buclă la rularea acestor programe.

2.4 Optimizările infrastructurii de compilare LLVM

Sunt prezentate în continuare optimizări la compilare ale buclelor care optimizează paralelismul la nivel de buclă. Acest tip de paralelism poate fi colocalizat în același program sursă cu paralelismul la nivel de task care este îmbunătățit prin algoritmii de mapare propuși în teză. Optimizările asupra buclelor pot influența eficiența algoritmilor de mapare în sens pozitiv, atunci când, în urma aplicării optimizărilor și apoi a algoritmilor de mapare, calculele sunt mai uniform distribuite pe core-uri decât atunci când sunt aplicați doar algoritmii de mapare, sau, în sens negativ, atunci când rezultă o mai mică uniformitate după aplicarea optimizărilor.

Alături de optimizările prezentate, acest subcapitol mai include și prezentarea implementării originale a *algoritmului* ***loop fusion*** în compilatorul **LLVM** [2]. Activarea optimizărilor la compilarea programelor paralele, împreună cu algoritmii de mapare a firelor de execuție ***NUMA-BTDM*** și de clasificare a firelor ***NUMA-BTLP*** [87], m-au condus la obținerea unor îmbunătățiri ale aspectelor conexe care influențează execuția programelor paralele:

1. localizarea datelor
2. rata de obținere a datelor din memoria cache
3. volumul de date mutate din regiștri în memorie

și a aspectului ce ține direct de paralelizare: gestiunea firelor de execuție. Prin urmare, voi detalia în continuare optimizările și aspectele execuției programelor paralele pe care aceste optimizări le îmbunătățesc.

2.4.1 Optimizări la compilare ale buclelor care îmbunătățesc performanța execuției codului

Din timpul total de execuție al aplicațiilor care realizează un număr mare de calcule numerice, o parte considerabilă este alocată execuției buclelor încuibate, așadar prin optimizarea acestor construcții folosind modelul poliedral, este redus un procent semnificativ din timpul de execuție al aplicațiilor [19].

Fuziunea buclelor (***loop fusion***: fuziunea a două bucle în una singură) reduce timpul de execuție scurs cu gestiunea buclelor (incrementare variabilă inductivă, testare condiție de oprire) [30] și îmbunătățește *localizarea spațială* și *temporală* [31]. Pe de altă parte, în urma acestei optimizări, în cazul sistemelor NUMA este eliminat un volum mai mare de date din primul nivel al memoriei cache (*L1 cache thrashing*) și sunt mutate un număr mai mare de variabile din regiștri în memorie (register spilling) odată cu creșterea dimensiunii corpului buclei fuzionate [30]. Mutarea variabilelor din regiștri în memorie afectează direct operația de software pipelining în sensul că un număr redus de regiștri disponibili împiedică operația de *pipelining* [30].

Aplicarea atât a optimizării de fuziune a buclelor cât și a optimizării de ***partiționare a buclelor*** (***loop tiling***: partiționarea spațiului iterațiilor în bucăți mai mici așa încât o partiție să încapă în cache) se dovedește a fi eficientă în îmbunătățirea localizării datelor în cazul codului stencil [13]. Optimizarea de partiționare a buclelor îmbunătățește localizarea datelor, dar necesită sincronizări adiționale la limitele unei partiții a spațiului iterațiilor [13].

Fuziunea buclelor (în unele cazuri), partiționarea buclelor și ***interschimbarea buclelor*** (***loop interchange***: interschimbarea variabilelor de inducție a două bucle încuibate aflate pe nivele consecutive în construcția de bucle încuibate) optimizează accesele la memoria cache, rezultând într-un volum ridicat de date obținute din memoria cache [13].

Pentru bucle cu un număr mare de instrucțiuni în corp, optimizarea cu efect opus fuziunii buclelor, ***divizarea buclelor*** (***loop fission***: divizarea unei bucle în mai multe bucle cu același interval de iterare) reduce numărul de variabile mutate din regiștri în memorie, dar poate crește timpul de execuție alocat cu gestiunea buclei [31]. Prin izolarea dependențelor de date inter-iterații ale unei bucle într-o buclă separată, faza de optimizare poate aplica operația de pipelining asupra instrțiunilor SIMD preluate din corpul buclelor divizate care nu prezintă dependențe de date inter-iterații [31].

Derularea buclei (***loop unrolling***: replicarea corpului buclei în interiorul acestuia) reduce timpul de gestiune al buclei și este cel mai eficient pentru bucle care au corpul mic. Pentru că această optimizare crește dimensiunea corpului buclei, ea favorizează vectorizarea (automată) și/sau parallelizarea de tip SIMD, dar poate produce depășirea cache-ului de instrucțiuni (instruction cache overflow) sau mutarea unui numărul mare de variabile din regiștri în memorie. Performanța optimizării de derulare a buclelor depinde de numărul de ori în care corpul buclei este replicat [30]. Optimizările centrate pe date se ocupă cu studiul și implementarea utilizării eficiente a lățimii de bandă a memoriei [13]. *Planificarea task-urilor centrată pe îmbunătățirea localizării datelor* este o tehnică de reordonare a instrucțiunilor în corpul unei bucle cu scopul de a obține tipare optime de acces la memorie care îmbunătățesc performanța [32].

2.4.2 Descrierea și încadrarea în necesitate a optimizărilor la compilare ale buclelor și a altor optimizări care îmbunătățesc paralelizarea

Criteriul comercial. Atunci când scăderea performantei la excluderea unei optimizări care favorizează paralelizarea nu este atât de mare comparativ cu costul implementării acesteia, dezvoltatorii compilatoarelor comerciale sunt puși în postura de a decide dacă tehnica respectivă va fi sau nu implementată [33].

Criteriul evolutiv. Numărul de core-uri este în continuă creștere în cazul arhitecturilor moderne precum sistemele NUMA. În cazul acestora, distanța dintre core-uri și unitățile de memorie este neuniformă. De exemplu, cache-urile dintr-un nod NUMA sunt mai apropiate de unitățile de procesare ale nodului decât memoria DRAM [12], dar cache-urile unui alt nod pot fi mai departe de nodul NUMA, decât memoria DRAM. Tot în cazul sistemelor NUMA, dacă mai multe core-uri împart același cache care aparține primului nivel de cache (L1), aplicațiile care rulează pe acestea își pot influența una alteia performanța, deoarece fiecare aplicație își aduce datele de care are nevoie în cache-ul L1, eliminând datele celorlalte aplicații [34]. Acest lucru poartă numele de neconcordanță a cache-ului [34] sau cache

contention. Neconcordanța cache-ului este principala cauză de degradare a performanței în cazul sistemelor multi-core [34].

Criteriul de asigurare a performanței. Timpul petrecut cu gestiunea firelor de execuție este un alt factor major care infuențează performanța execuției aplicațiilor paralele. Creșterea granularității paralelizării și a numărului de bucle paralelizabile poate scădea performanța datorită creșterii timpului de gestiune a task-urilor la execuția pe arhitecturile multi-core [35]. Paralelizarea parțială se aplică codului care nu are un procent paralelizabil mare. Dacă astfel de programe ar fi în întregime paralelizate, atunci timpul de execuție total la care se adaugă cel alocat gestiunii task-urilor ar fi mai mare [35] decât dacă programele nu ar fi paralelizate sau ar fi paralelizate doar parțial.

Timpul total de execuție poate fi minimizat dacă timpul scurs cu gestiunea task-urilor este eliminat pe cât posibil din fluxul critic de execuție. Acest lucru se realizează prin generarea a cât mai puține task-uri, ceea ce reduce apoi numărul de comutări între stările "*"în așteptare"*" și "*activ*" sau de mutări de pe un core pe altul, task-urile fiind executate mai mult timp înainte de a-și schimba starea [36].

Tehnicile la compilare care favorizează paralelizarea automată prin reducerea timpului scurs cu gestiunea task-urilor sunt următoarele [35]: ***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.

Timpul scurs cu gestiunea task-urilor este infuențat de latența de comunicare, care poate fi mascată dacă se suprapune cu faza de comunicare [37], în cazul modelului de calcul paralel *Message Passing Interface* (MPI) [38]. Fuziunea buclelor și partiționarea buclelor favorizează suprapunerea dintre calcule și comunicare, prin creșterea paralelismului la nivel de instrucțiune și a ratei calculelor, prin îmbunătățirea localizării datelor și prin reducerea timpului scurs cu gestiunea buclei [35].

În scopul reducerii comunicării între unitățile de procesare, compilatoare precum Maunam [37] folosesc următoarele transformări: extinderea și reducerea dinamică a dimensiunii tabloului, extragerea parțială de iterații din buclă (partial loop peeling: este un caz special de divizare a buclelor prin care se execută în afara buclei primele sau ultimele câteva iterații ale buclei) și alinierea buclei (loop alingment: minimizarea acceselor la memorie din corpul buclei [39]). *Paralelizarea* și *vectorizarea buclelor* aplicate împreună cu fuziunea sau cu divizarea buclelor îmbunătățesc localizarea echilibrată a datelor la execuția aplicațiilor [40].

Criteriul de optimizare. Spre deosebire de alți algoritmi de fuziune a buclelor care fuzionează mai multe bucle într-o singură trecere realizând analize complexe pentru identificarea dependențelor care pot apărea între instrucțiunile din corpurile multiplelor bucle, algoritmul pe care l-am implementat în **LLVM** [2] fuzionează doar două bucle într-o trecere și este apelat pentru fiecare buclă din cod, cautându-se o buclă cu care aceasta să fuzioneze.

Algoritmul de fuziune a buclelor prezentat în această lucrare este aplicat programelor sursă ***C/C++***.

Algoritmul de decizie asupra fuziunii constituie o parte componentă a algoritmului de fuziune a buclelor, alături de fuziunea propriu-zisă a buclelor. Acest algoritm de decizie realizează o analiză a îndeplinirii criteriilor de fuziune și a

existenței dependențelor de date, pentru a concluziona dacă cele două bucle pot fuziona. În caz afirmativ, corpul buclei sursă se va adăuga la sfârșitul corpului buclei destinație, operație denumită fuziunea buclelor. Algoritmul de decizie asupra fuziunii este implementat prin funcția potFuzionaBuclele și fuziunea propriu-zisă a buclelor este implementată prin funcția fuzioneazaBucle.

Algoritmul de decizie asupra fuziunii testează următoarele 3 aspecte care dacă sunt îndeplinite, buclele pot fuziona: (1) buclele au același număr de iterații, (2) nu există alt cod între cele două bucle și (3) nu sunt dependențe de date între cele două bucle, ceea ce clasifică aceste două bucle ca independente, conform algortimului. Criteriul de independență nu este echivalent cu paralelizarea, ceea ce face posibil ca două bucle care sunt independente să nu fie posibil să fie executate în paralel. Dependențele de date gestionate de algoritm sunt cele de tipul celor din Tabelul 2.

Tabel 2. Cazuri tratate de algoritmul de fuziune a buclelor

Exemple de cod	**Buclele fuzionează?**

for i = 0:n
  result += a[i]
for j = 0:n
  b[j] = a[j] + 1
	Da, chiar dacă variabila de inducție diferă

for i = 0:n
  result += a[i]
for i = 0:n
  b[i] = a[i] + 1
	Nu, deoarece buclele nu au același număr de iterații

for i = 0:n
  result += a[i]
for i = 0:n
  b[i] = a[i] + result
	Nu, pentru că variabilă result, folosită în a doua buclă, este calculată doar la finalul primei bucle

for i = 0:n
  result += a[i]
sumvec(a,b,n)
for i = 0:n
  b[i] = a[i] + 1
	Nu, deoarece există cod inserat între cele două bucle

i = 0
while i < n
  result += a[i]
  i = i + 1
i = 0
while i < n
  b[i] = a[i] + 1
  i = i + 1
	Da, și buclele while sunt tratate la fel ca buclele for

Intrarea algoritmului este pe rând, fiecare buclă din programul sursă (buclă destinație) exceptând ultima buclă. Ieșirea algoritmului este bucla destinație fuzionată având adăugate intrucțiunile buclei sursă la sfârșitul copului său. Atât intrarea, ieșirea, cât și structurile de date cu care lucrează algoritmul sunt clase specifice **LLVM** [2], de exemplu clasa LoopInfo care stochează informații legate de structura buclelor. Codul 2 prezintă pseudocodul algoritmului de fuziune a buclelor.

Cod 2. Algoritmul de fuziune a buclelor

```
funcția cautaBuclaPentruFuziune(parametru L)
    daca bucla L nu conține nici un bloc de baza (corpul buclei L este vid)
        ieșire forțată din funcție
```

```
se obține un pointer către primul bloc de baza din bucla L
se obține un pointer către funcția F care conține acel bloc de bază
pentru fiecare buclă din funcția F
    dacă bucla coincide cu L
        dacă L nu este ultima buclă din funcție
            se apelează potBucleleFuziona(parametru L, parametru L_NEXT):
                dacă numărul de iterații ale lui L si L_NEXT nu este același
                    se returnează fals
                dacă numărul de blocuri de bază dintre L si L_NEXT este mai mare ca 2
                    se returnează fals
                dacă numărul de blocuri de bază dintre L si L_NEXT este 2
                    condiția 1: dacă primul bloc de bază are un număr de instrucțiuni
                  diferit de 2
                        se returnează fals
                    condiția 2: dacă cele două instrucțiuni nu sunt o instrucțiune de
                    comparare și un salt
                        se returnează fals
                    condiția 3: dacă cel de-al doilea bloc de bază nu este blocul preheader
                    al buclei L_NEXT
                        se returnează fals
                dacă numărul de blocuri de bază dintre L si L_NEXT este 1
                    dacă conditiile 1 si 2 de mai sus nu sunt îndeplinite
                        se returnează fals
                dacă există dependențe de date între cele două bucle cum ar fi: prima
                buclă scrie o dată citită de cealaltă buclă sau prima buclă are dependențe
                inter-iterații care împiedică fuziunea cu a doua buclă
                    se returnează fals
                dacă valoarea returnată de funcția potBucleleFuziona este adevărat
                    se apelează fuzioneazaBucle(parametru L, parametru L_NEXT,
                    parametru F):
                        saltul condiționat de la finalul primei bucle se înlocuiește cu un salt
                  necondiționat către cea de-a doua buclă
                        saltul necondiționat de la finalul celei de-a doua bucle către
                  începutul acesteia se înlocuiește cu un salt necondiționat către
                          începutul primei bucle în toate nodurile phi
                        se copiază toate nodurile phi din primul bloc de bază al celei de-a
                          doua bucle în primul bloc de bază al primei bucle
                        se mută blocul preheader al celei de-a doua bucle la inceputul
                          primei bucle
                        se actualizează nodurile phi în cea de-a doua buclă
                        se șterge primul bloc de bază dintre cele două bucle
                        se adaugă toate blocurile de bază a celei de-a doua bucle la finalul
                          primei
                        se șterge bucla a doua
    ieșire forțată din buclă [dacă bucla coincide cu L]
```

Fig. 2 prezintă schema bloc a ***algoritmului de fuziune a buclelor***.

Avantajul utilizării infrastructurii de compilare **LLVM** [2] pentru implementarea algortimului de fuziune a buclelor este faptul că aplicarea acestui algoritm nu depinde de aplicarea altor optimizări la compilare, ci depinde de algoritmul de decizie asupra fuziunii. Execuția prezentului algoritm de decizie necesită foarte puțin timp adăugat timpului total de compilare. Acest fapt se datorează abordării sistematice și implementării simplificate posibil datorită existenței claselor din implementarea **LLVM** [2] și a funcțiilor membre care stochează informații despre codul sursă de intrare în compilator, într-o manieră ordonată.

Deși conceput ca o optimizare poliedrală datorită analizei asupra operațiilor de citire și scriere din corpul buclei, am implementat algoritmul de fuziune a buclelor în **LLVM** [2] și nu în optimizorul poliedral Polly [19], pentru a putea utiliza structurile de date anterior menționate furnizate de infrastructură.

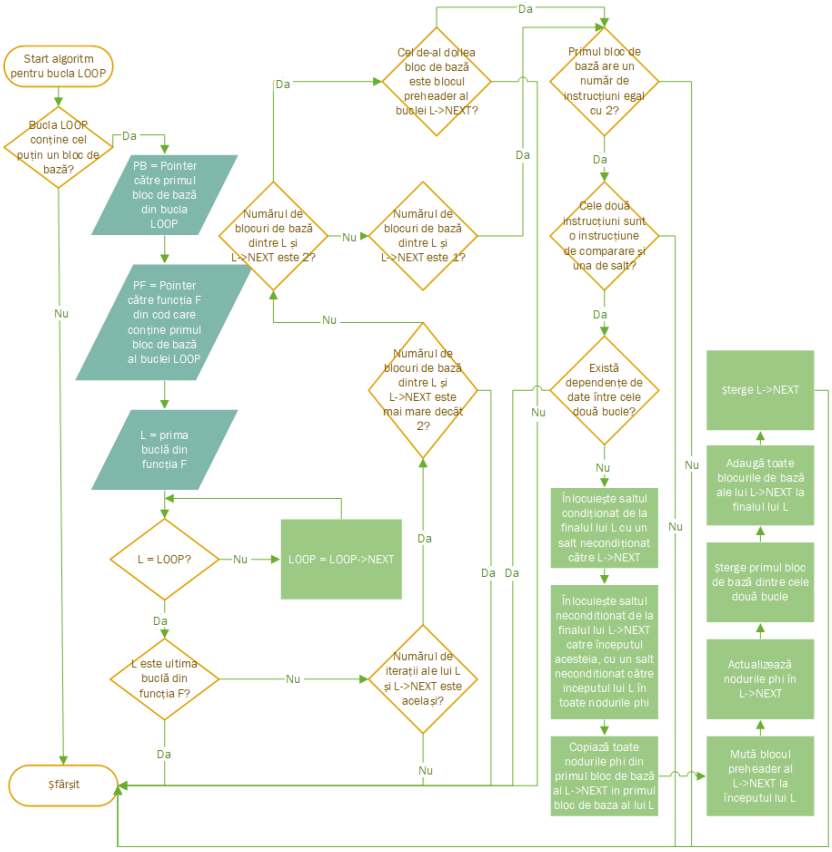

Figura 2. Schema bloc a algoritmului de fuziune a buclelor

Fuziunea buclelor este utilă la optimizarea paralelismului la nivel de buclă deoarece fuziunea buclelor înjumătățește numărul de fire de execuție care ar fi fost necesar pentru execuția fiecărei iterații (relația între iterații și firele de execuție fiind de unu-la-unu). Numărul redus de fire de execuție reduce timpul de gestiune al acestora ceea ce determină în lanț, reducerea consumului de energie pe sisteme NUMA.

Algoritmul de fuziune a buclelor îmbunătățește semnificativ performanța execuției, pe sisteme NUMA, a buclelor cu corpul de dimensiune mică care utilizează modelul de calcul paralel bazat pe memorie partajată, pentru obținerea paralelismului la nivel de buclă. În acest caz, algoritmul de fuziune a buclelor scade liniar consumul de energie odată cu scăderea timpului de execuție. Performanța

codului care folosește paralelism la nivel de buclă este îmbunătățită cu până la 56% și consumul de energie cu până la 15% [36]. Îmbunătățirile sunt datorate obținerii localizării datelor astfel: după fuziune, elementele unui tablou care până atunci erau accesate în corpul ambelor bucle, sunt după optimizare accesate din corpul unei singure bucle, fiind necesare mai puține operații de aducere a datelor din memorie. Odată cu creșterea timpului de execuție, totodată echipamentul hardware nefiind încărcat la capacitate maximă, scade și consumul de energie. Sistemele NUMA sunt cele care cu precădere evită această încărcare a hardware-ului la capacitate maximă prin intermediul politicilor de mapare a firelor de execuție care distribuie uniform (echilibrat) încărcarea pe unitățile de procesare.

Localizarea echilibrată a datelor este un concept definit în contextul calculului paralel pe sisteme NUMA (performanța acestor sisteme depinde de locația datelor în ierarhia de memorie [36]) și implică două strategii [36]: (1) firele de execuție care accesează (scriu sau citesc) aceleași date ar trebui plasate în execuție în unități de procesare astfel încât să folosească ambele același cache din primul nivel de cache în ierarhia de memorie (maparea firelor de execuție) și (2) firele de execuție trebuie executate pe nodul NUMA unde datele sunt plasate în ierarhia de memorie (maparea datelor). Maparea datelor îmbunătățește *lățimea de bandă* și *latența*.

Îmbunătățirea mapării datelor scade numărul de accese la ultimele nivele ale memoriei cache sau DRAM, dar timpul de execuție pare să fie influențat și de alți factori, devreme ce acesta nu scade liniar cu scăderea cantității de accese la memorie [36]. Schimbarea datelor de intrare implică o nouă mapare dinamică a datelor [36]. Totodată, migrarea dinamică, deși realizată cu scopul îmbunătățirii mapării firelor de execuție și a datelor, se poate dovedi nenecesară în unele
cazuri, în care nu produce decât degradarea performanței [36].

Localizarea echilibrată a datelor conduce la îmbunătățirea paralelismului.

2.5 Analiza impactului compilării asupra timpului de execuție și a consumului de energie

2.5.1 Acuratețea predicțiilor la compilare a timpului de execuție și a altor aspecte legate de execuția aplicațiilor paralele

Aplicațiile paralele pot avea atât un *comportament static* cât și un *comportament dinamic* [5]. Schimbările în comportamentul dinamic al aplicației sunt cele care pot influența în mod negativ acuratețea predicțiilor statice.

Comportamentul dinamic variază de la o execuție la alta în funcție de [5]:

1. datele de intrare și de plasamentul datelor în memorie
2. numărul firelor de execuție

Comportamentul dinamic al programelor paralele se poate schimba și în timpul execuției acestora în funcție de [5]:

1. *modelul de programare al pipeline-ului* folosit
2. *tehnicile de work-stealing* [42]

3. crearea sau distrugerea de fire de execuție
4. alocarea sau dezalocarea memoriei

Dacă nici unul din aspectele dinamice de mai sus nu intervine pentru a schimba execuția, așa cum este ea preconizată, a programului paralel, atunci comportamentul acestuia poate fi considerat in totalitate static [5].

Migrarea firelor de execuție și/sau a *paginilor de memorie* în timpul execuției programelor paralele poate fi necesară în cazul unei porțiuni de cod care are *comportament dinamic* [5]. În cazul în care porțiunea de cod are comportament static, determinarea celor două tipuri de *afinități* (*afinitatea firelor de execuție între ele* și *afinitatea unui fir de execuție*) se poate face în timpul procesului de compilare, evitându-se încărcarea execuției programului paralel cu execuția algoritmilor de mapare.

Factori care îngreunează predicția la compilare a aspectelor legate de execuția programelor paralele sunt următorii [43]:

1. nu toate caracteristicile execuției sunt cunoscute (comportament dinamic)
2. timpul de acces al datelor volatile și non-volatile este stocastic și nondeterministic
3. 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 nondeterministic al execuție programului paralel

Termenul de ***task*** este folosit în continuare cu semnificația unui set de job-uri paralele. Un ***job paralel*** [44] este o cantitate de muncă indivizibilă care poate rula pe un core care comunică cu alte core-uri. În contextul cercetării din această lucrare, termenul de task este utilizat cu o altă semnificație, și anume, în ***Pthreads*** [45], un task reprezintă un fir de execuție care execută o funcție [45].

Predicția timpului de execuție a unui task poate să difere de timpul de execuție real. Prin urmare, autorii din [43] ridică problema dacă mai este necesară sau nu aplicarea ***algoritmilor Greedy***, în astfel de cazuri, pentru a determina predicția timpului de execuție.

În cazul în care predicția timpului de execuție a unui *task* este determinată printr-o funcție cu două argumente: numărul maxim de operații în virgulă flotantă și numărul maxim de operații cu întregi care pot fi executate de *task*-ul respectiv, valoarea acestei funcții, care diferă de la o arhitectură la alta, reprezintă o estimare a timpului de execuție a unui *task*. Această estimare are avantajul că nu depinde de schimbările în comportamentul dinamic al programului (atunci când Hyper-Threading nu este activat), deoarece sunt luate în calcul doar numărul maxim de operații. Această estimare care poate fi utilizată în a compara timpul de execuție a două *task*-uri și în a prezice care dintre ele se execută mai repede, furnizând apoi această informație algoritmului de mapare. O variantă simplificată a acestei funcții este definită prin suma celor două argument. Algoritmii propuși în această lucrare utilizează la determinarea celui mai puțin încărcat core, o funcție definită de numărul de fire de execuție mapate pe core-ul respectiv.

2.5.2 Modul în care compilarea influențează consumul de energie la rulare

Codul are un impact mai mare asupra performanței și a consumului de energie decât arhitectura handware pe care rulează și nu există o optimizare care să îmbunătățească atât performanța cât și consumul de energie, pentru orice tip de cod [40]. Această lucrare susține afirmația anterioară prin faptul că există optimizări la compilare care pot îmbunătății atât performanța cât și consumul de energie al codului atunci când acestea nu degradează localizarea datelor. Un exemplu de astfel de optimizare este ***fuziunea buclelor***.

Performanța codului (în acest caz performanța codului implică toate aspectele care țin de performanța un sistem pe care acesta se execută, nu doar timpul de execuție) este îmbunătățită prin optimizarea fiecărei funcții din cod, scăzând astfel consumul de energie al fiecărui core [46].

Temperatura ridicată a hardware-ului cauzată de utilizarea sistemului la capacitate maximă afectează fiabilitatea și disponibilitatea sistemului, crescând consumul de energie [47]. Paralelizarea pare a fi cea mai promițătoare soluție la această problemă, în condițiile în care aceasta este optimă. Compilarea are și ea rolul de a îmbunătăți execuția paralelă prin faza de optimizare.

Un pas important în optimizarea consumului de energie în calculul paralel este stabilirea unei relații între paralelizare și puterea dinamică consumată, mai exact în ce fel modifică paralelizarea puterea dinamică. La rândul ei, puterea dinamică influențează puterea statică, în sensul că o rată scăzută de obținere a datelor din memoria cache și traficul pentru asigurarea coerenței cache-ului extind timpul de execuție.

Consumul de *putere statică* al unei aplicații reprezintă o fracțiune mare din puterea totală și scade odată cu timpul de execuție [12]. Această afirmație se explică prin faptul că, în contextul unei execuții paralele optime, timpul în care resurse hardware sunt în stare "în așteptare" (sleep mode) este mai mic decât în alte cazuri, deoarece execuția optimă se încheie mai rapid. Cu alte cuvinte, în același timp în care resursele hardware sunt alimentate cu putere statică se pot realiza mai multe sau mai puține procesări, în funcție de gradul de optimizare al execuției paralele.

Puterea dinamică, pe de altă parte, este puterea consumată adițional puterii statice cu acțiunile care implică executarea instrucțiunilor programului [12]. Puterea dinamică consumată de un core la un moment dat depinde de viteza de procesare a acestuia, de tensiunea de alimentare și de task-urile care rulează pe acesta [30]. Se consumă putere dinamică la alimentarea unităților aritmetice și logice, la alimentarea structurilor superscalare utilizate în reordonarea instrucțiunilor (reordonarea unei cantități mai mici de instrucțiuni se traduce în mai puțină putere dinamică consumată) și la alimentarea cache-ului (cu cât mai puține nivele de cache traversate per operație de memorie, cu atât puterea dinamică consumată este mai mică) [12].

În afara clasificării puterii în *statică* și *dinamică* mai există și alte clasificări, ca de exemplu cea după criteriul componentă care consumă. PowerPack [39] este un utilitar de măsurare a consumului de energie care grupează consumul de putere în următoarele trei categorii: energie pentru CPU (de obicei, CPU consum

aproximativ 50%-60% din energia totală consumată [48]), energie pentru memorie, energia pentru unitatea de hard-disk și energia pentru placa de bază.

Calculul paralel este împărțit, din punct de vedere al utilizării memoriei, în două categorii: *calcul paralel bazat pe schimb de mesaje între procese* și *cel bazat pe memorie partajată*.

În contextul execuției paralele bazate pe schimb de mesaje între procese, metoda de ajustare dinamică a tensiunii de alimentare și a frecvenței procesorului (DVFS: *Dynamic Voltage and Frequency Scaling*) poate fi aplicată în timpul fazei de comunicare prin mesaje (prin care se transmit și se primesc date între procese) folosind modelul de calcul paralel MPI [38, 50].

DVFS reduce consumul de energie al nodurilor NUMA care au un volum mic de procesări de efectuat și implicit, al aplicațiilor cu o repartizare neuniformă a calculelor de procesat [50]. Repartizarea calculelor de procesat se poate modifica la compilare prin câteva dintre optimizările detaliate anterior în raport, prin care se separă calculele din fluxul critic de execuție de cele necritice și astfel se poate aplica DVFS pentru procesoarele care execută calcule necritice, reducându-se consumul de energie al acestora. Calculele necritice sunt procesările realizate de firele care nu necesită execuție imediată.

Bibliotecile care implementează modelul de calcul paralel bazat pe memorie partajată precum ***OpenMP*** [51] sau ***Pthreads*** [45] folosesc accesele la memorie ca modalitate de comunicare [34]. ***Biblioteca OpenMP*** este utilizată în special pentru obținerea paralelismului la nivel de buclă și este suportată de unele compilatoare moderne (**LLVM** [2] și **GCC** [20]), care au implementate optimizări ce îmbunătățesc performanța execuției aplicațiilor paralele care utilizează biblioteca. Deși sunt bazate pe schimb de mesaje, implementările ***MPI*** cum ar fi ***MPICH2*** [53] sau ***Open MPI*** [54], conțin optimizări în scopul comunicării bazate pe memorie partajată [41].

Maparea firelor de execuție și a datelor pe sisteme NUMA este un aspect al calculului paralel. Îmbunătățirea mapării reduce timpul de execuție al aplicațiilor paralele, reducând consumul de putere statică [41]. În plus, o mapare îmbunătățită reduce traficul de date, ceea ce conduce la reducerea *puterii dinamice* consumate și implicit la reducerea consumului total de putere [41]. Fiind consumatoare de timp, este indicat ca *reasignarea dinamică a firelor de execuție* să fie realizată la compilare. Apare necesitatea deciziei de mapare a firelor de execuție la compilare (obiectul cercecetării din această lucrare).

Firele de execuție pot fi mapate manual, de către programator, care inserează în cod apeluri specifice din bibliotecile de mapare. În acest caz, acesta este indicat să studieze înainte, tiparul de acces la memorie al firelor de execuție. ***TABARNAC*** [49] este o aplicație care furnizează utilizatorului modul în care memoria este accesată, grupând accesele pe fire de execuție sau pe structuri de date și scoțând în evidență problemele legate de performanța codului (cu precădere cele legate de accesele la memorie). Această aplicație este o altă soluție practică, neautomată, care ar putea fi folosită cu scopul îmbunătățirii mapării firelor de execuție. Această abordare însă, necesită timp și cunoștiințe în ceea ce privește optimizarea acceselor la memorie. Prin urmare, este indicată, și din acest motiv, maparea automată a firelor de execuție la compilare.

2.6 Maparea firelor de execuție

Capitolul este menit să argumenteze importanța algoritmilor de mapare a firelor de execuție pe core-uri în îmbunătățirea aspectelor ce țin de execuția aplicațiilor paralele cum ar fi performanța (timpul de execuție) și consumul de energie.

La execuția aplicațiilor paralele pe sisteme NUMA, *localizarea echilibrată a datelor* este obținută prin *maparea firelor de execuție* așa încât firele care comunică cel mai des să fie plasate în execuție cât mai aproape în termeni de timp și distanță NUMA și să împartă același prim nivel de cache [55]. Maparea firelor de execuție se realizează după următoarele considerații [56]: (1) firele sunt plasate în execuție pe unitățile de procesare cele mai apropiate în termeni de distanță NUMA de locul de execuție al celorlalte fire cu care acestea comunică și (2) cantitatea de comunicare a firelor de execuție deja asignate pe fiecare unitate de procesare trebuie să fie mai mică sau egală cu media de comunicare per unitate de procesare (cantitatea de comunicare este dată de numărul de accese la memoria partajată).

Maparea firelor de execuție bazată pe conceptul de afinitate îmbunătățește *localizarea* și *echilibrul comunicării dintre firele de execuție* [5]. În contextul studiului din această lucrare, comunicarea dintre două fire de execuție se referă la datele utilizate în comun de acele fire de execuție. *Localizarea comunicării* dintre firele de execuție se poate îmbunătăți prin plasarea firelor de execuție care comunică între ele, cât mai aproape în ierarhia hardware pentru a putea utiliza cache-urile comune și interconexiunile rapide [5]. *Echilibrul comunicării* dintre firele de execuție presupune: distribuirea în mod echilibrat a comunicării gestionate de cache-uri și de interconexiuni și utilizarea cu precădere a primelor nivele de cache pentru a deservi accesele la memorie [5]. Dacă aplicația nu este echilibrată în ceea ce privește comunicarea dintre firele de execuție (de exemplu, unele fire comunică mai mult decât celelalte [5] sau anumite core-uri sunt indisponibile [56]) atunci balansarea comunicării poate îmbunătății performanța totală [5].

În contextul sistemelor cu acces la memorie comună, maparea firelor de execuție mai este numită și maparea task-urilor când se folosesc task-uri pentru obținerea paralelismului, cum este cazul ***MPI*** [5].

2.6.1 Tipare de mapare și reacția fiecăruia la schimbări în comportamentul dinamic al aplicațiilor paralele

Maparea firelor de execuție bazată pe *conceptul de afinitate* este realizată prin politicile următoare:

1. *politica de alocare*: fiecare fir de execuție este asignat unui anumit core utilizând funcțiile sistemului de operare sau opțiunile mediului de rulare și rămâne asignat pe acel core până când își încheie execuția. Această politică nu impune în plus un timp de gestiune a firelor de execuție, dar prezintă dezavantajul că nu poate reacționa la schimbările fluxului de execuție al codului survenite în timpul execuției programului. De exemplu, atunci când fluxul de execuție se schimbă, se vor executa instrucțiuni care pot avea nevoie de alte date decât cele deja prezente în cache-ul core-ului pe care se execută firul, iar firul nu poate fi mutat pe un alt core unde aceste date

există deja în cache, deoarece el rămâne asignat pe același core, conform acestei politici. Prin urmare, dezavantajul este ca noile date trebuie aduse (și) în memoria cache a core-ului pe care se execută firul.

2. *politica de migrare*: firele de execuție sunt migrate de pe un core pe altul în timpul execuției în funcție de comunicarea dintre acestea la momentul anterior migrării. Acest tip de politică de migrare impune extinderea timpului de execuție cu un timp adițional necesar execuției funcțiilor de migrare; de cele mai multe ori însă, acestă politică determină prelungirea și mai mult a timpului de execuție datorită numărului ridicat de date care nu pot fi obținute din cache (cache misses) [57]. Cu toate acestea, poate gestiona schimbări în comportamentul dinamic al programului paralel.

În contextul tiparelor de mapare, *maparea* este denumită *statică* dacă este decisă înainte de execuția programului și nu este modificată pe parcursul execuției acestuia [58], în caz contrar, poartă numele de *mapare dinamică*.

2.6.2 Informații necesare în realizarea unei mapări eficiente

În cazul în care maparea firelor de execuție este realizată la rulare, un consum de energie considerabil este indus datorită *reasignării firelor de execuție* [59].

În determinarea *mapării/migrării* optime a firelor de execuție, sunt necesare două informații:

1. 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
2. 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*** [60])

O mapare eficientă este, în general, o operație globală, ceea ce presupune ca afinitatea CPU a tuturor firelor de execuție să fie determinată în cadrul unei singure operații. Pentru un număr mic de fire de execuție, această operație nu este una costisitoare în cazul majorității algoritmilor de mapare [61].

2.6.3 Clasificarea algoritmilor de mapare

Clasificarea algoritmii de mapare descriși în următorul subcapitol, se face în funcție de următoarele criterii:

3. Momentul mapării:
   - algoritmi de mapare statici: *Limited Best Assingment* (LBA) [43] și alți ***algoritmi Greedy*** descriși în [43], SCOTCH [58]
   - algoritmi de mapare dinamici: *Opportunistic Load Balancing* (OLB) [43], METIS [63,64] și algoritmul de mapare parte a setului de utilitare Zoltan [64]
4. Metoda de mapare utilizată:
   - algoritmi care se bazează pe partiționarea grafului de comunicare între task-uri: SCOTCH [58], METIS [63,64], algoritmul de mapare parte a setului de utilitare Zoltan [64]

- algoritmi care identifică tipare în graful de comunicarea între task-uri, metodă descrisă în [65]
- algoritmi care determină afinitatea CPU a firelor de execuție, utilizată în maparea sau migrarea firele de execuție pe core-uri, prin reprezentarea sub formă de arbore a ierarhiei de fire de execuție: ***NUMA-BTDM*** [55]

5. Considerarea arhitecturii hardware pe care rulează aplicația, în realizarea mapării:
   - algoritmi care nu țin cont de arhitectura hardware pe care rulează aplicația: OLB [43], SCOTCH [58], METIS [63,64], algoritmul de mapare parte a setului de utilitare Zoltan [64]
   - algoritmi care realizează maparea în funcție de arhitectura hardware pe care rulează aplicația: Treematch [18,66], EagerMap [67]

2.6.4 Algoritmi de mapare

Maparea firelor de execuție este o *problemă polinomială non-deterministă* care necesită un algoritm eficient de mapare [36].

2.6.4.1 Algoritmi de mapare naivi

Algoritmi naivi de mapare sunt considerați OLB [43] și LBA [43]. OLB [43] plasează fiecare job în ordinea apariției acestora pe următoarea unitate de procesare disponibilă iar algoritmul LBA [43] asignează job-ul pe mașina pe care este așteptat să se execute cel mai rapid, făcând abstracție de încărcarea core-ului respectiv, inclusiv de încărcarea produsă de job-ul care se asignează.

2.6.4.2 Algoritmi de mapare bazați pe metoda partiționării grafurilor

Determinarea unei mapări optime a firelor de execuție este o *problemă de tip NP-Hard* [68], de aceea mulți algoritmi folosesc aproximări pentru a reduce complexitatea la o complexitate polinomială [61]. Majoritatea algoritmilor utilizează grafuri care descriu comunicarea între *task*-urile aplicației sau care descriu ierarhia hardware și folosesc metoda partiționării grafurilor [63] sau tehnici de identificarea de tipare în grafuri [65] pentru a îmbunătăți maparea firelor de execuție. Câteva exemple de astfel de algoritmi sunt: SCOTCH [58], METIS [63,64] și algoritmul de mapare parte a setului de utilitare Zoltan [64] discutați în continuare.

Algoritmul static de mapare din Pachetul software SCOTCH

Pachetul software SCOTCH [58] conține un algoritm static de mapare a firelor de execuție bazat pe bipartiționarea recursivă a grafului de fire de execuție în care ponderea atașată muchiilor descrie comunicarea dintre acestea, cât și bipartiționarea grafului de unități de procesare ale arhitecturii țintă. În determinarea mapării se ține cont de programele care deja rulează pe arhitectura țintă, astfel încât maparea să minimizeze costul comunicării dintre firele de execuție. Acest lucru este realizat prin mapare a firelor de execuție care comunică intens pe unități de procesare alăturate [58].

Algoritmul METIS de partiționare a grafului de comunicare între task-uri

Algoritmul METIS de partiționare nestructurată a grafului de comunicare între task-uri este structurat în 3 etape [62].

Prima etapă este bazată pe partiționarea grafului în *p* partiții, nu neapărat cu același număr de noduri, astfel încât numărul de muchii care conectează noduri din partiții diferite să fie minim [62]. Algoritmul pornește de la o partiție în care se află la început toate nodurile grafului și extrage la fiecare pas câte un nod din mulțimea nodurilor neextrase. Nodul extras este ales astfel încât muchia care leagă nodul de unul din nodurile din partiția formată anterior să fie cu valoare maximă, formând ***arborele parțial de acoperire maximă*** (acest principiu stă la baza algoritmului lui Prim, prin care se obține ***arborele minim de acoperire***) [62]. Apoi, nodul extras și nodul adiacent cu el dintre nodurile extrase anterior sunt grupate împreună formând o nouă partiție alcătuită dintr-un singur nod. Nodul rezultat are ponderea egală cu suma ponderilor celor două noduri din care a fost compus și muchia prin care nodul rezultat se leagă de arborele maxim de acoperire este egală cu suma ponderilor muchiilor prin care se leagă nodurile din care a fost compus de arbore maxim de acoperire, sumă din care se scade costul conectării celor două noduri [62]. Rezultatul acestei etape este diminuarea numărului de noduri și muchii din graf [62].

A doua etapă a algoritmului METIS [62] este ***bipartiționarea grafului*** astfel încât suma ponderilor nodurilor din fiecare partiție să fie apropiată de semisuma totală a ponderilor grafului și costul total al tăierii muchiilor (în urma căreia se obțin cele două partiții) să fie minim (se încearcă mai multe astfel de partiționări până când se obține diferența cea mai mare de densitate dintre cele două partiții).

A treia etapă a algoritmului METIS [62] este constituită din operațiile inverse primei etape. În această etapă, graful bipartit obținut în etapa anterioară este mărit succesiv ca număr de noduri și muchii și în final, transformat înapoi în graful inițial [62].

Algoritmul de partiționare a hipergrafului de task-uri din utilitarul Zoltan

Algoritmul de mapare prin ***partiționarea hipergrafului***, parte a utilitarului Zoltan, se bazează pe partiționarea multinivel a nodurilor grafului în *k* mulțimi disjuncte și tăierea hipermuchiilor (noțiunea de hipermuchie este definită ca un subset de noduri) care au cel puțin două noduri aparținând de partiții diferite [64]. Se taie cu prioritate hipermuchiile care au noduri în cât mai multe partiții. Ideea acestui algoritm este de a aproxima hipergraful într-o secvență de mai multe hipergrafuri mai reduse care reflectă hipergraful original [64].

Prima etapă a algoritmului constă în obținerea unui hipergraf mai redus [64].

În cea de-a doua etapă se partiționează succesiv graful obținut anterior până când se ajunge la un număr de *k* partiții (o alternativă este partiționarea direct, printr-o singură operație, în *k* partiții) [64].

Următoarea etapă constă în alegerea și expandarea celui mai dens hipergraf. Pentru obținerea unui încărcări optime la rulare, se consideră *k* egal cu *p*, numărul de procesoare. Algoritmul permite ***bipartiționarea hipergrafului*** chiar

dacă *k* nu este o putere a lui 2, partițiile rezultând, în acest caz, neidentice în ceea ce privește numărul de noduri [64].

2.6.4.3 Algoritmi de mapare care țin cont de arhitectura hardware

Arhitecturile hardware bazate pe memorie partajată pot fi reprezentate sub forma unui arbore, ceea ce a condus la dezvoltarea unor algoritmi și mai eficienți decât cei care folosesc grafuri [63,70]. Două exemple de astfel de algoritmi sunt Treematch [63,64], EagerMap [67].

Algoritmul de mapare Treematch

Algoritmul Treematch [66] realizează o mapare a tiparului de comunicare al aplicației pe o arhitectură NUMA. Algoritmul utilizează o matrice de comunicare între procese, în care liniile și coloanele reprezintă procesele iar elementele matricii definesc cantitatea de comunicare între procese. Se urmărește formarea de grupuri de câte două procese, cu proprietatea că un proces este conținut într-un singur grup și formarea unui graf de incompatiblități în care nodurile sunt toate grupurile posibile de câte două procese și muchiile sunt prezente între nodurile care sunt incompatibile (două noduri sunt incompatibile dacă au un proces comun). Setul de grupuri se identifică utilizând un ***algoritm Greedy***, considerând că nodurile din interiorul fiecărui grup comunică cel mai mult între ele și mai puțin cu nodurile din celelalte grupuri. Algoritmul este o problemă de tip NP-Hard, necesitând aplicarea uneia dintre următoarelor euristici pentru a i se reduce complexitatea:

1. utilizând un ***algoritm Greedy***, se ordonează crescător nodurile după valoarea acestora și se atribuie pe rând nodurilor indecși începând cu indexul 0, ținând cont că valoarea unui proces este dată de însumarea cantităților de comunicare cu celelalte procese, iar valoarea unui nod se stabilește ca fiind diferența dintre suma cantităților totală de comunicare cu toate celelalte procese ale celor două procese din grup și cantitatea de comunicare ale celor două procese între ele
2. se ordonează descrescător nodurile după valoarea acestora și se atribuie nodurilor, parcurse pe rând în ordine inversă, indecși începând cu indexul 0
3. utilizând un ***algoritm Greedy***, se ordonează descrescător nodurile după media valorilor nodului(nodurilor) vecin(e) din graf și se atribuie acestora pe rând indecși începând cu 0

În urma identificării setului de grupuri a câte două procese, se formează o nouă matrice de dimensiune *n* x *n*, unde *n* este cardinalul setului obținut. Elementul de poziția (*i*,*j*) a matricii reprezintă suma cantităților de comunicare dintre fiecare proces din grupul cu indexul *i* și fiecare proces din grupul cu indexul *j*.

Se construiește, în continuare, un arbore, care corespunde arhitecturii hardware, astfel: dacă arborele are 3 nivele atunci, pe nivelul 3 sunt plasate frunzele din arbore care corespund core-urilor, pe nivelul 2, cache-urile comune core-urilor, pe nivelul 1, cache-urile necomune core-urilor iar pe nivelul 0, memoria principală. Fiecărui core *i* se atribuie un proces astfel: primele două core-uri care împart același cache comun execută fiecare câte un proces din primul grup de două procese, următoarele două core-uri care deservesc același cache execută fiecare câte un proces din al doilea grup de două procese, ș.a.m.d.

Se urmărește reducerea succesivă a înălțimii arborelui cu câte o unitate. Dacă înalțimea este n, se calculează numărul de noduri fiu ale unui nod de pe nivelul n-1 din arbore. Arborele binar plin utilizat are proprietatea că numărul nodurilor fiu este același pentru toate nodurile de pe un anumit nivel. Numărul de noduri fiu ale unui nod de pe nivelul n-1 trebuie să dividă exact numărul total de grupuri de procese. Dacă nu se întâmplă acest lucru, se adaugă atâtea grupuri cât e nevoie ca numărul de noduri fiu să dividă numărul de grupuri, cu proprietatea că aceste grupuri nu comunică cu nici un alt grup, după care se aplică euristicile de mai sus pentru a forma grupuri care conțin fiecare un număr de procese egal cu numărul de noduri fiu ale nodurilor de pe nivelul n-1. În continuare, se notează grupurile nou formate cu câte un index începând cu indexul 0, fiecare grup formând câte un proces virtual, și se formează o nouă matrice de dimensiunea m x m, unde m este câtul împărțirii numărului de grupuri la numărul de noduri fiu ale nodurilor de pe nivelul n-1. Se procedează la fel până când se ajunge la un singur grup, adică un singur proces virtual.

Maparea proceselor pe unitățile de procesare se realizează pornind de la grupul rezultat anterior (procesul virtual obținut), format din două subprocese virtuale, astfel: primul subproces virtual se mapează în subarborele stâng iar cel de-al doilea subproces virtual se mapează în subarborele drept. Cele două subprocese virtuale se expandează fiecare la două sau mai multe subsubprocese virtuale care se mapează începând cu fiul stâng al subprocesului virtual de care aparțin. Se continuă procesul până când se ajunge la procese fizice care vor fi mapate fiecare pe frunzele arborelui (pe core-uri).

Algoritmul Greedy de mapare a task-urilor EagerMap

Algoritmul de mapare EagerMap [67] mapează bazat pe comunicarea între task-uri (algoritmul are efect în special atunci când comunicarea este structurată), funcționând doar în cazul arhitecturilor simetrice reprezentate sub formă de arbore.

Algoritmul primește două intrări: matricea de comunicare între task-uri și arborele care descrie arhitectura hardware și în care nodurile reprezintă unitățile de procesare, memoriile cache, procesoarele și nodurile NUMA, iar muchiile, legăturile dintre acestea. În implementarea algoritmului se folosește un tablou în care fiecare element reprezintă numărul de unități de procesare, numărul de memorii cache de pe ultimul nivel, numărul de noduri NUMA etc.

Primul pas realizat de algoritm reprezintă constituirea de grupuri, primul grup fiind cel al task-urilor aplicației, cel de-al doilea grup fiind un grup format din grupuri de task-uri în care numărul de grupuri de task-uri este egal cu numărul de core-uri ș.a.m.d. Pe măsură ce se adaugă câte un nivel în ierarhia arborescentă de grupuri, se urcă câte un nivel în arborele care descrie arhitectura hardware, numărul de grupuri din grupul înfășurător fiind egal cu numărul de unități hardware de pe nivelul curent. Procesul continuă până când se ajunge la rădăcina arborelui. Grupurile se formează în așa fel încât task-urile care comunică cel mai mult între ele sunt plasate în același grup. Pentru formarea grupurilor se folosește un algoritm Greedy care selectează la fiecare pas task-ul care prezintă cea mai mare cantitate de comunicare relativ la task-urile deja adăugate în grup, reluându-se această iterație până când se ajunge la numărul maxim de noduri admis într-un grup.

La fiecare adăugare a câte unui nivel de grupuri se redimensionează matricea comunicării între grupuri, noua dimensiune fiind de n x n, unde n este

numărul curent de grupuri și se initializează fiecare element (i,j) al matricii cu suma comunicării fiecărui task din grupul i cu fiecare task din grupul j.

Cel de-al doilea pas îl constituie maparea task-urilor pe core-uri printr-un algoritm recursiv astfel: fiecărui grup din grupul curent i se aignează o unitate hardware de pe nivelul respectiv și în cazul fiecărui grup se apelează algoritmul recursiv pentru grupurile sale componente asignate următorului nivel din ierarhia hardware.

2.6.4.4 Algoritmi de mapare bazați pe istoricul rulărilor precedente

Sunt descriși, în continuare, câțiva algoritmi existenți de mapare a firelor de execuție care folosesc istoricul rulărilor precedente pentru a determina o mapare eficientă a firelor de execuție:

4. Algoritmul naiv LBA [43] descris mai sus
5. ***Algoritmii Greedy*** care țin cont de timpul estimat de execuție a firelor pe fiecare core (dacă firul de execuție nu poate rula pe un anume core timpul de execuție se setează la o valoare foarte mare) și de încărcarea fiecărui core [43]. Acești algoritmi folosesc un ***tablou bidimensional*** *A* de dimensiune *n x m*, unde n este numărul de job-uri, m este numărul de core-uri și A_{ij} reprezintă timpul de execuție estimat al job-ului *i* pe core-ul *j*. Doi astfel de algoritmi sunt descriși în cele cu urmează:
   - Primul ***algoritm Greedy*** are complexitatea algoritmică O(nm) [43]. Pentru a seta elementele ***tabloului bidimensional*** *A* se procedează astfel: se determină valoarea A_{1j}, astfel încât $A_{1j} \leq A_{1k} \; \forall \; k = 1,m$. Se asigneaza apoi primul job pe core-ul *j* și se adaugă valoarea A_{1j} la toate $A_{ij} \; \forall \; i = 2,n$. Se determină pentru fiecare $p = 2,n$, valoarea A_{pj} astfel încât $A_{pj} \leq A_{pk} \; \forall \; k = 1,m$. Se asignează apoi job-ul *p* core-ului *j* și se adaugă valoarea A_{pj} la toate $A_{ij} \; \forall \; i = p+1,n$. Se continuă să se asigneze fiecare job pe core-ul pe care rulează în cel mai scurt timp, ținând cont de asignările job-urilor anterioare pe acel core. Job-urile sunt asignate în ordinea de procesare a acestora.
   - Cel de-al doilea ***algoritm Greedy*** are complexitate $O(n^2m)$ [43] și este compus din doi sub-algoritmi de mapare diferiți. Algoritmul alege maparea care prezintă timpul estimat de rulare cel mai mic, însumând timpii de pe toate core-urile. Cei doi sub-algoritmi sunt similari primului ***algoritm Greedy***, cu mențiunea că, în cazul acestora, job-urile nu mai sunt asignate în ordinea în care sunt procesate.
     - Primul sub-algoritm se bazează pe ideea finalizării cât mai repede a execuției job-urilor planificate ultimele și este alcătuit din următorii pași:
       - Se intializează mulțimea job-urilor rămase neasignate cu toate job-urile
       - $\forall \; i \in$ mulțimii job-urilor rămase neasignate, se identifică $A_{ij} \leq A_{ik} \; \forall \; k \in$ mulțimii core-urilor și se denumesc aceste elemente A_{ij} din matrice ca fiind A_{imin_i}
       - Se determina *p* astfel încât $A_{pmin\_p} \leq A_{imin\_i} \; \forall \; i \in$ mulțimii job-urilor rămase neasignate

- Se extrage job-ul *p* din mulțimea job-urilor rămase și se asignează mașinii min_p
- Se adaugă A_{pmin_p} la valorile A_{imin_p}, $\forall\, i \in$ mulțimii job-urilor rămase neasignate
- Dacă mulțimea job-urilor rămase neasignate nu este vidă, se trece la Pasul ii.

▪ Cel de-al doilea sub-algoritm diferă de primul în sensul că în Pasul iii, se identifică *p* așa încât $A_{pmin\_p} \geq A_{imin\_i}$, $\forall\, i \in$ mulțimii job-urilor neasignate (sub-algoritmul minimizează în fiecare pas cel mai defavorabil caz de execuție).

2.6.5 Comparație între algoritmii de mapare

Gradul de adaptare al execuției aplicațiilor software la arhitectura hardware este redus pentru majoritatea aplicațiilor. Sunt însă aplicații care pot să beneficieze de avantajele arhitecturilor moderne complexe. Un exemplu de astfel de aplicații sunt cele care utilizează ***MPI***, care permit corelarea tiparului de comunicare al aplicației cu arhitectura hardware pe care rulează aceasta, corelare în urma căreia se obține o corespondență între procesul ***MPI*** și core-urile mașinii pe care rulează aplicația.

Printre primele direcții în maparea proceselor a fost cea a lui ***Kruskal*** și ***Snir***, care modelează problema printr-un flux de informații [9]. Framework-ul ***MPIPP*** [3] ia în considerare un context multi cluster și împrăștie procesul ***MPI*** pe diverse clustere utilizate de aplicație. Mai târziu, algoritmii de mapare care folosesc grafuri pentru a determina maparea proces ***MPI*** - core au devenit larg răspândiți, integrându-se chiar în diverse implementări ale modelului de calcul paralel ***MPI***, cum ar fi implementările furnizate de Hewlett-Packard [70], IBM [71], sau cea din [69]. În toate aceste cazuri, algoritmi de mapare sunt specifici ***MPI*** și nu iau în considerare complexitatea arhitecturilor moderne NUMA și nici topologia acestora. Algoritmul de mapare a task-urilor Treematch [62] nu poate funcționa pe orice tip de graf, ci doar pe un arbore, spre deosebire de algoritmul SCOTCH [72]. Dezvoltarea algoritmilor precum Treematch a fost necesară deoarece particularizarea grafului printr-o structură de arbore determină apariția unor specificități privind maparea. Comparând ***algoritmii Greedy*** din [43] între ei, s-a ajuns la concluzia că performanța primului ***algoritm Greedy*** este cu până la 15% mai bună decât performanța celui de-al doilea algorithm [43].

Și algoritmul ***NUMA-BTDM*** [55] grupează firelor de execuție care comunică intens în scopul executării lor cât mai aproape unul față de celălalt – în plus, ține cont de predecesorii firelor de execuție în arborele de generare a firelor de execuție, unde nodurile fiu ale unui nod reprezintă firele de execuție generate de către firul aferent nodului.

2.6.6 Aspecte care influențează direct eficiența algoritmilor de mapare pe sisteme NUMA

Aspecte privitoare la arhitectura hardware. Factorul care influențează maparea firelor de execuție și a datelor pe sistemele NUMA este asimetria

inter-conectivității între nodurile unui astfel de sistem: conexiunile pot avea lățime de bandă diferită sau viteză mai mare într-o direcție de transmitere a datelor decât în cealaltă, pot fi împărțite de un număr diferit de noduri sau pot fi chiar unidirecționale [73]. Conform [73], maparea firelor de execuție trebuie realizată astfel încât lățimea de bandă totală pe care sunt transmise datele în urma respectivei mapării, să fie cât mai mare. Lățimea totală de bandă este, conform [73], un factor mai determinant decât numărul de hopuri traversate în cazul acceselor la distanță la alte noduri NUMA.

Cu cât numărul procesoarelor din sistemele multi-procesor este mai mare, cu atât devine mai importantă problema *mapării și a localizării datelor* în reducerea timpului de execuție al unei aplicații paralele [74].

Aspecte privitoare la execuția programelor paralele. O mapare eficientă a firelor de execuție/proceselor pe sisteme NUMA poate fi realizată atunci când această mapare este efectuată cu scopul obținerii localizării datelor la execuția programelor paralele [74]. Pentru a realiza o astfel de mapare algoritmul este următorul: se identifică un procesor pe care se va executa primul proces, iar următoarele procese planificate în execuție sunt mapate pe procesoarele aflate la un hop distanță față de procesorul ales inițial până la epuizarea acestora, apoi pe procesoarele aflate la două hopuri distanță, ș.a.m.d [74]. Acest algoritm se poate aplica și pentru core-uri în loc de procesoare, respectiv fire de execuție în loc de procese, astfel: se identifică core-ul pe care se execută primul fir, iar următoarele fire care sunt planificate în execuție sunt mapate pe același core pe care se execută primul fir până la epuizarea resurselor acelui core (a firelor logice disponibile per core), următoarele fire planificate în execuție fiind mapate pe core-urile aflate la un hop distanță față de primul core ales, până la epuizarea tuturor resurselor fiecărui core și până la epuizarea core-urilor, ca mai apoi maparea să fie realizată pe core-urile aflate la două hopuri distanță ș.a.m.d.

2.7 Biblioteci utilizate pentru mapare

2.7.1 Biblioteci actuale pentru maparea firelor de execuție

TBBNUMA [76] este o bibliotecă de calcul paralel portabilă și compozabilă care împiedică firele de execuție să fie asignate aleator pe core-uri. Biblioteca este derivată din Intel Threading Blocks (TBB) și destinată dezvoltării aplicațiilor paralele pentru rularea pe sisteme NUMA [76].

QThreads [77] este o bibliotecă pentru calcul paralel disponibilă în sistemul de operare Linux (cu o alternativă și pentru sistemul de operare Windows) care furnizează o modalitate de a controla maparea firelor de execuție și poate fi extinsă cu optimizări care au în vedere execuția pe sisteme NUMA.

Biblioteca actuală și reprezentativă pentru maparea firelor de execuție este biblioteca ***Pthreads*** [45], care permite maparea explicită de către utilizator a firelor de execuție pe core-uri prin funcții specifice cum ar fi:

1. ***int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset)*** - setează afinitatea CPU a unui fir de execuție pentru core-urile pe care nu este deja mapat

2. ***int pthread_getaffinity_np(pthread_t thread, size_t cpusetsize, const cpu_set_t *cpuset)*** - returnează afinitatea CPU a unui fir de execuție în parametrul de ieșire CPUset

2.7.2 Dezavantajele bibliotecilor actuale privind maparea

Planificarea task-urilor care ține cont de consumul de energie trebuie să fie transparentă programatorului [75], deci și maparea firelor de execuție pe unitățile de procesare, pentru a eficientiza procesul de mapare încă din faza de dezvoltare a codului.

Marea majoritate a limbajelor de programare și a bibliotecilor actuale nu numai ca nu au suport explicit pentru maparea firelor de execuție pe sistemele NUMA, ci nu au nici un fel de suport explicit pentru sistemele NUMA [76].

Optimizarea acceselor la memoria din sistemele NUMA necesită extinderea limbajele de programare în scopul furnizării la execuție, a detaliilor despre arhitectura pe care rulează în acel moment codul. Acest fapt determină ca optimizările pentru un anumit sistem NUMA să fie neportabile pe un sistem NUMA diferit [76].

Optimizările sistemelor NUMA nu sunt compozabile, neadaptându-se dinamic la schimbarea numărului de core-uri, ci sunt mai degrabă orientate pe un anumit sistem NUMA [76].

Dezavantajul limbajelor de programare extensibile este că ele oferă foarte puține detalii despre disponibilitatea resurselor hardware, considerându-le întotdeauna în proporție de 100% disponibile, ceea ce reduce efectul optimizărilor asupra acceselor la memorie la rularea aplicației optimizate în paralel cu altele pe sisteme NUMA [76].

Limbajele de programare extensibile nu oferă suport explicit pentru controlarea mapării firelor de execuție și a datelor, ci este mai degrabă necesară cunoașterea implementării limbajului de programare sau a compilatorului pentru a mapa eficient firele de execuție pe core-uri [76]. Limitarea de transparență în controlul mapării firelor de execuție, conduce la dificultăți în obținerea localizării echilibrate a datelor la rularea programelor implementate folosind limbajele existente [76].

2.8 Aspecte conexe care influențează maparea firelor de execuție

2.8.1 Politici de planificare a firelor de execuție în sistemul de operare Linux

În Linux, sunt două categorii de *politici de planificare a firelor de execuție*, descrise în Tabelul 3 [78]. Fiecare fir de execuție are asignată o politică de planificare prin care se specifică prioritatea de execuție a acelui fir (prioritatea 0 reprezintă prioritatea cea mai mică). Fiecărei priorități de execuție îi corespunde o listă separată în care se păstrează firele de execuție care au acea prioritate, într-o anumită ordine.

Planificatoarele firelor de execuție ca *Completely Fair Scheduler* (CFS) [80],

utilizat în prezent de Linux, se axează pe echilibrarea încărcării și utilizarea în mod justificat a resurselor [8,81] și nu dețin informații despre comportamentul dinamic al aplicației, acesta neinfluențând planificarea. Apare necesitatea dezvoltării de algoritmi de mapare a firelor de execuție care să suprascrie efectul planificatoarelor din sistemul de operare și care să asigneze în mod explicit firele de execuție pe core-uri [5], paticularizând maparea în funcție de caracteristicile aplicației.

Tabel 3. Categorii de politici de planificare a firelor de execuție în Linux și politicile aferente categoriilor

Categoria de politică de planificare	**Politicile de planificare**
Politicile de planificare în timp real a firelor de execuție	***SCHED_FIFO (first in-first out):*** - politica implicită a acestei categorii, presupune existența și lansarea în prealabil în execuție a unui fir marcat cu o prioritate mai mare decât firul de tip ***SCHED_FIFO*** (de exemplu, firele de tip ***SCHED_OTHER***, ***SCHED_BATCH*** sau ***SCHED_IDLE***) și care implică execuția mai apoi a firului ***SCHED_FIFO*** - execuția unui fir de tip ***SCHED_FIFO*** implică următorul scenariu: 1. firul ***SCHED_FIFO*** este plasat în capul listei aferente priorității lui atunci când un fir cu prioritate mai mare, care impune execuția firului ***SCHED_FIFO***, este (re)lansat în execuție 2. firul ***SCHED_FIFO*** va fi (re)lansat în execuție atunci când toate firele cu prioritate mai mare care impun execuția lui sunt (re)blocate (lansarea în execuție a unui fir mai prioritar decât firul ***SCHED_FIFO***, blochează execuția acestuia) 3. atunci când firul ***SCHED_FIFO*** este (re)lansat în execuție, acesta va fi inserat la coada listei aferente priorității lui - alt scenariu care ar fi posibil la execuția unui fir ***SCHED_FIFO*** este următorul: 1. sunt apelate funcțiile ***sched_setscheduler(2)***, ***sched_setparam(2)*** sau ***sched_setattr(2)*** care plasează firul ***SCHED_FIFO*** în capul listei de priorități atunci când acesta este prezent în listă pe o altă poziție, fiind considerat "runnable" (firele ***SCHED_FIFO*** mai pot fi plasate și la coada listei prin apelul funcției ***sched_yield(2)***, care, în plus, blochează execuția firului) 2. un alt fir ***SCHED_FIFO*** aflat deja în capul listei ar putea rula atunci când se apelează funcțiile de la punctul 1, caz în care pentru acest fir se execută Punctele 1-3 de mai sus
	SCHED_RR (round-robin): - este descrisă ca fiind o îmbunătățire a politicii de planificare ***SCHED_FIFO***, totodată, tot ce este descris mai sus pentru ***SCHED_FIFO*** fiind valabil și pentru această politică, cu mențiunile: - spre deosebire de firele ***SCHED_FIFO***, firele ***SCHED_RR*** pot să ruleze doar cel mult o perioadă maximă de timp - dacă un fir ***SCHED_RR*** rulează mai mult decât perioada considerată maximă, atunci firul este plasat la coada listei aferente priorității sale - un fir ***SCHED_RR*** care a fost lansat în execuție în urma Pașilor 1-3 ai primului scenariului de execuție al politicii ***SCHED_FIFO*** de mai sus, se va executa atât timp cât i-a rămas până când să-și atingă perioada maximă de execuție (perioada maximă a firului de execuție este obținută prin apelul funcției ***sched_rr_get_interval(2)***)

<table>
<tr><td rowspan="3">Politicile de planificare normale ale firelor de execuție (nu sunt în timp real)</td><td>

SCHED_NORMAL (se mai numește și ***SCHED_OTHER***):
- este politica implicită pentru planificarea firelor de execuție partajate în timp și nu necesită mecanisme în timp real
- fire ***SCHED_OTHER*** pot fi doar cele care au prioritatea 0, prin urmare, acestea sunt mai puțin prioritare decât firele de tipul politicilor în timp real, iar alegerea unui fir ***SCHED_OTHER*** se face dintre firele din lista cu prioritate 0 parcurgând următorii pași:
1. se determină prioritatea firelor din interiorul listei pe baza atributului "nice value":
- acest atribut specific în Linux fiecărui fir de execuție în parte, favorizează sau nu selecția firului, primind o mai mare însemnătate în ceea ce privește luarea acestei decizii începând cu Linux 2.6.23 (de exemplu, o diferență de o unitate între valoarea atributelor a două fire de execuție se traduce într-un factor de 1.25 cu care unul din fire este selectat cu o prioritate mai mare decât celălalt pentru a fi executat)
- atributul poate fi modificat folosind funcțiile ***nice(2)***, ***setpriority(2)*** sau ***sched_setattr(2)***
2. prioritatea determinată conform Pasului 1 este incrementată de fiecare dată când se întâlnește un moment de timp în care firul este disponibil să ruleze, dar execuția acestuia este respinsă de planificator de fire de execuție Completely Fair Scheduler (CFS) [80], introdus ca planificator implicit începând cu Linux 2.6.23
3. se alege cea mai mică valoare dintre valorile atributelor tuturor firelor de execuție (de exemplu, valoarea -19 pentru atributul "nice value" reprezintă prioritatea cea mai mare)
Notă: Un procent mare al firelor de execuție ale unei aplicații care au o valoare foarte mică pentru atributul "nice value" (+19), va conduce la întârzierea/prelungirea execuției acelei aplicații.

</td></tr>
<tr><td>

SCHED_BATCH:
- se aseamănă cu politica ***SCHED_OTHER*** prin faptul că:
- firele ***SCHED_BATCH*** sunt selectate tot pe bază unei priorități calculate dinamic utilizând atributul "nice value", cu mențiunea că planificatorul de fire de execuție nu aplică o penalitate atât de mare firelor când respinge execuția acestora
- fire ***SCHED_BATCH*** pot fi doar cele care au prioritatea 0
- politica este utilă în următoarele cazuri:
- în cazul firelor de execuție care nu comunică și pentru care nu se dorește scăderea valorii atributului "nice value"
- în cazul firelor de execuție care necesită o planificare deterministă, fără interacțiune între acestea care ar putea determina lansarea în execuție a altor fire

</td></tr>
<tr><td>

SCHED_IDLE:
- politica este potrivită pentru firele de execuție cu prioritate foarte mică dacă firele ar fi de tipurile ***SCHED_OTHER*** sau ***SCHED_BATCH*** (mai mică chiar și decât valoarea +19 care reprezintă cea mai mică prioritate la politicile ***SCHED_OTHER*** și ***SCHED_BATCH***, dată de atributul "nice value")
- fire ***SCHED_IDLE*** pot fi doar cele care au prioritatea 0

</td></tr>
</table>

2.8.2 Impactul alocării memoriei asupra eficienței mapării firelor de execuție

Alocarea memoriei descrisă în contextul migrării datelor. Gestionarea memoriei în multe din sistemele de operare actuale se bazează pe alocarea așa-numită leneșă a memoriei care are proprietatea că o pagină de memorie fizică este alocată în contextul firului de execuție care o folosește primul, proprietate denumită first-touch [79]. Prin intermediul acestei proprietăți, sistemul de operare poate aloca paginile de memorie fizică în nodul de memorie plasat lângă core-ul care utilizează datele din aceste pagini [79]. În sisteme cu ierarhii de memorii cum sunt sisteme NUMA, ar putea fi necesară migrarea acestor date. În sistemele NUMA, migrarea datelor are un impact mare asupra performanței și a consumului de energie al mașinilor paralele [83,84]. Algoritmii propuși în această lucrare urmăresc să evite necesitatea oricărei migrări dinamice a datelor printr-o mapare eficientă a firelor de execuție la compilare.

Alocarea memoriei descrisă în contextul migrării firelor de execuție. O posibilitate este aceea de a considera că firele nu sunt plasate în execuție în locul potrivit. Dacă însă un fir de execuție care alocă primul o pagină de memorie fizică, nu este firul care va accesa acea pagină cel mai des în viitor, atunci acest lucru este un indiciu că firul nu a alocat pagina dintr-un loc de execuție potrivit. Sunt unele aplicații care, din acest motiv, alocă non-automat paginile de memorie fizică în timpul fazei de inițializare, pentru a asigura o distanță cât mai mică între firele de execuție și datele accesate [79].

Algoritmii de mapare contribuie – alături de alocarea eficientă a memoriei – la asigurarea proximității execuției firelor față de memoria utilizată, favorizând accesele locale la memoria cache (în cazul sistemelor NUMA, accesele locale sunt accese la memoria cache comună care poate corespunde primului nivel și/sau celui de-al doilea nivel de cache) în detrimentul acceselor la distanță (în cazul sistemelor NUMA, accesele la distantă sunt accese la memoria principală sau la memoria cache aferentă nivelelor de cache până la nivelul doi ale unei alte unități de procesare).

2.9 Reducerea consumului de energie prin maparea firelor de execuție

Adesea necesitatea reducerii consumului de energie apare în cazul dispozitivelor mobile și, mai apoi ca frecvență, în cazul centrelor de date (*data centers*) [84]. În cazul dispozitivelor mobile, consumul de energie afectează durata de viață a bateriei și limitează utilizarea dispozitivului. În ceea ce privește centrele de date, consumul de energie limitează numărul de echipamente funcționale din centre și afectează costurile de operare.

Optimizarea performanței este de multe ori considerată un substitut al metodelor de reducere a consumului de energie, având în vedere că sistemele care rulează într-un timp mai scurt, cel mai probabil vor consuma mai puțină energie. Deși optimizarea performanței este un pas important în direcția reducerii consumului de energie, acest lucru poate să nu fie suficient sau câteodată chiar să producă efectul opus în ceea ce privește consumul de energie. De exemplu, calculul paralel poate să îmbunătățească performanța prin reducerea timpului de procesare [84].

Totuși, salvarea și restaurarea contextului de execuție, planificarea firelor de execuție, degradarea localizării datelor ar putea să ducă la un consum de energie în plus de energie comparativ cu execuția secvențială [84], aspectele anterioare fiind totodată direcții de optimizare a paralelismului.

Cea mai importantă sursă de consum de energie în cazul calculatoarelor, este considerată a fi cea consumată de CPU, iar cea mai puțin semnificativă dintre sursele de consum de energie care nu sunt neglijabile, este memoria, în timp ce pentru dispozitivele mobile, componentele cele mai consumatoare de energie raportat la energia totală necesară funcționării calculatorului sunt ecranul, GPU-ul și CPU-ul (în această ordine), memoria fiind din nou clasată pe ultimul loc, conform [84].

În scopul optimizării consumului de energie este esențial a alege platforma hardware și algoritmul software care ar conduce la reducerea consumului de energie [84]. Conform [85], *procesoarele eterogene* sunt în continuă răspândire, ele permițând reducerea consumului de energie datorită diversității acestora în ceea ce privește performanța core-urilor componente. *Procesoarele eterogene* conțin atât core-uri rapide cât și core-uri mai puțin performante, ceea ce vine în ajutor aplicațiilor care, pentru a rula optim, impun ca lățimea de bandă a memoriei și alte aspecte ce țin de execuție, să fie diferite în cazul execuției unui fir față de execuția unui alt fir [85]. Se întâmplă frecvent ca, atunci când apare o nouă arhitectură, să se impună o optimizare a aplicației deja implementată pentru ca aceasta să țină cont de caracteristicile noii arhitecturi hardware (sau de caracteristicile noi ale unei arhitecturi existente) printr-un algoritm care să conducă la reducerea consumului de energie la rularea aplicației pe acea arhitectură. În cazul aplicațiilor ***multi-threading***, modificarea aplicației poate fi evitată și o optimizare care ia în calcul arhitectura poate fi obținută indirect, prin intermediul unui algoritm de mapare a firelor de execuție care ține cont de caracteristicile hardware ale CPU-ului și ale memoriei în realizarea mapării. Un astfel de algoritm este *Distributed Intensity Online-Energy* (DIO-E) [85]. Algoritmul mapează pe core-uri mai rapide aplicațiile care au rată mare de eșuare a obținerii datelor din cache și pe core-uri mai puțin performante apllicațiile care au această rată mai mică (cu cât este mai mare rata, cu atât core-ul trebuie să fie mai performant) [85].

Factorii actuali care influențează reducerea consumului de energie sunt scăderea volumului de calcule și scăderea numărului de cereri de obținere/eliberare a resurselor [84]. Conform [85], multe dintre studiile în acest sens arată că performanța și consumul de energie pot fi optimizate atunci când fire de execuție sunt mapate core-uri de tipul potrivit și când, în urma mapării, fiecare core execută un singur fir. Creșterea numărului de cereri de obținere/eliberare a resurselor în cazul unui core cauzată de fire de execuție care rulează pe un același core și care utilizează în comun memoria cache a core-ului, poate degrada semnificativ performanța [86,87]. În alte cazuri, performanța firelor care rulează pe același core și utilizează în comun aceleași date poate să nu fie afectată atunci când scopul distribuirii pe același core a firelor este reducerea numărului de eșuări în obținerea datelor din memoria cache [85]. Totuși, autorii din [85] afirmă că maparea firelor trebuie să țină cont de numărul de cereri de obținere/eliberare a resurselor. O astfel de mapare poate fi realizată prin determinarea celui mai potrivit tip de core pentru un grup de fire de execuție care sunt mapate pe același core și utilizează date în comun [86,87].

2.10 Concluzii

Capitolul descrie inițial 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 [3], care contribuie la minimizarea latenței operațiilor de aducere a datelor din memorie atunci când datele sunt plasate optim în memorie [4] (prin optim se înțelege faptul că datele sunt gestionate de controlerul potrivit)
- timpii de acces la memorie sunt inegali [3], 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*) [5]

Sistemele NUMA optimizează accesele la memorie – prin avantajele anterior menționate – cu ajutorul algoritmilor de mapare care urmăresc îmbunătățirea localizării datelor.

Algoritmul de mapare propus în această teză este static, aspectele care îngreunează acuratețea predicțiilor la compilare ale algoritmilor de mapare asupra comportamentului dinamic al programelor paralele fiind [43]:

- nu toate caracteristicile execuției sunt cunoscute (comportament dinamic)
- timpul de acces al datelor volatile și non-volatile este stocastic și nondeterministic
- 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 nondeterministic al execuție programului paralel

Au mai fost prezentate în capitol 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.

Operațiile de mapare dinamică 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* [59].

Sun necesare două informații în scopul determinării *mapării/migrării* optime a firelor de execuție (statică sau dinamică) – acestea sunt:

- 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*** [60])

Capitol mai prezintă optimizări software care eficientizează execuția aplicațiilor paralele pe sistemele NUMA, precum și optimizări ale compilatorului

LLVM [2] prin care se realizează acest lucru. Câteva din optimizările care îmbunătățesc paralelizarea automată prin reducerea timpului scurs cu gestiunea task-urilor sunt următoarele [35]: ***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** [2] 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 Assingment* (LBA) [43], *Opportunistic Load Balancing* (OLB) [43] și alți ***algoritmi Greedy*** descriși în [43], SCOTCH [58], METIS [63,64], algoritmul de mapare parte a setului de utilitare Zoltan [64], algoritmi care identifică tipare în graful de comunicarea între task-uri prin metoda descrisă în [65], Treematch [18,66], EagerMap [67] și ***NUMA-BTDM*** [55] (prezentat în teză) în funcție de criteriile: momentul mapării, metoda de mapare utilizată, considerarea sau nu a arhitecturii hardware pe care rulează aplicația.

Factorii care influențează în mod direct eficiența mapării firelor de execuție pe sisteme NUMA sunt:

- *factorii ce țin de arhitectura hardware*: lățimea de bandă, care este indicat să fie cât mai mare [73] și, mai puțin important ca lățimea de bandă, numărul de hopuri traversate în cazul acceselor la distanță la alte noduri NUMA [73], precum și numărul numărul procesoarelor [74], 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 [74].

Am arătat în 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) [80], 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 [8,81] 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 prezentat factorii care cresc consumul de energie al procesării paralele comparativ cu cea secvențială, cum ar fi salvarea și restaurarea contextului de execuție, planificarea firelor de execuție sau degradarea localizării datelor [84]. Algoritmii de mapare urmăresc optimizarea acestor procesări consumatoare de energie care apar în cazul calculului paralel, prin urmare se urmărește optimizarea performanței prin care, indirect, poate fi redus consumul de energie.

3. Prezentarea algoritmilor NUMA-BTLP și NUMA-BTDM

3.1. Descrierea algoritmului NUMA-BTLP de clasificare a firelor de execuție

Algoritmul ***NUMA-BTLP*** [87] clasifică firele de execuție în următoarele trei categorii în urma unei analize statice a codului la compilare:

1. *fire de execuție autonome*: un fir de execuție autonom nu necesită alte date calculate în paralel cu acesta de către alte fire de execuție și nu trebuie să fie executat pe același nod NUMA unde sunt executate celelalte fire de execuție (cu care nu comunică); astfel firele de execuție autonome sunt repartizate în execuție cât mai uniform posibil pe unitățile de procesare; criteriul de uniformitate este unul din criteriile de obținere a localizării echilibrate a datelor, implicând ca volumul de calcule al fiecărei unități de procesare să nu depășească media per unitate de procesare [39]; acest criteriu contribuie printre altele, la îmbunătățirea *localizării echilibrate a datelor* în cazul algoritmilor prezentați în lucrare
2. *fire de execuție alăturate*: sunt lansate în execuție cât mai apropiat în termeni de distanță NUMA de firele de execuție în raport cu care este alăturat, acest lucru însemnând pe aceleași CPU-uri ca fiecare dintre acestea; un fir de execuție considerat alăturat îndeplinește cel puțin un criteriu din următoarele:
    - execută o funcție care are cel puțin un parametru de ieșire citit de un fir de execuție sau un parametru de intrare scris de un fir
    - scrie o dată globală citită de un fir de execuție sau citește o dată globală scrisă de un fir
    - este creat, în mod frecvent, în corpul unei bucle (în mod frecvent, fire de execuție create utilizând același apel ***pthread_create*** din corpul buclei, sunt alăturate între ele)
3. *fire de execuție amânate*: sunt definite ca fire de execuție generate de un fir, care nu necesită a fi executate imediat în raport cu firul generator, celelalte fire care au același părinte și care rulează în paralel neutilizând datele pe care firul amânat le scrie; un fir de execuție amânat este, prin urmare, un fir care calculează date necritice (datele nu aparțin fluxului critic de execuție), alăturat în raport cu firul generator și autonom în raport cu alte fire și poate fi identificat conform următoarelor:
    - execută o funcție care are parametri de ieșire citiți exclusiv de firul de execuție generator, după scrierea datei de către firul amânat sau parametri de intrare scrișI de firul generator
    - nu este inclus în corpul unei bucle, în mod frecvent

3.2 Analiza tipurilor de fire de execuție definite de algoritmul NUMA-BTLP

3.2.1 Fire de execuție autonome

Firele de execuție autonome nu au dependențe de date cu nici un alt fir – un alt fir nu scrie nici o dată pe care firul autonom o citește, firul autonom nu scrie nici o dată citită de vreun alt fir și firul autonom citește datele citite doar de toate firele. În Codul 3 se observă un exemplu de fire de execuție autonome, create de firul de execuție principal, utilizând apeluri ***pthread_create*** (firele corespund nodurilor de pe același nivel din arborele de generare al firelor de execuție, detaliat ulterior). Firele utilizează date diferite, primul fir incrementând variabila *arg_thr0* și cel de-al doilea fir decremenentând variabila *arg_thr1*, prin urmare acestea sunt autonome și deci sunt setate pe core-uri diferite: primul fir pe core-ul 0 și cel de-al doilea fir pe core-ul 1, considerând ca acestea rulează pe un sistem cu cel puțin două core-uri.

Cod 3. Exemplu de fir de execuție autonom în raport cu un alt fir autonom

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 2

void * inc(void *args) {
    (*(int *)args)++; // se incrementează parametrul funcției
    return args;
}

void * dec(void *args) {
    (*(int *)args)--; // se decrementează parametrul funcției
    return args;
}

int main(int argc, char **argv) {
    pthread_t *thr = (pthread_t *)malloc(NUM_THREADS*sizeof(pthread_t));
    void * arg_thr0 = malloc(sizeof(void)), * arg_thr1 = malloc(sizeof(void));

    *(int *)arg_thr0 = atoi(argv[2]); // se primește din linia de comandă parametrul funcției
                                      // executate de fir se creează primul firul de execuție
    if(pthread_create(&thr[0], NULL, inc, arg_thr0)) { // dacă crearea firului a eșuat
        return EXIT_FAILURE;
    }
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr0; // se declară afinitatea CPU a primului fir de execuție
        CPU_ZERO(&CPUset_thr0); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(0, &CPUset_thr0); // se setează afinitatea CPU a firului așa încât acesta să
                                                        // ruleze pe core-ul cu indicele 0
        // se mapează firul pe core-ul cu indicele 0
        if(pthread_setaffinity_np(thr[0], sizeof(cpu_set_t), &CPUset_thr0)) { // dacă
                                                                    // maparea a eșuat
            return EXIT_FAILURE;
```

```
        }
    }

    *(int *)arg_thr1 = atoi(argv[3]); // se primește din linia de comandă parametrul funcției
                                      // executate de fir se creează al doilea fir de execuție
    if(pthread_create(&thr[2], NULL, dec, arg_thr1)) { // dacă crearea firului a eșuat
        return EXIT_FAILURE;
    }
    else // daca firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr1; // se declară afinitatea CPU a celui de-al doilea fir de execuție
        CPU_ZERO(&CPUset_thr1); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(1, &CPUset_thr1); // se setează afinitatea CPU a firului așa încât acesta să
                                                        // ruleze pe core-ul cu indicele 1
        // se mapeaza firul pe core-ul cu indicele 1
        if(pthread_setaffinity_np(thr[2], sizeof(cpu_set_t), &CPUset_thr1)) { // dacă
                                                                    // maparea a eșuat
            return EXIT_FAILURE;
        }
    }
    void *status;
    int i;
    for (i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], &status); // se așteaptă finalizarea tuturor firelor de execuție
        printf("\%d ", *(int *)status);
    }
}
```

3.2.2 Fire de execuție alăturate

Firele de execuție alăturate pot fi considerate astfel în raport cu fire generate în același nivel sau în raport cu firul generator, acest lucru putând fi observat în exemplul din Codul 4. În Codul 4, firul de execuție cu identificatorul 1 – execută funcția *read_first_element* – generează firele de execuție cu identificatorii 2 – execută funcția *read_all_elements* - și 3 – execută funcția *read_third_element*. Firele de execuție 1 și 2 utilizează ambele primul element din șirul de întregi alocat dinamic *arg_thr*, firul de execuție 2 utilizează toate elementele din șir și firele de execuție 2 și 3 utilizează ambele ultimul element din șirul *arg_thr*, ținând cont că numarul de elemente din șir este 3. Întrucât utilizează date comune, firul de execuție 2 este alăturat în raport cu firul generator 1, deci va fi mapat pe același core 0 ca firul 1, iar firul de execuție 3 este alăturat în raport cu firul de execuție 2 de pe același nivel, deci firul 3 va fi mapat pe același core 0 ca firul 2, de unde rezultă că toate cele trei fire sunt mapate pe același core 0.

Cod 4. Exemple de fir de execuție alăturat în raport cu firul generator respectiv în raport cu un alt fir de pe același nivel în ierarhia de generare a firelor de execuție

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 3

pthread_t *thr;

void * read_all_elements(void *args) {
```

```
    printf("read_all_elements: \%d \%d \%d\n", *(int *)args, *(((int *)args) + 1), *(((int
*)args) + 2));
    return args + 1;
}

void * read_third_element(void *args) {
    printf("read_third_element: \%d\n", *(((int *)args) + 2));
    return args + 2;
}

void * read_first_element(void *args) {
    printf("read_first_element: \%d\n", *(int *)args);

    // se creează cel de-al doilea fir de execuție
    if(pthread_create(&thr[2], NULL, read_all_elements, args)) { } // dacă crearea firului a
                                                                   // eșuat
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr0; // se declară afinitatea CPU a celui de-al doilea fir de execuție
        CPU_ZERO(&CPUset_thr0); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(0, &CPUset_thr0); // se setează afinitatea CPU a firului așa încât acesta să
                                                        // ruleze pe core-ul cu indicele 0
        pthread_setaffinity_np(thr[2], sizeof(cpu_set_t), &CPUset_thr0); // se mapează
                                                        // firul pe core-ul cu indicele 0
    }

    // se creează cel de-al treilea fir de execuție
    if(pthread_create(&thr[3], NULL, read_third_element, args)) { } // dacă crearea firului a
                                                                    // eșuat
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr1; // se declară afinitatea CPU a celui de-al treilea fir de execuție
        CPU_ZERO(&CPUset_thr1); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(0, &CPUset_thr1); // se setează afinitatea CPU a firului așa încât acesta să
                                                        // ruleze pe core-ul cu indicele 0
        pthread_setaffinity_np(thr[3], sizeof(cpu_set_t), &CPUset_thr1); // se mapează
                                                        // firul pe core-ul cu indicele 0
    }
    return args;
}

int main(int argc, char **argv) {
    thr = (pthread_t *)malloc(NUM_THREADS*sizeof(pthread_t));
    void * arg_thr = malloc(NUM_THREADS*sizeof(void));

    // se primește din linia de comandă un șir care este parametrul funcției executată de fire
    *(int *)arg_thr = atoi(argv[2]);
    *(((int *)arg_thr) + 1) = atoi(argv[3]);
    *(((int *)arg_thr) + 2) = atoi(argv[4]);
    // se creează primul fir de execuție
    if(pthread_create(&thr[1], NULL, read_first_element, arg_thr)) { // dacă crearea firului a
                                                                     // eșuat
        return EXIT_FAILURE;
    }
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr0; // se declară afinitatea CPU a primului fir de execuție
        CPU_ZERO(&CPUset_thr0); // se resetează afinitatea CPU a firului pe 0
```

```
        CPU_SET(0, &CPUset_thr0); // se setează afinitatea CPU a firului așa încât acesta să
                                                                    // ruleze pe core-ul cu indicele 0
        // se mapează firul pe core-ul cu indicele 0
        if(pthread_setaffinity_np(thr[1], sizeof(cpu_set_t), &CPUset_thr0)) { // dacă
                                                                    // maparea a eșuat
            return EXIT_FAILURE;
        }
    }

    void *status;
    int i;
    for (i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], &status); // se asteaptă finalizarea tuturor firelor de execuție
    }
}
```

3.2.3 Fire de execuție amânate

Firele de execuție amânate pot fi considerate astfel doar în raport cu firul de execuție generator, acest lucru putând fi observat în exemplul din Codul 5. În Codul 5, firul de execuție amânat 2 nu are dependențe de date cu nici un alt fir exceptând firul generator 1 – firul 2 scrie o dată care este utilizată târziu de firul generator 1, chiar înainte de încheierea acestuia, prin urmare acesta poate fi mapat pe un alt core (core-ul 1) decât firul de execuție generator (mapat pe core-ul 0) pentru a putea balansa încărcarea core-urilor. Balansarea este realizată prin maparea firului amânat pe core-ul cel mai puțin încărcat (core-ul 1 este primul cel mai puțin încărcat core, core-ul 0 executând firul 1), permițând cu prioritate firelor alăturate în raport cu firul generator să fie mapate pe același core ca firul generator. Firele alăturate 3 și 4 (unul în raport cu celălalt și ambele în raport cu firul generator 1) scriu datele utilizate de firul generator 1, firul generator având nevoie de aceste date cu mult înainte de a procesa datele scrise de firul amânat (deși acesta este creat primul).

Cod 5. Exemplu de fir de execuție amânat

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 3

pthread_t *thr;

void * writes_data(void *args) {
    printf("Un fir\n");
    *(int *)args = 0;
    return args;
}

void * writes_other_data(void *args) {
    printf("Un alt fir\n");
    *(((int *)args) + 1) = 1;
    *(((int *)args) + 2) = 2;
    return args;
}
void * execute_thread(void *args) {
```

```
    // se creează cel de-al doilea fir de execuție
    if(pthread_create(&thr[2], NULL, writes_data, args)) { } // dacă crearea firului a eșuat
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr2; // se declară afinitatea CPU a celui de-al doilea fir de execuție
        CPU_ZERO(&CPUset_thr2); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(1, &CPUset_thr2); // se setează afinitatea CPU a firului așa încât acesta să
                                                                    // ruleze pe core-ul cu indicele 1
        pthread_setaffinity_np(thr[2], sizeof(cpu_set_t), &CPUset_thr2); // dacă maparea a
                                                                    // eșuat
    }

    // se creează cel de-al treilea fir de execuție
    if(pthread_create(&thr[3], NULL, writes_other_data, args)) { } // dacă crearea firului a
                                                                    // eșuat
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr3; // se declară afinitatea CPU a celui de-al treilea fir de execuție
        CPU_ZERO(&CPUset_thr3); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(0, &CPUset_thr3); // se setează afinitatea CPU a firului așa încât acesta să
                                                                    // ruleze pe core-ul cu indicele 0
        pthread_setaffinity_np(thr[3], sizeof(cpu_set_t), &CPUset_thr3); // dacă maparea a
                                                                    // eșuat
      }

    // se creează cel de-al patrulea fir de execuție
    if(pthread_create(&thr[4], NULL, writes_other_data, args)) { } // dacă crearea firului a
                                                                    // eșuat
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr4; // se declară afinitatea CPU a celui de-al doilea fir de execuție
        CPU_ZERO(&CPUset_thr4); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(1, &CPUset_thr4); // se setează afinitatea CPU a firului așa încât acesta să
                                                                    // ruleze pe core-ul cu indicele 0
        pthread_setaffinity_np(thr[4], sizeof(cpu_set_t), &CPUset_thr4); // dacă maparea a
                                                                    // eșuat
    }

    void *status;
    pthread_join(thr[4], &status); // se așteaptă finalizarea firului alăturat cu identificatorul 4
    pthread_join(thr[3], &status); // se așteaptă finalizarea firului alăturat cu identificatorul 3

    /* porțiune de cod care utilizează datele scrise de firul amânat cu identificatorul 2 */

    pthread_join(thr[2], &status); // se asteaptă finalizarea firului amânat cu identificatorul 2
    printf("\%d\n", *(int *)status);

    return args;
}

int main(int argc, char **argv) {
    thr = (pthread_t *)malloc(NUM_THREADS*sizeof(pthread_t));
    void * arg_thr = malloc(NUM_THREADS*sizeof(void));

    // se primeste din linia de comanda un sir care este parametrul functiei executata de fire
    *(int *)arg_thr = atoi(argv[2]);
    *(((int *)arg_thr) + 1) = atoi(argv[3]);
    *(((int *)arg_thr) + 2) = atoi(argv[4]);
```

```
    // se creează primul fir de execuție
    if(pthread_create(&thr[1], NULL, execute_thread, arg_thr)) { // dacă crearea firului a
                                                                  // eșuat
        return EXIT_FAILURE;
    }
    else // dacă firul a fost creat cu succes
    {
        cpu_set_t CPUset_thr1; // se declară afinitatea CPU a primului fir de execuție
        CPU_ZERO(&CPUset_thr1); // se resetează afinitatea CPU a firului pe 0
        CPU_SET(0, &CPUset_thr1); // se setează afinitatea CPU a firului așa încât acesta să
                                                    // ruleze pe core-ul cu indicele 0
        if(pthread_setaffinity_np(thr[1], sizeof(cpu_set_t), &CPUset_thr1)) { // dacă
                                                                  // maparea a eșuat
            return EXIT_FAILURE;
        }
    }

    void *status;
    pthread_join(thr[1], &status); // se așteaptă finalizarea tuturor firelor de execuție din
                                                                  // funcție
}
```

3.3 Reprezentarea sub formă de arbore a ierarhiei de generare a firelor de execuție

Informațiile unui fir de execuție – tipul și identificatorul firului, tipul și identificatorul firului generator – sunt salvate într-o structură denumită Nod. Într-un program paralel care folosește ***Pthreads*** [45] pentru obținerea paralelismului la nivel de task, firele de execuție pot fi reprezentate sub forma unui arbore în următorul mod: un nod de pe nivelul i din arbore care stochează informația legată de firul de execuție i_k este fiu pentru un nod de pe nivelul $j = i - 1$ care stochează informația legată de firul de execuție j_l atunci când firul de execuție i_k este generat de firul de execuție j_l printr-un apel ***pthread_create*** în codul firului j_l, $\forall\ i \geq 1$ și $\forall\ k = 1,n_i$, $l = 1,n_j$, unde n_i și n_j reprezentă numărul de noduri de pe nivelul i, respectiv de pe nivelul j, rădăcina aborelui stocând informația legată de firul principal de execuție. De exemplu, pentru un program cu următoarele fire de execuție, reprezentarea firelor de execuție sub formă de arbore este ilustrată în Fig. 3:

1. firul de execuție principal are identificatorul -1 și acestuia nu îi este asignat nici un tip
2. firul de execuție cu identificatorul 0 este autonom și este generat de firul de execuție principal
3. firul de execuție cu identificatorul 1, generat și el de firul de execuție principal, este autonom în raport cu firul cu identificatorul 0 și alăturat în raport cu un alt fir (cel cu identificatorul 2)
4. firul de execuție cu identificatorul 2 este alăturat în raport cu firul autonom generator cu identificatorul 0
5. firul de execuție cu identificatorul 3 este alăturat în raport cu firul de execuție generator cu identificatorul 2, care este și acesta alăturat
6. firul de execuție cu identificatorul 4 este amânat în raport cu firul de execuție generator cu identificatorul 2 care este alăturat
7. firul de execuție cu identificatorul 5 este amânat și este generat de firul de execuție autonom cu identificatorul 1.

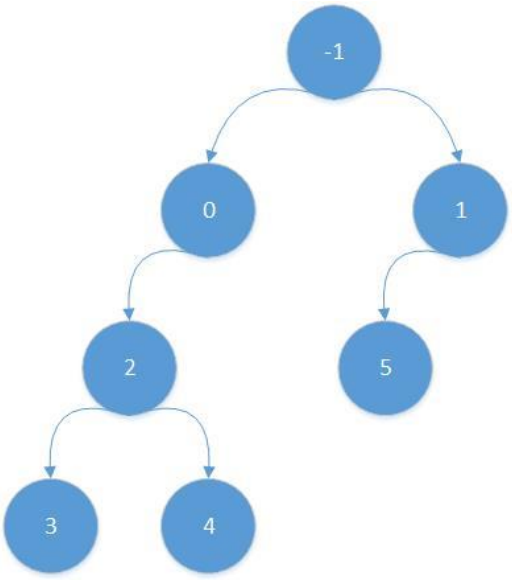

Figura 3. Reprezentarea ierarhiei de generare a firelor de execuție sub formă de arbore

3.4 Reprezentarea sub formă de arbore a tiparelor de comunicare între firele de execuție

Reprezentarea comunicării între firele de execuție este creată de algoritmul ***NUMA-BTLP*** [87] printr-un arbore a tiparelor de comunicare între firele de execuție, după următoarele considerente: un nod de pe nivelul i din arbore care stochează informația legată de firul de execuție i_k este fiu pentru un nod de pe nivelul $j = i - 1$ care stochează informația legată de firul de execuție j_l atunci când firul de execuție i_k utilizează datele scrise sau citite în prealabil de firul de execuție j_l, $\forall\ i \geq 1$ și $\forall\ k = 1,n_i$, $l = 1,n_j$, unde n_i și n_j reprezentă numărul de noduri de pe nivelul i, respectiv de pe nivelul j, rădăcina aborelui stocând informația legată de firul principal de execuție. Dacă atât firul de execuție i_k cât și firul de execuție j_l utilizează prima dată câte o dată utilizată apoi și de celălalt, de exemplu firul de execuție i_k scrie o dată care este apoi citită de firul de execuție j_l iar firul de execuție i_k scrie o altă dată care este apoi citită de firul de execuție j_l atunci relația de părinte-fiu este stabilită în funcție de care fir este creat primul astfel: dacă firul de execuție i_k este creat primul atunci acesta va fi parintele firului de execuție j_l și invers, dacă firul de execuție j_l este creat primul, atunci acesta este părintele lui i_k.

Pentru a exemplifica un arbore de tipare de comunicare între firele de execuție, am definit tiparul de comunicare între firele de execuție din "Anexa A1 – Aplicarea algoritmilor ***NUMA-BTDM*** în mod automat și ***NUMA-BTLP*** în mod neautomat pe un exemplu de cod cu un anumit tipar de comunicare între firele de execuție", ilustrat grafic în Fig. 4. Exemplul din Fig. 4 își propune să utilizeze toate tipurile de fire de execuție definite în această lucrare, într-o reprezentare grafică sub

formă de arbore construit după următorul tipar:

1. două fire de execuție autonome;
2. un fir de execuție alăturat în raport cu primul fir de execuție autonome de la punctul 1;
3. un fir de execuție alăturat în raport cu firul de execuție de la punctul 2;
4. un fir de execuție amânat în raport cu firul de execuție de la punctul 2;
5. un fir de execuție amânat în raport cu cel de-al doilea fir de execuție autonom de la punctul 1;
6. ciclul se reia pentru următoarele șase fire de execuție începând cu punctul 1.

Exemplul își mai propune să ilustreze grafic următoarele aspecte:

1. un fir de execuție autonom este fiu în arborele de comunicare al firului de execuție generator
2. un fir de execuție alăturat în raport cu un altul, este părintele în arborele de comunicare al celui de-al doilea fir
3. un fir de execuție amânat este, de asemenea, fiul al firului de execuție generator

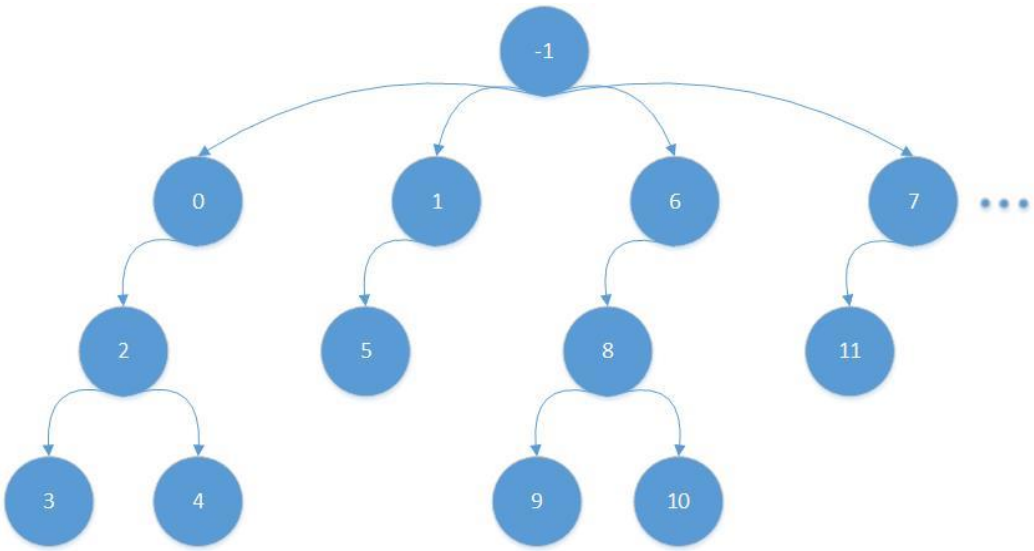

Figura 4. Reprezentarea tiparului de comunicare a firelor de execuție sub formă de arbore

În "Anexa A1 – Aplicarea algoritmilor ***NUMA-BTDM*** în mod automat și ***NUMA-BTLP*** în mod neautomat pe un exemplu de cod cu un anumit tipar de comunicare între firele de execuție" se poate observa că fiecărui fir îi este asignată o anumită porțiune din ***tabloul*** alocat dinamic *args*, iar limitele acestei porțiuni din tablou se calculează în funcție de identificatorul firului de execuție ca în Fig. 5 unde firele de execuție 0, 2, 3 și 4 scriu toate elementele din porțiunea din ***tabloul*** *args* situată între indexul 0 și indexul 999, firele 1, 5 citesc ambele toate elementele din porțiunea din tabloul args situată între indexul 1000 și indexul 1999 ș.a.m.d. Atunci când firele de execuție utilizează o zona comună de memorie, acestea fac parte din același subarbore al nodului rădăcină (nodul rădăcină reprezintă firul principal de execuție). Acest lucru este ilustrat grafic în Fig. 4, unde firele de execuție 0,2,3 și 4

aparțin unui subarbore al nodului rădăcină, firele de execuție 1 și 5 aparțin și ele unui alt subarbore al nodului rădăcină ș.a.m.d.

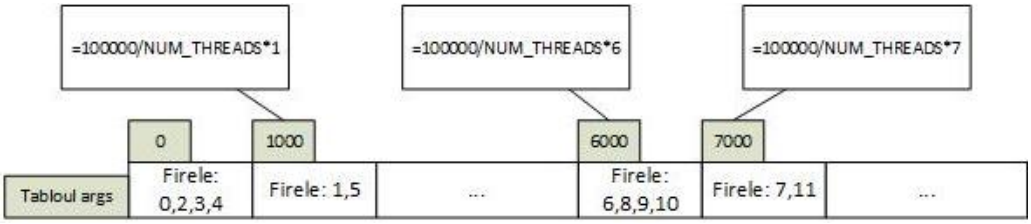

Figura 5. Reprezentarea grafică a zonelor de memorie din tabloul alocat dinamic args utilizate de firele de execuție

3.5 Implementarea algoritmului NUMA-BTLP în pseudocod

Relația între firele de execuție și apelurile de funcții este de unu-la-unu în ***Pthreads*** [45], firelor de execuție fiindu-le atașat un pointer la funcție la crearea acestora. Codul 6 descrie analiza pe care compilatorul o realizează pe codul sursă la întâlnirea în cod a unui grup funcții numit modul în **LLVM** [2]. Înainte de rularea pasului de optimizare ***NUMA-BTLP*** [87], este indicat să se ruleze pasul de optimizare *Loop-Invariant Code Motion* (LICM) pe fiecare buclă din cod. În urma aplicării acestei optimizări, un apel ***pthread_create*** care este în corpul unei buclei, poate fi mutat în afara buclei, dacă nu depinde de iterațiile acesteia, simplificând astfel execuția programului. Analiza presupune următorii pași aplicați pentru fiecare parametru de ieșire al funcției executate de firul notat cu A și pentru fiecare dată globală scrisă sau citită de funcție [87]:

1. Este identificat firul generator al firului de execuția A, notat PA, și este traversat în preordine subarborele care are ca rădăcină nodul firului de execuție PA
2. Pentru fiecare fir din coada de traversare sunt executate următoarele:
    - Dacă firul execută o funcție care primește ca parametri o dată scrisă de firul excuție A, atunci cele două fire de execuție, cel din coada de traversare și firul A, sunt setate ca alăturate
    - Dacă firul de execuție citește sau scrie o data globală utilizată de firul de executie A, atunci cele două fire de execuție, cel din coada de traversare și firul A, sunt setate ca alăturate
    - Dacă firul de execuție are o dependență de date cu firul de execuție PA și nu are nici o dependență de date cu un alt fir, se setează firul de execuție A ca amânat. Altfel, dacă firul de execuție PA coincide cu firul principal de execuție, se setează firul de execuție A ca autonom. În orice alt caz, se setează firul ca amânat.

Funcția aplicaAlgoritmulNUMABTLPPentruFirul – Codul 6 – apelează funcția obtineFireDependenteDeFirPrinTraversareaInPreordineASubarboreluiCuRadacina care traversează în preordine subarborele cu rădăcina în părintele nodului pentru care a fost apelată funcția aplicaAlgoritmulNUMABTLPPentruFirul, adică acel nod care corespunde firului de execuție denumit în continuare A. Pentru fiecare nod traversat, care corespunde câte unui fir de execuție, se verifică, în funcția nuAreFunctiaDependenteDeDateCuVreunParametruAlFunctiei, ca parametrii funcției

atașate firului A să nu fie utilizați în funcția atașată firului traversat și, în funcția nuAreFunctiaDependenteDeDateCuVreoDataGlabalaScrisaDeFunctiei ca datele globale scrise de firul A în funcția sa atașată, să nu fie utilizate de firul traversat, în funcția atașată acestuia. Analiza dependențelor de date se face prin analize de tip alias între un operand și o instrucțiune în funcția nuEsteParametrulDependentDpdvAliasDeInstructiune sau între un operand și o instrucțiune încuibată unei alte instrucțiuni, în funcția nuEsteParametrulDependentDeOInstructiuneIncuibataAInstructiunii. În cazul în care există vreo dependență de date între firul traversat și firul A, adică una din funcțiile nuAreFunctiaDependenteDeDateCuVreunParametruAlFunctiei sau nuAreFunctiaDependenteDeDateCuVreoDataGlabalaScrisaDeFunctiei, returnează adevărat, atunci cele două fire sunt considerate alăturate unul în raport cu celălalt și intersecția core-urilor pe care mapate e nevidă (firul A este mapat pe toate core-urile pe care este mapat firul traversat, dar nu și invers), altfel, dacă firul de execuție A are dependențe de date doar cu firul său generator, acesta se consideră amânat și este mapat pe cel mai puțin încărcat core (identificat static), altfel (dacă nu se identifică nici o dependență de date), firul A se consider a fi de tipul autonom.

Cod 6. Analiza algoritmului ***NUMA-BTLP*** apelata de fiecare data atunci cand este întâlnit un apel al functiei ***pthread_create***

```
funcția aplicăAlgoritmulNUMABTLPPentruFirul(fir)
    firPărinte <- obțineFirulPărinteÎnIerarhiaDeGenerareAlFirului(fir)
    fireDependenteDeFir <-
obțineFireDependenteDeFirPrinTraversareaÎnPreordineASubarboreluiCuRădăcina(firPărinte, fir)
    setează TipulFirului DacăEsteAmânat(fireDependenteDeFir, fir, firPărinte)
    setează TipulFirelor DacăFirulEsteAlăturat(fireDependenteDeFir, fir)
    setează TipulFirului DacăEsteAutonom(fireDependenteDeFir, fir)

funcția
obțineFireDependenteDeFirPrinTraversareaInPreordineASubarboreluiCuRădăcina(firCurent, fir)
    funcțieFirCurent <- obțineObiectulDeTipFuncțieAtașatFirului(firCurent)
    funcțieFir <- obțineObiectulDeTipFuncțieAtașatFirului(fir)
    nuEsteDependent <- aplicăAlgoritmulNUMABTLPPentruFuncția(funcțieFirCurent,funcțieFir)
    daca nuEsteDependent = fals atunci
        adaugăLaCoadaListeiFirelorDependenteFirul(firCurent)
    pentru fiecare fiu al firului firCurent execută
        obțineFireDependenteDeFirPrinTraversareaÎnPreordineASubarboreluiCuRădăcina(fiu,fir)

funcția aplicăAlgoritmulNUMABTLPPentruFuncția(funcție, funcțieCuCareSeCompară)
    nuEsteDependent <- nuAreFuncțiaDependențeDeDateCuVreunParametruAlFuncției(funcție,
funcțieCuCareSeCompara) sau
nuAreFunctiaDependenteDeDateCuVreoDataGlabalaScrisaDeFunctie(functie,
funcțieCuCareSeCompară)
    dacă nuEsteDependent = fals atunci
        returnează fals
    returnează adevarat

funcția nuAreFuncțiaDependențeDeDateCuVreunParametruAlFuncției(funcție,
funcțieCuCareSeCompară)
    nuEsteParametrulDependent <- fals
    pentru fiecare argument al funcției funcțieCuCareSeCompară execută
        A <- obținePrinCastUnObiectDeTipInstrucțiuneDinObiectul(argument)
        pentru fiecare bloc BB din funcția funcție execută
            pentru fiecare instrucțiune I din BB execută
```

```
            nuEsteParametrulDependent <- nuEsteParametrulDependent și
nuEsteParametrulDependentDpdvAliasDeInstrucțiune(I, A) și
nuEsteParametrulDependentDeOInstrucțiuneÎncuibatăAInstrucțiunii(I, A, fals)
            dacă nuEsteParametrulDependent = fals atunci
                returnează fals
    returnează adevărat

funcția nuEsteParametrulDependentDpdvAliasDeInstrucțiune(I, A)
    AA <- obțineUnObiectDeTipAnalizaAlias()
    rezultatAA <- obțineRezultatulAnalizeiAliasPentruInstrucțiunile(I,A)
    dacă rezultatAA <> nici un alias atunci
        IPtr <- obținePrinCastUnObiectDeTipAdresăAInstrucțiuniiDinObiectul(I)
        APtr <- obținePrinCastUnObiectDeTipAdresăAInstrucțiuniiDinObiectul(A)
        dacă IPtr <> nul și APtr <> nul atunci
            operandDeTipPointerScrisDeIPtr <-
obțineOperandulDeTipPointerScrisDeInstrucțiunea(IPtr)
            operandDeTipPointerScrisDeAPtr <-
obțineOperandulDeTipPointerScrisDeInstrucțiunea(APtr)
            dacă operandDeTipPointerScrisDeIPtr = operandDeTipPointerScrisDeAPtr atunci
                SI <- obținePrinCastUnObiectDeTipInstrucțiuneDeScriereDinObiectul(I)
                dacă SI <> nul atunci
                    returnează fals
            pentru fiecare instrucțiune IDep care utilizează instrucțiunea I execută
                SI <- obtinePrinCastUnObiectDeTipInstrucțiuneDeScriereDinObiectul(IDep)
                dacă SI <> nul atunci
                    returnează fals
    returnează adevărat

funcția nuEsteParametrulDependentDeOInstrucțiuneÎncuibatăAInstrucțiunii(I, arg,
nuEsteDependent)
    A <- obținePrinCastUnObiectDeTipValoareDinObiectul(arg)
    pentru fiecare operand al instrucțiunii I execută
        inst <- obținePrinCastUnObiectDeTipInstrucțiuneDinObiectul(operand)
        dacă inst <> nul atunci
            SI <- obținePrinCastUnObiectDeTipInstrucțiuneDeAtribuireDinObiectul(inst)
            op <- obțineOperandulAtribuitDeInstrucțiunea(SI)
            opv <- obținePrinCastUnObiectDeTipValoareDinObiectul(op)
            dacă SI <> nul și opv = A atunci
                returnează fals
        altfel
            returnează (nuEsteDependent = nuEsteDependent și
nuEsteParametrulDependentDpdvAliasDeInstrucțiune(inst,arg) și
nuEsteParametrulDependentDeOInstrucțiuneÎncuibatăAInstrucțiunii(inst,arg,nuEsteDependent)
    returnează fals

funcția nuAreFuncțiaDependenteDeDateCuVreoDatăGlabalăScrisăDeFuncție(funcție,
funcțieCuCareSeCompară)
    pentru fiecare bloc BB din funcția funcțieCuCareSeCompară execută
        pentru fiecare instrucțiune I din BB execută
            primulOperand <- obținePrimulOperandDacăEsteDatăGlobalăAlInstrucțiunii(I)
            dacă primulOperand <> nul atunci
                esteOperand <- esteOperandÎnVreoInstrucțiuneAFuncției(funcție,primulOperand)
                dacă esteOperand = adevărat atunci
                    returnează fals
    returnează adevărat

funcția obținePrimulOperandDacăEsteDatăGlobalăAlInstrucțiunii(I)
    SI <- obținePrinCastUnObiectDeTipInstrucțiuneDeAtribuireDinObiectul(I)
```

```
    dacă SI <> nul atunci
        primulOperand <- obținePrimulOperandAlInstrucțiunii(SI)
        esteDatăGlobală <- esteDatăGlobalăOperandul(primulOperand)
    dacă esteDatăGlobală = adevărat
        returnează primulOperand
    returnează nul

funcția esteOperandÎnVreoInstrucțiuneAFuncției(functie, operandulCăutat)
    pentru fiecare bloc BB din funcție
        pentru fiecare instrucțiune I din BB
            pentru fiecare operand O al instrucțiunii I:
                dacă O = operandulCăutat atunci
                    returnează adevărat
    returnează fals
```

Analiza din Codul 6 este apelată mai jos, în Codul 7, pentru fiecare grup de funcții din codul de intrare, numit modul. Funcția din Codul 7 primește ca parametru un modul în reprezentare intermediară a compilatorului, identifică în cadrul fiecărei funcții din modul, apeluri ***pthread_create*** dacă acestea există și inserează, după fiecare dintre acestea, câte un apel ***pthread_setaffinity_np***, prin care se mapează firul aferent conform algoritmilor ***NUMA-BTLP*** [88] și ***NUMA-BTDM*** [55].

Cod 7. Aplicarea analizei algoritmului ***NUMA-BTLP*** pentru fiecare funcție în parte din codul de intrare

```
funcția rulează PentruModulul(M)
    populează IerarhiaDeGenerare(M)
    inserează DetaliileArhitecturiiDacă FuncțiaEsteMain(M)
    inserează FirulDeExecuțieCorespunzătorFuncțieiMainÎnIerarhiaDeComunicare()
    pentru fiecare functie F din modulul M
        pentru fiecare bloc de bază BB din funcția F
            pentru fiecare instrucțiune I din blocul de bază BB
                CI <- obținePrinCastUnObiectDeTipApelDeFuncțieDinObiectul(I)
                dacă CI <> nul și atunci
                    numeFuncțieApelată <- obțineNumeleFuncției(CI)
                    dacă numeFuncțieApelată = "pthread_create" atunci
                        iteratorArgumenteFuncție <-
                                    obțineAdresaPrimuluiArgumentAlFuncțieiApelate(FCI)
                        identificatorFirCreat <-
          obținePrinCastUnObiectDeTipConstantăIntregDinObiectul(iteratorArgumenteFuncție)
                        verificăPrinAserțiuneValiditateaPrimuluiParametruAlFuncțieApelate(CI)
                        iteratorArgumenteFuncție <- iteratorArgumenteFuncție + 2
                        CIA <-
          ObținePrinCastUnObiectDeTipApelDeFuncțieDinObiectul(iteratorArgumenteFuncție)
                        FA <- obțineFuncțiaApelatăDinObiectulDeTipApelDeFuncție(CIA)
                        verificăPrinAserțiuneValiditateaParametrului3AlFuncțieApelate(FA)
                        fireFiu <- obțineDinIerarhiadeGenerareFireleCareExecutăFuncția(FA)
                        pentru fiecare firFiu din firele fiu fireFiu:
                            firPărinte <- obțineFirulPărinteAlFirului(firFiu)
                            aplicăAlgoritmulNUMABTLPPentruFirul(firPărinte) – Codul 6
                            adaugăFirulLaCoadaListeiDeFireIdentificatePânăÎnAcelMoment(firFiu) -
***NUMA-BTDM*** [55] Codul 8
                            numărFire <- obțineNumărulDeFireIdentificatePînăÎnAcelMoment()
                            esteAutonom <-
recalculează AfinitateaCPUATuturorFirelorAutonomeDacăFirulEsteAutonom(firFiu,numărFire) -
***NUMA-BTDM*** [55] Codul 8
```

```
                    firFiuInIerarhiaDeComunicare <-
                            obțineEchivalentulInIerarhiaDeComunicareAlFirului(firFiu)
                    esteAlăturat <-
            setează AfinitateaCPUAFiruluiDacăEsteAlăturat(firFiuInIerarhiaDeComunicare)
    - NUMA-BTDM [55] Codul 8
                    esteAmânat <-
        setează AfinitateaCPUAFiruluiDacăEsteAmânat(firFiuInIerarhiaDeComunicare) -
NUMA-BTDM [55] Codul 8
                    dacă esteAutonom ==adevărat sau esteAmânat == adevărat
                        afinitateaCPU <- obțineAfinitateaCPUAFirului(firFiu)
                    dacă esteAlăturat == adevărat
                        listaFire <- obțineFireleÎnRaportCuCareFirulEsteAlăturat(firFiu)
                        pentru fiecare firAlăturat din listaFire:
                            afinitateCPUFirAlăturat <-
                                                obțineAfinitateaCPUAFirului(firAlăturat)
                            afinitateCPU<-
                                aplicăCPU_ORAsupra(afinitateCPUFirAlăturat,afinitateaCPU)
                    dimensiuneCPUSet <-
            obțineObiectulDeTipValueCuValoareaDimensiuneaÎnOcteți ATipului(cpu_set_t)
                    CPUSet <- obțineObiectulDeTipValueCuValoarea(afinitateCPU)
                    inserează DupăApelulPThreadCreateUnApelPThreadSetAffinityNpCu
Parametrii(identificatorFirCreat, dimensiuneCPUSet, CPUSet)
    retunează fals
```

3.6 Implementarea algoritmului NUMA-BTDM în pseudocod

Algoritmul ***NUMA-BTDM*** [55] de mapare a firelor de execuție din teză, utilizează apeluri ale funcțiilor din biblioteca ***Pthreads*** [45] prezentate mai sus pentru a împiedica sistemul de operare să aplice la rulare politicile de planificare a *task*-urilor fără a ține cont de caracteristicile aplicației paralele. Astfel, algoritmul ***NUMA-BTDM*** [55] permite aplicației să controleze maparea firelor de execuție prin apelul funcțiilor de mai sus cu parametrii setați de compilator (sunt setate core-urile pe care să ruleze firul de execuție) pe baza caracteristicilor statice ale codului.

Se consideră următoarele:

1. în cazul bibliotecii ***Pthreads*** [45] relația între funcții și fire de execuție este de unu-la-unu, adică unui fir de execuție creat prin apelul funcției ***pthread_creat*** i se asociază o funcție (nu pot fi zero sau mai multe funcții, ci o singură funcție) pe care acesta o va executa
2. o funcția execută un singur *task* (conform regulilor de scriere a codului este indicat ca o funcție să facă un singur lucru, să fie o unitate indivizibil din punct de vedere logic)

Algoritmul ***NUMA-BTDM*** [55] din Codul 8 este apelat de ***NUMA-BTLP*** [88] la identificarea fiecărui fir de execuție. ***NUMA-BTDM*** [55] calculează afinitatea CPU a firului de excuție, așa încât firele să fie mapate cât mai uniform pe core-uri, în următorul mod:

1. firele de execuție autonome sunt mapate uniform pe unitățile de procesare pentru a balansa încărcarea core-urilor (funcția recalculează AfinitateaCPUATuturorFirelorAutonomeDacăFirulEsteAutonom)

2. firele de execuție alăturate sunt mapate pe aceleași core-uri ca firele de execuție în raport cu care sunt alăturate, pentru a îmbunătății localizarea datelor (funcția setează**A**finitateaCPUAFiruluiDacă**E**steAlăturat) firele de executie amânate sunt mapate fiecare pe cel mai puțin încărcat core până la momentul respectiv, încărcarea fiind determinată de numărul de fire de execuție asignate acelui core, număr determinat static (funcția setează**A**finitateaCPUAFiruluiDacă**E**steAmânat)

NUMA-BTDM [55] decide, prin urmare, la compilare, core-ul/core-urile pe care să fie executat fiecare fir de execuție în funcție de tipul acestuia.

Efectele produse de algoritmul ***NUMA-BTDM*** sunt:

1. Maparea uniformă a firelor de execuție pe core-urile sistemului ceea ce îmbunătățește *localizarea echilibrată a datelor* pe sisteme NUMA
2. Optimizarea *ratei de obținere a datelor din memoria cache* pe sisteme NUMA datorită alăturării în termeni de distanță NUMA a firelor de execuție care folosesc aceleași date și prin urmare împart același cache, nemai fiind necesară aducerea acelorași date în mai multe cache-uri diferite

Cod 8. Algoritmul ***NUMA-BTDM***

```
funcția
recalculeazăAfinitateaCPUATuturorFirelorAutonomeDacăFirulEsteAutonom(fir,sloturiLiberePână
Acum)
    tipFir <- obțineTipulFirului(fir)
    dacă tipFir = autonom atunci
        setează**P**eZeroAfinitateaTuturorFirelorDeExecuțieAutonome()
        resetează**D**imensiuneaColecției(indicatorÎncărcare, numărUnități**D**eProcesareLogice)
        slotCurent <- 0
        poziție**P**U = 0.0
        interval = sloturiLiberePânăAcum / numărUnități**D**eProcesareFizice
        cât timp numărulDeFireAutonome < sloturiLiberePânăAcum și slotCurent <
                                                                sloturiLiberePânăAcum execută
            firCurent <- obțineFirDinSlotul(slotCurent)
            tipFirCurent <- obțineTipulFirului(firCurent)
            dacă tipFirCurent = autonom atunci
                PU <- floor(poziție**P**U) mod numărulUnități**I**orDeProcesareFizice
                numărUnități**D**eProcesare <-
                                    obțineNumărulDeUnități**D**eProcesareAsignateFirului(firCurent)
                cât timp numărUnități**D**eProcesare < numărUnități**D**eProcesareFizice execută
                    esteUnitateaSetată <-
                                     esteUnitateaDeProcesareDejaSetată**P**entruFirul(PU,firCurent)
                    dacă esteUnitateaSetată = fals atunci
                        adaugă**L**aCoadaListeiDeUnități**D**eProcesareAsignateFiruluiUnitateaDe
Procesare(PU)
                        actualizează**A**finitateaCPUAFiruluiCuUnitateaDeProcesare(firCurent,PU)
                        incrementează**I**ndicatorulDeÎncarcareAlUnității**D**eProcesare(PU)
                        incrementează**N**umărulDeUnități**D**eProcesareAleFirului(firCurent)
                    PU <- (PU + numărUnități**D**eProcesareLogice) mod
                                                                numărUnități**D**eProcesareFizice
                poziție**P**U <- poziție**P**U + interval
                numărulDeFireAutonome <- numărulDeFireAutonome + 1
            slotCurent <- slotCurent + 1

funcția setează**A**finitateaCPUAFiruluiDacă**E**steAlăturat(fir)
    tipFir <- obțineTipulFirului(fir)
    părinteFir <- obține**P**ărinteleFirului(fir)
    tipFir**P**ărinte <- obțineTipulFirului(părinteFir)
```

```
    primulSlotDisponibil <- obținePrimulSlotDisponibil()
    dacă tipFir = alăturat atunci
        dacă părinteFir <> nul și tipPărinteFir = autonom si tipFir = alăturat atunci
            setează NumărulDeUnități DeProcesareAsignateSlotuluiCuValoarea(primulSlot
Disponibil,număr Unități DeProcesareLogice)
            j <- 0
            pentru adresa fiecărei unități de procesare PU din lista unităților de procesare ale
                                                                    slotului primulSlotDisponibil
                PU <- j
                j <- j + număr Unități DeProcesareLogicePerCore
                actualizează AfinitateaCPUA Firului DeLa SlotulCu Unitatea DeProcesare(PU,primul
SlotDisponibil)
                incrementează IndicatorulDeÎncărcare Al Unității DeProcesare(PU)
        ieșire din funcție
    setează NumărulDeUnități DeProcesareAsignateSlotuluiCuValoarea(primulSlotDisponibil,
                                obține NumărulDeUnități DeProcesareAsignate Firului(firCurent))
    copiază Unitățile DeProcesare Ale Firului ParinteÎn Fir()
    slotPărinteFir <- obține SlotDin Firul(părinteFir)
    pentru fiecare unitate de procesare PU din lista unităților de procesare ale slotului
                                                                                slotPărinteFir
        actualizează AfinitateaCPUA Firului DeLa SlotulCu Unitatea DeProcesare(PU,primulSlot
Disponibil)
        incrementează IndicatorulDeÎncarcare Al Unității DeProcesare(PU)

funcția setează AfinitateaCPUA Firului DacăEsteAmânat(fir)
    ceaMaiPuținÎncărcată UnitateDeProcesare <-
                                            obține CeaMaiPutinÎncărcată UnitateDeProcesare()
    slotFir <- obține SlotDin Firul(fir)
    pentru adresa fiecărei unități de procesare PU din lista unităților de procesare ale slotului
                                                                                    slotFir
        PU <- ceaMaiPutinÎncărcată UnitateDeProcesare
        actualizează AfinitateaCPUA Firului DeLa SlotulCu Unitatea DeProcesare(PU,slotFir)
        incrementează IndicatorulDeÎncărcare Al Unității DeProcesare(PU)
        ceaMaiPuținÎncărcată UnitateDeProcesare
        număr Unități DeProcesareLogicePerCore <- ceaMaiPuținÎncărcată UnitateDeProcesare +
                                                    număr Unități DeProcesareLogicePerCore
    setează NumărulDeUnități DeProcesareAsignateSlotuluiCuValoarea(slotFir,număr Unități De
ProcesareLogicePerCore)
```

3.7 Concluzii

Cercetarea vizează optimizarea aplicaţiilor paralele C/C++ care utilizează biblioteca ***Pthreads*** [45] 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.

Analiză statică clasifică firele de execuție în trei tipuri: autonome, alăturate și amânate (algoritmul ***NUMA-BTLP*** [87]), maparea statică în funcție de tipul firului de execuție (algoritmul ***NUMA-BTDM*** [55]) 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.

Scăderea 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 aceleași core-uri şi celelalte fire sunt distribuite uniform pe core-uri.

Algoritmii ***NUMA-BTLP*** [88] și ***NUMA-BTDM*** [55] sunt implementați în compilatorul **LLVM** [2]. Algoritmii utilizează două structuri arborescente în implementarea acestora: arborele de generare al firelor de execuție și arborele de comunicare al firelor de execuție [88].

Arborele de generare al firelor de execuție este construit conform următoarelor reguli [88]:

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 funcțiile ataşate firelor create din funcţia main, reprezintă al treilea nivel din arbore ş.a.m.d.

Arborele de generare al firelor de execuție 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 [87].

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 dependenţe 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.

Algoritmii mai utilizează o structură arborescentă – arborele de comunicare al firelor de execuție [88]. Acesta este construit pe baza următoarelor reguli [88]:

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 [87]. Analiza identifică dependențele de date între fir și toate firele din subarborele cu

rădăcina părintele firului, dacă aceste dependențe există [87]. Cu ajutorul claselor specifice din compilatorului **LLVM** [2], 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*** [55]) 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.

Am detaliat, în încheierea capitoului, 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*** [88]);
2. Creează arborele de generare a firelor de execuţie din reprezentarea intermediară **LLVM IR** [2] a codului de intrare (***NUMA-BTLP*** [88]);
3. Determină tipul fiecărui fir de execuţie pe baza unei analize statice şi îl adaugă în arborele de comunicare (***NUMA-BTLP*** [88]);
4. Mapează firele de execuţie pe baza arborelui de comunicare obţinut în pasul anterior (***NUMA-BTDM*** [55]);
5. Identifică în reprezentarea intermediară **LLVM IR** [2] a codului de intrare, câte un apel ***pthread_create***, care creează firul de execuţie (***NUMA-BTLP*** [88]);
6. Adaugă după fiecare ***pthread_create***, câte un apel ***pthread_set_affinity_np***, care setează afinitatea CPU a firului creat conform mapării obţinute în pasul anterior (***NUMA-BTLP*** [87]).

Elementele de noutate ale celor doi algoritmi sunt:

1. Abilitatea de a permite aplicațiilor paralele C/C++ care utilizează ***Pthreads*** [45] 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 aspectelor dinamice, precum numărului de fire de execuție, în faza de compilare, prin inserarea de către

algoritmul ***NUMA-BTLP*** [87], la compilare, imediat după fiecare apel ***pthread_create***, a câte unui apel ***pthread_setaffinity_np***, care mapează firul sau firele create de ***pthread_create***, indiferent de numărul acestora

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 [88]

4 Rezultate experimentale

4.1 Obiective specifice propuse la implementarea algoritmilor și criterii de îndeplinire

1. Demonstrarea portabilității algoritmilor propuși în lucrare pe diferite sisteme NUMA (cu număr diferit de core-uri) prin inserarea, la compilare, în codul sursă de intrare în reprezentare intermediară, a comenzilor Linux care obțin numărul total de core-uri fizice și număr de core-uri logice per **CPU**, specifice arhitecturii hardware pe care se execută programul. Inserția este realizată de către algoritmul ***NUMA-BTLP*** [87]. Obiectiv realizat pentru orice arhitectură hardware și pentru sistemul de operare Linux prin Anexa A2 – Implementarea algoritmilor ***NUMA-BTLP*** și ***NUMA-BTDM***
2. Reducerea timpului de execuție pe sisteme NUMA a aplicațiilor paralele care folosesc ***Pthreads*** [45] și/sau ***OpenMP*** [51] cu timp de execuție considerabil (de ordinul zecilor de minute/orelor) prin aplicarea celor doi algoritmi la compilare. Această reducere a timpului de execuție intervine atunci când se obține la rulare un timp de îmbunătățire T_{imb} mai mare decât timpul de mapare T_m (timpul de mapare este timpul consumat cu execuția apelurilor ***pthread_setaffinity_np***). Obiectiv realizat prin secțiunea de rezultate experimentale dedicată aplicației de referință **CPU** [90].
3. Reducerea și explicarea motivului reducerii consumului total de putere prin cei doi algoritmi. Această reducere a consumului de energie intervine atunci când se reduce consumul de putere al aplicației paralele optimizate E_{opt} comparativ cu consumul neoptimizat E_{neopt}, obținute respectând procedura de măsurare a consumului de putere care folosește dispozitivul WattUp și utilitarul ***turbostat***, descrisă mai jos. Obiectiv realizat prin rezultatele experimentale obținute pentru aplicațiile de referință **CPU** [90] și **Context Switch** [91] și prin secțiunea de discuții a rezultatelor experimentale.

4.2 Materiale și metode de obținere a rezultatelor experimentale

4.2.1 Arhitectura hardware pe care au fost obținute rezultatele experimentale

Rezultatele experimentale sunt obținute pe o mașină Fujitsu Workstation Celsius R930 Power care rulează sub sistemul de operate Linux (Ubuntu 17.10 artful). Mașina are 2 procesoare Intel Xeon E5-2630 v2 ivy bridge, fiecare procesor având 6 core-uri (în total 12 core-uri), fiecare core având 2 core-uri logice când opțiunea *HyperThreading* este activată.

Sistemul conține în fiecare procesor 2 nivele de cache privat (L1 și L2) și 1 nivel de cache comun tuturor core-urilor (L3).

Împărțind un cache comun din nivelul L3, de dimensiune mai mare ca memoriile cache de pe L2, firelor de execuție amânate le este permis să utilizeze în comun mai multe date, dar, pe de altă parte, se traversează mai multor nivele de cache pentru obținerea datelor.

Sistemul a fost configurat pe rând ca fiind UMA, respectiv NUMA, prin modificarea unei opțiuni în BIOS, la pornirea sistemului.

Măsurătorile de consum de putere al CPU-ului au fost obținute cu ajutorul utilitarului software ***turbostat*** concomitent cu măsurătorile de consum de putere al sistemului, obținute utilizând dispozitivul specializat în măsurarea consumului de energie WattsUp. În ambele cazuri, rezultatele au fost furnizate sub formă de tabel în care, pe lângă măsurătorile de consum de putere, mai sunt prezente și alte statistici, fiind necesară conceperea mai multor programe sursă, câte unul pentru fiecare sursă de obținere a măsurătorilor (WattsUp, ***turbostat***) și mai apoi fiecare program modificat pentru fiecare aplicație testată în parte, programele fiind utilizate pentru extragerea consumului de putere din coloana aferentă din tabel.

4.2.2 Metodologia de obținere a rezultatelor experimentale de timp de execuție

Rezultatelor experimentale de timp de execuție s-au obținut rulând fiecare aplicație de 40 de ori și calculând media timpilor de execuție obținuți la fiecare rulare. Au fost obținute măsurători pe sistem NUMA și pe sistem UMA, atât atunci când algoritmul ***NUMA-BTDM*** a fost aplicat în prealabil la compilare, cât și atunci când algoritmul nu a fost aplicat.

4.2.3 Metodologia de obținere a rezultatelor experimentale de consum de putere

Consumul de putere al întregului sistem a fost măsurat utilizând dispozitivul WattsUp și un program aferent de colectare a datelor utilizat de sistemul de operare Linux și consumul CPU-ului a fost măsurat folosind utilitarul software ***turbostat***. Rata de achiziție a consumului de putere utilizând atât dispozitivul WattsUp cât și utilitarul ***turbostat*** este de 1 s. Datele obținute din cele două surse au fost corelate pentru a arăta că valorile obținute concomitent din cele două surse diferă prin aceeași valoare, constantă în timp (ținând cont de faptul că experimentele au fost supuse acelorași condiții de măsurare pe toată durata realizării experimentelor).

Au fost obținute 2 executabile, unul pentru aplicația optimizată utilizând algoritmul ***NUMA-BTDM*** [55] și celălalt pornind de la aplicația neoptimizată. Pentru realizarea rezultatelor experimentale, fiecare din cele două executabile au fost rulate de 40 de runde până la încheierea timpului de execuție sau maxim 15 minute (după acest interval de timp execuția fiind stopată utilizând apelul sistem Linux timeout), atât pe sistem NUMA cât și pe sistem UMA. Fiecare rundă conține un număr de măsurători de consum de putere egal cu numărul de secunde în care rulează aplicația în runda respectivă. Pentru fiecare din cele 40 de runde au fost obținute

minimul, media, maximul, varianța și deviația standard a valorilor obținute la fiecare 1 s în runda respectivă.
Nota: Metodologia se aplică fiecărei aplicații testate, mai puțin uneia dintre aplicații, căreia nu i se aplică în întregime.

Metodologia de măsurare a consumului de putere al CPU-ului de mai sus a fost aplicată în toate cazurile următoare:

1. aplicația a fost optimizată la compilare folosind algoritmul ***NUMA-BTDM*** și a fost rulată fără să ruleze nici un alt proces în paralel cu aceasta
2. aplicația nu au fost optimizată și au fost rulată fără să ruleze nici un alt proces în paralel cu aceasta
3. nici un proces nu a fost rulat obținându-se consumul de putere în stare "în așteptare"

4.3 Simularea algoritmului NUMA-BTLP pentru aplicațiile testate

Aplicațiile testate sunt următoarele aplicații de referință din industria IT:

1. **CPU-X** Benchmark [89]
2. **CPU** Benchmark [90]
3. **Context Switch** Benchmark [91]

Fiind vorba de un proces de compilare și execuție predefinit și specific fiecăreia din aplicațiile testate, proces care a necesitat instalarea mai multor programe de care depind compilarea și/sau rularea aplicațiilor și care implică folosirea comezilor Linux ***cmake*** și ***make***, am preferat să aplic algoritmul ***NUMA-BTLP*** [87] în mod neautomat, asupra acestor aplicații, decât să le compilez utilizând compilatorul clang din infrastructura de compilare **LLVM** [2] care include algoritmul ***NUMA-BTLP*** [87], astfel: am inserat în codul aplicațiilor apeluri ale funcției ***pthread_setaffinity_np*** care realizează maparea firele de execuție pe core-urile specificate la apel prin parametrul de tip ***cpu_set_t***, reprezentând afinitatea CPU a firului de execuție (core-urile pe care este mapat firul). Am determinat afinitatea CPU pentru fiecare fir de execuție în parte, printr-o analiză a dependențelor de date din cod, pentru fiecare aplicație, și am setat afinitatea CPU aplicând succesiv următorii pași:

1. Utilizând macrodefiniția cu parametri ***CPU_ZERO***, care inițializează afinitatea CPU (adică structura ***cpu_set_t*** primită ca parametru), nefiind specificat nici un core pe care să se execute firul
2. Utilizând macrodefiniția cu parametrii ***CPU_SET***, care primește ca parametri:
   - numărul de ordine al core-ului ca prim parametru (core-urile începând cu numărul 0 sunt setate de la dreapta la stânga în reprezentarea pe biți a afinității CPU)
   - structura de date ***cpu_set_t*** transmisă prin adresă ca al doilea parametru și care va conține afinitatea CPU a firului de execuție

   care setează pe valoarea 1, în afinitatea CPU transmisă ca parametru, bitul care corespunde core-ului dat ca prim parametru (dacă bitul nu are deja valoarea 1)

Codul 9 conține un exemplu de program prin care se setează afinitatea CPU a unui fir de execuție, stocată în variabila *cpu_affinity_mask*. Utilizând ***CPU_SET***, firul de execuție este setat să ruleze pe core-urile 0 și 2 și apoi, prin apelul funcției

pthread_setaffinity_np, firul de execuție este mapat pe aceste core-uri, după crearea acestuia prin apelul funcției ***pthread_create***.

Cod 9. Setarea afinității CPU a unui fir de execuție și maparea corespunzătoare afinității acestuia

```
#include <pthread.h>
#include <stdio.h>

void * function(void * arg) {
    printf("Thread with id \%d is executing", pthread_self());
}

int main(int argc, char **argv)
{
    pthread_t* thread = (pthread_t *)malloc(sizeof(pthread_t));
    void *arg = malloc(sizeof(void));
    if(pthread_create(thread, NULL, function, arg)) {
        return EXIT_FAILURE;
    }
    cpu_set_t CPU_affinity_mask;
    CPU_ZERO(&CPU_affinity_mask);
    CPU_SET(0, &CPU_affinity_mask);
    CPU_SET(2, &CPU_affinity_mask);
    if(pthread_setaffinity_np(*thread, sizeof(cpu_set_t), &CPU_affinity_mask)) {
        return EXIT_FAILURE;
    }
    return 0;
}
```

4.4 Aplicația de referință CPU-X

4.4.1 Descrierea aplicației

Aplicația **CPU-X** [89] rulează în buclă infinită, afișând, prin intermediul unei interfețe grafice, parametri hardware legați de configurația memoriei și de funcționarea sistemului, parametri ai sistemului de operare precum și ai execuției aplicației în sine, pe care îi actualizează la fiecare secundă, cu excepția celor care nu își schimbă valoarea în timp.

Cu privire la utilizarea bibliotecii ***Pthreads*** [45] în implementarea aplicației, aceasta apelează într-o buclă ***for*** o funcție care creează firele de execuție utilizând această bibliotecă. În acest caz, firele de execuție se consideră a fi create în corpul buclei, deși create indirect printr-o funcție apelată din corp, sunt marcate, conform algoritmului ***NUMA-BTLP*** [87], ca alăturate, fiind mapate pe aceleași core-uri, conform algoritmului ***NUMA-BTDM*** [55], prin inserția în codul aplicației a unei porțiuni de cod similare cu cea din Codul 9.

4.4.2 Metodologia de obținere a rezultatelor experimentale de consum de putere specifică aplicației de referință CPU-X

Metodologia de obținere a rezultatelor experimentale de consum de putere pentru aplicația de referință **CPU-X** [89] diferă de metodologia generală prin numărul de executabile testate și prin numărul de runde în care a fost rulat fiecare executabil în parte.

Aplicația **CPU-X** [89] rulează utilizând un număr de fire de execuție care poate varia dinamic sau static în intervalul [1, n], unde n este numărul de core-uri ale sistemului pe care este lansată aplicația, valoare obținută dinamic la lansarea în execuție. În cazul acestei aplicații, au fost obținute în prealabil 24 de executabile împărțite în două categorii: un număr de 12 pornind de la aplicația optimizată utilizând algoritmul ***NUMA-BTDM*** [55] și 12 pornind de la aplicația neoptimizată. Cele 12 executabile din fiecare categorie, rulează fiecare cu câte un număr de fire de execuție din intervalul menționat: primul executabil rulează utilizând 1 fir de execuție, cel de-al doilea executabil rulează utilizând 2 fire de execuție, ș.a.m.d. Pentru realizarea rezultatelor experimentale, au fost considerate, pentru fiecare categorie, câte 5 runde în care fiecare executabil rulează, pe rând, câte 15 minute. În total durata experimentului a fost de 1800 de minute.

4.4.3 Rezultate experimentale pe sisteme UMA

4.4.3.1 Timp de execuție

Nu au fost obținute rezultate experimentale. Aplicația **CPU-X** [89] rulează în buclă infinită.

4.4.3.2 Consum de putere

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când aplicația rulează neoptimizat și atunci când rulează optimizat*

Abscisa reprezintă numărul de fire de execuție cu care este lansată aplicația iar ordonata reprezintă consumul de putere în W al întregului sistem – în Fig. 6. Din Fig. 6 se observă că atunci când aplicația **CPU-X** [89] rulează pe un sistem Uniform Memory Access (UMA), consumul de putere al sistemului este mai mic atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat, diferențele fiind sub 1 W.

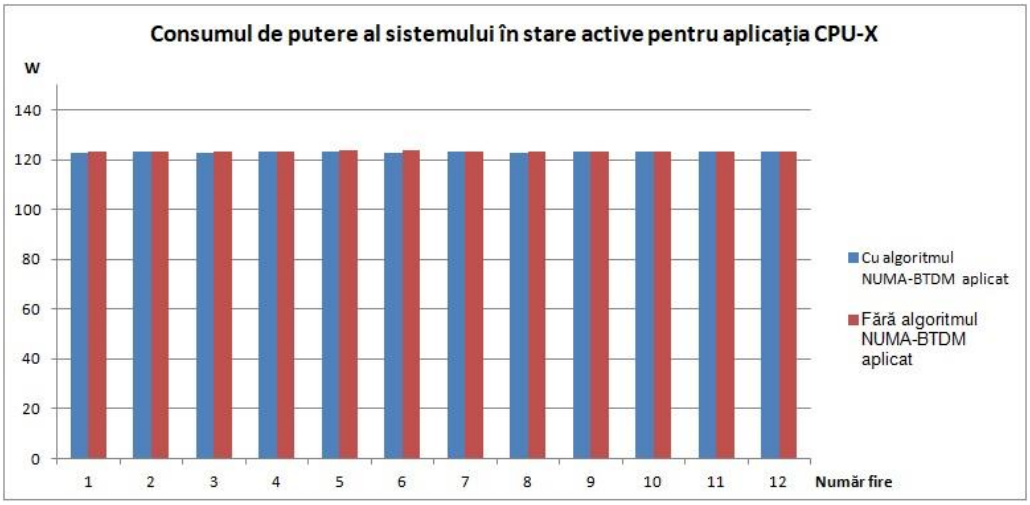

Figura 6. Consumul de putere (W) al sistemului UMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Se observă, în Fig. 7, că aplicația **CPU-X** [89] rulează pe sistem UMA cu consum de putere al CPU-ului mai mic atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat. În figură, abscisa are aceeași semnificație ca în Fig. 6, iar ordonata se referă la consumul de putere în W al CPU-ului. Spre deosebire de consumul de putere al sistemului din Fig. 6, unde optimizarea este mai mică de 1 W, în cazul Fig. 7, optimizarea consumului de putere al CPU-ului este mai mare de 1 W, fapt care se datorează optimizării acceselor la memorie prin creșterea ratei de obținere a datelor din memoria cache locală a core-ului pe care sunt mapate toate firele alăturate ale aplicației. Totuși, consumul de putere al sistemul nu este optimizat cu aceeași valoare datorită numărului mare de tranziții activ-”în așteptare”, rezultat în urma mapării tuturor

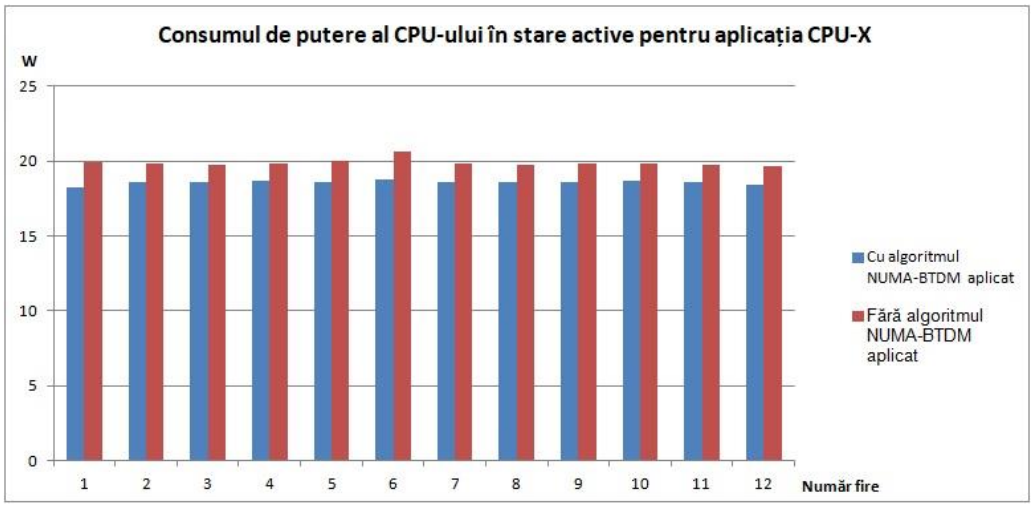

Figura 7. Consumul de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

firelor de execuție pe același core.

Consumul de putere al CPU-ului și cel al întregului sistem este ilustrat – în Fig. 8 – pentru o comparație valorică, atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când nu este aplicat. Se observă că măsurătorile sunt corelate și consumul CPU-ului este semnificativ mai mic decât cel al întregului sistem pentru aplicația **CPU-X** [89].

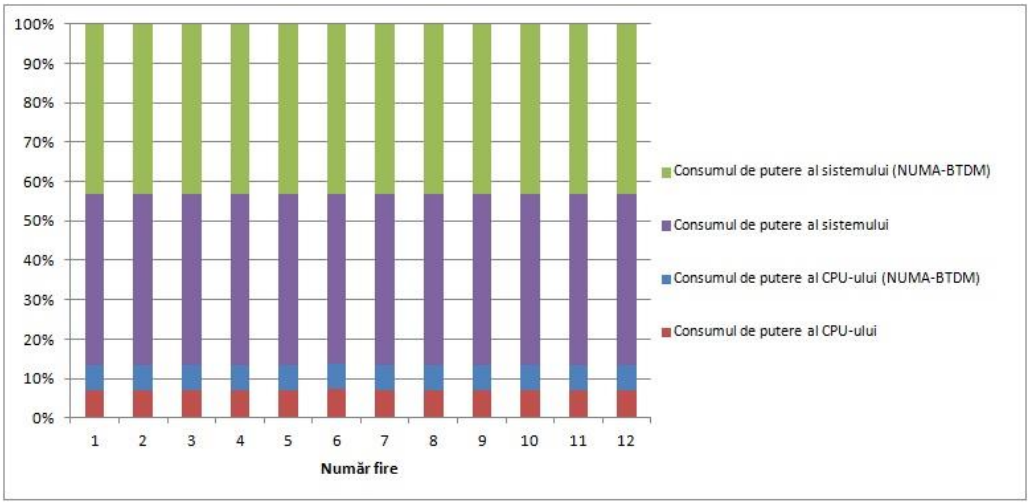

Figura 8. Consumul de putere (W) al CPU-ului și al întregului sistem UMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Celelalte componente ale sistemului cu excepția CPU-ului consumă putere constantă, astfel că, cu cât diferența între consumul de putere al întregului sistem UMA și cel al CPU-ului este mai mare, cu atât consumul de putere al CPU-ului este mai mic. Scăderea consumului de putere al CPU-ului se datorează reducerii numărului de operații cu memoria principală, ca urmare a unei mapări eficiente a firelor de execuție. Reducerea operațiilor cu memoria principală se obține prin maparea firelor alăturate, între ele, pe aceleași core-uri și prin maparea uniformă, pe câte un core separat, a fiecărui fir autonom, ceea ce scade probabilitatea ca firul autonom să elimine din cache datele folosite curent de mai multe alte fire (alăturate). Fig. 9 ilustrează această diferență pentru aplicația **CPU-X** [89], cu și fără algoritmul ***NUMA-BTDM*** [55] aplicat. De exemplu, atunci când aplicația **CPU-X** [89] optimizată rulează cu 7 fire de execuție, diferența este mare, ceea ce indică un consum de putere mic. Acest lucru se poate vedea și din Figura 7.

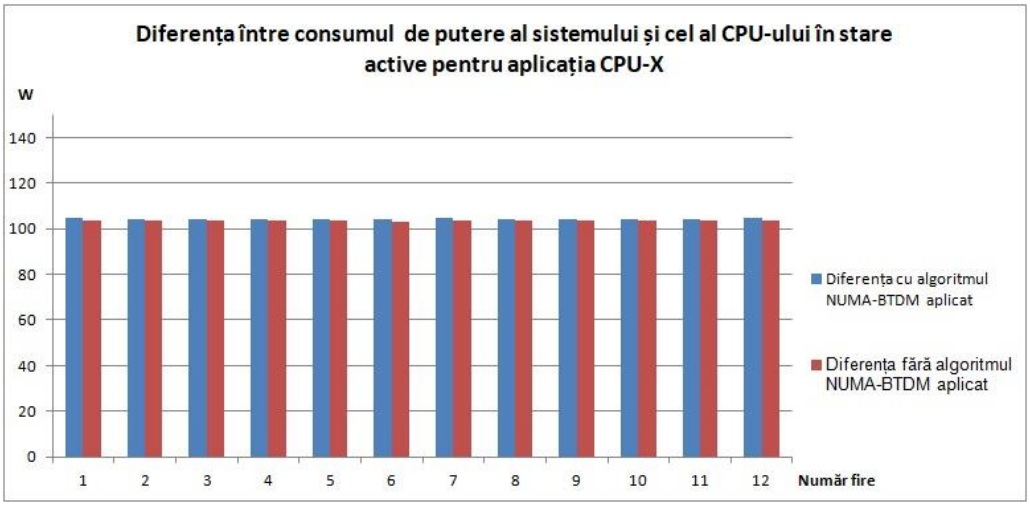

Figura 9. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Figura 10 indică faptul că, mediile a câte 900 de valori de consum de putere, obținute utilizând dispozitivul WattsUp în runda 1 de rularea pe sistem UMA, câte una pentru fiecare executabil, sunt corelate cu cele obținute utilizând utilitarul ***turbostat***. Fiecare valoare medie este calculată ca fiind media aritmetică a celor 900 de valori obținute câte una în fiecare secundă, executabilul rulând timp de 900 s (15 min).

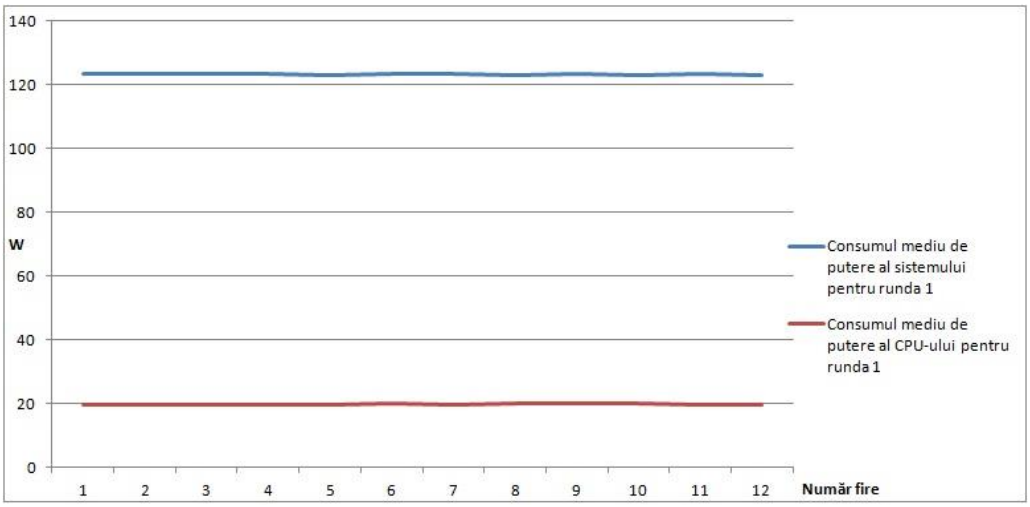

Figura 10. Valorile medii de consum de putere (W) al CPU-ului din sistemul UMA și al sistemului pe care rulează aplicația **CPU-X** obținute în runda 1 pentru diferite valori ale numărului de fire de execuție

Figura 11 indică faptul că valorile de consum de putere obținute în runda 1 din cele două surse, WattsUp, respectiv ***turbostat***, sunt corelate. Termenul de

corelare este utilizat în continuare în lucrare cu același sens și anume: în condițiile în care măsurătorile au fost obținute fără vreun alt proces rulat în paralel și cu echipamentele periferice (monitorul) deconectate, creșterea consumului de putere al CPU-ului cu o anumită valoare, va determina creșterea consumului de putere al întregului sistem, cu aceeași valoare. Similar, s-au întocmit grafice pentru toate rundele și a fost remarcată aceeași observație.

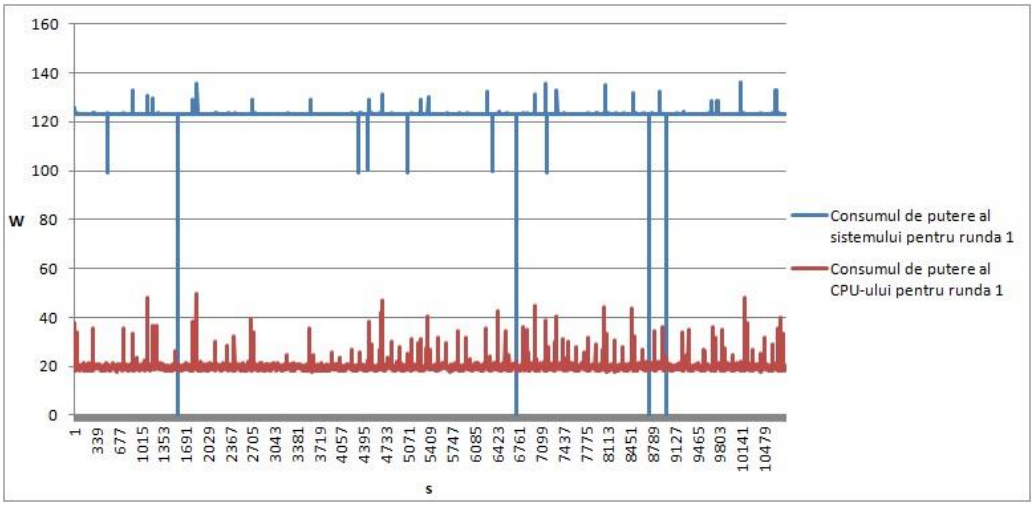

Figura 11. Valorile de consum de putere (W) al CPU-ului din sistemul UMA și al sistemului pe care rulează aplicația **CPU-X** obținute în runda 1

Consumul de putere al sistemului comparativ cu cel al CPU-ului *în stare "în așteptare"*

Fig. 12 prezintă consumul de putere al sistemului UMA în stare "în așteptare" (fără nici o aplicație rulând). Fiecare valoare de pe abscisă corespunde mediei aritmetice a câte 900 de valori de consum putere a sistemului în stare "în așteptare" obținute câte una la fiecare secundă: valoarea 1 de pe abscisă corespunde mediei aritmetice a primelor 900 de valori de consum de putere al sistemului, valoarea 2 de pe abscisă corespunde mediei aritmetice a următoarelor 900 de valori de consum de putere al sistemului, ș.a.m.d. Sistemul în stare "în așteptare" consumă ~122 W și măsurătorile obținute au varianța 7.98.

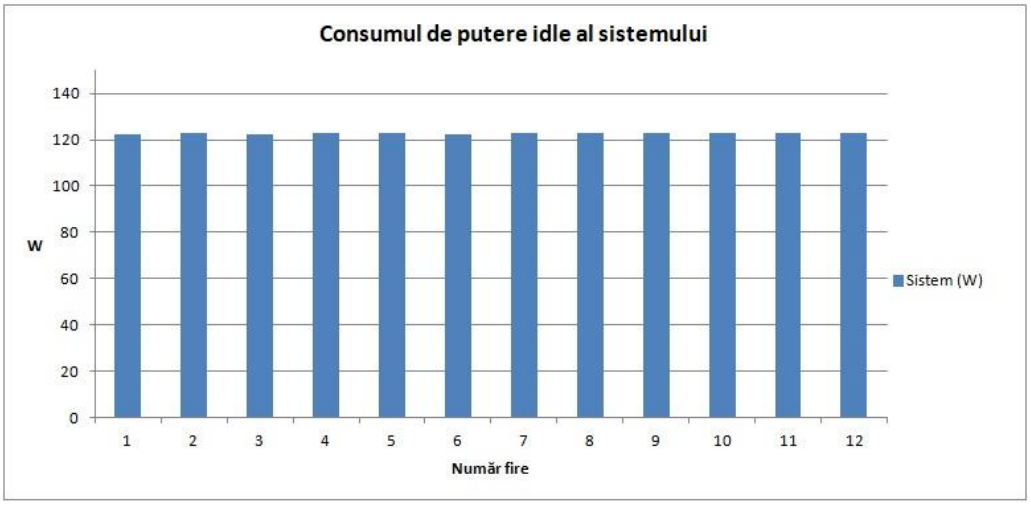

Figura 12. Valorile medii de consum de putere (W) al sistemului UMA în stare "în așteptare"

Fig. 13 prezintă consumul de putere al CPU-ului din sistemul UMA în stare "în așteptare" (fără nici o aplicație rulând). Fiecare valoare de pe abscisă corespunde mediei aritmetice a câte 900 de valori de consum putere a CPU-ului în stare "în așteptare" obținute câte una la fiecare secundă: valoarea 1 de pe abscisă corespunde mediei aritmetice a primelor 900 de valori de consum de putere al CPU-ului, valoarea 2 de pe abscisă corespunde mediei aritmetice a următoarelor 900 de valori de consum de putere al CPU-ului, ș.a.m.d.

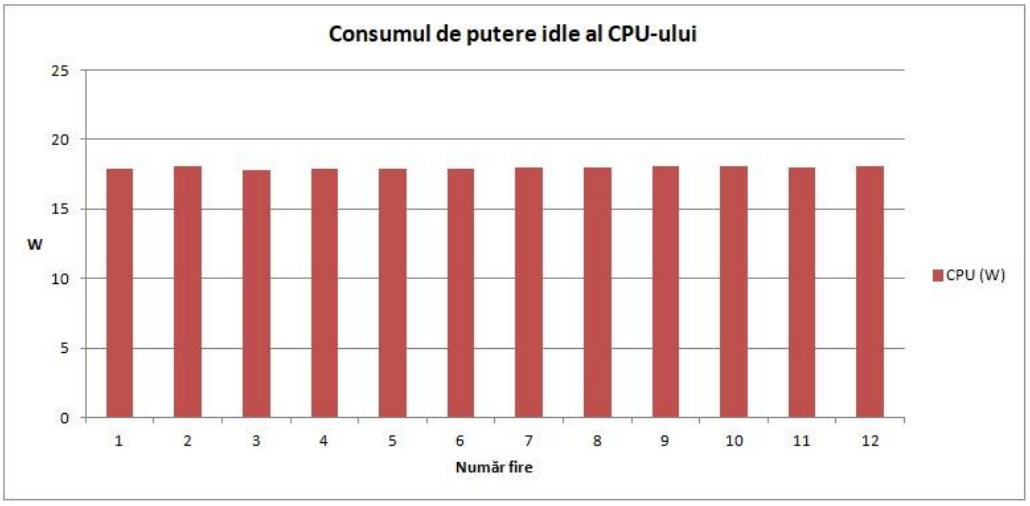

Figura 13. Valorile medii de consum de putere (W) ale CPU-ului din sistemul UMA în stare "în așteptare"

Bazat pe aceleași considerente ca în cele două figuri anterioare, Fig. 14 prezintă comparativ consumul de putere al CPU-ului și al întregului sistem în corelație.

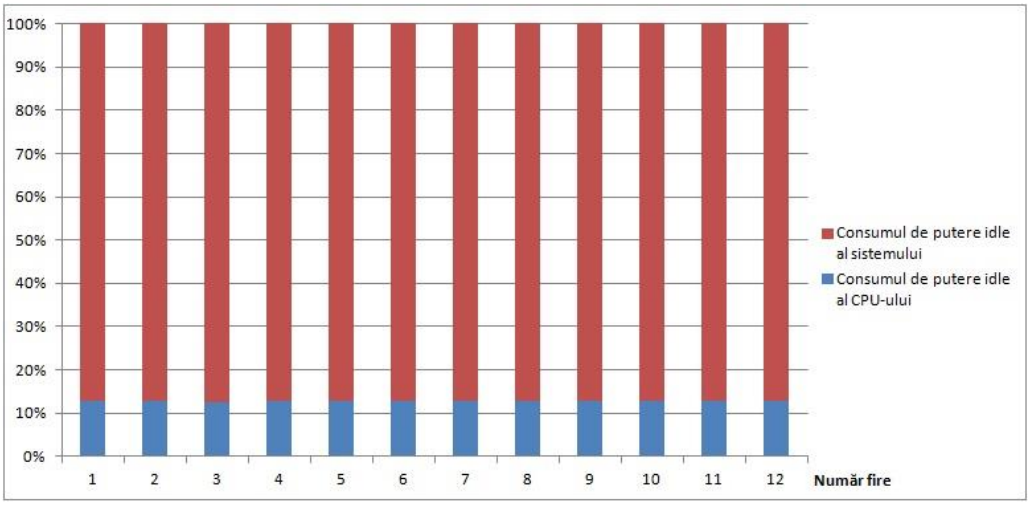

Figura 14. Consumul de putere (W) al CPU-ului din sistemul UMA și al întregului sistem în stare "în așteptare" corelate

Se observă – în Fig. 15 – că diferența între consumul de putere al întregului sistem UMA și cel al CPU-ului, ambele în stare "în așteptare", variază impredictibil în timp, dar cu diferențe foarte mici între valorile obținute. Și în această figură valorile de pe abscisă corespund câte unei diferențe dintre media aritmetice a 900 de valori de consum de putere al sistemului și media aritmetică a 900 de valori de consum de putere al CPU-ului (înregistrate concomitent).

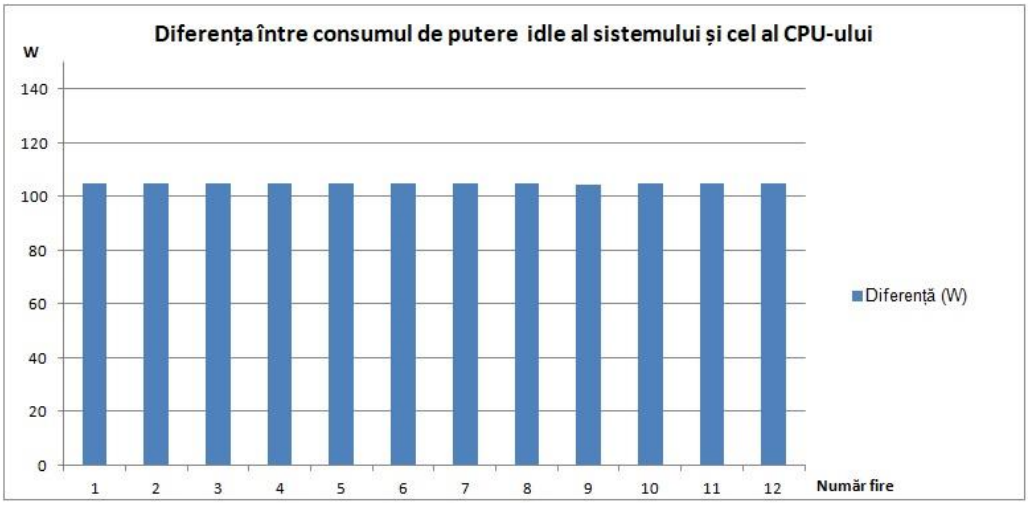

Figura 15. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului în stare "în așteptare"

*Consumul de putere al aplicației **CPU-X** obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atunci când aplicația rulează neoptimizat și atunci când rulează optimizat*

Tabelul 4 prezintă comparativ rezultatele experimentale pe sistem UMA ale aplicației **CPU-X** [89], pentru cazul când aceasta este optimizată folosind algoritmul ***NUMA-BTDM*** [55] și pentru cazul când nu este optimizată. Prima coloană indică numărul de fire cu care rulează aplicația, stabilit static. A doua coloană prezintă comparativ consumul de putere al aplicației **CPU-X** [89] optimizate și al aceleiași aplicații, neoptimizate. A treia coloană descrie optimizarea în W a aplicației **CPU-X** [89] rulată pe sistem UMA (obținută din diferența dintre consumul de putere la rularea aplicației neoptimizate și cel la înregistrat la rularea aplicației optimizate), iar ultima coloană ilustrează această optimizare în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Din Tabelul 4 se observă că pentru un număr mic de fire de execuție, procentul de optimizare al consumului de putere la aplicarea algoritmului ***NUMA-BTDM*** [55] variază între 58.7% și 80.2%.

Tabel 4. Rezultate experimentale pe sistem UMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației **CPU-X**

Număr fire	**Consum de putere (W) al aplicației CPU-X pe sistem UMA**		**Optimizarea obținută (W)**	**Procent de optimizare**
	Neoptimizată	**Optimizată**		
1	2.006	0.395	1.610	80.2
2	1.760	0.571	1.189	67.5
3	1.929	0.778	1.151	59.6
4	1.939	0.800	1.139	58.7
5	2.066	0.658	1.407	68.1
6	2.710	0.838	1.872	69.0
7	1.877	0.614	1.263	67.2
8	1.792	0.619	1.172	65.4
9	1.749	0.521	1.227	70.1
10	1.783	0.625	1.158	64.9
11	1.793	0.623	1.169	65.2
12	1.644	0.409	1.234	75.1

Se observă – în Fig. 16 – că pentru orice număr de fire de execuție cu care este lansată aplicația **CPU-X** [89], indicat pe abscisă, consumul de putere atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] este semnificativ mai mic decât atunci când aplicația nu este optimizată. Optimizarea maximă se înregistrează pentru 6 fire de execuție, adică o optimizare de 1.87 W/s, dintr-un total de 2.71 W.

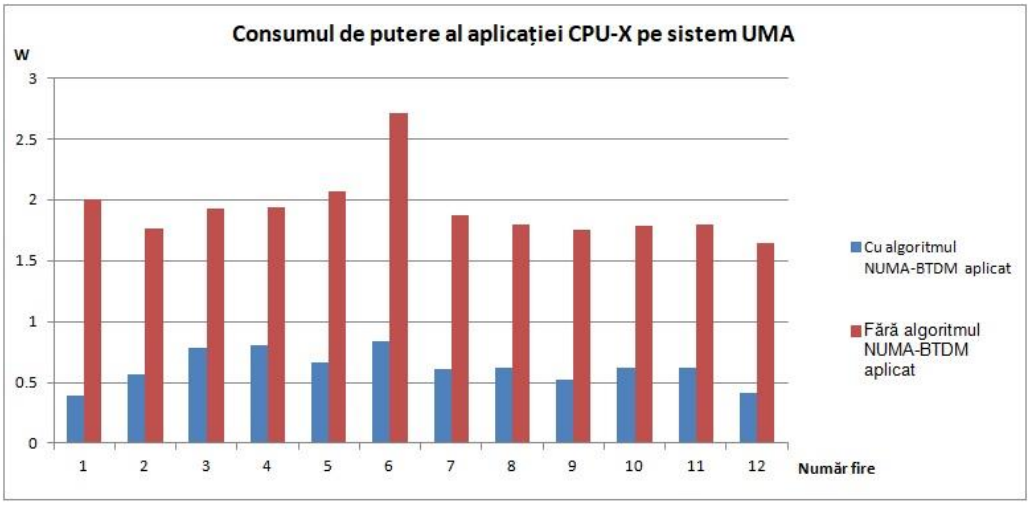

Figura 16. Consumul de putere (W) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție

Fig. 17 arată optimizarea de consum de putere la rularea aplicației **CPU-X** [89], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea este cuprinsă între valorile minimă de 1.13 W/s dintr-un total de 1.75 W/s pentru 4 fire de execuție și maximă de 1.87 W/s dintr-un total de 2.7 W/s pentru 6 fire de execuție.

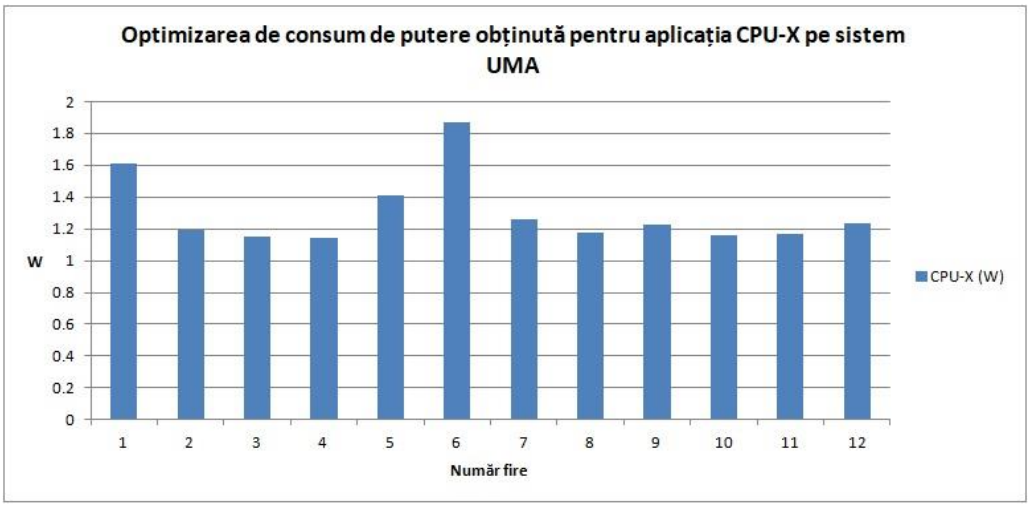

Figura 17. Optimizarea de consum de putere (W) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție

Fig. 18 arată optimizarea de consum de putere, în procente, la rularea aplicației **CPU-X** [89], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul maxim de optimizare de consum de putere este de 79% înregistrat pentru un singur fir de execuție. Procentul de optimizare mai mare în cazul unui fir de

execuție comparativ cu cazul mai multor fire de execuție, se datorează faptului că, deși firul de execuție nu comunică cu nici un alt fir, acesta fiind mapat pe un unic core, datele necesare execuției firului vor fi aduse din memorie doar de către acel core.

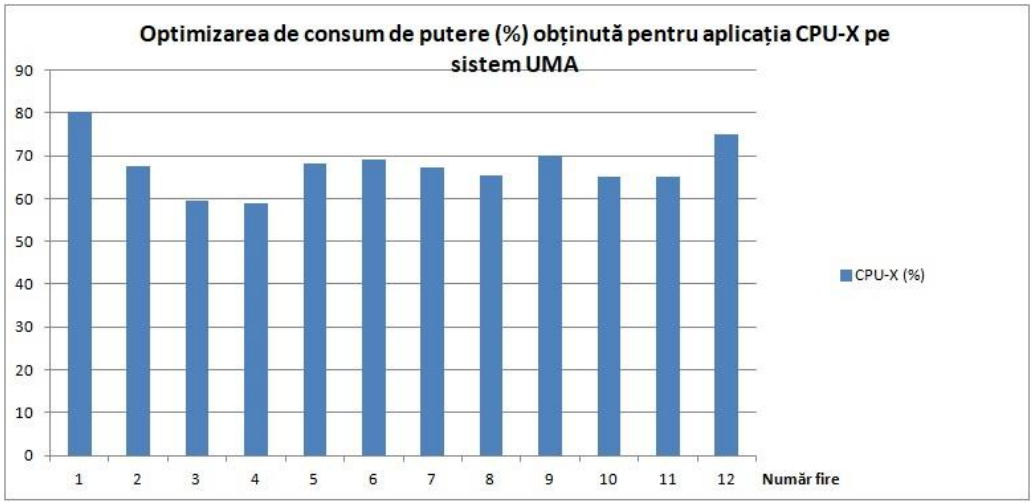

Figura 18. Optimizarea de consum de putere (%) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție

4.4.4 Rezultate experimentale pe sisteme NUMA

4.4.4.1 Timp de execuție

Nu au fost obținute rezultate experimentale. Aplicația **CPU-X** [89] rulează în buclă infinită.

4.4.4.2 Consum de putere

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când aplicația rulează neoptimizat și atunci când rulează optimizat*

În Fig. 19, abscisa reprezintă numărul de fire de execuție cu care este lansată aplicația **CPU-X** [89] iar ordonata reprezintă consumul de putere in W al întregului sistem când aplicația se execută. În figură se observă că aplicația **CPU-X** [89] consumă mai puțină putere la execuția pe sistem NUMA, atunci când este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], decât atunci când algoritmul nu este aplicat, observație valabilă pentru un număr de fire de execuție cuprins în intervalul [3,12].

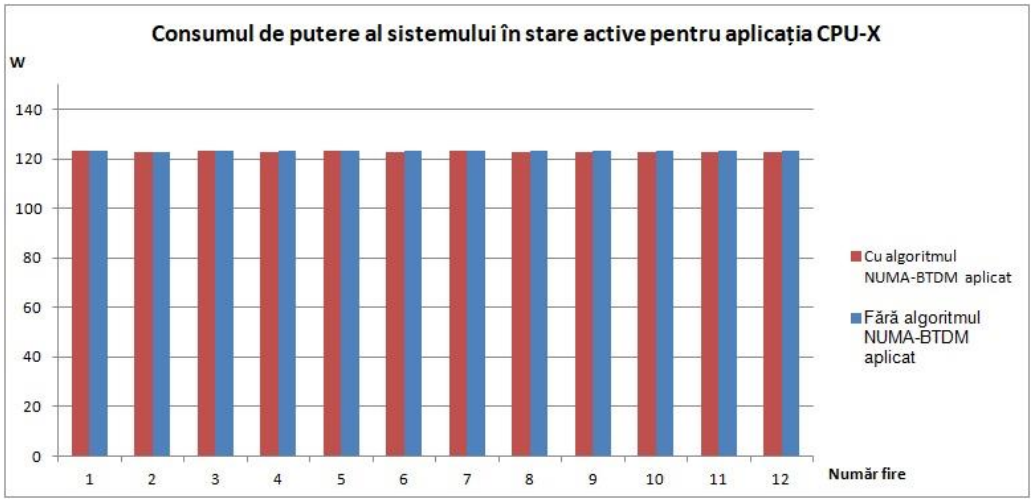

Figura 19. Consumul de putere (W) al sistemului NUMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

În Fig. 20, abscisa reprezintă numărul de fire de execuție cu care este lansată aplicația **CPU-X** [89] iar ordonata reprezintă consumul de putere in W al CPU-ului când aplicația se execută. În Fig. 20 se observă că, la execuția aplicației **CPU-X** [89] pe sistem NUMA, CPU-ul sistemului consumă mai puțină putere atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], decât atunci când aceasta nu este optimizată.

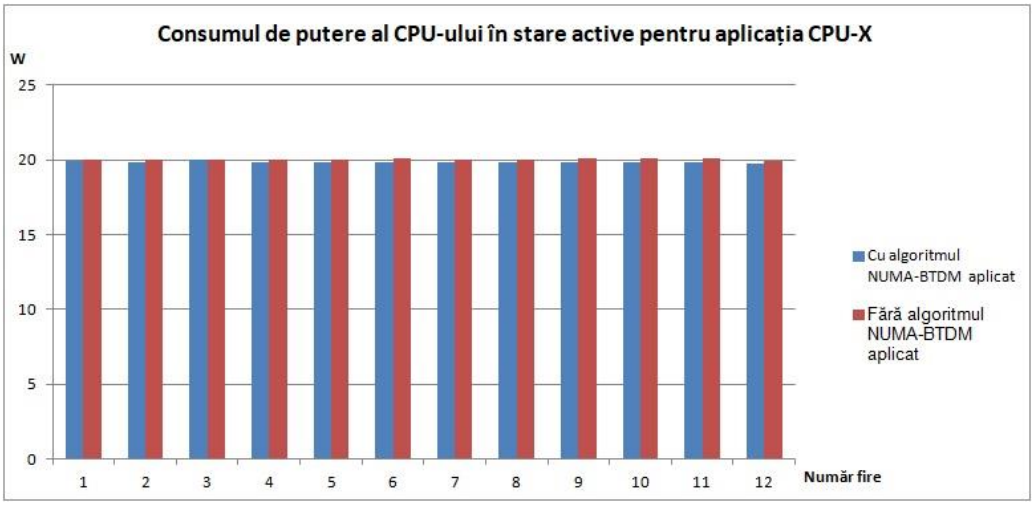

Figura 20. Consumul de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Fig. 21 ilustrează consumul de putere al CPU-ului și cel al întregului sistem NUMA pentru o comparație valorică, atât atunci când algoritmul ***NUMA-BTDM*** [55]

este aplicat cât și atunci când nu este aplicat. În figură, se observă că măsurătorile obținute pe sistem NUMA sunt corelate și consumul CPU-ului este semnificativ mai mic decât cel al întregului sistem pentru aplicația **CPU-X** [89].

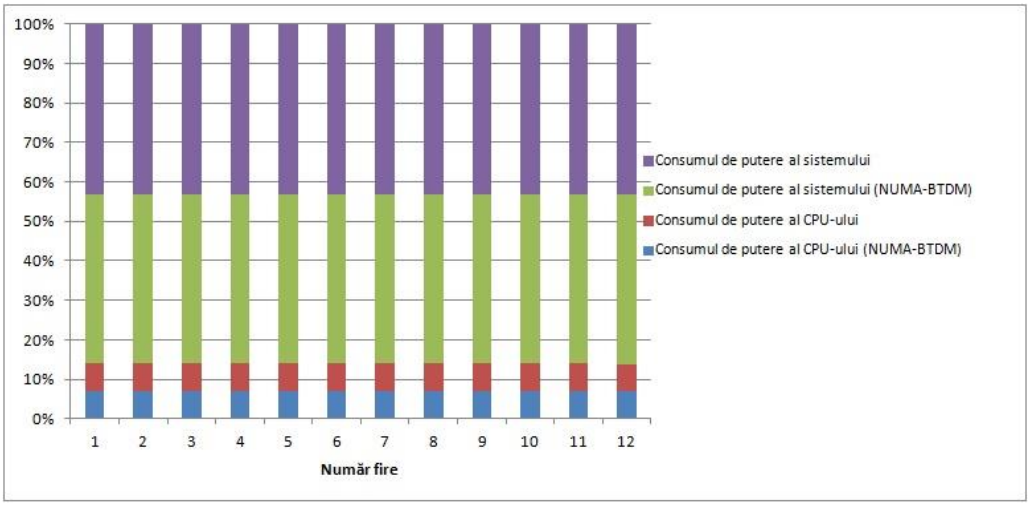

Figura 21. Consumul de putere (W) al CPU-ului și al întregului sistem NUMA pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Cu cât diferența între consumul de putere al întregului sistem NUMA și cel al CPU-ului este mai mare, cu atât execuția aplicației este mai optimă. Fig. 22 ilustrează această diferență pentru aplicația **CPU-X** [89] optimizată prin algoritmul ***NUMA-BTDM*** [55], cât și pentru aplicația **CPU-X** neoptimizată. Atunci când aplicația **CPU-X** [89] rulează cu mai multe de două fire de execuție, diferența în cazul în care algoritmul ***NUMA-BTDM*** [55] este aplicat este mare, ceea ce indică un consum scăzut de putere al aplicației optimizate. Acest lucru se poate vedea și din Figura 20.

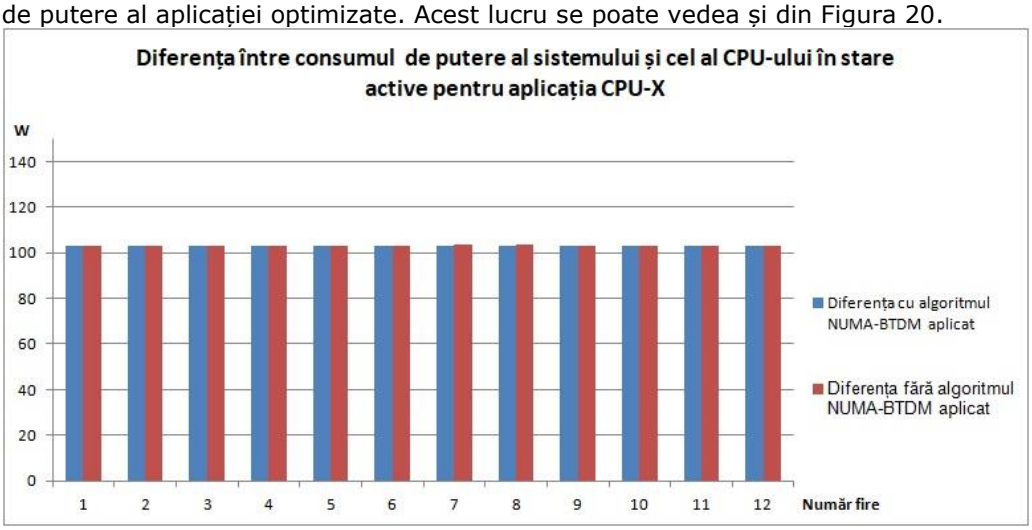

Figura 22. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului pe care rulează aplicația **CPU-X** cu diferite valori pentru numărul de fire de execuție

Fig. 23 indică faptul că, mediile a câte 900 de valori de consum de putere, obținute utilizând dispozitivul WattsUp în runda 1 de rularea pe sistem NUMA, câte o medie pentru fiecare executabil, sunt corelate cu cele obținute utilizând utilitarul ***turbostat***. Fiecare valoare medie este calculată ca fiind media aritmetică a 900 de valori obținute la fiecare 1 s timp de 900 s în care rulează fiecare executabil.

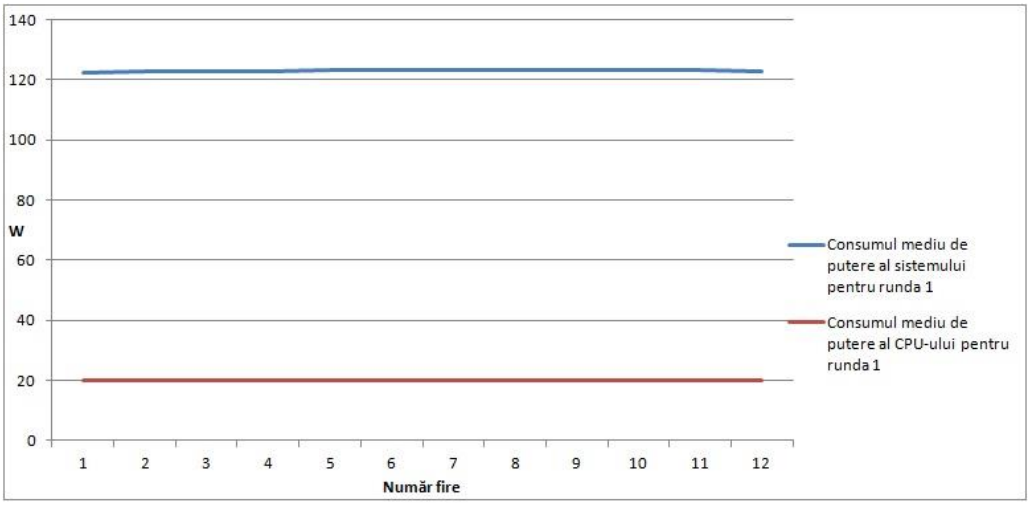

Figura 23. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **CPU-X** obținute în runda 1 pentru diferite valori ale numărului de fire de execuție

Figura 24 indică faptul că valorile de consum de putere obținute în runda 1 din cele două surse, WattsUp, respectiv ***turbostat***, pe sistem NUMA, sunt corelate. S-au întocmit grafice pentru toate rundele și a fost remarcată, în toate cazurile, corelarea măsurătorilor.

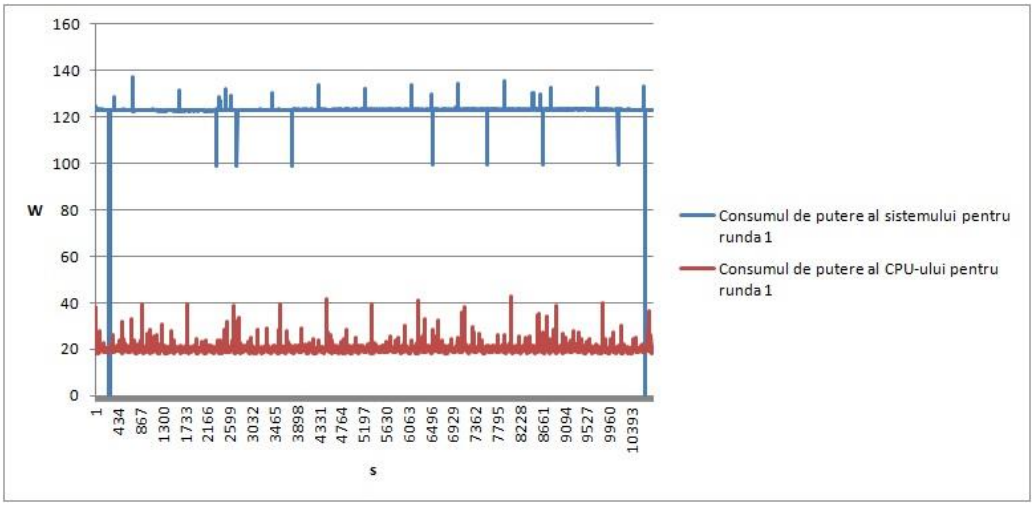

Figura 24. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **CPU-X** obținute în runda 1

Consumul de putere al sistemului comparativ cu cel al CPU-ului *în stare "în așteptare"*

Fig. 25 prezintă consumul de putere al sistemului NUMA în stare "în așteptare" (fără nici o aplicație rulând). Valorile de pe ordonată corespund mediei aritmetice a câte 900 de valori de consum putere a sistemului în stare "în așteptare" obținute câte una la fiecare o secundă: valoarea 1 de pe abscisă corespunde mediei aritmetice a primelor 900 de valori de consum de putere al sistemului, valoarea 2 de pe abscisă corespunde mediei aritmetice a următoarelor 900 de valori de consum de putere al sistemului, ș.a.m.d. Din Fig. 25 se observă că acest consum de putere este aproape constant în timp. Sistemul în stare "în așteptare" consumă ~122 W și măsurătorile obținute au varianța 7.35.

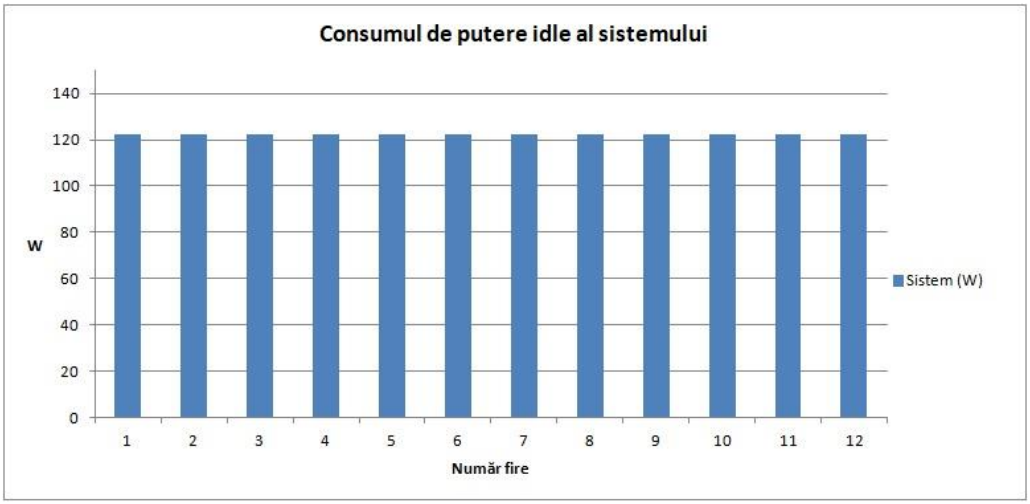

Figura 25. Valorile medii de consum de putere (W) al sistemului NUMA în stare "în așteptare"

Fig. 26 prezintă consumul de putere al CPU-ului din sistemul NUMA în stare "în așteptare" (fără nici o aplicație rulând). Fiecare valoare de pe abscisă corespunde mediei aritmetice a câte 900 de valori de consum putere a CPU-ului în stare "în așteptare" obținute la fiecare secundă: valoarea 1 de pe abscisă corespunde mediei aritmetice a primelor 900 de valori de consum de putere al CPU-ului, valoarea 2 de pe abscisă corespunde mediei aritmetice a următoarelor 900 de valori de consum de putere al CPU-ului, ș.a.m.d. La fel ca în Fig. 25, și în acest caz consumul de putere este aproape constant în timp.

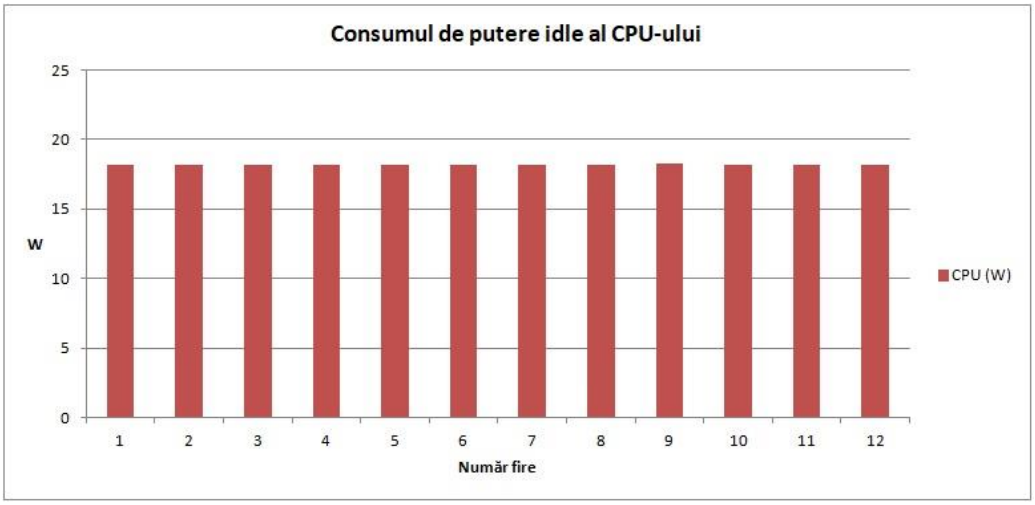

Figura 26. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA în stare "în așteptare"

Bazat pe aceleași considerente menționate în cazul celor două figuri anterioare, Fig. 27 prezintă comparativ consumul de putere al CPU-ului și al întregului sistem NUMA în corelație.

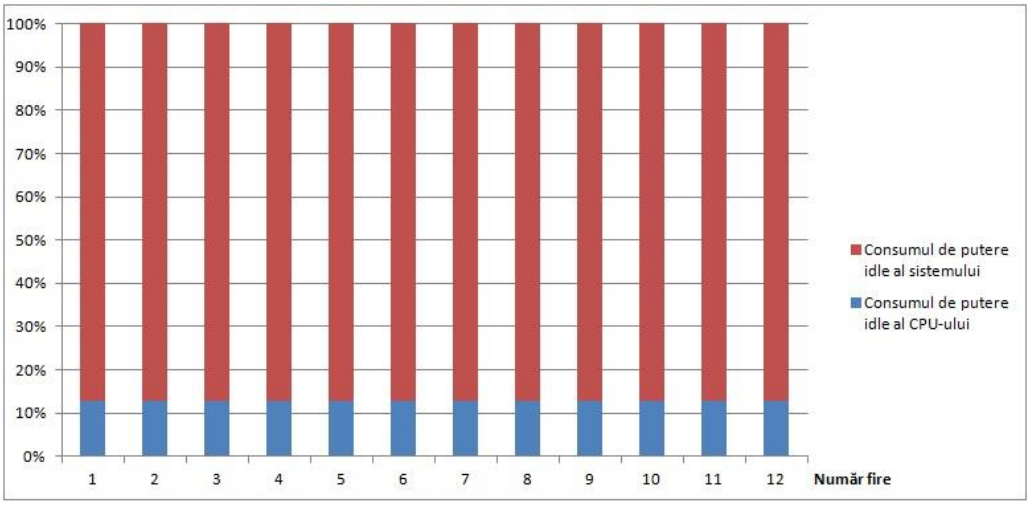

Figura 27. Consumul de putere (W) al CPU-ului din sistemul NUMA și al întregului sistem în stare "în așteptare" corelate

În Fig. 28 se observă că diferența între consumul de putere al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare", variază impredictibil în timp și deci, la fel și consumul CPU-ului și al sistemului. Asemenea figurilor anterioare, valorile de pe abscisă corespund câte unei medii aritmetice a 900 de valori de consum de putere.

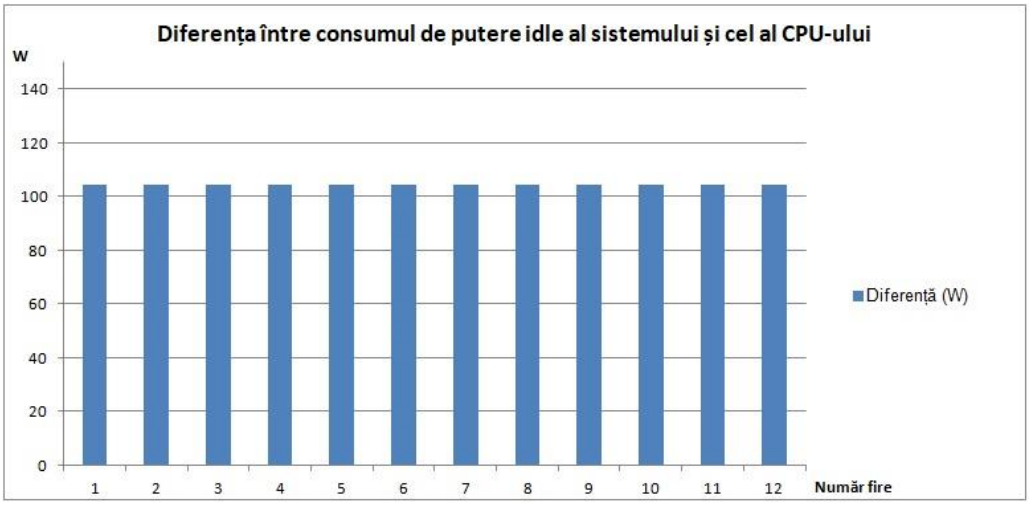

Figura 28. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare"

Consumul de putere al aplicației obținut din diferența între consumul de putere al CPU-ului *în stare activ și cel în stare "în așteptare", atunci când aplicația rulează neoptimizat și atunci când rulează optimizat, pe sistem NUMA*

Tabelul 5 prezintă comparativ rezultatele experimentale pe sistem NUMA ale aplicației **CPU-X** [89], pentru cazul când aceasta este optimizată folosind algoritmul ***NUMA-BTDM*** [55] și pentru cazul când nu este optimizată. Prima coloană indică numărul de fire de execuție ale aplicației, stabilit static. A doua coloană prezintă comparativ consumul de putere al aplicației **CPU-X** [89] optimizate și al aplicației **CPU-X** [89] neoptimizate. A treia coloană indică optimizarea în W, obținută pe sistem NUMA, la execuția aplicației **CPU-X** [89], ca fiind diferența dintre consumul de

Tabel 5. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației **CPU-X**

Număr fire	**Consum de putere (W) al aplicației CPU-X pe sistem NUMA**		**Optimizarea obținută (W)**	**Procent de optimizare**
	Neoptimizată	**Optimizată**		
1	1.801	1.706	0.094	5.24
2	1.846	1.720	0.126	6.84
3	1.815	1.807	0.007	0.41
4	1.866	1.680	0.186	9.96
5	1.868	1.666	0.201	10.79
6	1.870	1.617	0.252	13.49
7	1.916	1.711	0.204	10.66
8	1.842	1.653	0.189	10.29
9	1.887	1.592	0.295	15.62
10	1.879	1.619	0.259	13.82
11	1.891	1.666	0.224	11.88
12	1.720	1.553	0.166	9.68

putere la execuția aplicației **CPU-X** [89] neoptimizate și cel la obținut la execuția aplicației **CPU-X** [89] optimizate. Ultima coloană indică optimizarea anterior menționată, în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Procentul de optimizare la aplicarea algoritmului ***NUMA-BTDM*** [55] este cuprins în intervalul [0.41;15.62] pentru un număr de fire de execuție mai mic sau egal ca 12.

În Fig. 29 se observă că pentru orice număr de fire cu care este lansată în execuție aplicația **CPU-X** [89], indicat pe abscisă, consumul de putere al acesteia (indicat pe ordonată) atunci când este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] este mai mic decât atunci când aplicația nu este optimizată. Aplicația este optimizată cu până la 0.29 W/s, dintr-un total de 1.88 W/s, optimizare înregistrată pentru 9 fire de execuție.

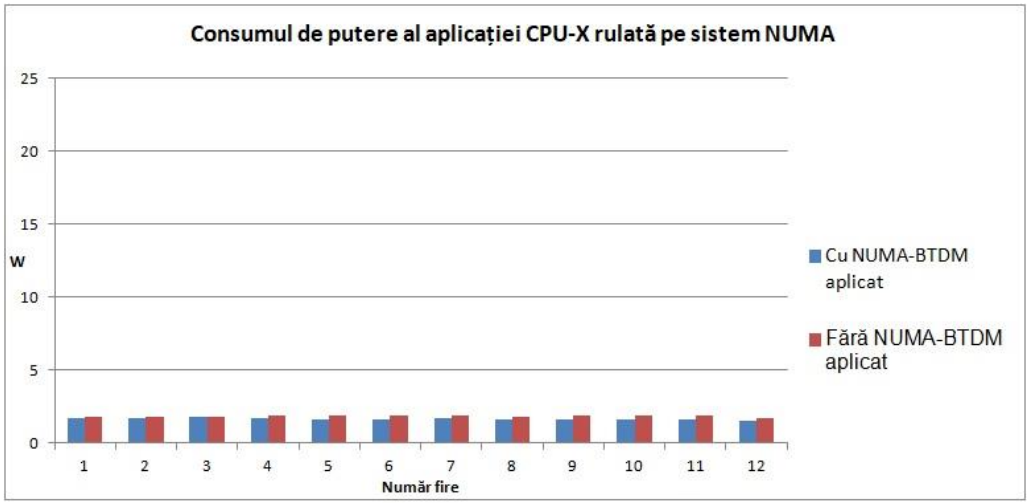

Figura 29. Consumul de putere (W) al aplicației **CPU-X** care rulează pe sistem NUMA cu diferite valori, stabilite static, pentru numărul de fire de execuție

Fig. 30 arată optimizarea de consum de putere la rularea aplicației **CPU-X** [89] pe sistem NUMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea este cuprinsă între valorile minimă de 0.0076 W înregistrată pentru pentru 3 fire de execuție și maximă de 0.2950 W înregistrată pentru 9 fire de execuție. Valorile de optimizare sunt raportate la 1 s.

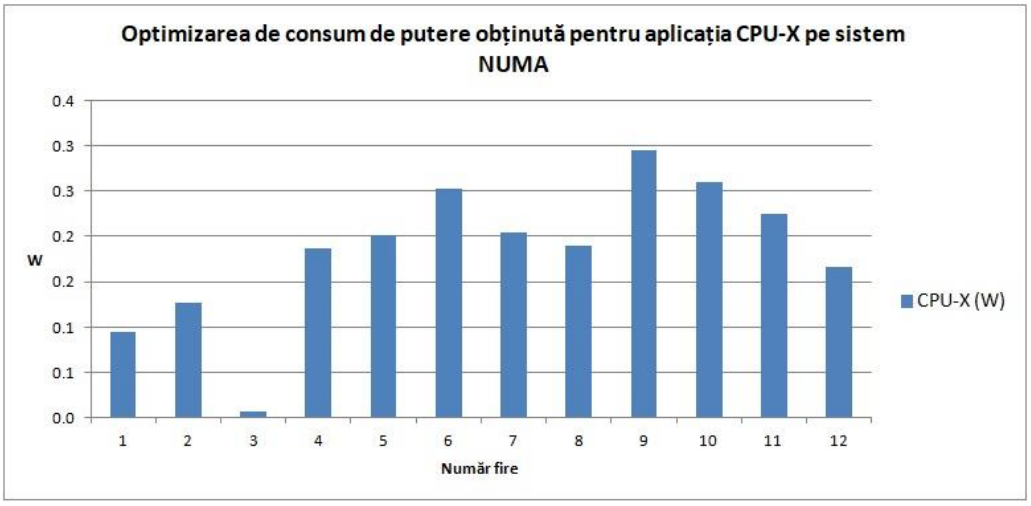

Figura 30. Optimizarea de consum de putere (W) al aplicației **CPU-X** care rulează pe sistem NUMA cu diferite valori, stabilite static, pentru numărul de fire de execuție

Fig. 31 arată optimizarea de consum de putere, în procente, la rularea aplicației **CPU-X** [89] pe sistem NUMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul de optimizare de consum de putere variază între valorile 0.41% pentru 3 fire de execuție și 15.62% înregistrat pentru 9 fire de execuție. În figură se observă că procentul de optimizare crește neliniar și aleator odată cu creșterea numărului de fire de execuție.

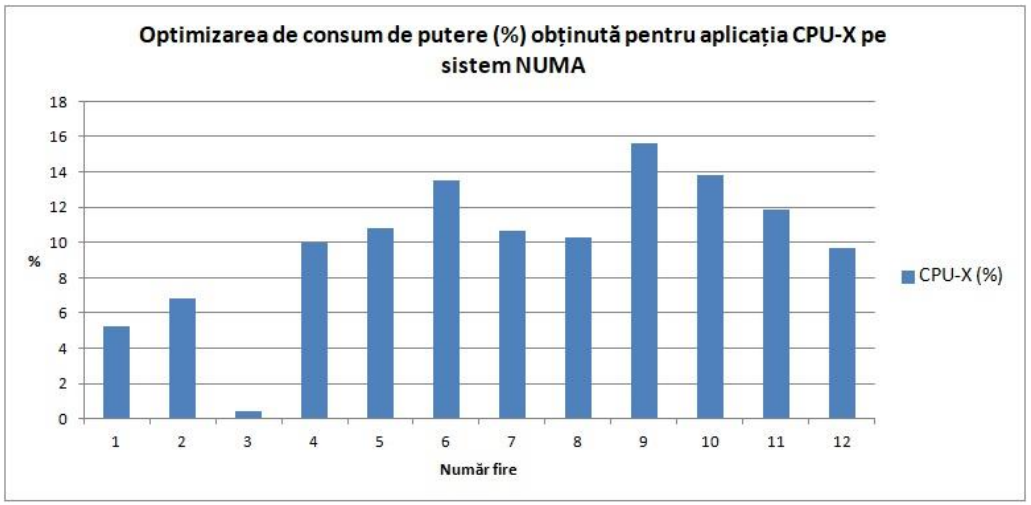

Figura 31. Optimizarea de consum de putere (%) al aplicației **CPU-X** care rulează cu diferite valori pentru numărul de fire de execuție

4.4.5 Comparație între rezultatele experimentale pe sisteme UMA și cele pe sisteme NUMA

4.4.5.1 Comparație pentru timp de execuție

Nu au fost obținute rezultate experimentale. Aplicația **CPU-X** [89] rulează în buclă infinită.

4.4.5.2 Comparație pentru consum de putere

Tabelul 6 prezintă comparativ rezultatele experimentale, pe sistem UMA și sistem NUMA, pentru aplicația **CPU-X** [89]. Prima coloană indică numărul de fire cu care rulează aplicația, stabilit static. A doua coloană descrie optimizarea în W, obținută la execuția aplicației **CPU-X** [89] pe sistem UMA și respectiv, pe sistem NUMA. Ultima coloană indică optimizarea în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Prin consultarea procentelor de optimizare din tabel, se observă că aplicația **CPU-X** [89] consumă mai puțină putere pe sistem UMA decât pe NUMA, pentru un număr de fire de execuție mai mic sau egal ca 12, datorită lățimii de bandă mai mare la UMA. Astfel, magistrala către memoria principală nu devine congestionată la UMA, fiind un număr mic de fire care utilizează date în comun.

Tabel 6. Rezultate experimentale pe sistem UMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației **CPU-X**

Număr fire	**Optimizarea obținută (W)**		**Procent de optimizare**	
	UMA	**NUMA**	**UMA**	**NUMA**
1	1.610	0.094	80.2	5.24
2	1.189	0.126	67.5	6.84
3	1.151	0.007	59.6	0.41
4	1.139	0.186	58.7	9.96
5	1.407	0.201	68.1	10.79
6	1.872	0.252	69.0	13.49
7	1.263	0.204	67.2	10.66
8	1.172	0.189	65.4	10.29
9	1.227	0.295	70.1	15.62
10	1.158	0.259	64.9	13.82
11	1.169	0.224	65.2	11.88
12	1.234	0.166	75.1	9.68

Fig. 32 arată optimizarea de consum de putere, în procente, la rularea aplicației **CPU-X** [89], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55], prezentată comparativ pentru sistemele UMA și NUMA. Procentul maxim de optimizare de consum de putere este 80.26% obținut pentru un fir de execuție la sistem UMA și 15.62% obținut pentru 9 fire de execuție la sistem NUMA, iar procentul mediu este 67.63% pentru sistem UMA și 9.89% pentru sistem NUMA.

Prin urmare, după aplicarea algoritmului ***NUMA-BTDM*** [55], această aplicație rulează în medie cu până la 57.74% mai optim pe sistem UMA decât pe sistem NUMA, din punct de vedere al consumului de energie.

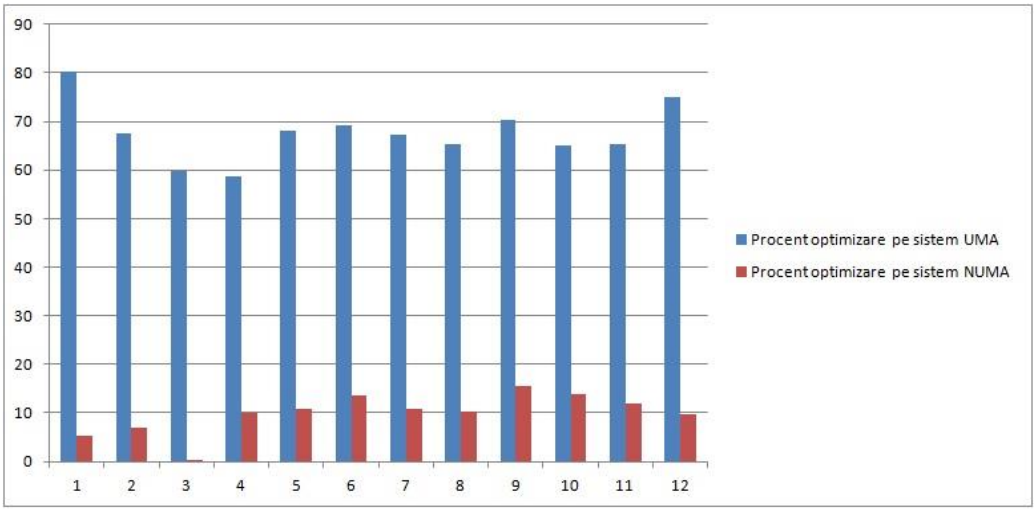

Figura 32. Optimizarea de consum de putere (%) al aplicației **CPU-X** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA

Tabelul 7 descrie comparativ pentru sistemul UMA și sistemul NUMA, rezultatele experimentale obținute din cele două surse, utilitarul software tubostat și dispozitivul hardware specializat WattsUp, împărțite pe categorii: minime, medii, maxime, varianța, deviație standard pentru fiecare din executabilele aplicației **CPU-X** [89], care diferă prin numărul de fire de execuție cu care sunt rulate. Valorile minimă, medie, maximă, valorile pentru varianța și respectiv pentru deviație standard au fost obținute fiecare din ambele surse de măsurare, pentru fiecare rundă de execuție a fiecărui executabil în parte. De exemplu: valoarea minimă din cele 900 de valori obținute în 900 s din sursa ***turbostat***, pentru executabilul care rulează în runda 1 utilizând 3 fire de execuție, este 18.2 W.

4.4.6 Concluzii ale rezultatelor experimentale pentru aplicația de referință CPU-X

În urma aplicării algoritmului ***NUMA-BTDM*** [55], rezultatele experimentale indică o optimizare în medie cu 0.28 W/s, a consumului de putere al sistemului NUMA pe care se execută aplicația **CPU-X** [89] 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 mediu al aplicației **CPU-X** [89] neoptimizată, consumul de putere mediu al aplicației **CPU-X** [89] optimizată prin algoritmul ***NUMA-BTDM*** [55]. Varianța optimizării se obține scăzând din varianța medie a măsurătorilor neoptimizate, varianța medie a măsurătorilor optimizate și are valoarea ~0.06 pentru sistemul NUMA și 0.13 pentru sistemul UMA. Varianța optimizării fiind mai mică decât optimizarea, se poate concluziona că există o optimizare, care 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 utilizat în comun de firele de execuție alăturate, mapate de către algoritmul ***NUMA-BTDM*** [55] 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.

Procentul de optimizare al aplicației **CPU-X** [89] este mai mare la sistem UMA decât la sistem NUMA datorită lățimii de bandă mai mare a sistemului UMA. Magistrala către memoria principală nu devine congestionată la sistemul UMA, datorită numărului mic de fire de execuție alăturate care utilizează aceleași date.

Tabel 7. Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMA pentru aplicația **CPU-X**

Categorie valori	**Tip sistem**	**Comparație între măsurătorile obținute din cele două surse, cu și fără aplicarea algoritmului NUMA-BTDM**	
		Utilitarul *turbostat*	**Dispozitivul WattsUp**
Minime	NUMA	Media valorilor minime obținute este 17.67 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat. Valorile minime sunt mai mare atunci când algoritmul nu este aplicat, având media 17.89 W.	Minimum când aplicația este optimizată este 98.8 W, înregistrat pentru 4 fire de execuție. Minimum când aplicația nu este optimizată este tot 98.8 W, înregistrat pentru 3 fire de execuție.
	UMA	Media valorilor minime obținute este 16.92 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat. Valorile minime sunt mai mici atunci când algoritmul nu este aplicat, având media 17.65 W.	Minimul atunci când algoritmul este aplicat este 98.8 W (înregistrat pentru 4 fire de execuție), mai mic decât minimul 99.2 W (înregistrat pentru 4 fire de execuție) obținut atunci când algoritmul nu este aplicat.
Medii	NUMA	Media valorilor medii obținute pentru cele 12 executabile atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este 19.84 W, fiind nesemnificativ mai mică decât media 20.02 W obținută atunci când algoritmul nu este aplicat.	Media valorilor medii pentru cele 12 executabile obținute atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este 122.94 W, fiind nesemnificativ mai mică decât media 123.22 W, obținută atunci când algoritmul nu este aplicat.
	UMA	Media valorilor medii pentru cele 12 executabile obținute atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este 18.57 W, mai mică decât media 19.87 W, obținută atunci când algoritmul nu este aplicat.	Media valorilor medii pentru cele 12 executabile când algoritmul ***NUMA-BTDM*** [55] este aplicat, este 123.01 W, fiind nesemnificativ mai mică decât media 123.35 W, obținută atunci când algoritmul nu este aplicat.

Maxime	NUMA	Media valorilor maxime obținute atunci când algoritmul ***NUMA-BTDM*** [55] a fost aplicat este 45.31 W (deviație de la medie în cazul 6 fire de execuție, înregistrându-se 55.05 W). Media valorilor maxime este mai mare atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat (49.12 W).	Media valorilor maxime atunci când algoritmul este aplicat este 135.88 W, fiind mai mică decât atunci când algoritmul nu este aplicat, când media este 136.44 W, înregistrându-se în acest caz valoarea cea mai mare dintre maxime de 141.4 W pentru 8 fire de execuție.
	UMA	Media valorilor maxime obținute atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 47.19 W, mai mare cu aprox. 2 W decât cea obținută la sistem NUMA (pentru 10 fire de execuție se înregistrează cea mai mare valoare dintre maxime, 53.03 W). Media valorilor maxime obținute atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, este 49.54 W, mai mari decât media obținută atunci când algoritmul este aplicat (cea mai mare valoare dintre maxime este 55.76 W, obținută pentru 6 fire de execuție).	Media valorilor maxime atunci când algoritmul este aplicat este 136.59 W, fiind mai mică decât atunci când algoritmul nu este aplicat, când media este 137.41 W si cea mai mare valoare dintre maxime este 144.5 W, fiind obținută pentru 6 fire de execuție. Comparativ cu valorile obținute pentru sistem NUMA, media valorilor maxime este cu aprox. 1 W mai mare la UMA atunci când algoritmul este aplicat și aprox. 2 W mai mare la UMA când algoritmul nu este aplicat.

Varianță	NUMA	Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, varianța medie pentru toate rundele de execuție a tuturor executabilelor, este 3.92 (valoarea maximă 19.74 este înregistrată pentru 3 fire de execuție). Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, valorile maxime pentru varianță, sunt 8.77, 6.06, respectiv 7.23, pentru 6, 7, respectiv 9 fire de execuție, varianța medie fiind 2.	Varianța medie atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 0.77, iar valoarea maximă este 2.13, obținută pentru 4 fire de execuție. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.71 și varianța maximă este 3.1, obținută pentru 9 fire de execuție.
	UMA	Varianța medie atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 3.12 (valoarea maximă 26.98 este înregistrată pentru 6 fire de execuție, în runda 5). Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 4.57 (valoarea maximă a variației 81.35 este înregistrată pentru 6 fire de execuție în runda 3), mai mare decât atunci când algoritmul este aplicat, deci măsurătorile sunt mai apropiate de medie când algoritmul ***NUMA-BTDM*** [55] este aplicat. Măsurătorile obținute atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat sunt mai dispersate față de medie la sistem NUMA decât la UMA, iar atunci când algoritmul nu este aplicat, se obține opusul.	Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, valoarea maximă pentru varianță 2.26 este înregistrată pentru 2 fire de execuție, varianța medie fiind 0.53. Varianța medie atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 0.66 iar valoarea maximă 2.56, înregistrată tot pentru 5 fire de execuție.

Deviație standard	NUMA	Media deviațiilor standard obținute atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este 0.3 W. Valorile maxime ale deviației standard obținută pentru fiecare executabil neoptimizate sunt 2.96 W, respectiv 2.68 W, înregistrate în runda 4 pentru 6, respectiv 9 fire de execuție.	Valorile maxime pentru deviația standard atunci când algoritmul ***NUMA-BTDM*** [55] a fost aplicat sunt 1.46 W, obținuți pentru 4 fire de execuție iar valoarea medie a deviației standard este 0.82 W. Media valorilor obținute atunci când nu este aplicat algoritmul ***NUMA-BTDM*** [55] este 0.76 W, valoarea maximă pentru deviația standard fiind 1.76 W, pentru a 9 fire de execuție.
	UMA	Media deviațiilor standard atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este de 1.66 W, deviația standard maximă fiind de 5.19 W obținută pentru 6 fire de execuție în runda 5. Deviația standard atunci când algoritmul nu este aplicat are valoarea maximă 9.01 W obținută în runda 3 pentru 6 fire de execuție, deviația standard medie fiind 1.8 W.	Valoarea maximă pentru deviația standard atunci când algoritmul ***NUMA-BTDM*** [55] a fost aplicat este 1.5 W obținută pentru 2 fire de execuție, iar valoarea medie a deviației standard este 1.71 W. Media valorilor obținute atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 0.75 W, valoarea maximă pentru deviația standard fiind 1.6, înregistrate pentru 5 fire de execuție.

Conform aceluiași tip de raționament ca cel din paragraful anterior și bazat pe datele din Tabelul 7, rezultă că, consumul CPU-ului la execuția aplicației **CPU-X**, nu este optimizat (optimizare 0.18 W cu o variantă a optimizării de 1.92), datorită degradării de performanță produsă de numărul mare de tranziții activ-”în așteptare” și invers, firele fiind toate mapate pe același core. Rezultă însă în final o optimizarea a consumului de putere al întregului sistem, datorită optimizării operațiilor cu memoria, optimizare care este mai mare decât degradarea datorată numărului mare de tranziții activ-”în așteptare”.

Conform Tabelului 5, 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.

4.5 Aplicația de referință CPU

4.5.1 Descrierea aplicației

Aplicația de referință **CPU** [90] este compusă din aplicațiile **CPU** [90], **Flops** [90] și **Iops** [90].

Aplicația de referință **CPU** [90] creează firele de execuție în două funcții, *calFlops* și *calIops*, care sunt apelate din funcția principală. Funcția *calFlops* calculează numărul de operații în virgulă flotantă per secundă dintr-un total de 33 de instrucțiuni, executate de fiecare fir de execuție în parte, numărul posibil de fire

fiind 1, 2 sau 4 fire de execuție. Similar, funcția *calIops* calculează numărul de operații cu întregi per secundă.

Părintele firelor de execuție create de cele două funcții, *calFlops* și *calIops*, este firul principal și între firele create de firul principal nu există nici o dependență de date, prin urmare firele de execuție sunt marcate ca autonome. Atât în funcția *calFlops* cât și în funcția *calIops*, sunt create grupuri de fire de execuție care conțin fiecare câte 1,2, respectiv 4 fire de execuție și care sunt distribuite cât mai uniform pe core-uri de către algoritmul ***NUMA-BTDM*** [55] astfel: firul din grupul cu 1 fir de execuție este mapat pe core-ul 0, firele din grupul cu 2 fire de execuție sunt mapate primul pe core 0, iar cel de-al doilea pe core-ul 6, firele din grupul de 4 fire de execuție sunt mapate primul pe core-ul 0, cel de-al doilea pe core-ul 3, cel de-al treilea pe core-ul 6, iar cel de-al patrulea pe core-ul 9. Cele trei grupuri de câte 1, 2, respectiv 4 fire de execuție sunt create concomitent, pe același nivel în ierarhia de generare a firelor de execuție.

Aplicația **Flops** [90] rulează timp de aproximativ 10 minute, timp în care creează de 600 de ori, câte un grup cu 4 fire de execuție care sunt mapate de algoritmul ***NUMA-BTDM*** [55] la fel ca grupul de 4 fire create de aplicația **CPU** [90]. Aplicația calculează câte operații în virgulă flotantă sunt realizate în medie în 1 s, timp de 600 s.

Aplicația **Iops** [90] realizează aceleași calcule ca și **Flops** [90], dar pentru operații cu întregi.

4.5.2 Rezultate experimentale pe sisteme UMA

4.5.2.1 Timp de execuție

Tabelul 8 prezintă valorile medii ale rezultatelor experimentale de timp de execuție, pe sistem UMA, pentru aplicația de referință **CPU** [90], formată din aplicațiile independente **CPU** [90], **Flops** [90], **Iops** [90], atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când nu este aplicat la compilare. Valorile medii pentru fiecare din cele 3 aplicații, sunt obținute calculând media aritmetică a 40 de timpi de execuție.

Din Tabelul 8, se observă că, pentru aplicația **CPU** [90], timpul de execuție scade în medie cu 0.0028 s, adică 0.47%. Procentul mic de optimizare se datorează faptului că mai multe fire autonome au fost mapate pe același core-uri, deși resursele hardware disponibile ar fi fost suficiente ca aceste fire să fie mapate fiecare pe core-uri diferite, ceea ce ar fi condus la scăderea numărului de treceri ale firelor din stare activă în stare "în așteptare" și invers (fiind mai puține fire asignate per core).

Pentru aplicațiile **Flops** [90] și **Iops** [90] nu se înregistrează o optimizare a timpului de execuție. Acest lucru se datorează timpului scurs cu execuția apelurilor de setare a afinității CPU, pentru un număr mare de fire de execuție (2400 în cazul fiecărei aplicații). Astfel, timpul scurs cu execuția apelurilor de setare a afinității CPU este mai mare decât optimizarea produsă de algoritmul ***NUMA-BTDM*** [55]. Atunci când numărul firelor de execuție este mare și calculele realizate de fiecare fir sunt reduse, timpul de execuție al unui fir este mic și este logic ca orice operație de mapare realizată asupra firului la execuție, cum este cea de setare a afinității CPU la execuție, să lungească timpul de execuție al firului cu o durată de timp

comparativă chiar cu cea a execuției sale. În general, atunci când timpul de execuție al firului este foarte mic, setarea afinității CPU la execuția acestuia poate să dubleze sau chiar să tripleze timpul de execuție.

Tabel 8. Rezultate experimentale de timp de execuție al aplicație **CPU** (formată din aplicațiile **CPU**, **Flops**, **Iops**) rulată pe sistem UMA, prezentate comparativ atunci când algoritmul ***NUMA-BTDM*** este aplicat și atunci când nu este aplicat

Aplicație	**Timp de execuție mediu (W)**	
	Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**
CPU	0.5868	0.5840
Flops	600.4846	600.4890
Iops	600.4706	600.4740

4.5.2.2 Consum de putere

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când aplicația* ***CPU*** *rulează neoptimizat și atunci când rulează optimizat*

Tabelul 9 prezintă comparativ rezultatele experimentale de consum de putere pentru aplicația **CPU** [90] atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] și atunci când aceasta nu este optimizată, rezultate obținute din cele două surse, utilitarul ***turbostat*** (consumul de putere al CPU-ului) și dispozitivul specializat WattsUp (consumul de putere al întregului sistem NUMA). Rezultatele arată o scădere în medie a consumului de energie cu aproximativ 10 W atunci când aplicația **CPU** [90] este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], scădere înregistrată păstrându-se același timp de execuție. Dispersia măsurătorilor de consum de putere față de valoarea medie este mai mică atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat: în cazul măsurătorilor obținute cu ajutorul utilitarului ***turbostat***, varianța 2.16 atunci când optimizarea este aplicată este mai mare decât varianța 7.35 atunci când optimizarea nu este aplicată, iar în cazul măsurătorilor obținute cu ajutorul dispozitivului WattsUp, varianța 15.02 atunci când optimizarea este aplicată este mai mică decât varianța 32.48 atunci când optimizarea nu este aplicată.

Tabel 9. Comparație între rezultatele experimentale pentru aplicația **CPU**, obținute din cele două surse, utilitarul ***turbostat***, respectiv dispozitivul WattsUp, atât când aceasta nu este optimizată, cât și când este optimizată

Categorie valori	**Sursă valori**	**Comparație între măsurătorile obținute din cele două surse, cu și fără aplicarea algoritmului NUMA-BTDM**	
		Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**
Minime	***turbostat***	54.25	46.44
	WattsUp	145.1	136.1
Medii	***turbostat***	57.14	47.85
	WattsUp	148.48	140.92
Maxime	***turbostat***	65.1	53.59
	WattsUp	163.3	157.1
Varianță	***turbostat***	7.35	2.16
	WattsUp	32.48	15.02
Deviație standard	***turbostat***	2.71	1.47
	WattsUp	5.69	3.87

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când fiecare din aplicațiile* ***Flops*** *și* ***Iops*** *rulează fără a fi optimizate și atunci când rulează după ce au fost optimizate*

În Fig. 33, abscisa reprezintă numărul rundei de rulare a celor două aplicații **Flops** [90] și **Iops** [90], iar ordonata reprezintă consumul de putere in W al întregului sistem la rularea pe rând a celor două aplicații. Din figură se observă că, în majoritatea rundelor, sistemul UMA cu aplicația **Flops** [90] rulând consumă mai multă putere pe sistem UMA atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat. Degradarea de consum de putere al sistemului UMA din majoritatea rundelor, se datorează unui dezavantaj pe care algoritmii ***NUMA-BTLP*** [87] și ***NUMA-BTDM*** [55] îl prezintă și anume acela că algoritmii nu țin cont în realizarea mapării de numărul total de fire de execuție create de o buclă deoarece acest număr depinde de numărul de iterații, număr care se consideră că nu poate fi determinat static (chiar dacă în unele cazuri acest lucru este posibil). Dacă algoritmii ar fi ținut cont de numărul de iterații, ar fi folosit toate core-urile pentru a mapa cât mai uniform firele de execuție, ținând cont că numărul firelor de execuție (2400) depășește cu mult numărul core-urilor logice (24). În realitate, algoritmii mapează uniform cele 4 fire de execuție din corpul buclei for cu 600 de iterații, ca și cum ar fi doar 4 fire de execuție în total. Astfel, rezultă mai multe fire de execuție mapate pe același core, în loc ca acestea să fie distribuite uniform pe mai multe core-uri. Acest lucru determină creșterea numărului de tranziții din stare "în așteptare" în stare activă a core-urilor și invers, ceea ce degradează timpul de execuție și consumul de putere pe sistem UMA. Aceeași observație este valabilă și pentru aplicația **Iops** [90], dar, în acest caz, consumul de putere al sistemului UMA este degradat în mai puține runde decât la **Flops** [90]. În medie, luând în considerare toate rundele, consumul de putere al sistemului UMA la rularea aplicației **Flops** [90] este cu 0.7 W mai mare atunci când este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], iar în cazul aplicației **Iops** [90], consumul de putere al sistemului UMA este în medie cu 0.38 W mai mult atunci când aceasta este optimizată.

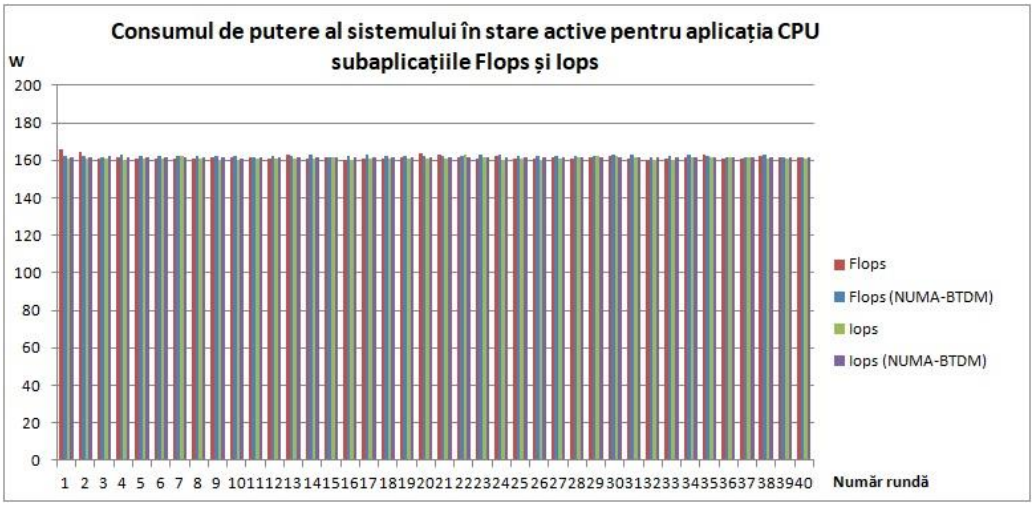

Figura 33. Consumul de putere (W) al sistemului UMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde

Fig. 34 descrie consumul de putere al CPU-ului din sistemul UMA. În figură, abscisa are aceeași semnificație ca în Fig. 33 iar ordonata reprezintă consumul putere al CPU-ului exprimat in W. Se observă că în jumătate din runde, CPU-ul din sistemul UMA cu aplicația **Flops** [90] rulând, prezintă un consum de putere mai redus atunci când aplicația este optimizată, consumul de putere al CPU-ului din sistemul UMA fiind în medie mai mic (luând în considerare toate rundele) cu 0.13 W atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], decât atunci când algoritmul nu este aplicat. Aceleași observații sunt valabile și pentru aplicația **Iops** [90], numărul rundelor în care aplicația optimizată, utilizând ***NUMA-BTDM*** [55], se execută cu consum de putere al CPU-ului mai redus, fiind mai mic decât la **Flops** [90], optimizarea medie de consum de putere fiind tot de 0.13 W.

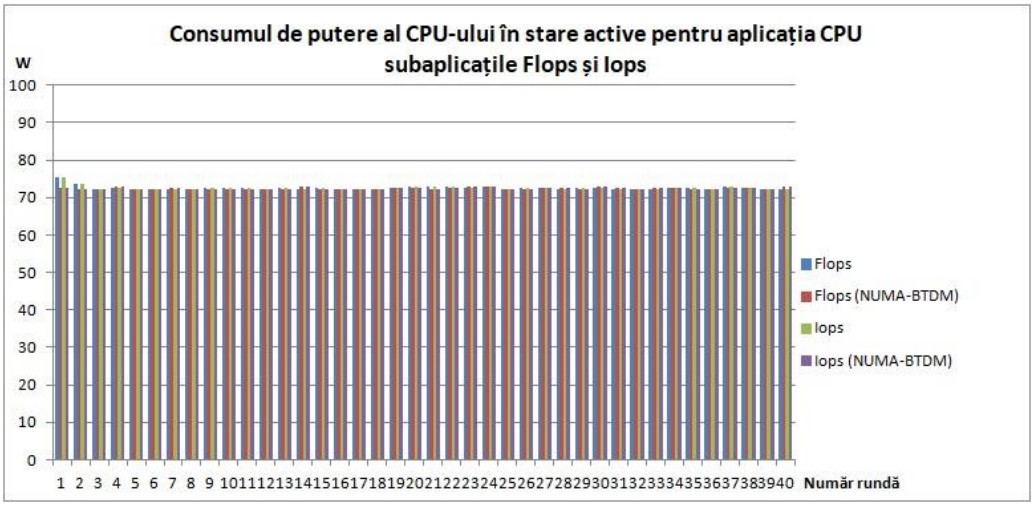

Figura 34. Consumul de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde

Fig. 35 ilustrează consumul de putere al CPU-ului și cel al întregului sistem UMA pe care rulează aplicația **Flops** [90]. Rolul acestei figuri este de a furniza o comparație valorică asupra măsurătorilor de consum de putere obținute din cele două surse (***turbostat***, WattsUp), atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când acesta nu este aplicat. Pentru aplicația **Flops** [90], se observă că, și în cazul sistemului UMA, măsurătorile din cele două surse sunt corelate și consumul CPU-ului este aproximativ jumătate din cel al întregului sistem.

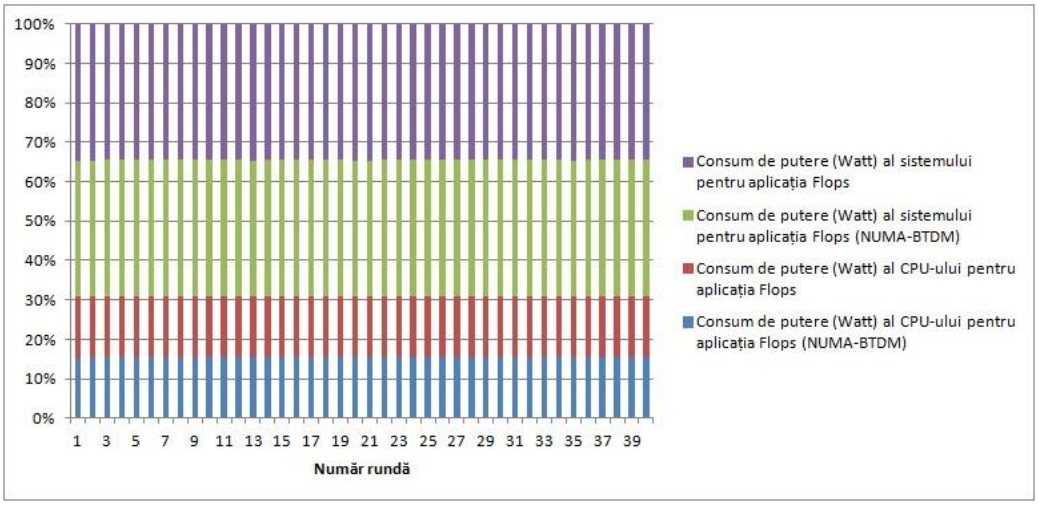

Figura 35. Consumul de putere (W) al CPU-ului și a întregului sistem UMA pe care rulează aplicația **Flops** în diferite runde

Fig. 36 ilustrează consumul de putere al CPU-ului și cel al întregului sistem UMA pe care rulează aplicația **Iops** [90], fiind valabile aceleași observații ca cele din Fig. 35.

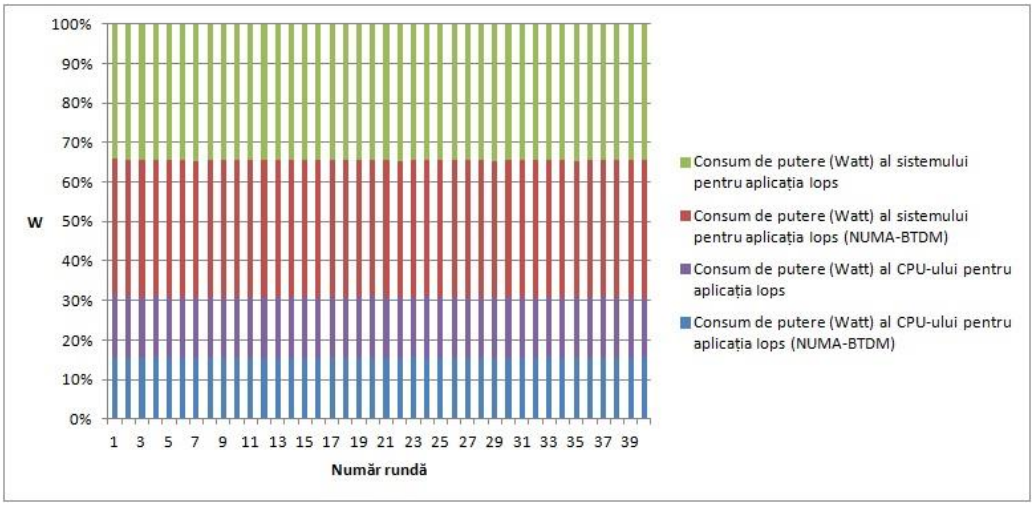

Figura 36. Consumul de putere (W) al CPU-ului și al întregului sistem UMA pe care rulează aplicația **Iops** în diferite runde

Cu cât diferența dintre consumul de putere al întregului sistem UMA și cel al CPU-ului este mai mare, cu atât execuția aplicației este mai performantă. Fig. 37 ilustrează această diferență pentru aplicațiile **Flops** [90] și **Iops** [90], cu și fără algoritmul ***NUMA-BTDM*** [55] aplicat. Se poate observa că diferența este mai mare în majoritatea cazurilor la aplicația **Flops** [90] decât la **Iops** [90]. O altă remarcă este că valorile extreme de consum de putere, adică acele valori care sunt cele mai

dispersate față de valoarea medie, se înregistrează atunci algoritmul ***NUMA-BTDM*** [55] nu este aplicat, în cazul ambelor aplicații. Aplicarea algoritmului contribuie la minimizarea dispersiei valorilor față de valoarea medie, ceea ce conduce la stabilizarea execuției, în ambele cazuri (**Flops** [90] și **Iops** [90]).

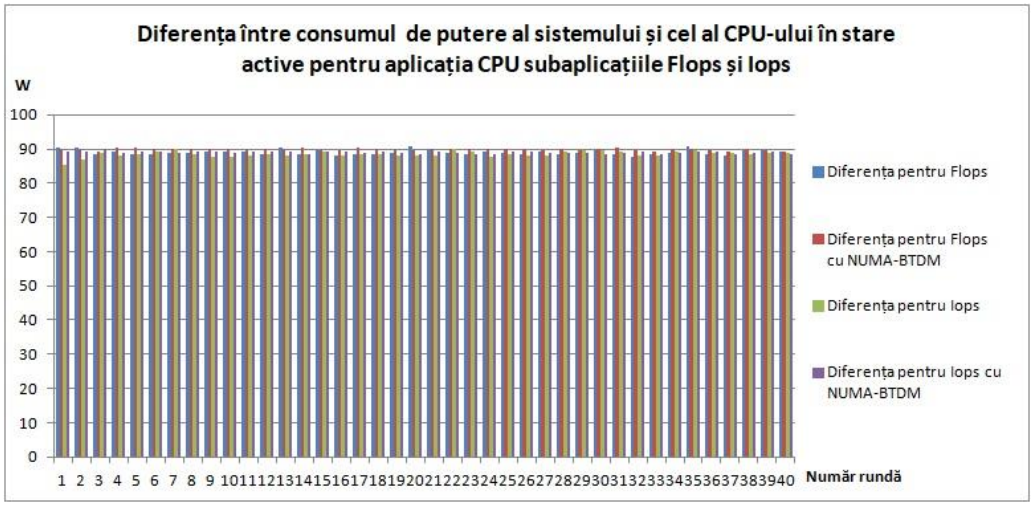

Figura 37. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului pe care rulează aplicațiile **Flops** și **Iops** în diferite runde

Figura 38 indică faptul că, pentru aplicația **Flops** [90], valorile medii per rundă de rulare (indicate pe abscisă) ale aplicației **Flops** [90], obținute pe sistem UMA din cele două surse (WattsUp și ***turbostat***), sunt corelate. Fiecare valoare medie este calculată ca fiind media aritmetică a celor 600 de valori obținute la fiecare secundă de rulare a aplicației **Flops** [90]. A fost obținut un grafic și pentru aplicația **Iops** [90], constatându-se și aici corelarea dintre cele două reprezentări.

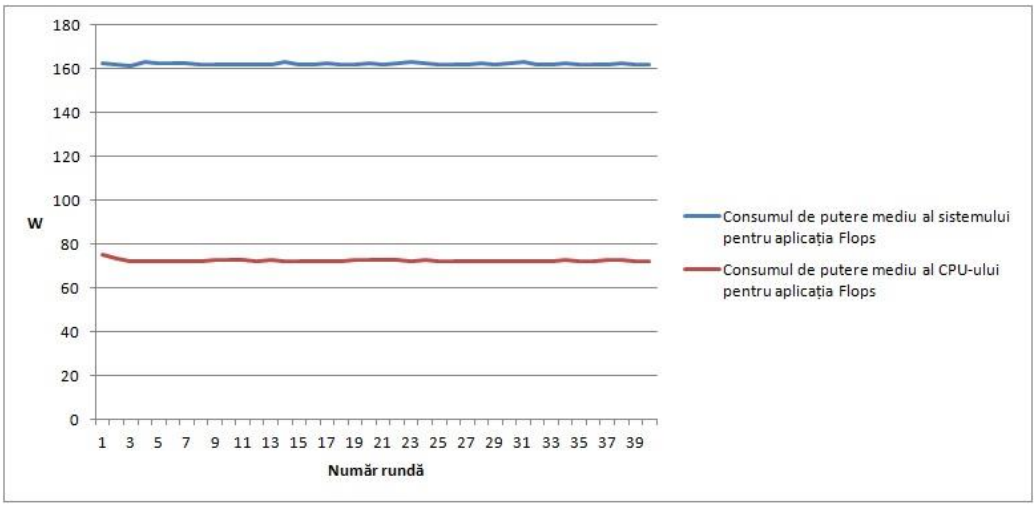

Figura 38. Valorile medii de consum de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **Flops** obținute pentru fiecare rundă în parte

Fig. 39 indică faptul că valorile de consum de putere obținute pe sistem UMA în runda 1 pentru aplicația **Flops** [90] din cele două surse, WattsUp, respectiv ***turbostat***, sunt corelate. Similar, s-au întocmit grafice pentru toate rundele de rulare a aplicației **Flops** [90] și a fost remarcată aceeași observație.

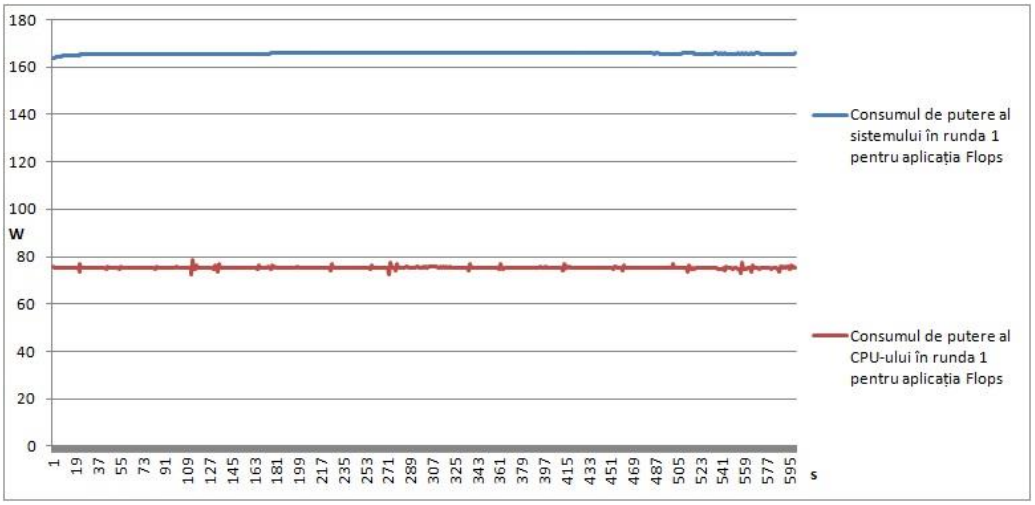

Figura 39. Valorile de consum de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **Flops** obținute în runda 1

Fig. 40 indică faptul că valorile de consum de putere al aplicației **Iops** [90], obținute în runda 1, pe sistem UMA, din cele două surse, WattsUp, respectiv ***turbostat***, sunt corelate. Similar, s-au întocmit grafice pentru toate rundele de execuție a aplicației **Iops** [90] și a fost remarcată aceeași observație.

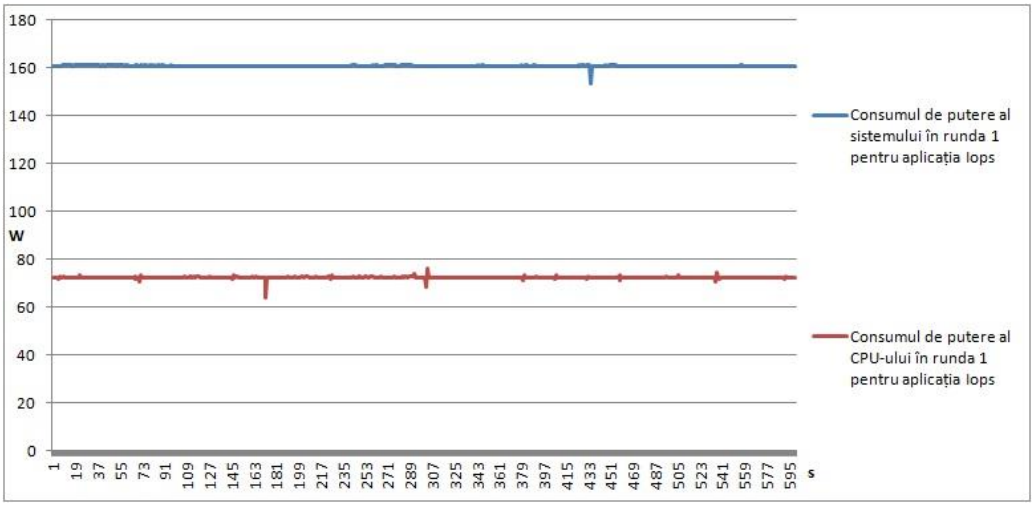

Figura 40. Valorile de consum de putere (W) al CPU-ului din sistemul UMA pe care rulează aplicația **Iops** obținute în runda 1

Consumul de putere al sistemului comparativ cu cel al CPU-ului *în stare "în așteptare"*

Fig. 41 prezintă consumul de putere mediu al sistemului UMA în stare "în așteptare" (fără nici o aplicație rulând), obținut pentru fiecare din cele 40 de runde de funcționare a sistemului în stare "în așteptare". Valorile proiectate pe ordonată corespund câte unei medii aritmetice a câte 600 de valori de consum putere a sistemului în stare "în așteptare" obținute câte una la fiecare 1 s: valoarea 1 de pe abscisă corespunde mediei aritmetice a primelor 600 de valori de consum de putere, valoarea 2 de pe abscisă corespunde mediei aritmetice a următoarelor 600 de valori de consum de putere, ș.a.m.d. Valoarea 600 reprezintă numărul de secunde în care rulează fiecare din aplicațiile **Flops** [90] și **Iops** [90]. În Fig. 41 se observă că acest consum de putere este aproape constant în timp, variațiile de la o rundă la cealaltă fiind mici, de cel mult 1 W, raportat la valoarea medie a tuturor rundelor, de 122.6 W pentru **Flops** [90] și 122.55 W pentru **Iops** [90].

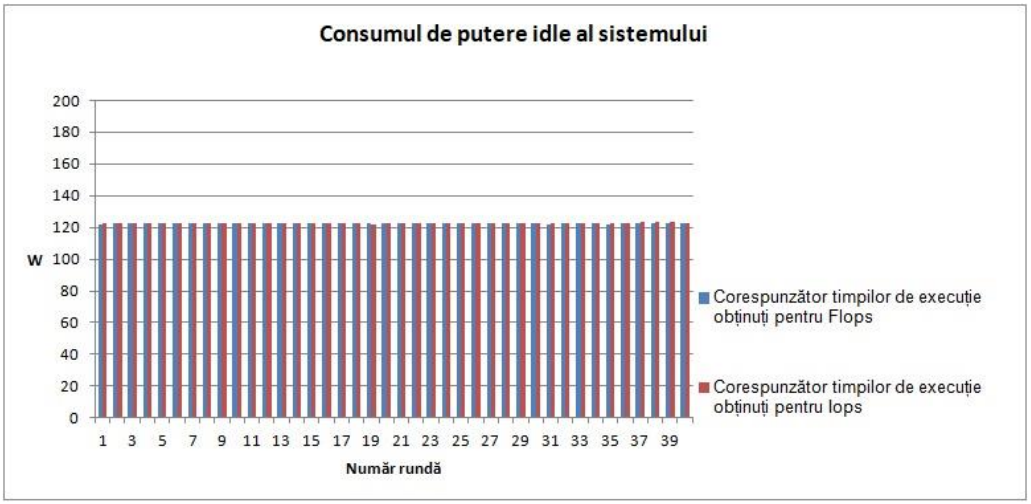

Figura 41. Valorile medii de consum de putere (W) al sistemului UMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare

Fig. 42 prezintă consumul de putere al CPU-ului din sistemul UMA în stare "în așteptare" (fără nici o aplicație rulând), obținut pentru fiecare din cele 40 de runde de funcționare a sistemului în stare "în așteptare". La fel ca în Figura 41, ordonata corespunde valorii mediei aritmetice a 600 de valori de consum putere al CPU-ului în stare "în așteptare", obținute câte una la fiecare 1 s. Și în acest caz consumul de putere este aproape constant în timp. Valoarea medie a tuturor rundelor este 18.22 W pentru **Flops** [90] și 18.38 W pentru **Iops** [90].

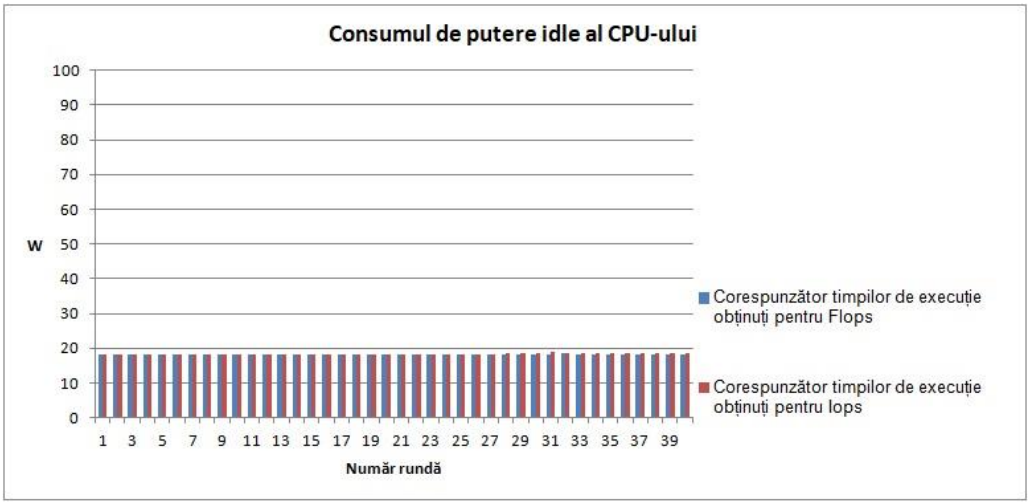

Figura 42. Valorile medii de consum de putere (W) al CPU-ului din sistemul UMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare

În Fig. 41 și Fig. 42 se observă că măsurătorile de consum de putere al CPU-ului și al întregului sistem, sunt corelate. Similar, Fig. 43 descrie consumul de putere al CPU-ului și al întregului sistem în corelație. De asemenea, se observă în plus că, consumul de putere al CPU-ului este aproximativ 25% din consumul de putere al întregului sistem UMA.

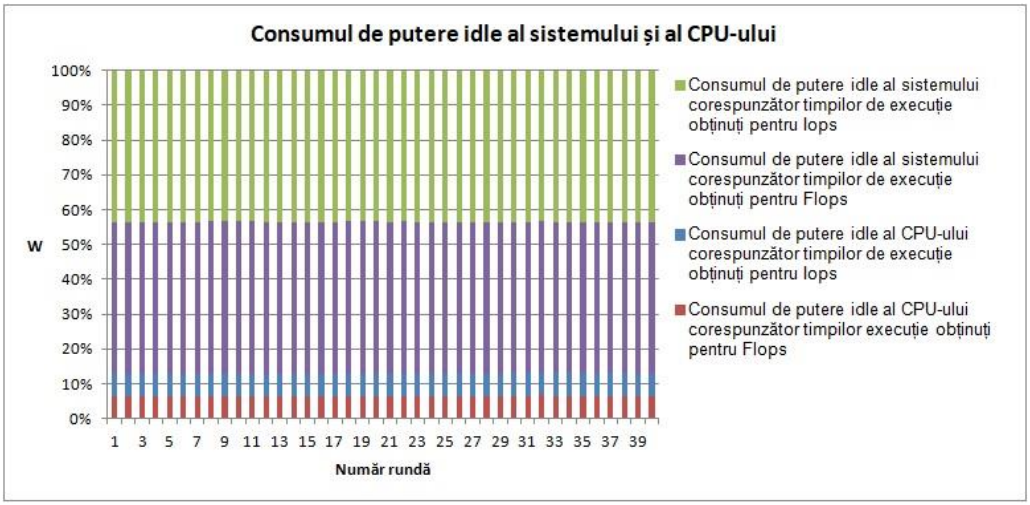

Figura 43. Consumul mediu de putere (W) al CPU-ului din sistemul UMA și al întregului sistem în stare "în așteptare", corelate, valori rezultate conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**

În Fig. 44 se observă că diferența între consumul de putere al întregului sistem UMA și cel al CPU-ului în stare "în așteptare", variază impredictibil de la o rundă (abscisa) la cealaltă. Pentru fiecare rundă este figurată grafic valorea medie a măsurătorilor obținute în acea rundă.

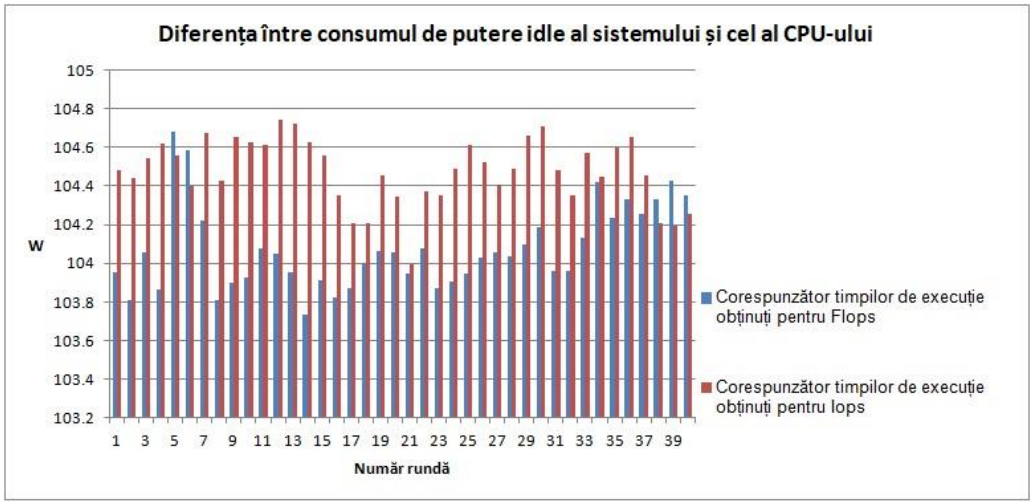

Figura 44. Diferența per rundă dintre consumul mediu de putere (W) al întregului sistem UMA și cel al CPU-ului în stare "în așteptare", rezultată conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**

*Consumul de putere al aplicației **CPU**, obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atât atunci când aplicația rulează neoptimizat cât și atunci când rulează optimizat, pe sistem UMA*

Tabelul 10 prezintă rezultatele experimentale pe sistem UMA ale aplicației **CPU**, atât în cazul în care acestea sunt optimizate utilizând algoritmul ***NUMA-BTDM*** [55] și cât și în cazul în când nu sunt optimizate. Prima coloană indică numărul rundei în care rulează aplicația. Coloanele 2-3 prezintă comparativ consumul de putere al aplicației **CPU** [90] optimizate și al aplicației **CPU** [90] neoptimizate. Coloana 4 descrie optimizarea în procente a aplicației **CPU** [90] rulate pe sistem UMA (obținută din diferența dintre consumul de putere la rularea aplicației neoptimizate și cel la înregistrat la rularea aplicației optimizate), iar coloana 5 ilustrează această optimizare în W. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Se observă o optimizare de până la 38.66% sau 17.91 W la aplicarea algoritmului ***NUMA-BTDM*** [55].

În Fig. 45 se observă că, în toate rundele, consumul de putere al aplicației **CPU** [90] este mai mic atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când aceasta nu este optimizată.

Tabel 10. Rezultate experimentale pe sistem UMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației **CPU**

Număr secundă	**Consum de putere (W)**		**Optimizarea obținută**	
	Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**	**Exprimată în %**	**Exprimată în W**
1	44.83	27.83	37.92	17
2	41.63	27.95	32.86	13.68
3	32.95	24.97	24.22	7.98
4	25.39	13.98	44.94	11.41
5	30.68	27.76	9.52	2.92
6	27.76	19.42	30.04	8.34
7	35.5	30.21	14.90	5.29
8	37.46	27.48	26.64	9.98
9	35.75	26.47	25.96	9.28
10	36.85	27.04	26.62	9.81
11	36.42	27.88	23.45	8.54
12	34.62	25.75	25.62	8.87
13	34.44	27.86	19.11	6.58
14	35.93	28.34	21.12	7.59
15	37.33	27.92	25.21	9.41
16	37.21	27.02	27.39	10.19
17	38.19	29.79	22.00	8.40
18	37.66	28.95	23.13	8.71
19	37.81	28.78	23.88	9.03
20	36.43	27.82	23.63	8.61
21	35.92	29.52	17.82	6.40
22	37.49	29.82	20.46	7.67
23	46.33	28.42	38.66	17.91

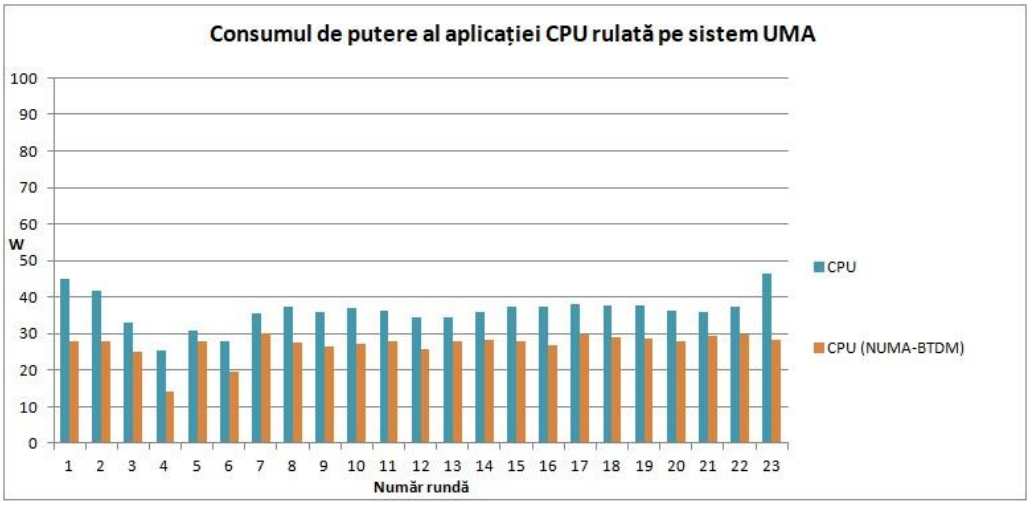

Figura 45. Consumul de putere (W) al aplicației **CPU** care rulează pe sistem UMA în diferite runde

Fig. 46 arată optimizarea de consum de putere la rularea aplicației **CPU** [90] pe sistem UMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55].

Optimizarea este cuprinsă între valorile minimă de 5.29 W și maximă de 17.91 W dintr-un total de 36.28 W, în medie. Optimizarea obținută se înregistrează la fiecare 1 s.

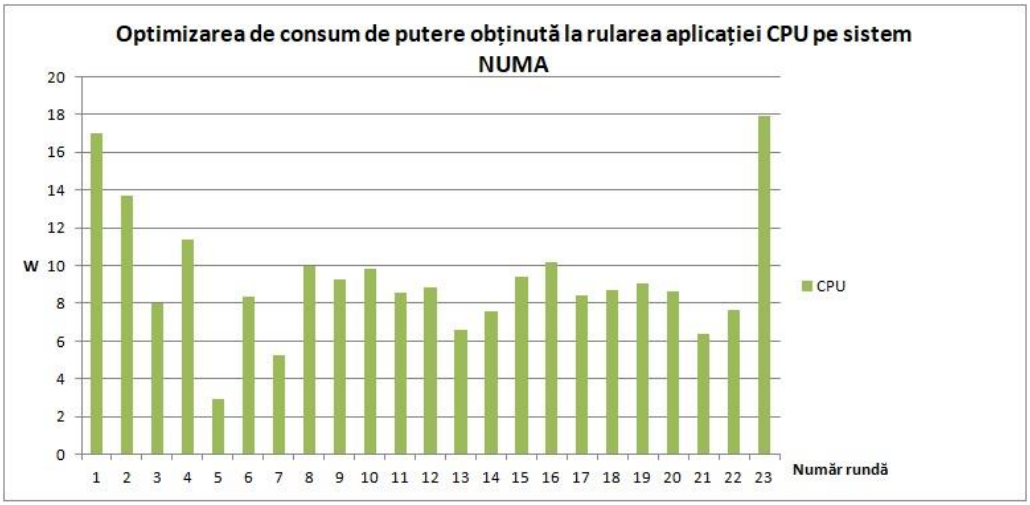

Figura 46. Optimizarea de consum de putere (W) al aplicației **CPU** care rulează pe sistem UMA în diverse runde

Fig. 47 descrie optimizarea de consum de putere, în procente, la rularea aplicației **CPU** [90] pe sistem UMA, obținută în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul de optimizare de consum de putere variază între valorile 9.52% și 44.94%.

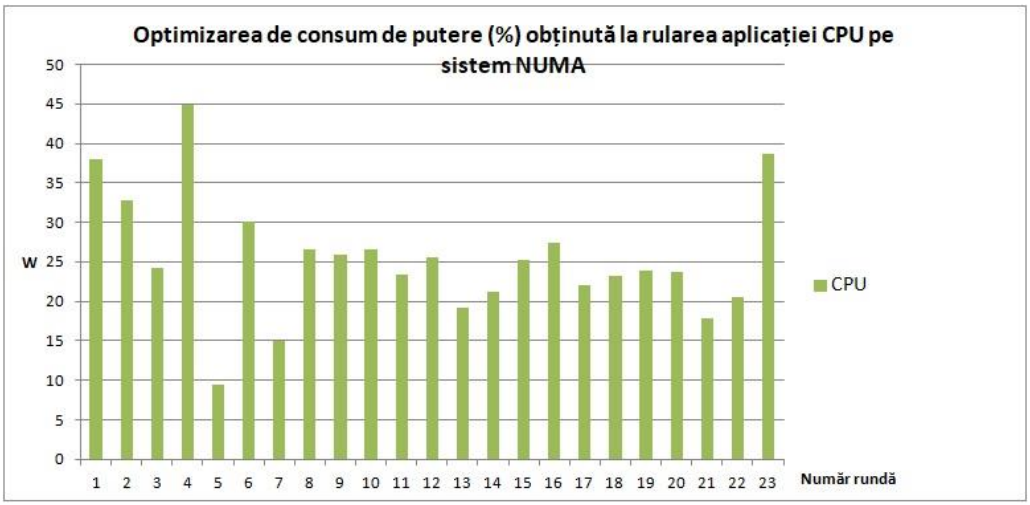

Figura 47. Optimizarea de consum de putere (%) al aplicației **CPU** care rulează pe sistem UMA în diverse runde

*Consumul de putere al aplicațiilor **Flops** și **Iops**, obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atât atunci când aplicațiile rulează neoptimizat cât și atunci când rulează optimizat, pe sistem UMA*

Tabelul 11 prezintă rezultatele experimentale pe sistem UMA ale aplicațiilor **Flops** [90] și **Iops** [90], atât în cazul în care acestea sunt optimizate utilizând algoritmul ***NUMA-BTDM*** [55] și cât și în cazul în când nu sunt optimizate.

Tabel 11. Rezultate experimentale pe sistem UMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicațiilor **Flops** și **Iops**

Număr rundă	**Consum de putere fără NUMA-BTDM**		**Consum de putere cu NUMA-BTDM**		**Optimizarea obținută (W)**		**Procent de optimizare**	
	Flops	**Iops**	**Flops**	**Iops**	**Flops**	**Iops**	**Flops**	**Iops**
1	57.15	57.10	54.33	54.29	2.815	2.815	4.926	4.930
2	55.55	55.52	54.02	53.99	1.522	1.522	2.741	2.742
3	54.08	54.03	53.98	53.93	0.098	0.098	0.181	0.181
4	54.40	54.27	54.89	54.76	-0.484	-0.484	-0.890	-0.892
5	54.16	54.04	54.13	54.01	0.030	0.030	0.055	0.055
6	53.89	53.95	54.06	54.11	-0.163	-0.163	-0.303	-0.303
7	53.93	53.72	54.24	54.03	-0.305	-0.305	-0.566	-0.568
8	54.04	53.95	54.00	53.91	0.040	0.040	0.073	0.073
9	54.51	54.31	53.94	53.75	0.565	0.565	1.037	1.041
10	54.49	54.41	53.93	53.85	0.558	0.558	1.024	1.025
11	54.44	54.31	53.91	53.79	0.522	0.522	0.959	0.961
12	54.16	54.16	53.93	53.93	0.227	0.227	0.419	0.419
13	54.60	54.43	54.16	54.00	0.434	0.434	0.794	0.796
14	54.30	54.18	54.73	54.60	-0.425	-0.425	-0.784	-0.785
15	54.31	54.20	54.05	53.94	0.262	0.262	0.482	0.483
16	53.84	53.82	53.93	53.91	-0.090	-0.090	-0.168	-0.168
17	54.00	53.80	54.24	54.03	-0.231	-0.231	-0.427	-0.429
18	54.09	54.07	53.90	53.87	0.197	0.197	0.365	0.365
19	54.50	54.48	54.33	54.31	0.168	0.168	0.309	0.309
20	54.83	54.70	54.57	54.43	0.264	0.264	0.481	0.482
21	54.83	54.80	53.98	53.96	0.845	0.845	1.542	1.543
22	54.56	54.75	54.09	54.28	0.472	0.472	0.865	0.862
23	54.20	54.25	54.74	54.79	-0.539	-0.539	-0.994	-0.993
24	54.71	54.52	54.67	54.49	0.039	0.039	0.072	0.072
25	54.07	54.09	53.90	53.92	0.164	0.164	0.303	0.303
26	54.34	54.24	54.03	53.93	0.311	0.311	0.572	0.573
27	54.15	54.25	54.31	54.41	-0.158	-0.158	-0.292	-0.292
28	54.10	53.75	54.43	54.08	-0.333	-0.333	-0.616	-0.620
29	54.51	54.04	54.36	53.89	0.150	0.150	0.275	0.277
30	54.21	53.71	54.58	54.07	-0.361	-0.361	-0.666	-0.672
31	54.08	53.22	54.45	53.60	-0.375	-0.375	-0.694	-0.705
32	53.54	53.79	53.57	53.82	-0.028	-0.028	-0.052	-0.051
33	53.98	53.55	54.53	54.10	-0.543	-0.543	-1.007	-1.015
34	54.23	54.02	54.25	54.04	-0.018	-0.018	-0.034	-0.034
35	54.20	53.76	54.03	53.59	0.174	0.174	0.320	0.323
36	54.08	53.60	54.06	53.58	0.019	0.019	0.036	0.036
37	54.39	54.22	54.35	54.18	0.040	0.040	0.073	0.073
38	54.33	53.92	54.28	53.87	0.051	0.051	0.094	0.095
39	54.02	53.60	54.03	53.61	-0.012	-0.012	-0.023	-0.023
40	53.86	53.56	54.55	54.25	-0.685	-0.685	-1.272	-1.279

Prima coloană indică numărul rundei în care rulează pe rând, fiecare aplicație. Coloanele 2-5 prezintă comparativ consumul de putere al aplicațiilor optimizate **Flops** [90] și **Iops** [90] și ale aplicațiilor **Flops** [90] și **Iops** [90] neoptimizate. Coloanele 6-7 descriu optimizarea în W a aplicațiilor **Flops** [90] și **Iops** [90] rulate pe sistem NUMA (obținută din diferența dintre consumul de putere la rularea aplicației neoptimizate și cel la înregistrat la rularea aplicației optimizate), iar coloanele 8-9 ilustrează această optimizare în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Se observă o optimizare relativ redusă atât în cazul **Flops** [90] cât și în cazul **Iops** [90], la aplicarea algoritmului ***NUMA-BTDM*** [55].

În Fig. 48 se observă că, în majoritatea rundelor (figurate pe abscisă), consumul de putere al aplicației **Iops** [90] este mai mare atunci când aplicația nu este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când aceasta este optimizată. La aplicația **Flops** [90], optimizarea de consum de putere este prezentă în mai puțin de jumătate din runde și are valori mai mici ca la **Iops** [90].

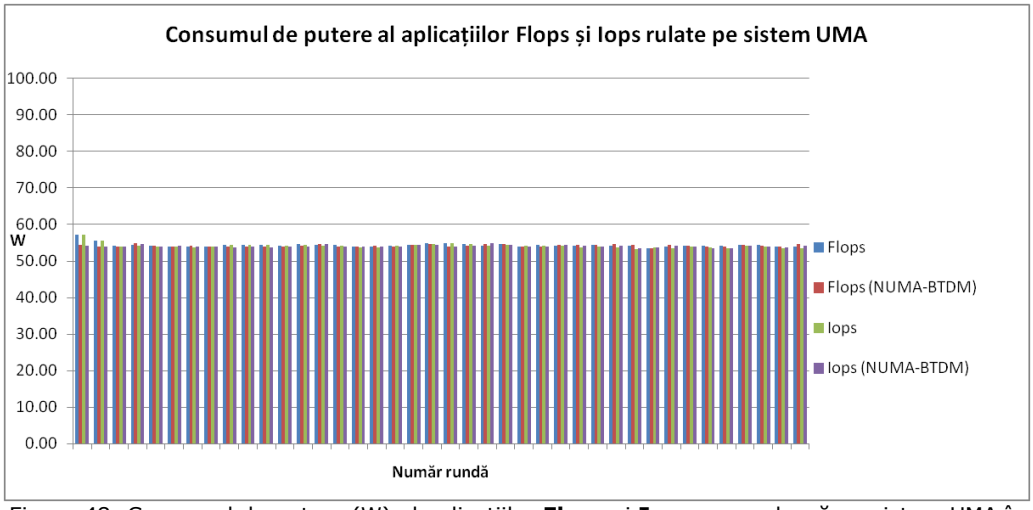

Figura 48. Consumul de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem UMA în diferite runde

Fig. 49 descrie optimizarea de consum de putere la rularea aplicațiilor **Flops** [90] și **Iops** [90] pe sistem UMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea este cuprinsă între valorile minimă de 0.036 W și maximă de 1.59 W pentru **Flops** [90] și valorile minimă 0.007 W și maximă 1.164 W pentru **Iops** [90]. Optimizarea obținută se înregistrează la fiecare 1 s.

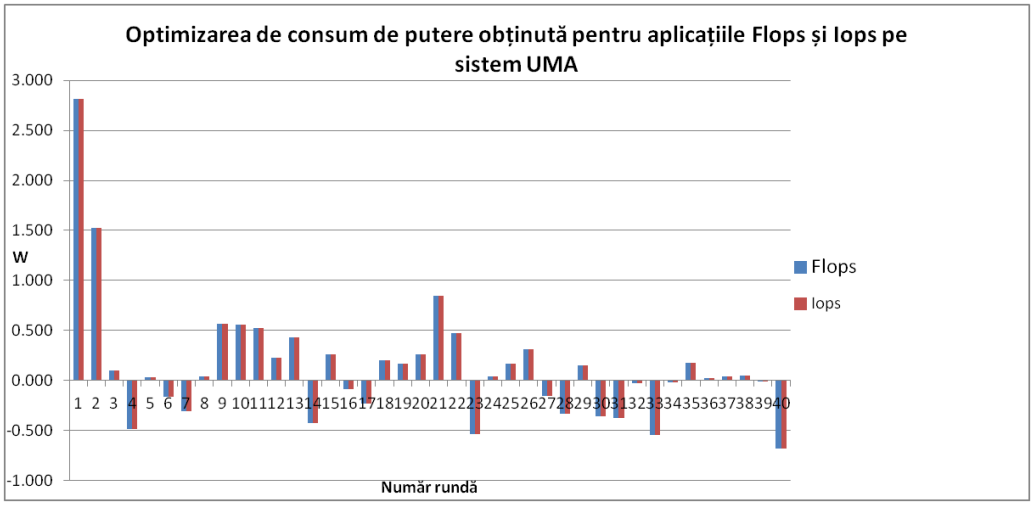

Figura 49. Optimizarea de consum de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem UMA în diverse runde

Fig. 50 arată optimizarea de consum de putere, în procente, la rularea aplicațiilor **Flops** [90] și **Iops** [90] pe sistem UMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul de optimizare de consum de putere variază între valorile 0.07% și 1.93% pentru **Flops** [90] și 0.01% și 2.11% pentru **Iops** [90].

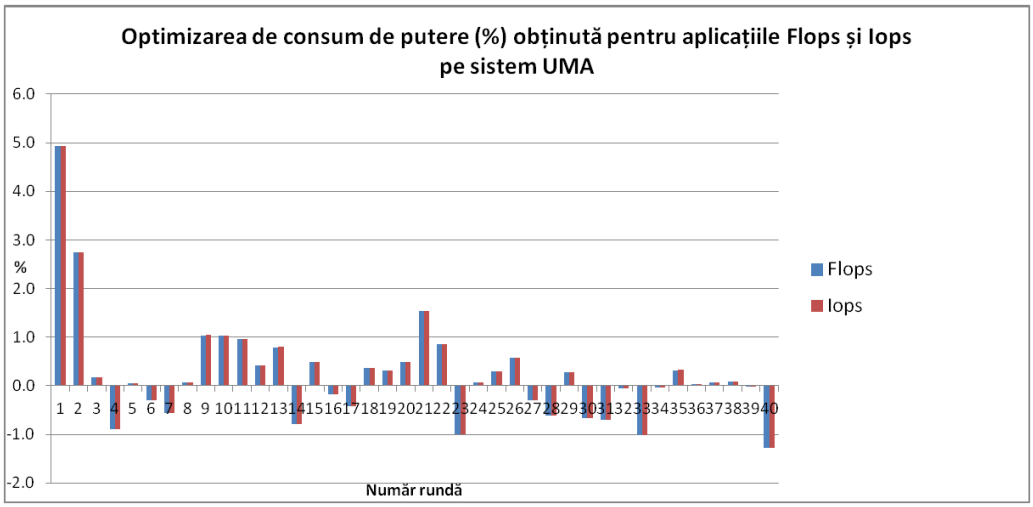

Figura 50. Optimizarea de consum de putere (%) al aplicațiilor **Flops** și **Iops** care rulează pe sistem UMA în diverse runde

4.5.3 Rezultate experimentale pe sisteme NUMA

4.5.3.1 Timp de execuție

Tabelul 12 prezintă valorile medii ale rezultatelor experimentale de timp de execuție, pe sistem NUMA, pentru aplicația **CPU** [90], formată din aplicațiile **CPU** [90], **Flops** [90], **Iops** [90], atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când nu este aplicat la compilare. Valorile medii pentru fiecare din cele 3 aplicații, sunt obținute calculând media aritmetică a 40 de timpi de execuție.

În Tabelul 12, se observă că, pentru aplicația **CPU**, timpul de execuție scade în medie cu 0.0035 s, adică 0.61 %. Procentul mic de optimizare se datorează faptului că mai multe fire autonome au fost mapate pe același core-uri, deși resursele hardware disponibile ar fi fost suficiente ca aceste fire să fie mapate fiecare pe core-uri diferite, ceea ce ar fi condus la scăderea numărului de treceri ale firelor din stare activă în stare "în așteptare" și invers (fiind mai puține fire asignate per core).

Pentru aplicațiile **Flops** [90] și **Iops** [90] nu se înregistrează o optimizare a timpului de execuție. Acest lucru se datorează timpului scurs cu execuția apelurilor de setare a afinității CPU, setare realizată pentru un număr mare de fire de execuție (2400 în cazul fiecărei aplicații). Timpul scurs este în acest caz mai mare decât optimizarea produsă de algoritmul ***NUMA-BTDM*** [55]. Acesta este un caz tipic în care optimizarea nu este obținută. În general, atunci când numărul de fire de execuție este mare și calculele realizate de fiecare fir sunt reduse, este logic ca setarea afinității CPU la rulare să lungească timpul de execuție al firului cu o durată de timp comparativă chiar cu timpul său de execuție. Firul executând puține calcule, timpul său de execuție este scurt, maparea dublând sau chiar triplând timpul său de execuție.

Tabel 12. Rezultate experimentale de timp de execuție al aplicației **CPU** (formată din aplicațiile **CPU**, **Flops**, **Iops**) rulată pe sistem NUMA, prezentate comparativ atunci când algoritmul ***NUMA-BTDM*** este aplicat și atunci când nu este aplicat

Aplicație	**Timp de execuție mediu (W)**	
	Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**
CPU	0.5710	0.5675
Flops	600.4992	600.5054
Iops	600.4989	600.5020

4.5.3.2 Consum de putere

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când aplicația* ***CPU*** *rulează neoptimizat și atunci când rulează optimizat*

Tabelul 13 prezintă comparativ rezultatele experimentale de consum de putere pentru aplicația **CPU** atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] și atunci când aceasta nu este optimizată, rezultate obținute din

cele două surse, utilitarul ***turbostat*** (pentru consumul de putere al CPU-ului) și dispozitivul specializat WattsUp (pentru consumul de putere al întregului sistem NUMA). Rezultatele arată o scădere în medie a consumului de energie cu aproximativ 3 W/s atunci când aplicația **CPU** este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], scădere înregistrată păstrându-se același timp de execuție. Dispersia măsurătorilor de consum de putere față de valoarea medie este mai mică atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat: în cazul măsurătorilor obținute cu ajutorul utilitarului ***turbostat***, varianța 3.16 atunci când optimizarea este aplicată este mai mică decât varianța 30.09 atunci când optimizarea nu este aplicată, iar în cazul măsurătorilor obținute cu ajutorul dispozitivului WattsUp, varianța mare 3308.09 a aplicației **CPU** optimizate se datorează valorilor nule obținute datorită erorilor de măsurare, în timp ce valorile nenule sunt apropiate de medie, la fel ca la măsurătorile obținute utilizând ***turbostat***.

Tabel 13. Comparație între rezultatele experimentale pentru aplicația **CPU**, obținute din cele două surse, utilitarul ***turbostat***, respectiv dispozitivul WattsUp, atât când aceasta nu este optimizată, cât și când este optimizată

Categorie valori	**Sursă valori**	**Comparație între măsurătorile obținute din cele două surse, cu și fără aplicarea algoritmului NUMA-BTDM**	
		Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**
Minime	***turbostat***	45.83	45.40
	WattsUp	137.90	137
Medii	***turbostat***	50.47	47.86
	WattsUp	143.03	149.46
Maxime	***turbostat***	61.63	52.38
	WattsUp	158.10	160.50
Varianță	***turbostat***	30.09	3.16
	WattsUp	45.63	119.16
Deviație standard	***turbostat***	5.49	1.82
	WattsUp	6.76	11.21

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când fiecare din aplicațiile **Flops** și **Iops** rulează fără a fi optimizate și atunci când rulează după ce au fost optimizate*

În Fig. 51, abscisa reprezintă numărul rundei de rulare a celor două aplicații **Flops** [90] și **Iops** [90], iar ordonata reprezintă consumul de putere in W al întregului sistem la rularea pe rând a celor două aplicații. Din figură se observă că, în jumătate dintre runde, sistemul NUMA consumă mai puțină putere când rulează aplicația **Flops** [90] optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat. Aceeași observație este valabilă și pentru aplicația **Iops** [90], dar, în acest caz, numărul de runde în care sistemul pe care rulează aplicația optimizată consumă mai puțină putere este mai mare decât cel al rundelor în care sistemul pe care rulează aplicația neoptimizată consumă mai puțină putere. Totuși în medie, luând în considerare toate rundele, sistemul cu aplicația **Iops** [90] rulând consumă cu 0.06 W mai mult atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], comparativ cu consumul de putere al sistemului cu aplicația **Flops** [90] rulând, care este în medie cu 0.34 W mai mic atunci când aplicația este optimizată.

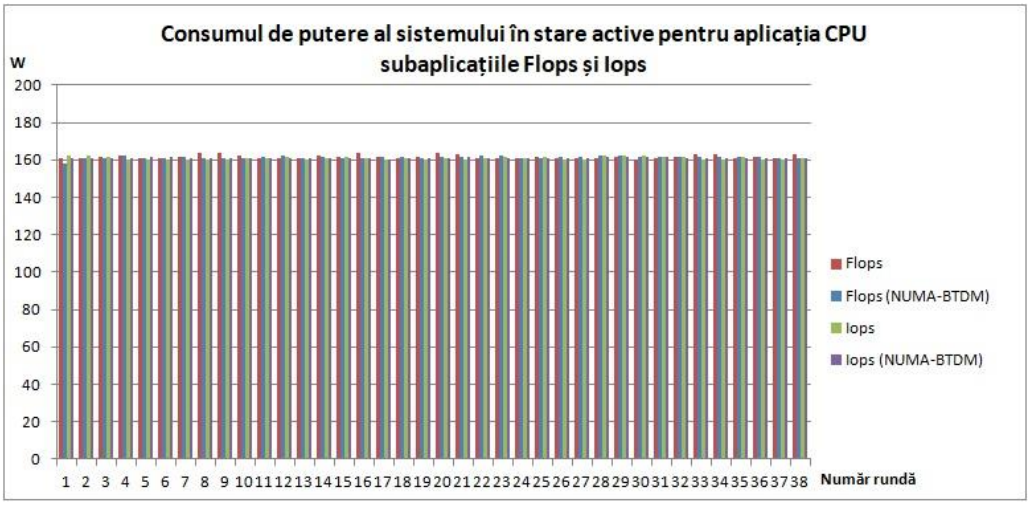

Figura 51. Consumul de putere (W) al sistemului NUMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde

Fig. 52 descrie consumul de putere al CPU-ului din sistemul NUMA. În figură, abscisa are aceeași semnificație ca în Fig. 51 iar ordonata reprezintă consumul putere al CPU-ului exprimat in W. Se observă că în mai mult de jumătate din runde, consumul de putere al CPU-ului cu aplicația **Flops** [90] rulând este mai optim atunci când aplicația nu este optimizată. Totuși consumul de putere al CPU-ului din sistemul NUMA este în medie mai mic (luând în considerare toate rundele) cu 0.14 W atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55], decât atunci când algoritmul nu este aplicat. Aceleași observații sunt valabile și pentru aplicația **Iops** [90], numărul rundelor în care consumul de putere al CPU-ului cu aplicația rulând fiind mai mare decât la **Flops** [90], când aceasta este optimizată utilizând ***NUMA-BTDM*** [55] și optimizarea medie de consum de putere fiind mai mică decât la **Flops** [90] (0.12 W).

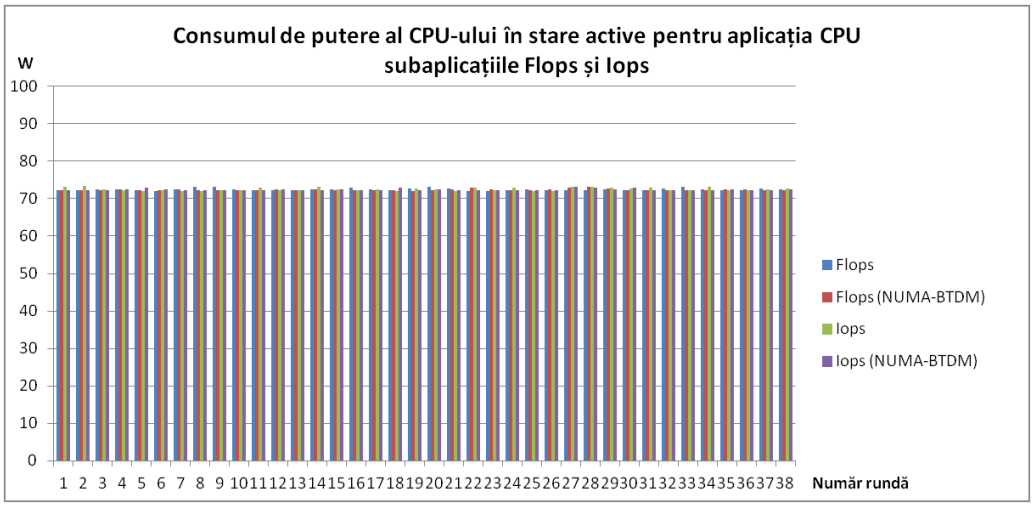

Figura 52. Consumul de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicațiile **Flops** și **Iops** în diferite runde

Fig. 53 ilustrează consumul de putere al CPU-ului și cel al întregului sistem NUMA pe care rulează aplicația **Flops** [90]. Rolul acestei figuri este de a furniza o comparație valorică asupra măsurătorilor de consum de putere obținute din cele două surse (***turbostat***, WattsUp), atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când acesta nu este aplicat. Pentru aplicația **Flops** [90], se observă că, și în cazul sistemului NUMA, măsurătorile sunt corelate și consumul CPU-ului este aproximativ jumătate din cel al întregului sistem.

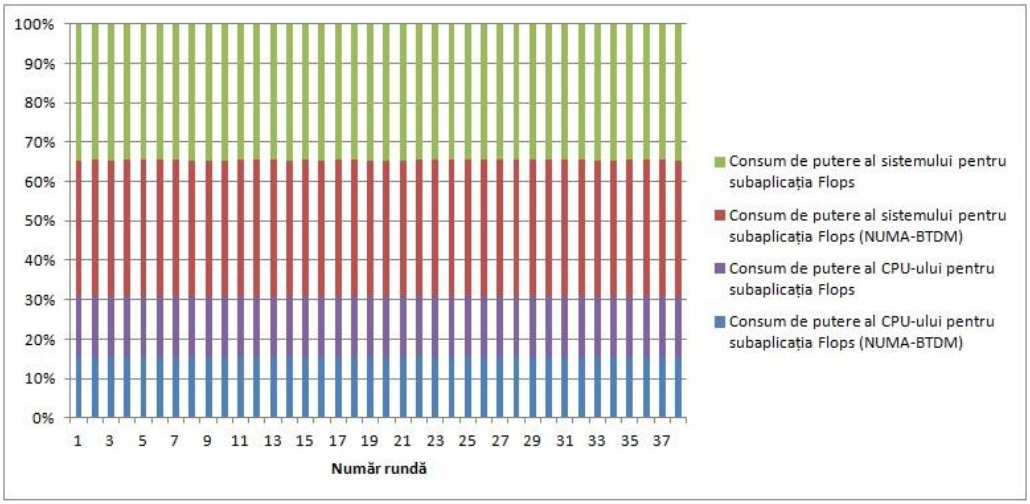

Figura 53. Consumul de putere (W) al CPU-ului și al întregului sistem NUMA pe care rulează aplicația **Flops** în diferite runde

Fig. 54 ilustrează consumul de putere al CPU-ului și cel al întregului sistem NUMA pe care rulează aplicația **Iops** [90], fiind valabile aceleași observații ca cele din Figura 53.

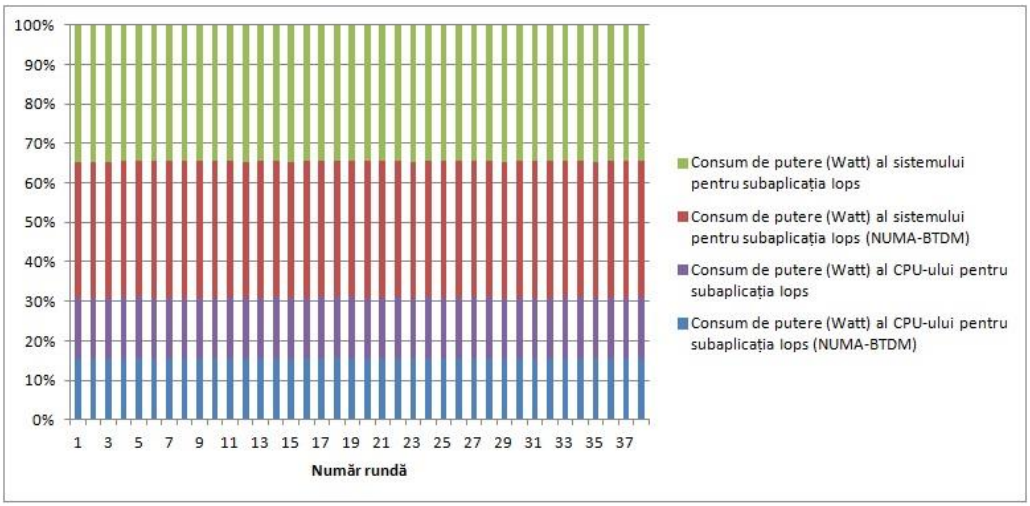

Figura 54. Consumul de putere (W) al CPU-ului și al întregului sistem NUMA pe care rulează aplicația **Iops** în diferite runde

Cu cât diferența între consumul de putere al întregului sistem NUMA și cel al CPU-ului este mai mare, cu atât execuția aplicației este mai optimă. Fig. 55 ilustrează această diferență pentru aplicația **CPU** aplicațiile **Flops** [90] și **Iops** [90], cu și fără algoritmul ***NUMA-BTDM*** [55] aplicat. Se poate observa că diferența este mai mare în majoritatea cazurilor la aplicația **Flops** [90] decât la **Iops** [90]. O altă remarcă este că valorile extreme de consum de putere, adică acele valori care sunt cele mai dispersate fată de valoarea medie, se înregistrează atunci algoritmul ***NUMA-BTDM*** [55] nu este aplicat, în cazul ambelor aplicații. Aplicarea algoritmului ***NUMA-BTDM*** [55] contribuie la minimizarea dispersiei valorilor față de valoarea medie, ceea ce conduce la stabilizarea execuției, în ambele cazuri (**Flops** [90] și **Iops** [90]).

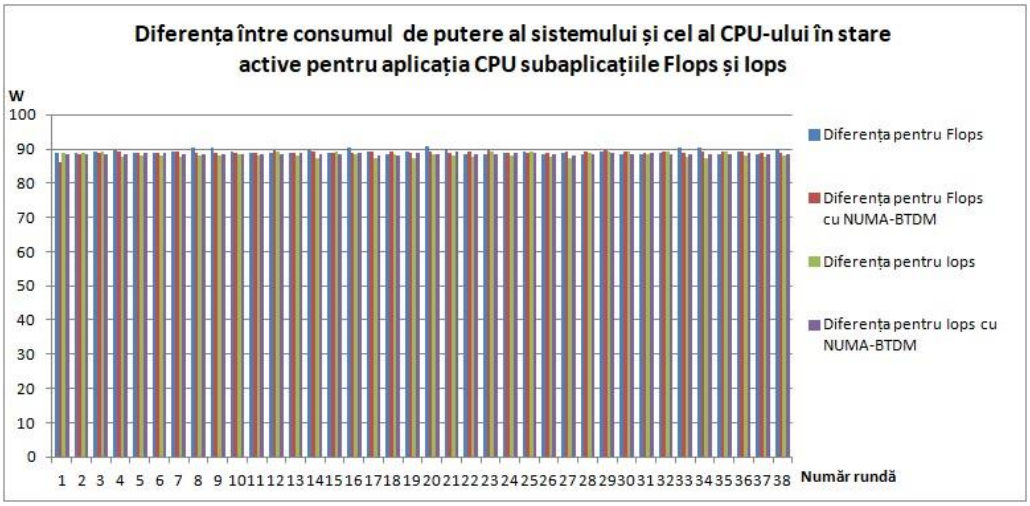

Figura 55. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului pe care rulează aplicațiile **Flops** și **Iops** în diferite runde

Figura 56 indică faptul că, pentru aplicația **Flops** [90], valorile medii, câte una pentru fiecare rundă de rulare a aplicației, obținute pe sistem NUMA din cele două surse (WattsUp și ***turbostat***), sunt corelate. Fiecare valoare medie este calculată ca fiind media aritmetică a celor 600 de valori obținute câte una la fiecare secundă de rulare a aplicației **Flops** [90]. Similar, a fost obținută o figură și pentru aplicația **Iops** [90], constatându-se și în acest caz corelarea dintre cele două grafice.

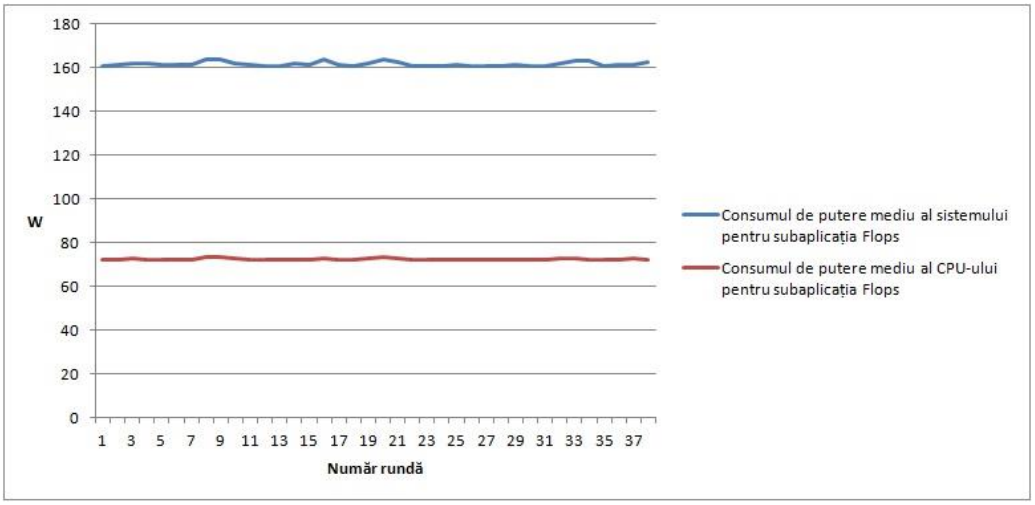

Figura 56. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **Flops** obținute pentru fiecare rundă în parte

Figura 57 indică faptul că valorile de consum de putere obținute în runda 1 pentru aplicația **Flops** [90] din cele două surse, WattsUp, respectiv ***turbostat***, pe sistem NUMA, sunt corelate. Similar, s-au întocmit grafice pentru toate rundele de rulare a aplicației **Flops** [90] și a fost remarcată aceeași observație.

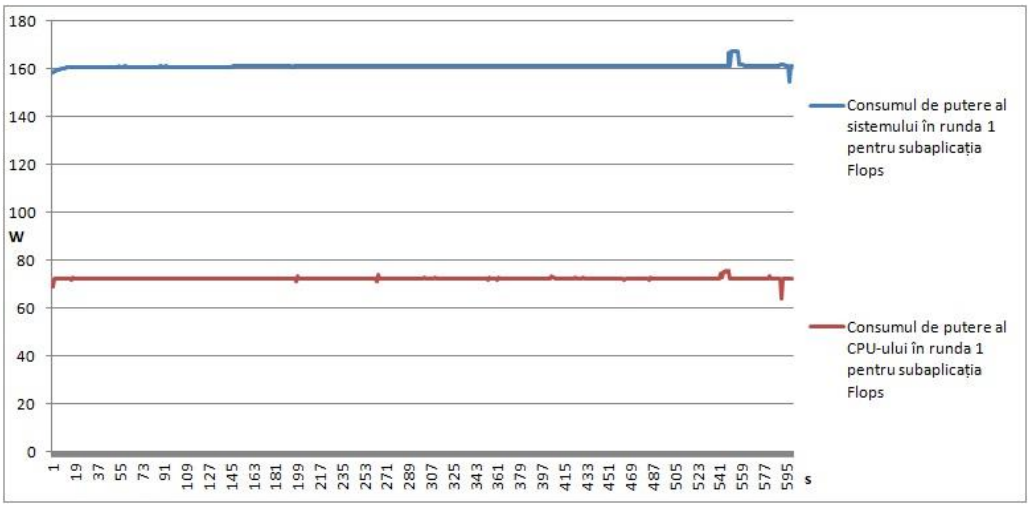

Figura 57. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **Flops** obținute în runda 1

Figura 58 indică faptul că valorile de consum de putere obținute în runda 1 pentru aplicația **Iops** [90] din cele două surse, WattsUp, respectiv ***turbostat***, pe sistem NUMA, sunt corelate. Similar, s-au întocmit grafice pentru toate rundele de rulare a aplicației **Iops** [90] și a fost remarcată aceeași observație.

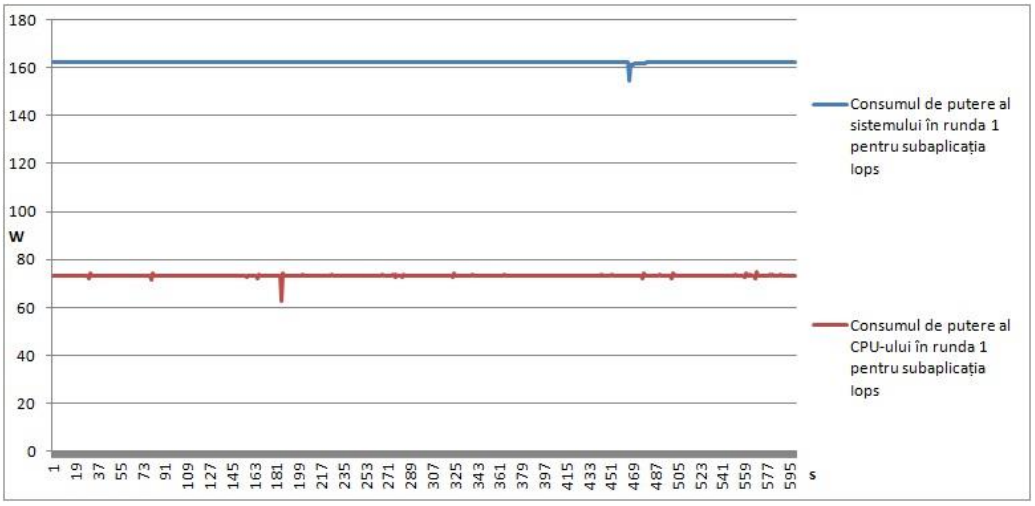

Figura 58. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA pe care rulează aplicația **Iops** obținute în runda 1

Consumul de putere al sistemului comparativ cu cel al CPU-ului *în stare "în așteptare"*

Fig. 59 prezintă consumul de putere mediu al sistemului NUMA în stare "în așteptare" (fără nici o aplicație rulând), obținut pentru fiecare din cele 40 de runde de funcționare a sistemului în stare "în așteptare". Fiecare valoare de pe abscisă corespunde rundei de rulare, iar ordonata corespunde mediei aritmetice a câte 600 de valori de consum putere a sistemului în stare "în așteptare" obținute la câte una fiecare secundă: valoarea 1 de pe abscisă corespunde mediei aritmetice a primelor 600 de valori de consum de putere, valoarea 2 de pe abscisă corespunde mediei aritmetice a următoarelor 600 de valori de consum de putere, ș.a.m.d. Valoarea 600 reprezintă numărul de secunde în care ar rula fiecare din aplicațiile **Flops** [90] și **Iops** [90], dacă ar fi lansate în execuție. Din Fig. 59 se observă că acest consum de putere este aproape constant în timp, variațiile de la o rundă la cealaltă fiind mici, de cel mult 1 W, comparativ cu valoarea medie a tuturor rundelor, de 122.38 W pentru **Flops** [90] și 122.54 W pentru **Iops** [90].

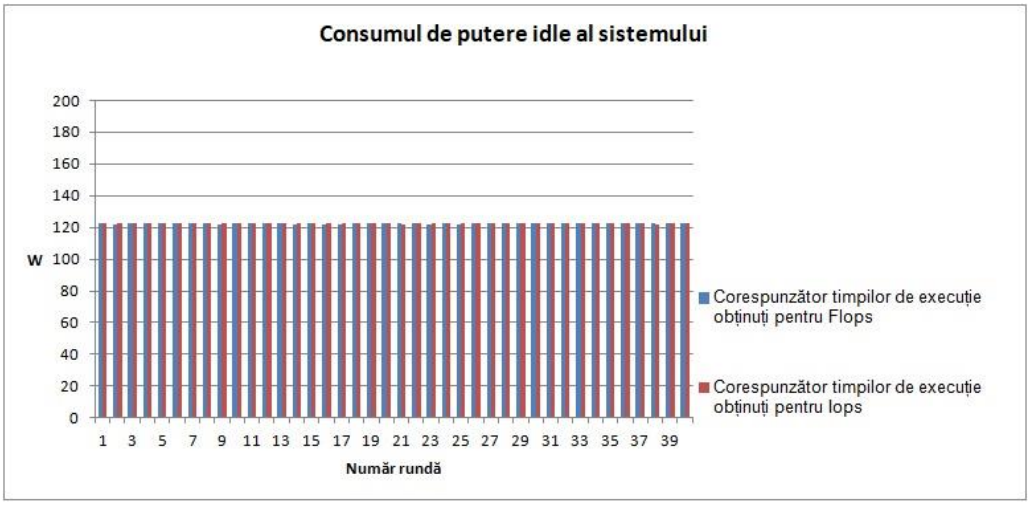

Figura 59. Valorile medii de consum de putere (W) al sistemului NUMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare

Fig. 60 prezintă consumul de putere al CPU-ului din sistemul NUMA în stare "în așteptare" (fără nici o aplicație rulând), obținut pentru fiecare din cele 40 de runde de funcționare a sistemului în stare "în așteptare". La fel ca în Figura 59, fiecare valoare de pe abscisă corespunde rundei de rulare, iar ordonata mediei aritmetice a câte 600 de valori de consum putere a CPU-ului în stare "în așteptare" obținute la fiecare secundă. Și în acest caz consumul de putere este aproape constant în timp, observându-se că variațiile consumului de putere al sistemului sunt cauzate în mare parte de variațiile consumului de putere al CPU-ului (variațiile consumului de putere al CPU-ului de la o rundă la cealaltă sunt de cel mult 0.7 W, fiind relativ mici comparativ cu valoarea medie a tuturor rundelor, 18.30 W pentru **Flops** [90] și 18.06 W pentru **Iops** [90]).

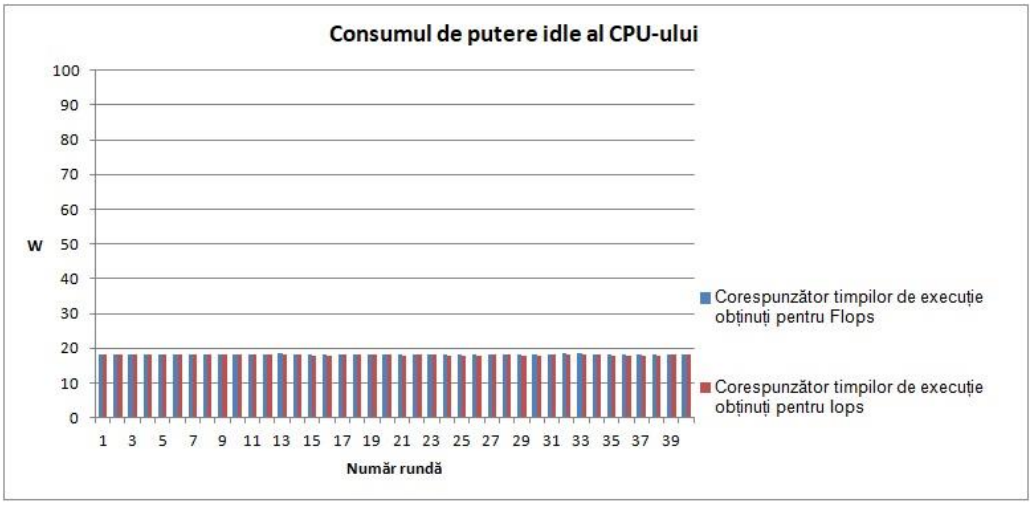

Figura 60. Valorile medii de consum de putere (W) al CPU-ului din sistemul NUMA în stare "în așteptare" corespunzătoare timpilor de execuție ai aplicațiilor **Flops** și **Iops**, obținuți în stare active, în rundele de rulare

Din cele două figuri, Fig. 59 și Fig. 60, se poate observa că măsurătorile de consum de putere al CPU-ului și al întregului sistem, sunt corelate. Bazat pe aceleași considerente ca în cazul celor două figuri anterioare, Fig. 61 descrie consumul de putere al CPU-ului și al întregului sistem în corelație. De asemenea, se observă că, consumul de putere al CPU-ului este aproximativ 15% din consumul de putere al întregului sistem NUMA.

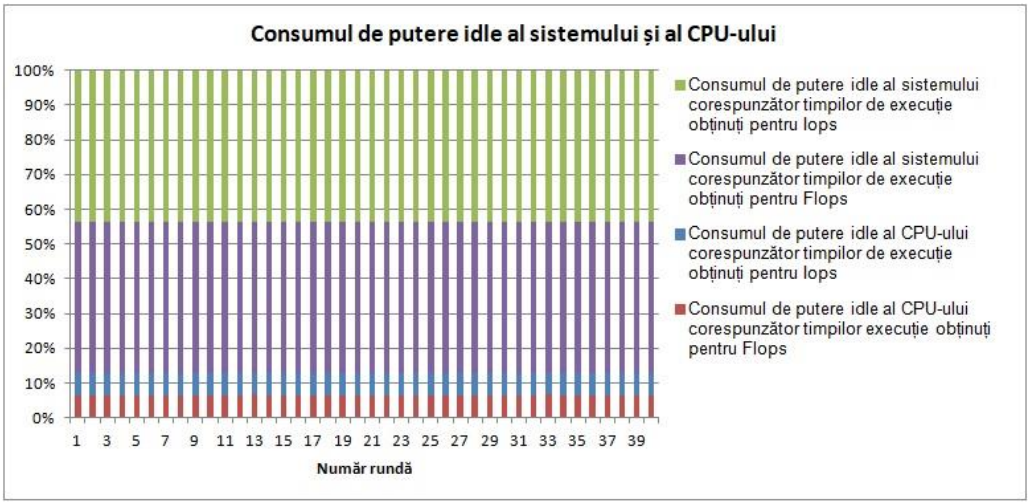

Figura 61. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA și al întregului sistem în stare "în așteptare", corelate, valori rezultate conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**

Din Fig. 62 se observă că diferența între consumul de putere al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare", variază impredictibil în medie, de la o rundă la cealaltă.

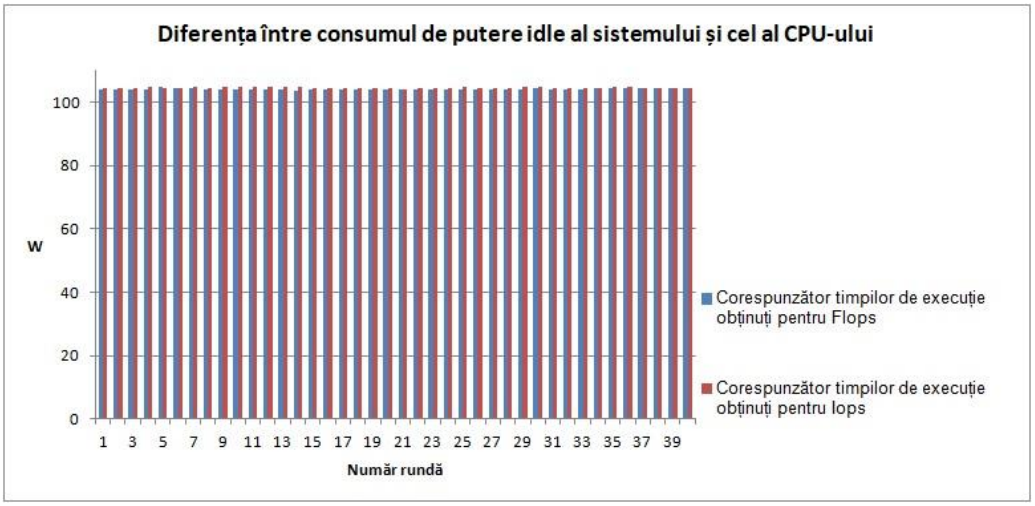

Figura 62. Diferența per rundă dintre consumul mediu de putere (W) al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare", rezultată conform timpilor de execuție ai aplicațiilor **Flops** și **Iops**

*Consumul de putere al aplicației **CPU**, obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atât atunci când aplicația rulează neoptimizat cât și atunci când rulează optimizat, pe sistem NUMA.*

Tabelul 14 prezintă rezultatele experimentale pe sistem NUMA ale aplicației **CPU** [90], atât în cazul în care aceasta este optimizată folosind algoritmul ***NUMA-BTDM*** [55] și cât și în cazul în când nu este optimizată. Prima coloană indică numărul rundei în care rulează aplicația. Coloanele 2-3 prezintă comparativ consumul de putere al aplicației optimizate și al aceleiași aplicații, neoptimizate. Coloana 4 descrie optimizarea în W a aplicației **CPU** rulate pe sistem NUMA (obținută din diferența dintre consumul de putere la rularea aplicației neoptimizate și cel la înregistrat la rularea aplicației optimizate), iar coloana 5 ilustrează această optimizare în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Se observă o optimizare de 2.61 W/s în medie, la aplicarea algoritmului ***NUMA-BTDM*** [55].

Tabel 14. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației **CPU**

Număr secundă	**Consum de putere (W)**		**Optimizarea obținută**	
	Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**	**Exprimată în %**	**Exprimată în W**
1	39.67	27.06	12.61	31.79
2	27.18	18.19	8.99	33.08
3	30.71	22.47	8.24	26.83
4	39.62	30.76	8.86	22.36
5	39.68	25.88	13.80	34.78
6	30.82	27.46	3.36	10.90
7	28.73	26.92	1.81	6.30
8	28.08	27.79	0.29	1.03
9	31.26	28.62	2.64	8.45
10	25.19	24.74	0.45	1.79
11	27.04	28.29	-1.25	-4.62
12	25.55	26.13	-0.58	-2.27
13	17.41	18.43	-1.02	-5.86
14	19.51	17.01	2.50	12.81
15	21.83	22.21	-0.38	-1.74
16	19.95	18.55	1.40	7.02
17	25.54	27.21	-1.67	-6.54
18	27.03	26.86	0.17	0.63
19	26.57	26.55	0.02	0.08
20	25.68	25.73	-0.05	-0.19
21	26.91	26.42	0.49	1.82
22	26.66	29.36	-2.70	-10.13
23	33.31	31.26	2.05	6.15

Din Fig. 63 se observă că, în majoritatea rundelor, consumul de putere al aplicației **CPU** este mai mic atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când aceasta nu este optimizată.

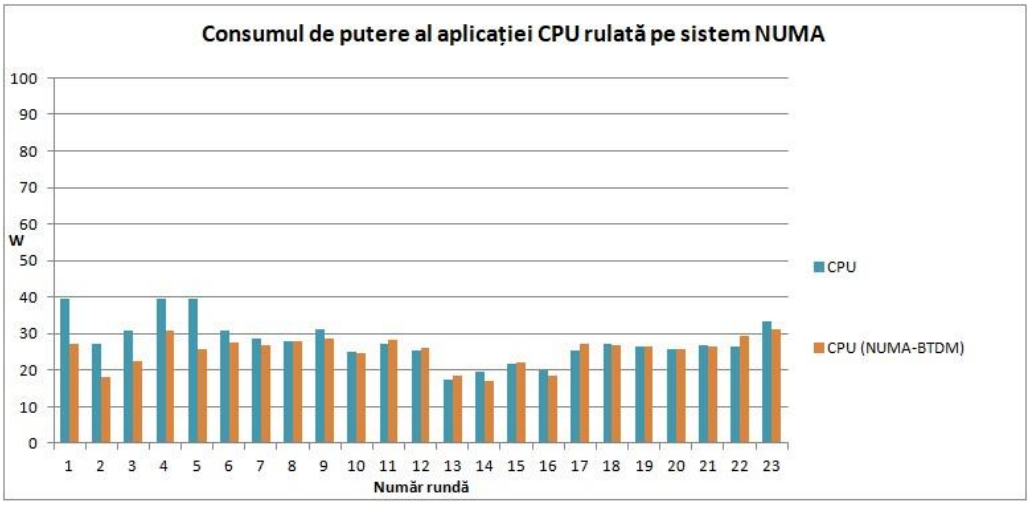

Figura 63. Consumul de putere (W) al aplicației **CPU** care rulează pe sistem NUMA în diferite runde

Fig. 64 arată optimizarea de consum de putere la rularea aplicației **CPU** pe sistem NUMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea este cuprinsă între valorile minimă de 0.08 W și maximă de 34.78 W. Optimizarea obținută se înregistrează la fiecare 1 s.

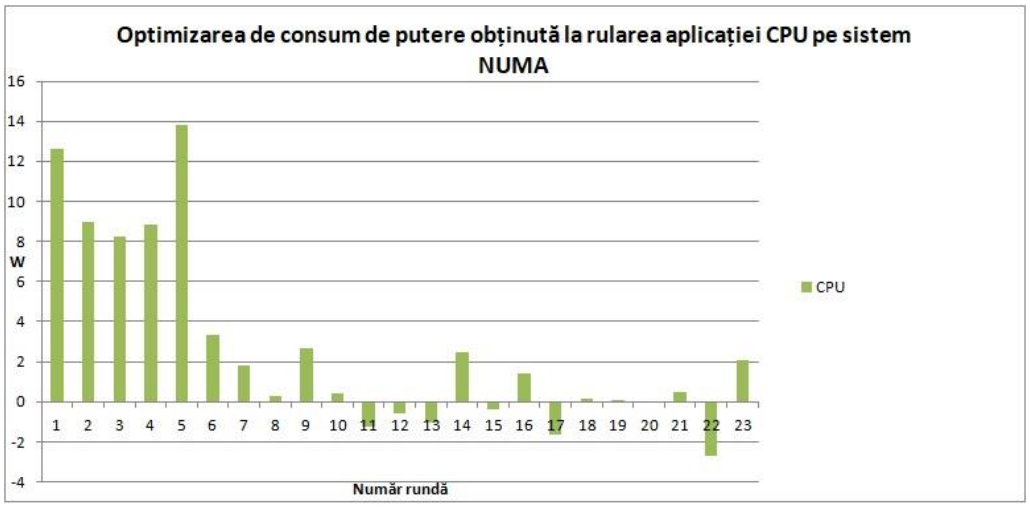

Figura 64. Optimizarea de consum de putere (W) al aplicației **CPU** care rulează pe sistem NUMA în diverse runde

Fig. 65 arată optimizarea de consum de putere, în procente, la rularea aplicației **CPU** pe sistem NUMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul de optimizare de consum de putere variază între valorile 0.02% și 13.80%.

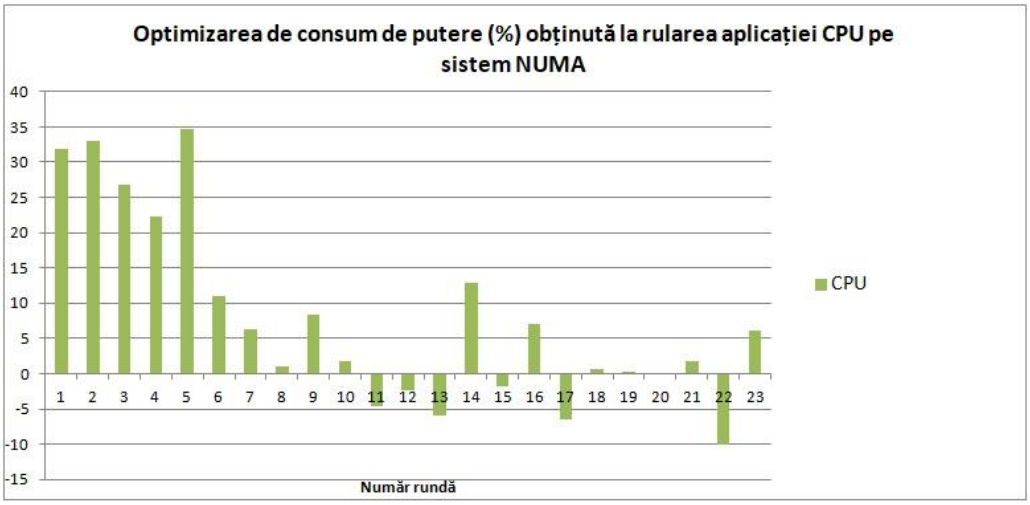

Figura 65. Optimizarea de consum de putere (%) al aplicației **CPU** care rulează pe sistem NUMA în diverse runde

*Consumul de putere al aplicațiilor **Flops** și **Iops**, obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atât atunci când aplicațiile rulează neoptimizat cât și atunci când rulează optimizat, pe sistem NUMA.*

Tabelul 15 prezintă rezultatele experimentale pe sistem NUMA ale aplicațiilor **Flops** [90] și **Iops** [90], atât în cazul în care acestea sunt optimizate folosind algoritmul ***NUMA-BTDM*** [55] și cât și în cazul în când nu sunt optimizate. Prima coloană indică numărul rundei în care rulează aplicațiile, stabilit static. Coloanele 2-5 prezintă comparativ consumul de putere al celor două aplicații optimizate și ale acelorași două aplicații, neoptimizate. Coloanele 6-7 descriu optimizarea în W a aplicațiilor **Flops** [90] și **Iops** [90] rulate pe sistem NUMA (obținută din diferența dintre consumul de putere la rularea aplicației neoptimizate și cel la înregistrat la rularea aplicației optimizate), iar coloanele 8-9 ilustrează această optimizare în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100. Se observă o optimizare relativ redusă atât în cazul **Flops** [90] cât și în cazul **Iops** [90], la aplicarea algoritmului ***NUMA-BTDM*** [55].

Din Fig. 66 se observă că, în majoritatea rundelor, consumul de putere al aplicației **Iops** [90] este mai mare atunci când aplicația nu este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când aceasta este optimizată, spre deosebire de aplicația **Flops** [90], unde optimizarea de consum de putere este prezentă doar în jumătate din runde și are valori mai mici ca la **Iops** [90].

Tabel 15. Rezultate experimentale pe sistem NUMA prezentând optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicațiilor **Flops** și **Iops**

Număr rundă	Consum de putere fără ***NUMA-BTDM***		Consum de putere cu ***NUMA-BTDM***		Optimizarea obținută (W)		Procent de optimizare	
	Flops	**Iops**	**Flops**	**Iops**	**Flops**	**Iops**	**Flops**	**Iops**
1	53.97	55.21	53.85	54.18	0.122	1.026	0.23	1.86
2	53.85	55.26	53.88	54.09	-0.031	1.164	-0.06	2.11
3	54.42	54.26	53.99	54.08	0.428	0.180	0.79	0.33
4	54.22	54.12	54.18	54.36	0.036	-0.244	0.07	-0.45
5	53.91	53.94	53.94	54.81	-0.030	-0.863	-0.06	-1.60
6	53.83	54.06	53.90	54.32	-0.070	-0.262	-0.13	-0.48
7	54.07	53.91	54.12	54.05	-0.055	-0.146	-0.10	-0.27
8	54.85	53.86	53.79	53.96	1.059	-0.097	1.93	-0.18
9	54.96	54.13	53.96	54.06	0.998	0.067	1.82	0.12
10	54.30	54.18	53.94	54.12	0.354	0.060	0.65	0.11
11	54.13	54.91	54.14	54.08	-0.011	0.826	-0.02	1.51
12	53.89	54.10	54.14	54.30	-0.255	-0.202	-0.47	-0.37
13	53.54	54.09	53.62	54.10	-0.076	-0.008	-0.14	-0.02
14	54.05	55.14	54.06	54.21	-0.009	0.932	-0.02	1.69
15	54.25	54.60	53.98	54.56	0.267	0.041	0.49	0.07
16	54.66	54.24	53.93	54.22	0.727	0.018	1.33	0.03
17	54.17	54.50	53.97	54.11	0.198	0.386	0.36	0.71
18	54.06	53.97	53.90	54.89	0.162	-0.918	0.30	-1.70
19	54.40	54.67	53.82	54.15	0.577	0.516	1.06	0.94
20	55.01	54.43	54.02	54.44	0.990	-0.003	1.80	-0.01
21	54.33	54.14	54.25	54.23	0.088	-0.084	0.16	-0.15
22	53.86	54.94	54.74	54.17	-0.878	0.767	-1.63	1.40
23	53.90	54.18	54.27	54.17	-0.376	0.007	-0.70	0.01
24	53.93	55.09	53.88	54.34	0.050	0.744	0.09	1.35
25	54.35	54.15	54.12	54.25	0.225	-0.095	0.41	-0.18
26	54.00	54.19	54.13	54.41	-0.123	-0.219	-0.23	-0.40
27	53.89	54.98	54.54	54.95	-0.649	0.033	-1.20	0.06
28	54.04	55.08	54.86	54.93	-0.824	0.151	-1.52	0.27
29	54.12	55.17	54.47	54.55	-0.350	0.619	-0.65	1.12
30	53.88	54.79	54.01	55.07	-0.131	-0.286	-0.24	-0.52
31	54.05	54.84	54.00	54.25	0.049	0.597	0.09	1.09
32	54.37	53.97	53.85	54.05	0.524	-0.085	0.96	-0.16
33	54.47	54.31	53.69	54.15	0.785	0.163	1.44	0.30
34	54.33	55.06	54.00	54.19	0.323	0.867	0.59	1.57
35	54.05	54.26	54.18	54.37	-0.128	-0.108	-0.24	-0.20
36	54.11	54.32	54.19	54.25	-0.083	0.071	-0.15	0.13
37	54.49	54.57	54.05	54.23	0.445	0.341	0.82	0.63
38	54.25	54.83	54.03	54.53	0.221	0.292	0.41	0.53
39	54.72	35.82	54.21	36.52	0.510	-0.695	0.93	-1.94
40	54.81	-0.34	53.98	0.05	0.838	-0.394	1.53	115.46

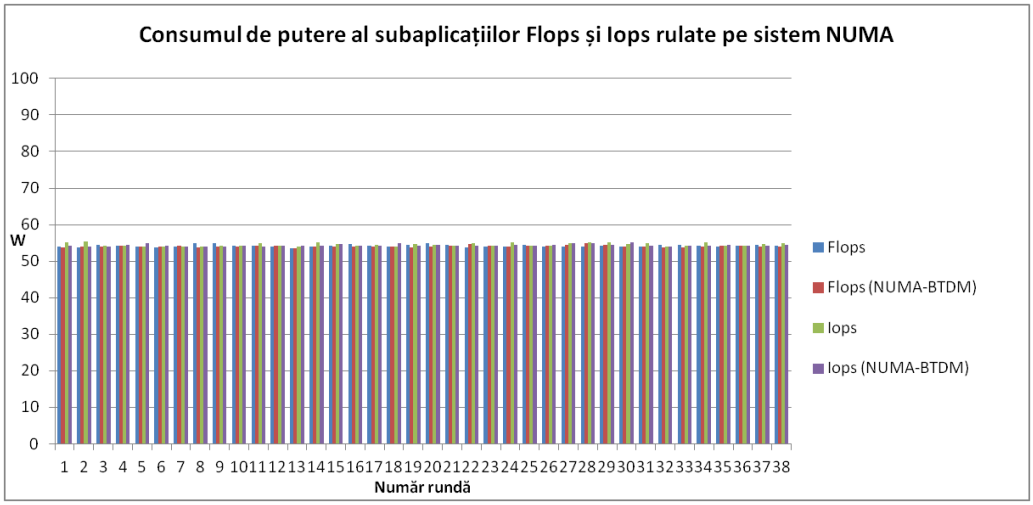

Figura 66. Consumul de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem NUMA în diferite runde

Fig. 67 arată optimizarea de consum de putere la rularea aplicațiilor **Flops** [90] și **Iops** [90] pe sistem NUMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea este cuprinsă între valorile minimă de 0.04 W și maximă de 1.06 W pentru **Flops** [90] și valorile minimă 0.01 W și maximă 1.16 W pentru **Iops** [90]. Optimizarea obținută se înregistrează la fiecare 1 s.

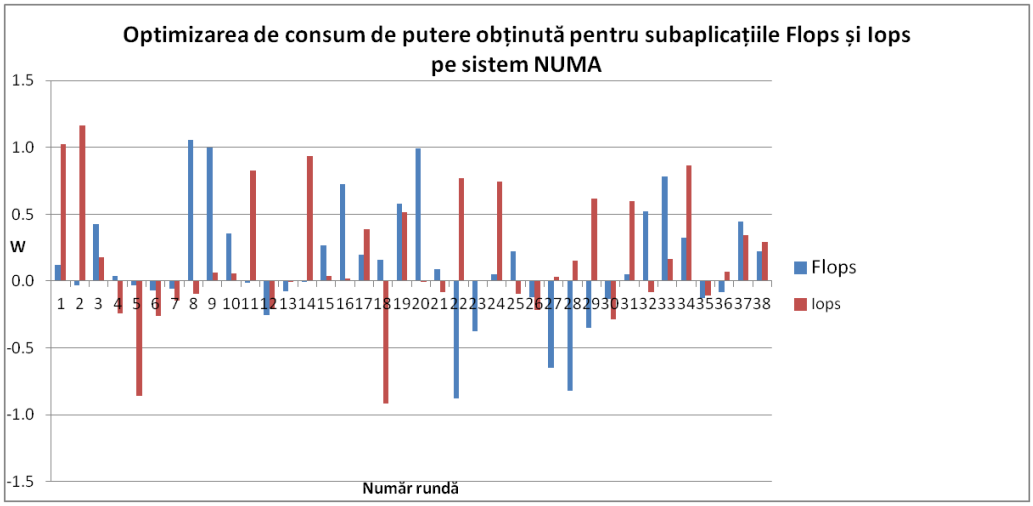

Figura 67. Optimizarea de consum de putere (W) al aplicațiilor **Flops** și **Iops** care rulează pe sistem NUMA în diverse runde

Fig. 68 arată optimizarea de consum de putere, în procente, la rularea aplicațiilor **Flops** [90] și **Iops** [90] pe sistem NUMA, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul de optimizare de consum de putere variază între valorile 0.067% și 1.931% pentru **Flops** [90] și 0.013% și 2.106% pentru **Iops** [90].

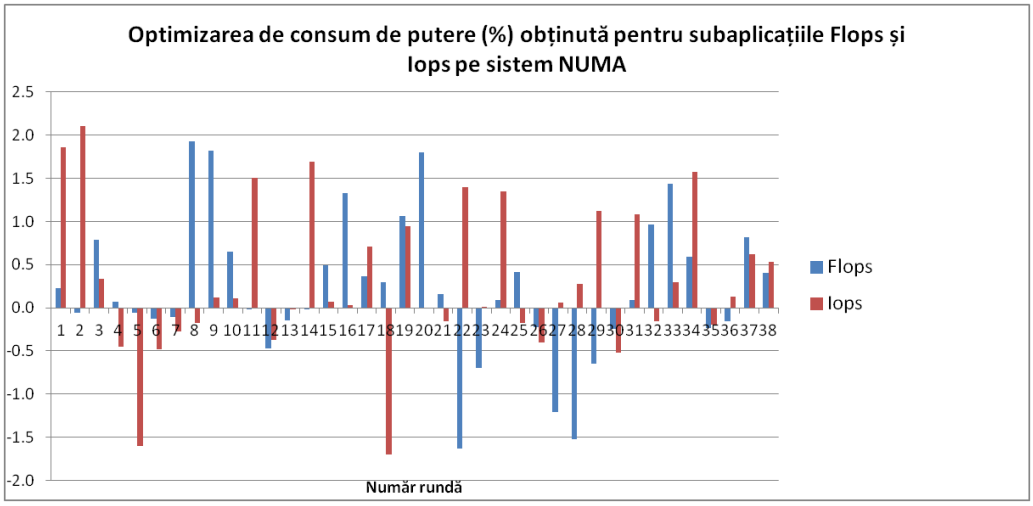

Figura 68. Optimizarea de consum de putere (%) al aplicațiilor **Flops** și **Iops** care rulează pe sistem NUMA în diverse runde

4.5.4 Comparație între rezultatele experimentale pe sisteme UMA și cele pe sisteme NUMA

4.5.4.1Timp de execuție

Tabelul 16 prezintă comparativ rezultatele experimentale de timp de execuție, pentru aplicația de referință **CPU**, rulată pe sistem UMA și sistem NUMA. Din tabel reies următoarele concluzii:

3. timpul de execuție pe sistem UMA este în medie mai mic decât pe sistem NUMA pentru aplicațiile **Flops** [90] și **Iops** [90] și mai mare pe sistem UMA decât pe NUMA pentru aplicația **CPU** [90]
4. timpul de execuție mediu, atât pe sistem UMA cât și pe sistem NUMA, al aplicației **CPU** cu algoritmul ***NUMA-BTDM*** [55] aplicat este mai mic decât cel al aplicației fără algoritm aplicat iar în cazul aplicațiilor **Flops** [90] și **Iops** [90] cu număr mare de fire de execuție se obține reversul, degradarea de performanță fiind însă nesemnificativă (de ordinul milisecundelor raportat la un timp total de execuție de 10 minute) și deci putându-se afirma că timpul de execuție se păstrează
5. în cazul aplicațiilor **CPU** [90] și **Flops** [90], măsurătorile obținute fără algoritmul ***NUMA-BTDM*** [55] aplicat sunt mai disipate față de valoarea medie decât cele obținute atunci când algoritmul este aplicat, atât pe sistem UMA cât și pe sistem NUMA, iar la aplicația **Iops** [90] se obține reversul.

Tabel 16. Comparație între rezultatele experimentale de timp de execuție obținute pe sisteme UMA și pe sisteme NUMA pentru aplicația **CPU**

Categorie valori	**Tip sistem**	**Valorile fiecărei categorii pentru măsurătorile obținute la rularea aplicației CPU**					
		Fără NUMA-BTDM			**Cu NUMA-BTDM**		
		CPU	**Flops**	**Iops**	**CPU**	**Flops**	**Iops**
Minime	UMA	0.58	600.46	600.47	0.57	600.47	600.47
	NUMA	0.55	600.47	600.48	0.56	600.48	600.47
Medii	UMA	0.59	600.48	600.47	0.58	600.49	600.47
	NUMA	0.57	600.50	600.50	0.57	600.51	600.50
Maxime	UMA	0.60	600.54	600.48	0.59	600.54	600.49
	NUMA	0.59	600.57	600.52	0.58	600.57	600.52
Varianță	UMA	0.00008	0.00074	0.00002	0.00004	0.00071	0.00007
	NUMA	0.00006	0.00029	0.00008	0.00003	0.00019	0.00011
Deviație standard	UMA	0.0087	0.0271	0.0039	0.0066	0.0266	0.0081
	NUMA	0.0080	0.0170	0.0086	0.0052	0.0137	0.0105

4.5.4.2 Consum de putere

Tabelul 17 prezintă comparativ rezultatele experimentale, pe sistem UMA și sistem NUMA, pentru aplicația **CPU**. Prima coloană indică număr rundei în care rulează aplicația. Aplicațiile **CPU** [90], **Flops** [90] și **Iops** [90] rulează timp de 40 de runde fiecare, în tabel fiind trecute doar primele 20 de runde. A doua coloană descrie optimizarea în W, obținută atât în cazul sistemului UMA cât și în cazul sistemului NUMA, pentru fiecare din cele trei aplicații componente **CPU** [90], **Flops** [90] și **Iops** [90] ale aplicației de referință **CPU** [90], iar ultima coloană descrie această optimizare în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100.

Tabel 17. Rezultate experimentale prezentând comparativ pentru sistem UMA și sistem NUMA, optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației de referință **CPU**

Rundă	**Optimizarea obținută (W)**						**Procent de optimizare**					
	UMA			**NUMA**			**UMA**			**NUMA**		
	CPU	**Flops**	**Iops**	**CPU**	**Flops**	**Iops**	**CPU**	**Flops**	**Iops**	**CPU**	**Flops**	**Iops**
1	17.0	2.8	2.8	12.6	0.1	1.0	37.9	4.9	4.9	31.8	0.2	1.9
2	13.7	1.5	1.5	9.0	0.0	1.2	32.9	2.7	2.7	33.1	-0.1	2.1
3	8.0	0.1	0.1	8.2	0.4	0.2	24.2	0.2	0.2	26.8	0.8	0.3
4	11.4	-0.5	-0.5	8.9	0.0	-0.2	44.9	-0.9	-0.9	22.4	0.1	-0.5
5	2.9	0.0	0.0	13.8	0.0	-0.9	9.5	0.1	0.1	34.8	-0.1	-1.6
6	8.3	-0.2	-0.2	3.4	-0.1	-0.3	30.0	-0.3	-0.3	10.9	-0.1	-0.5
7	5.3	-0.3	-0.3	1.8	-0.1	-0.1	14.9	-0.6	-0.6	6.3	-0.1	-0.3
8	10.0	0.0	0.0	0.3	1.1	-0.1	26.6	0.1	0.1	1.0	1.9	-0.2
9	9.3	0.6	0.6	2.6	1.0	0.1	26.0	1.0	1.0	8.4	1.8	0.1
10	9.8	0.6	0.6	0.4	0.4	0.1	26.6	1.0	1.0	1.8	0.7	0.1
11	8.5	0.5	0.5	-1.3	0.0	0.8	23.4	1.0	1.0	-4.6	0.0	1.5
12	8.9	0.2	0.2	-0.6	-0.3	-0.2	25.6	0.4	0.4	-2.3	-0.5	-0.4
13	6.6	0.4	0.4	-1.0	-0.1	0.0	19.1	0.8	0.8	-5.9	-0.1	0.0
14	7.6	-0.4	-0.4	2.5	0.0	0.9	21.1	-0.8	-0.8	12.8	0.0	1.7
15	9.4	0.3	0.3	-0.4	0.3	0.0	25.2	0.5	0.5	-1.7	0.5	0.1
16	10.2	-0.1	-0.1	1.4	0.7	0.0	27.4	-0.2	-0.2	7.0	1.3	0.0
17	8.4	-0.2	-0.2	-1.7	0.2	0.4	22.0	-0.4	-0.4	-6.5	0.4	0.7
18	8.7	0.2	0.2	0.2	0.2	-0.9	23.1	0.4	0.4	0.6	0.3	-1.7
19	9.0	0.2	0.2	0.0	0.6	0.5	23.9	0.3	0.3	0.1	1.1	0.9
20	8.6	0.3	0.3	-0.1	1.0	0.0	23.6	0.5	0.5	-0.2	1.8	0.0

Fig. 69 arată optimizarea de consum de putere, în procente, la rularea aplicației **CPU**, produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55] și prezentată comparativ pentru sistemele UMA și NUMA. Procentul maxim de optimizare de consum de putere este 44.93% obținut pentru sistem UMA și 34.77% pentru sistem NUMA, iar procentul mediu este 25.43% pentru sistem UMA și 7.58% pentru sistem NUMA. Prin urmare această aplicație rulează mai optim din puncte de vedere al consumului de energie pe sistem UMA decât pe NUMA, după aplicarea algoritmului ***NUMA-BTDM*** [55].

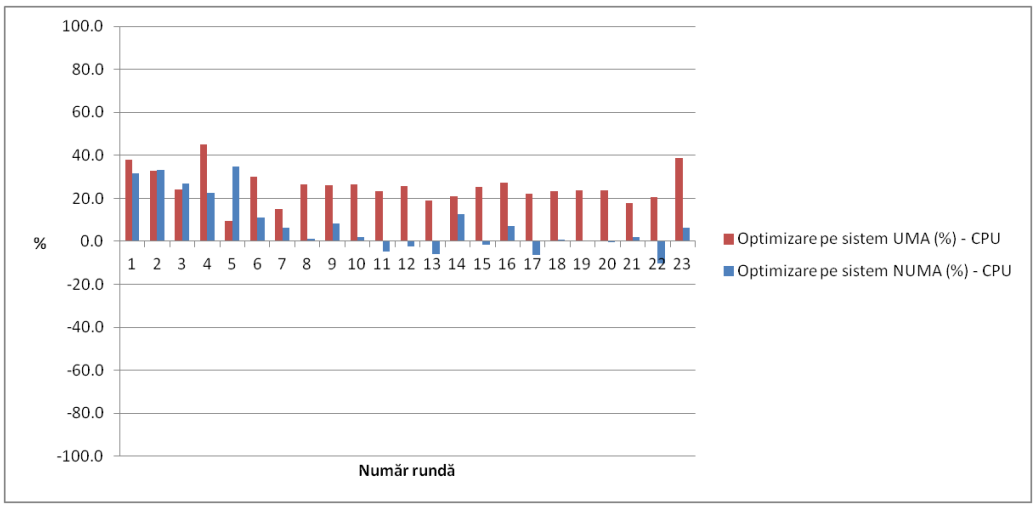

Figura 69. Optimizarea de consum de putere în procente al aplicației **CPU** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA

Fig. 70 descrie optimizarea de consum de putere, în procente, la rularea aplicației **Flops** [90], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55], prezentată comparativ pentru sistemele UMA și NUMA. Procentul maxim de optimizare de consum de putere este 2.81% la execuția pe sistem UMA și 1.06% la sistem NUMA, iar procentul mediu este 0.13% pentru sistem UMA și 0.15% pentru sistem NUMA. Prin urmare această aplicație rulează mai optim din puncte de vedere al consumului de energie pe sistem NUMA decât pe UMA, după aplicarea algoritmului ***NUMA-BTDM*** [55].

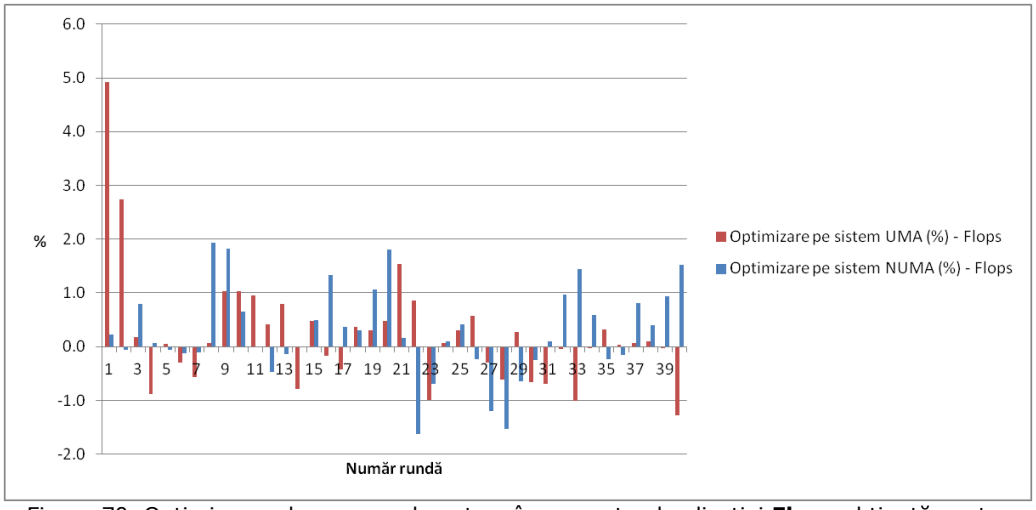

Figura 70. Optimizarea de consum de putere în procente al aplicației **Flops** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA

Fig. 71 arată optimizarea de consum de putere, în procente, la rularea aplicației **Iops** [90], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55], prezentată comparativ pentru sistemele UMA și NUMA. Procentul maxim de optimizare de consum de putere este 2.81% obținut la rularea pe sistem UMA și 1.16% obținut pentru sistem NUMA, iar procentul mediu de optimizare este 0.13% atât pentru sistem UMA cât și pentru NUMA.

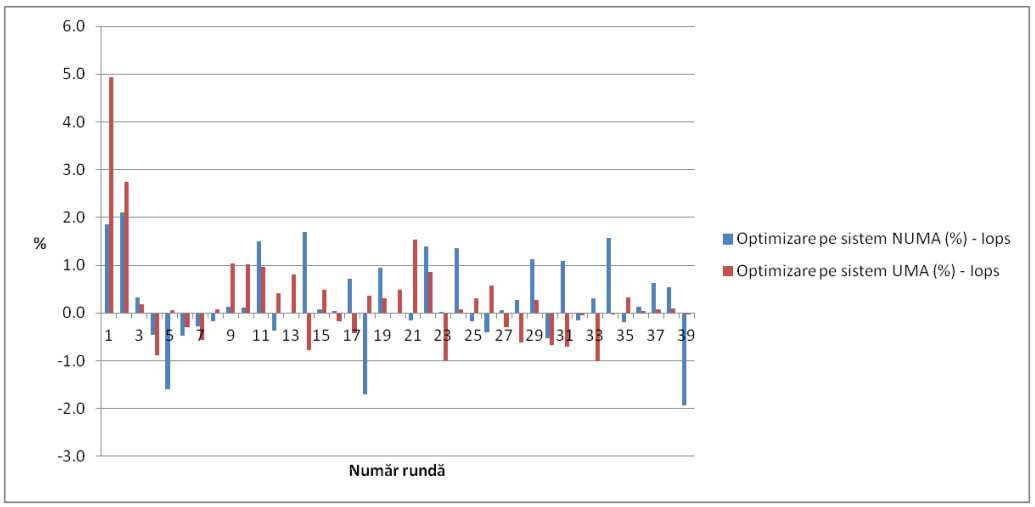

Figura 71. Optimizarea de consum de putere în procente al aplicației **Iops** obținută pentru diferite numere de fire de execuție și prezentată comparativ pentru sistemele UMA și NUMA

Tabelul 18 descrie comparativ pentru sistemul UMA și sistemul NUMA, rezultatele experimentale obținute din cele două surse, utilitarul software tubostat și dispozitivul hardware specializat WattsUp, împărțite în 5 categorii: minime, medii,

Tabel 18. Comparație între rezultatele experimentale pe sistem UMA și cele pe sistem NUMA pentru aplicația **CPU**

Categorie valori	**Aplicație**	**Tip sistem**	**Comparație între măsurătorile obținute din cele două surse, cu și fără aplicarea algoritmului NUMA-BTDM**	
			Utilitarul *turbostat*	**Dispozitivul WattsUp**
Minime	**CPU**	NUMA	Minimul atunci când ***NUMA-BTDM*** [55] este aplicat, 45.40 W, este mai mic decât 45.83 W, minimul atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Minimul când ***NUMA-BTDM*** [55] a fost aplicat este 137, iar minimul când ***NUMA-BTDM*** [55] nu a fost aplicat este 137.9 W, rezultând o optimizare de ~1 W.
		UMA	Minimul atunci când ***NUMA-BTDM*** [55] este aplicat, 46.44 W, este mai mic decât 54.25 W, minimul atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare cu până la 7.81 W raportat la valorile minime.	Minimul atunci când ***NUMA-BTDM*** [55] este aplicat, 136.10 W, este mai mic decât 145.10 W, minimul atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare cu până la 9 W raportat la valorile minime.
	Flops	NUMA	Media minimelor din fiecare rundă este 65.10 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 65.38 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, optimizarea fiind redusă.	Media minimelor din fiecare rundă este 155.48 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 156.14 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare de ~0.6 W.
		UMA	Media minimelor din fiecare rundă este 65.21 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 65 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Media minimelor din fiecare rundă este 155.78 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 156.1 W, atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare de ~0.3 W/s.
	Iops	NUMA	Media minimelor din fiecare rundă este 62.57 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, cu puțin mai mică decât 62.89 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Media minimelor din fiecare rundă este 155.47 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 150.01 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat.
		UMA	Media minimelor din fiecare rundă este 65.21 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 65 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Media minimelor din fiecare rundă este 159.64 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 154.70 W, rezultați atunci când ***NUMA-BTDM*** [55] nu este aplicat.

Medii	**CPU**	NUMA	Media atunci când ***NUMA-BTDM*** [55] este aplicat, 47.85 W, este mai mică decât 50.46 W, media atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare de până la 2.61 W în medie.	Media când ***NUMA-BTDM*** [55] a fost aplicat este 149.46 W și media atunci când ***NUMA-BTDM*** [55] nu a fost aplicat este 143.03 W.
		UMA	Media atunci când ***NUMA-BTDM*** [55] este aplicat, 47.85 W, este mai mică decât 57.14 W, media atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare cu până la 9.29 W în medie	Media atunci când ***NUMA-BTDM*** [55] este aplicat, 140.92 W, este mai mică decât 148.48 W, media atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare cu până la 7.56 W/s în medie.
	Flops	NUMA	Media rundelor este 72.37 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 72.51 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, optimizarea fiind redusă.	Media rundelor este 161.44 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 161.70 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare de până la 0.24 W/s în medie.
		UMA	Media rundelor este 72.43 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 72.56 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare redusă.	Media rundelor este 162.32 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 161.76 W, rezultați atunci când ***NUMA-BTDM*** [55] nu este aplicat.
	Iops	NUMA	Media rundelor este 70.58 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 70.71 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, optimizarea fiind redusă.	Media rundelor este 160.99 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 160.56 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat.
		UMA	Media rundelor este 72.43 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 72.56 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare redusă.	Media rundelor este 161.43 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 160.91 W, rezultați atunci când ***NUMA-BTDM*** [55] nu este aplicat.

Maxime	**CPU**	NUMA	Maximul atunci când ***NUMA-BTDM*** [55] a fost aplicat 52.38 W, este mai mic decât 61.63 W, maxim obținut atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Maximul atunci când algoritmul este aplicat de 160.50 W, este mai mare decât maximul de 158.10 W obținut atunci când algoritmul nu este aplicat.
		UMA	Maximul atunci când algoritmul ***NUMA-BTDM*** [55] a fost aplicat de 53.59 W, este mai mic decât cel obținut atunci când ***NUMA-BTDM*** [55] nu este aplicat, 65.10 W.	Maximul 157.1 W atunci când algoritmul este aplicat, este mai mare decât maximul 163.3 W atunci când algoritmul nu este aplicat, optimizarea maximă fiind 6.2 W/s.
	Flops	NUMA	Media maximelor din fiecare rundă este 73.93 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 75.19 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, optimizarea fiind de până la 1.26 W/s raportat la valorile maxime.	Media maximelor din fiecare rundă de 163.85 W obținută atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este mai mare decât 163.38 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat.
		UMA	Media maximelor din fiecare rundă este 74.79 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 75.54 W, atunci când algoritmul nu este aplicat, optimizarea fiind de maxim 0.75 W/s.	Media maximelor din fiecare rundă este 165.65 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, care este mai mare decât media 163.19 W, rezultată atunci când ***NUMA-BTDM*** [55] nu este aplicat.
	Iops	NUMA	Media maximelor din fiecare rundă este 72.25 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mică decât 73.95 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, optimizarea fiind de până la 1.7 W/s raportat la valorile maxime.	Media maximelor din fiecare rundă de 162.61 W obținută atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este mai mică decât 164.02 W, medie obținută atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare de până la 1.41 W/s raportat la valorile maxime.
		UMA	Media maximelor din fiecare rundă de 74.79 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, este mai mică decât 75.54 W, obținuți atunci când ***NUMA-BTDM*** [55] nu este aplicat, optimizarea fiind de până la 0.75 W/s, raportat la valorile maxime.	Media maximelor din fiecare rundă este 162.47 W atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, mai mare decât 163.64 W, medie obținută atunci când ***NUMA-BTDM*** [55] nu este aplicat, rezultând o optimizare de până la 1.17 W/s, raportat la valorile maxime.

Varianță	**CPU**	NUMA	Varianța atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 3.15. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța este 30.08.	Varianța atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 119.16. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța este 45.63.
		UMA	Varianța atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 2.16. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța este 7.35.	Varianța atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 15.02. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța este 32.48.
	Flops	NUMA	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.12 iar valoarea maximă este 0.31. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.19 și varianța maximă este 0.46.	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.201. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.34 și varianța maximă este 1.38.
		UMA	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.18 iar valoarea maximă este 0.38. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.30 și valoarea maximă este 2.22.	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 2.42 iar valoarea maximă este 87.37, iar atunci când algoritmul nu este aplicat, variantele medie și maximă sunt 9.02, respectiv 128.63.
	Iops	NUMA	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.142. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.231.	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.165. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.543.
		UMA	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.18 iar valoarea maximă este 0.38. Atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.30 și varianța maximă este 2.21.	Varianța medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.07 iar valoarea maximă a varianței este 0.99. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, varianța medie este 0.490 iar valoarea maximă a varianței este 3.807.

Deviație standard	**CPU**	NUMA	Deviația standard atunci când ***NUMA-BTDM*** [55] este aplicat este 1.81. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, deviația standard este 5.49, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Deviația standard atunci când ***NUMA-BTDM*** [55] este aplicat este 0.394. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, deviația standard este 6.76, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** [55] este aplicat.
		UMA	Deviația standard atunci când ***NUMA-BTDM*** [55] este aplicat este 1.47. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, deviația standard medie este 2.71, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 3.87. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, deviația standard este 5.69, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** [55] nu este aplicat.
	Flops	NUMA	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.35. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, deviația standard medie este 0.42, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** [55] nu este aplicat.	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.39. Atunci când ***NUMA-BTDM*** [55] nu este aplicat, deviația standard medie este 0.51, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** [55] nu este aplicat.
		UMA	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.41. Atunci când algoritmul nu este aplicat, deviația standard medie este 0.52, măsurătorile fiind mai disipate față de valoarea medie atunci când algoritmul nu este aplicat.	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.66. Atunci când algoritmul nu este aplicat, deviația standard medie este 1.54, măsurătorile fiind mai disipate față de valoarea medie atunci când algoritmul nu este aplicat.
	Iops	NUMA	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 1. Atunci când algoritmul nu este aplicat, deviația standard medie este 1.10, măsurătorile fiind mai disipate față de valoarea medie atunci când algoritmul nu este aplicat.	Deviația standard medie atunci când ***NUMA-BTDM*** [55] este aplicat este 0.357. Atunci când algoritmul nu este aplicat, deviația standard medie este 0.809, măsurătorile fiind mai disipate față de valoarea medie atunci când algoritmul nu este aplicat.

		UMA	Deviația standard medie atunci când algoritmul ***NUMA-BTDM*** este aplicat este 0.41. Atunci când algoritmul ***NUMA-BTDM*** nu este aplicat, deviația standard medie este 0.52, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** nu este aplicat.	Deviația standard medie atunci când algoritmul ***NUMA-BTDM*** este aplicat este 0.12. Atunci când algoritmul ***NUMA-BTDM*** nu este aplicat, deviația standard medie este 1.09, măsurătorile fiind mai disipate față de valoarea medie atunci când ***NUMA-BTDM*** nu este aplicat.

maxime, varianța și deviația standard, calculate pentru fiecare din aplicațiile **CPU** [90], **Flops** [90] și **Iops** [90]. A fost obținută câte o valoare din fiecare categorie pentru fiecare rundă în parte, iar rezultatele din tabel prezintă concluziile luând în considerare toate rundele.

4.5.5 Concluzii ale rezultatelor experimentale pentru aplicația de referință CPU

Timpul de execuție pe sistem NUMA pentru aplicația de referință **CPU** [90], compusă din aplicațiile **CPU** [90], **Flops** [90] și **Iops** [90], este optimizat cu o valoare foarte mică (0.03 s raportat la timpul total de execuție de 600 s al aplicațiilor **Flops** [90] și **Iops** [90] și 0.02 s raportat la timpul total de execuție de 0.59 s al aplicației **CPU** [90]), putându-se considera că timpul de execuție se păstrează. În cazul aplicațiilor **Flops** [90] și **Iops** [90], optimizarea produsă de algoritmul ***NUMA-BTDM*** [55] este mai mare, în realitate, decât cea rezultată în urma experimentelor (0.03 s), deoarece maparea unui număr mare de fire de execuție (2400), în cazul fiecăreia dintre cele doua aplicații, nu produce o degradare a timpului de execuție final (cum ar fi de așteptat atunci când se mapează un număr mare de fire de execuție prin apeluri ***pthread_setaffinity_np***), degradarea fiind acoperită de câștigul de timp de execuție rezultat în urma aplicării algoritmului ***NUMA-BTDM*** [55], rezultând în final un câștig de performanță.

Păstrând același timp de execuție, la rularea aplicației **CPU** [90] se observă o optimizare a consumului de putere al sistemului NUMA cu 0.9 W/s și al sistemului UMA cu 7.56 W/s, conform Tabelului 17. 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. Conform Tabelului 18, varianța optimizării este cu mult mai mare decât optimizarea în sine, deci aplicația **CPU** [90] nu se consideră optimizată. Din Tabelul 17, rezultă că optimizarea execuției aplicației **Flops** în urma aplicării algoritmului ***NUMA-BTDM*** [55] pe sistem NUMA de 0.6 W/s și de 0.3 W/s pe sistem UMA. Aceasta se consideră a fi o optimizare având în vedere că varianța optimizării este mai mică decât optimizarea în sine (varianța optimizării are valoarea 0.07 la NUMA, respectiv 0.12 la UMA). Optimizarea mică se datorează numărului mediu de fire de execuție autonome (2400), în acest caz algoritmii asigurând îndeplinirea doar a criteriului de echilibrare a execuției. Aplicația **Iops** [90] nu este optimizată de algoritmul ***NUMA-BTDM*** [55], rezultând un consum de putere mai mare în medie atunci când algoritmul este aplicat decât atunci când acesta nu este aplicat.

4.6 Aplicația de referință Context Switch

4.6.1 Descrierea aplicației

Aplicația **Context Switch** [91] conține două programe principale care ambele creează câte un singur fir de execuție în funcția main, considerat autonom. Cele două programe principale intră în execuție alternativ.

4.6.2 Rezultate experimentale pe sisteme UMA

4.6.2.1 Timp de execuție

Tabelul 19 prezintă comparativ timpii de execuție obținuți atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat și atunci când algoritmul este aplicat asupra aplicației de referință **Context Switch** [91]. În Tabelul 19, prima coloană reprezintă numărul rundei în care este rulată aplicația **Context Switch** [91], coloana a doua reprezintă timpul de execuție al aplicației **Context Switch** [91] fără algoritmul ***NUMA-BTDM*** [55] aplicat, iar cea de-a treia coloană, timpul de execuție pentru aplicația cu algoritmul ***NUMA-BTDM*** [55] aplicat (la compilare). Se observă că timpul de execuție pe sistem UMA crește atunci când asupra aplicației este aplicat algoritmul ***NUMA-BTDM*** [55] anterior rulării acesteia. Media timpilor de execuție pentru toate rundele atunci când algoritmul nu este aplicat este 41.61 W, mai mică decât media 47.23 W, obținută când algoritmul este aplicat.

Tabel 19. Rezultate experimentale de timp de execuție al aplicației **Context Switch** rulată pe sistem UMA, prezentate comparativ atunci când algoritmul ***NUMA-BTDM*** este aplicat și atunci când nu este aplicat

Număr rundă	**Timp de execuție fără algoritmul NUMA-BTDM aplicat**	**Timp de execuție cu algoritmul NUMA-BTDM aplicat**
1	45.191	48.482
2	40.101	47.949
3	40.6	46.688
4	40.835	47.391
5	42.851	47.152
6	41.252	48.566
7	41.833	47.335
8	40.48	46.851
9	41.771	45.105
10	40.726	46.96
11	40.524	47.273
12	39.936	48.614
13	40.605	46.814
14	42.441	47.365
15	42.707	47.745
16	43.479	47.601

17	43.924	44.433
18	41.14	46.487
19	40.901	48.103
20	42.733	45.642
21	40.642	46.631
22	46.517	46.299
23	44.132	48.148
24	44.462	47.721
25	43.31	47.536
26	40.863	46.438
27	40.639	47.367
28	40.304	47.285
29	41.519	48.6
30	40.859	47.308
31	40.843	47.716
32	41.18	47.208
33	40.714	47.692
34	41.856	46.33
35	40.358	46.662
36	40.277	47.951
37	40.821	48.613
38	40.146	47.582
39	39.773	46.821
40	41.493	46.777

Fig. 72 prezintă comparativ timpii de execuție obținuți în cele 40 de runde de rulare ale aplicației **Context Switch** [91] cu algoritmul ***NUMA-BTDM*** [55] aplicat, cât și fără algoritm aplicat. Din figură se observă o degradare a timpului de execuție al aplicației **Context Switch** [91] pe sistem UMA în urma aplicării algoritmului ***NUMA-BTDM*** [55], pentru majoritatea rundelor.

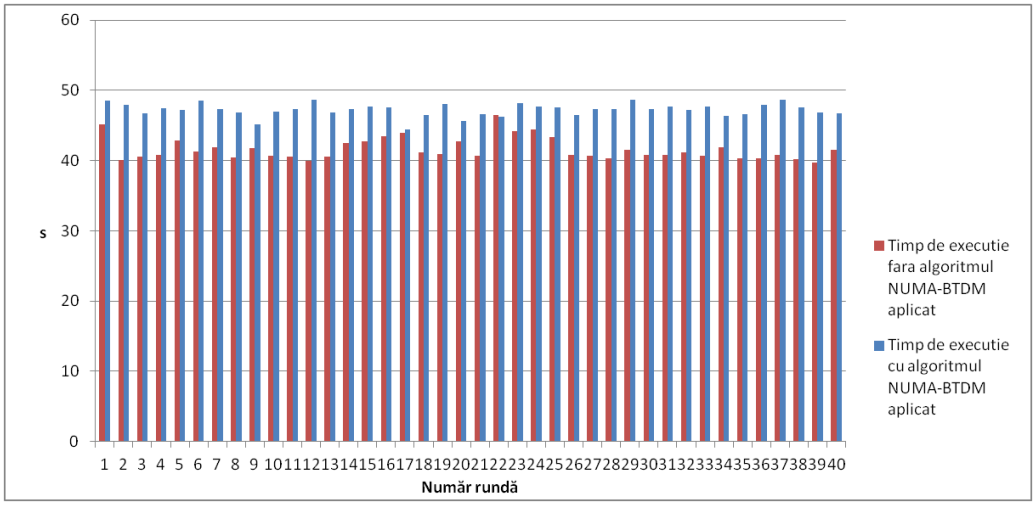

Figura 72. Timpul de execuție al aplicației **Context Switch** pe sistem UMA, atât atunci când algoritmul ***NUMA-BTDM*** este aplicat, cât și când algoritmul nu este aplicat

4.6.2.2 Consum de putere

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când aplicația rulează neoptimizat și atunci când rulează optimizat*

Din Fig. 73 se observă că atunci când aplicația **Context Switch** [91] rulează pe un sistem UMA, consumul de putere al sistemului este mai mic în mai multe de jumătate dintre runde atunci când aplicația este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat, în restul rundelor consumul fiind aproximativ egal în cele două cazuri. În figură, abscisa reprezintă numărul rundei iar ordonata reprezintă consumul mediu de putere în W al întregului sistem, pentru fiecare rundă.

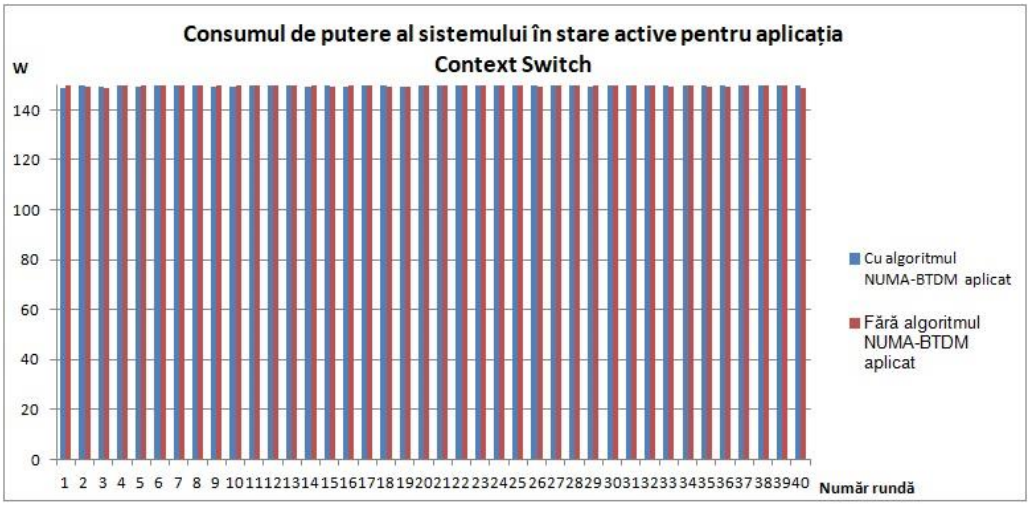

Figura 73. Consumul mediu de putere (W) al sistemului UMA obținut în fiecare rundă pentru aplicația **Context Switch**

Din Fig. 74 se observă că aplicația **Context Switch** [91] rulează pe sistem UMA cu consum de putere al CPU-ului mai mic, în majoritatea rundelor, atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat. În figură, abscisa are aceeași semnificație ca în Fig. 73, iar ordonata se referă la consumul mediu de putere al CPU-ului în fiecare rundă.

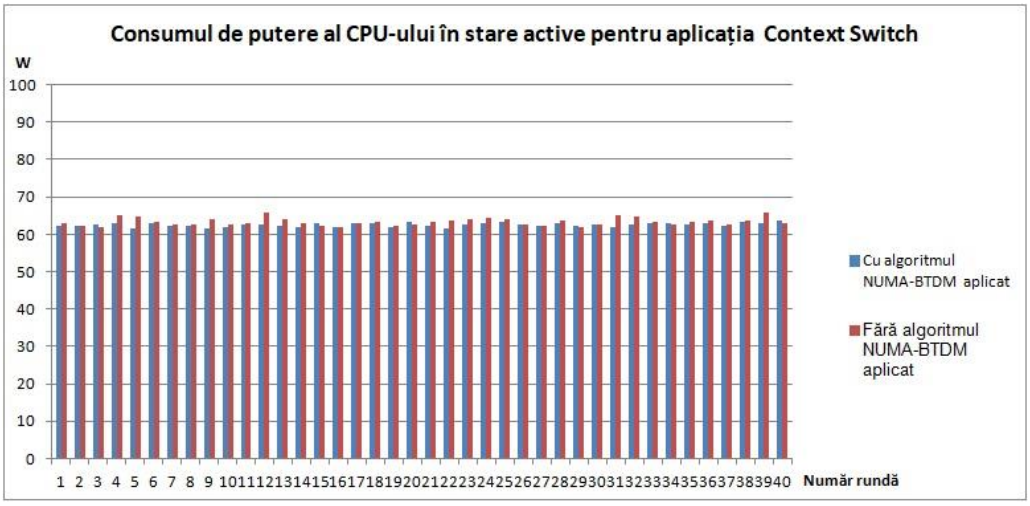

Figura 74. Consumul mediu de putere (W) al CPU-ului din sistemul UMA obținut în fiecare rundă la rularea aplicației **Context Switch**

Fig. 75 ilustrează consumul de putere al CPU-ului și cel al întregului sistem pentru o comparație valorică, atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când nu este aplicat. Se observă că măsurătorile sunt corelate și consumul CPU-ului este aproximativ 41% din cel al întregului sistem pentru aplicația **Context Switch** [91].

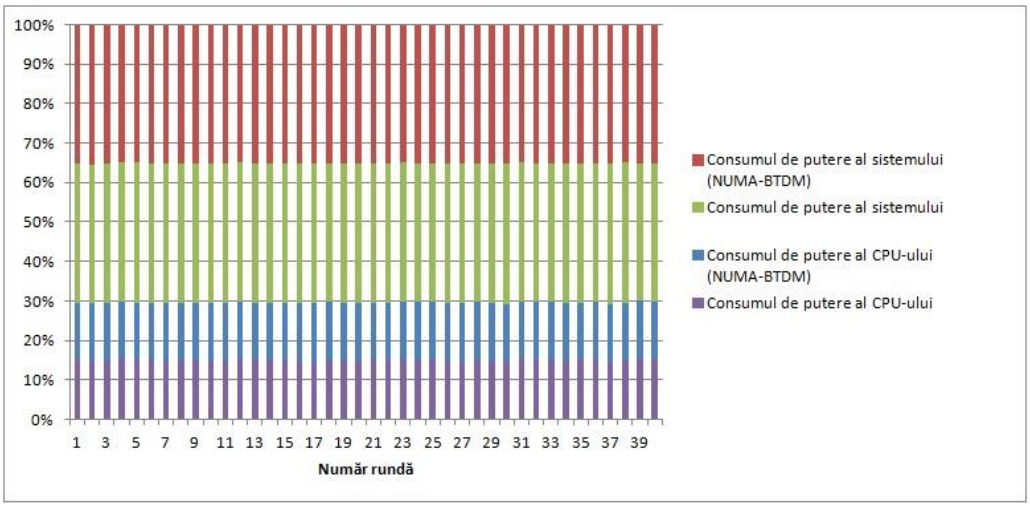

Figura 75. Consumul mediu de putere (W) al CPU-ului și al întregului sistem UMA obținut în fiecare rundă la rularea aplicației **Context Switch**

Cu cât diferența între consumul de putere al întregului sistem UMA și cel al CPU-ului este mai mare, cu atât aplicația rulează mai optim. Fig. 76 ilustrează diferența pentru aplicația **Context Switch** [91], cu și fără algoritmul ***NUMA-BTDM*** [55] aplicat. Diferența are o valoare aproape constantă pentru toate rundele,

media diferenței pentru cele 40 de runde fiind 87.05 W atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat și 87.31 W atunci când algoritmul este aplicat.

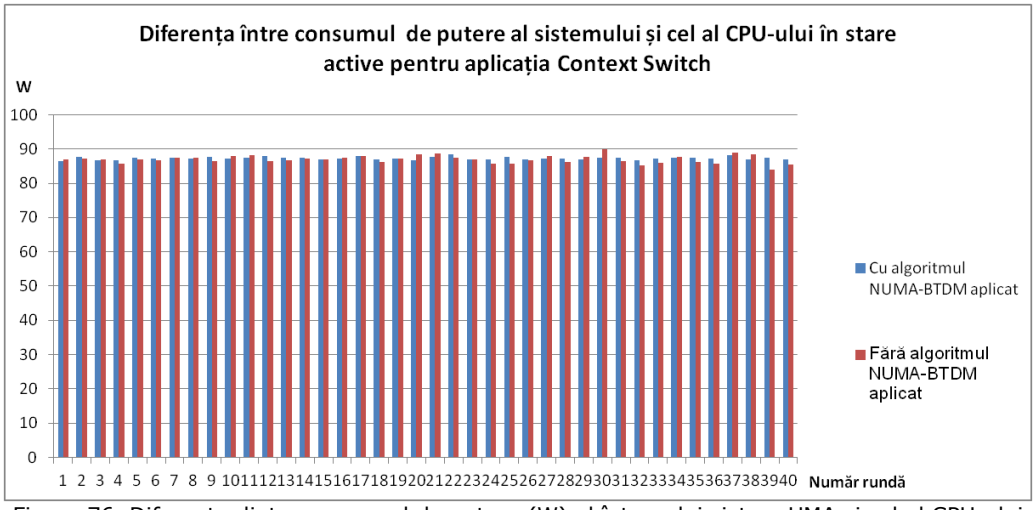

Figura 76. Diferența dintre consumul de putere (W) al întregului sistem UMA și cel al CPU-ului obținută pentru fiecare rundă la rularea aplicației **Context Switch**

Figura 77 indică faptul că valorile medii de consum de putere obținute câte una pentru fiecare rundă, din cele două surse, WattsUp, respectiv ***turbostat***, sunt corelate.

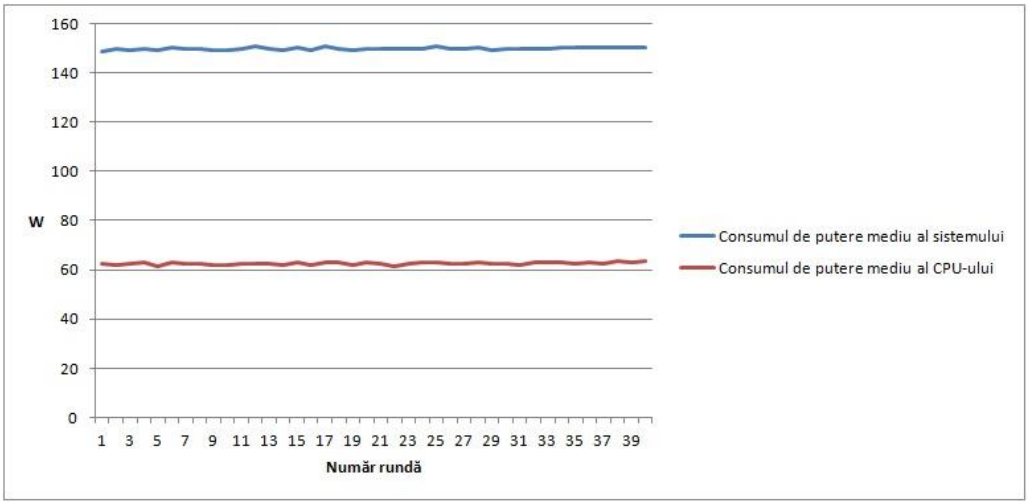

Figura 77. Consumul mediu de putere (W) al CPU-ului din sistemul UMA și al sistemului obținute în fiecare runda la rularea aplicației **Context Switch**

Figura 78 indică faptul că valorile de consum de putere obținute în runda 1 din cele două surse, WattsUp, respectiv ***turbostat***, la fiecare 1 s de rulare a aplicației, sunt corelate. Similar, s-au întocmit grafice și pentru alte câteva runde și

a fost remarcată aceeași observație. Forma de undă se datorează caracteristicilor aplicației, care rulează alternativ două programe principale.

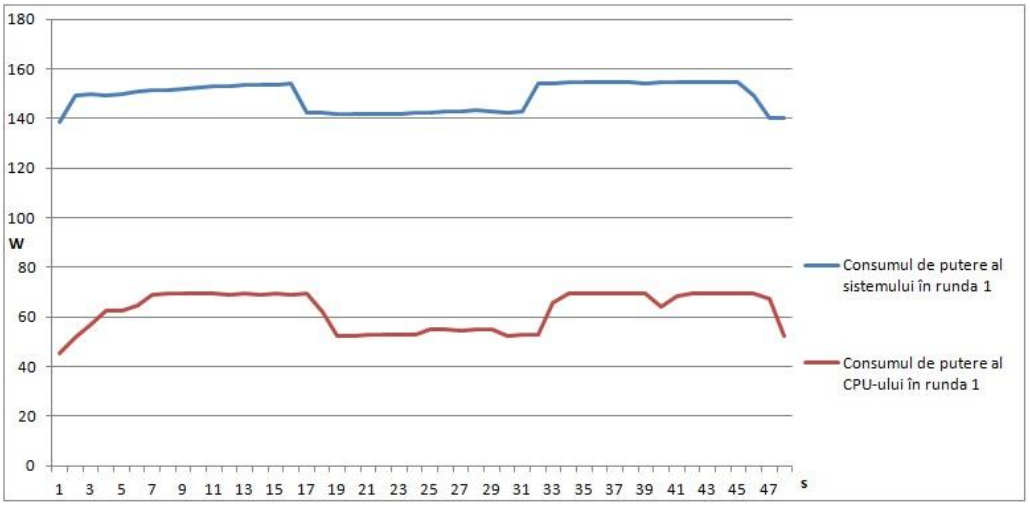

Figura 78. Valorile de consum de putere (W) al CPU-ului din sistemul UMA și al sistemului obținute la fiecare 1 s în runda 1 de rulare a aplicației **Context Switch**

Consumul de putere al sistemului comparativ cu cel al CPU-ului *în stare "în așteptare"*

Fig. 79 prezintă consumul de putere al sistemului UMA în stare "în așteptare" (fără nici o aplicație rulând). Valorile de pe abscisă reprezintă numărul rundei iar valorile de pe ordonată reprezintă consumul de putere "în așteptare" al sistemului înregistrat într-un timp egal cu timpul în care s-ar fi executat aplicația fără, respectiv cu algoritmul ***NUMA-BTDM*** [55] aplicat, pentru fiecare rundă.

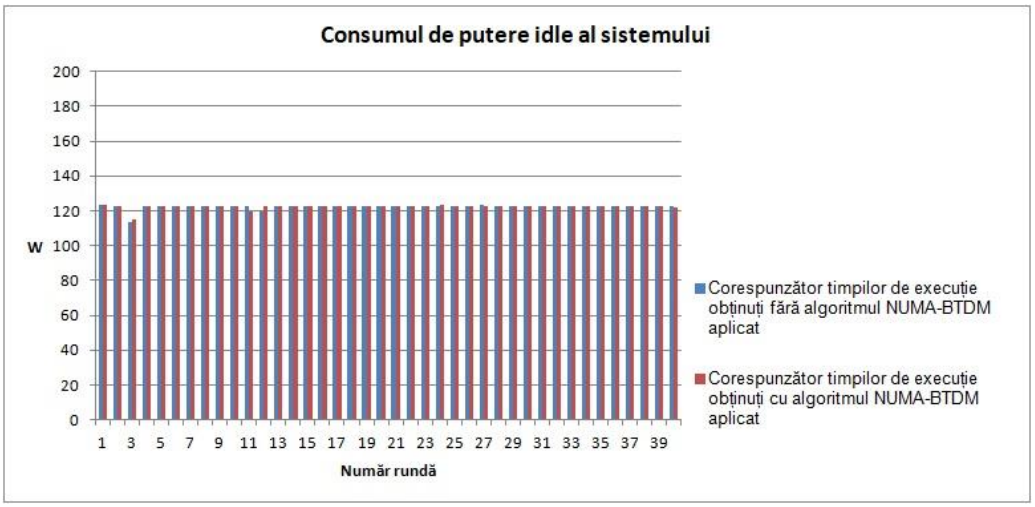

Figura 79. Consumul mediu de putere (W) al sistemului UMA în stare "în așteptare" obținut pentru fiecare rundă

De exemplu, în runda 1, prima valoare medie pentru consumul de putere "în așteptare" al sistemului este 123.28 W, obținută făcând media a 45 de valori de consum de putere înregistrate câte una la fiecare 1 s timp de 45.19 s, adică timpul de execuție obținut în prima rundă fără algoritmul ***NUMA-BTDM*** [55] aplicat, iar cea de-a doua valoare medie pentru consumul de putere "în așteptare" este 123.25 W, obținută făcând media a 48 de valori de consum de putere înregistrate câte una la fiecare 1 s timp de 48.48 s, adică timpul de execuție obținut în prima rundă cu algoritmul ***NUMA-BTDM*** [55] aplicat.

Fig. 80 prezintă consumul de putere al CPU-ului din sistemul UMA în stare "în așteptare" (fără nici o aplicație rulând). Valorile de pe abscisă reprezintă numărul rundei iar valorile de pe ordonată reprezintă consumul de putere "în așteptare" al CPU-ului înregistrat într-un timp egal cu timpul de execuție al aplicației fără, respectiv cu algoritmul ***NUMA-BTDM*** [55] aplicat, pentru fiecare rundă în parte. De exemplu, în runda 1, valoarea medie pentru consumul de putere "în așteptare" al CPU-ului este 19.52 W, obținută făcând media a 45 de valori de consum de putere înregistrate câte una la fiecare 1 s timp de 45.19 s, adică timpul de execuție obținut în prima rundă fără algoritmul ***NUMA-BTDM*** [55] aplicat. Tot în runda 1, o altă valoare medie pentru consumul de putere "în așteptare" este 19.32 W, obținută făcând media a 48 de valori de consum de putere înregistrate câte una la fiecare 1 s timp de timp de 48.48 s, adică timpul de execuție obținut la rularea aplicației **Context Switch** [91] cu algoritmul ***NUMA-BTDM*** [55] aplicat.

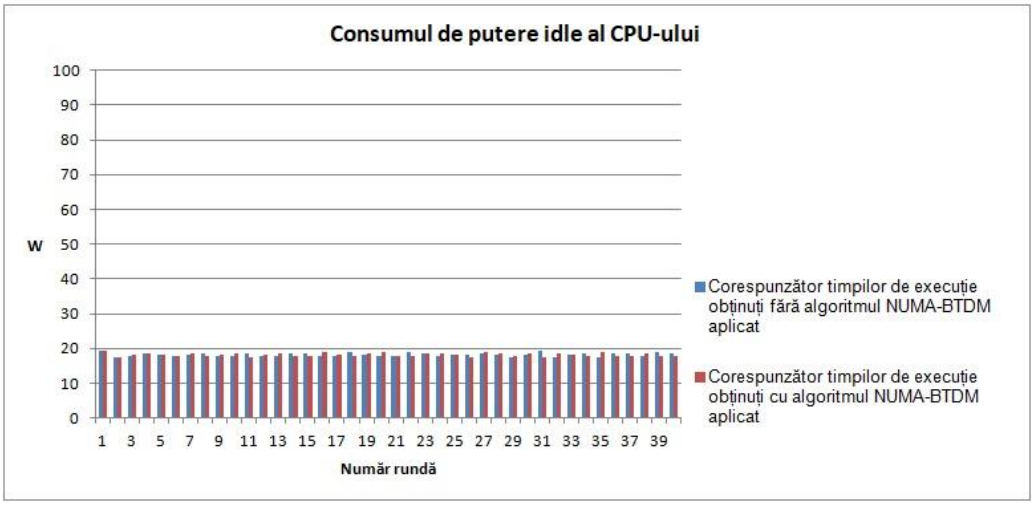

Figura 80. Consumul mediu de putere (W) al CPU-ului din sistemul UMA în stare "în așteptare" obținut pentru fiecare rundă

Considerând că semnificația abscisei și a ordonatei se păstrează, Fig. 81 descrie consumul de putere al CPU-ului și al întregului sistem în corelație.

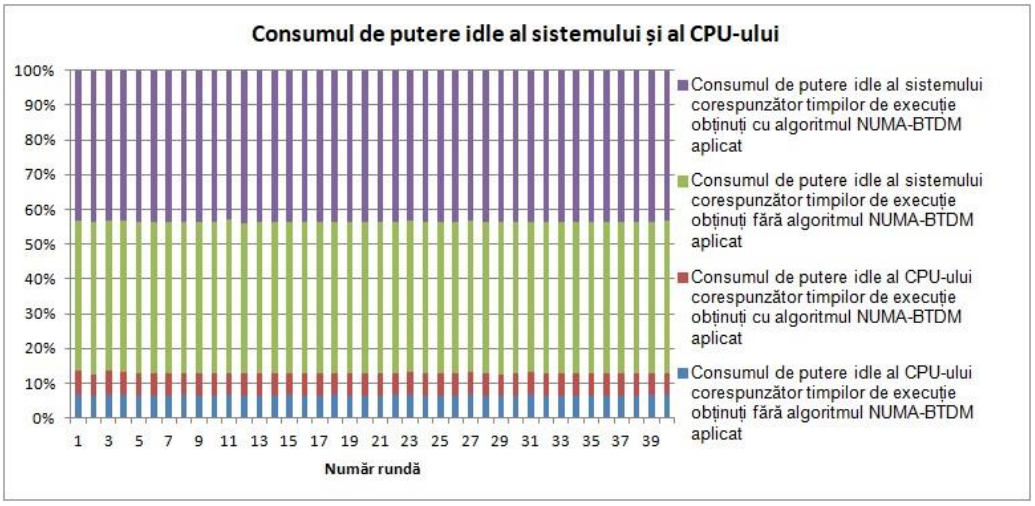

Figura 81. Consumul mediu de putere (W) al CPU-ului din sistemul UMA și al întregului sistem, ambele în stare "în așteptare", obținut pentru fiecare rundă

Din Fig. 82 se observă că diferența între consumul de putere al întregului sistem UMA și cel al CPU-ului în stare "în așteptare" în ambele cazuri, atât pentru timpi de execuție corespunzători rulării fără algoritmul ***NUMA-BTDM*** [55] aplicat (consumul de putere este marcat cu albastru), cât și pentru cei corespunzători rulării cu algoritmul ***NUMA-BTDM*** [55] aplicat (consumul de putere este marcat cu roșu), variază impredictibil în timp.

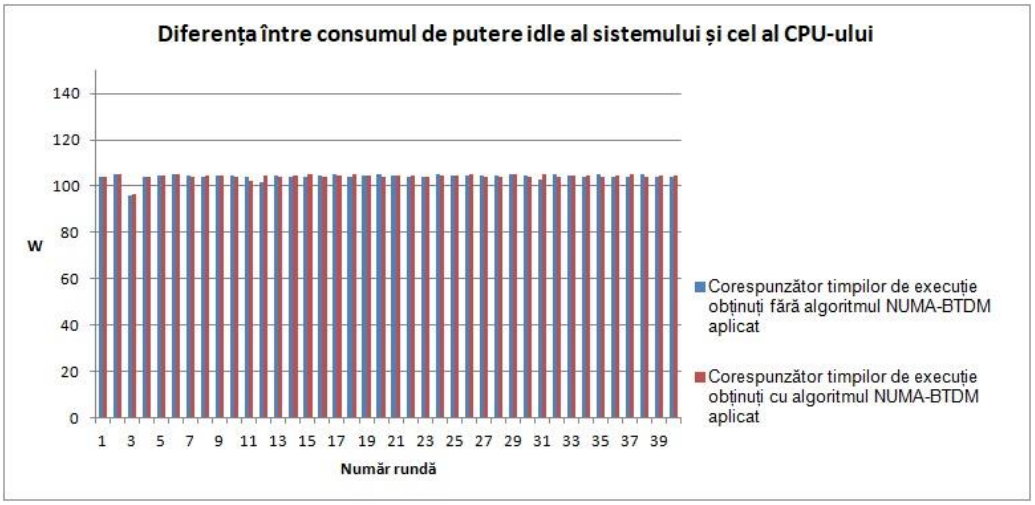

Figura 82. Diferența dintre consumul mediu de putere (W) al întregului sistem UMA și cel al CPU-ului, ambele în stare "în așteptare"

*Consumul de putere al aplicației **Context Switch** [91] obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atunci când aplicația rulează neoptimizat și atunci când rulează optimizat.*

Din Fig. 83 se observă că în majoritatea rundelor în care este lansată aplicația **Context Switch** [91], indicate pe abscisă, consumul de putere atunci când această aplicație este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] este semnificativ mai mic decât atunci când aplicația nu este optimizată. Totuși, în medie (luând în considerare toate rundele), consumul este optimizat cu doar 0.76 W.

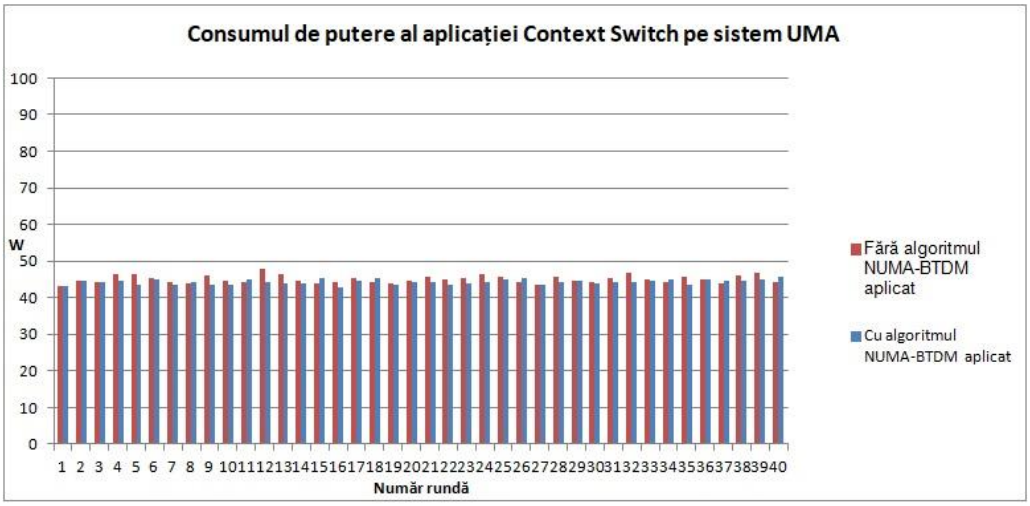

Figura 83. Consumul mediu de putere (W) al aplicației **Context Switch** obținut în fiecare rundă la rularea sistem UMA

Fig. 84 arată optimizarea de consum de putere la rularea aplicației **Context Switch** [91], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea

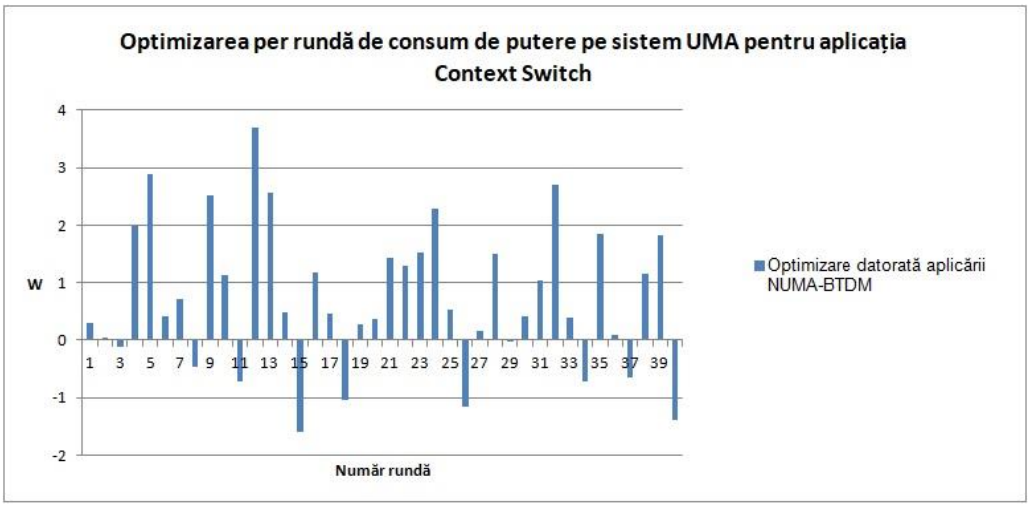

Figura 84. Optimizarea de consum de putere (W) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem UMA

este cuprinsă între valorile minimă de 0.04 W dintr-un total de 44.52 W înregistrat în runda 2 și maximă de 3.69 W dintr-un total de 48.05 W înregistrat în runda 12.

Fig. 85 arată optimizarea de consum de putere, în procente, la rularea aplicației **Context Switch** [91], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul maxim de optimizare de consum de putere este de 7.69% înregistrat în runda 12, iar procentul mediu este 1.57%.

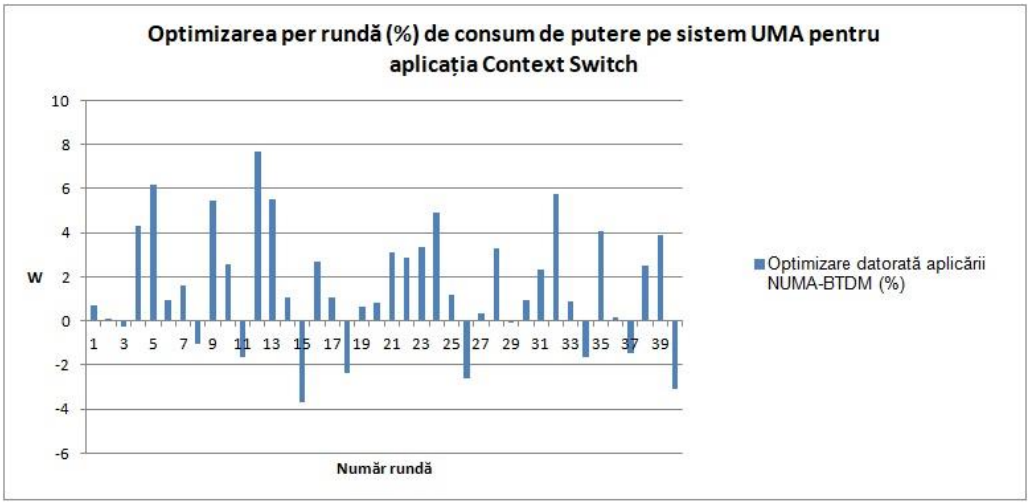

Figura 85. Optimizarea de consum de putere (%) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem UMA

4.6.3 Rezultate experimentale pe sisteme NUMA

4.6.3.1 Timp de execuție

Tabelul 20 prezintă comparativ timpii de execuție obținuți atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat și atunci când algoritmul este aplicat asupra aplicației **Context Switch** [91] rulate pe sistem NUMA. În Tabelul 20, prima coloană reprezintă numărul rundei în care este rulată aplicația **Context Switch** [91], coloana a doua reprezintă timpul de execuție pe sistem NUMA al aplicației **Context Switch** [91] fără algoritmul ***NUMA-BTDM*** [55] aplicat, iar cea de-a treia coloană, timpul de execuție pe sistem NUMA al aplicației cu algoritmul ***NUMA-BTDM*** [55] aplicat (la compilare). Se observă că timpul de execuție pe sistem NUMA crește atunci când asupra aplicației este aplicat algoritmul ***NUMA-BTDM*** anterior rulării acesteia. Creșterea însă nu este atât de semnificativă ca la sistemul UMA, media timpilor de execuție pentru toate rundele atunci când algoritmul nu este aplicat având valoarea de 41.56 W și fiind aproximativ egală cu media 41.90 W, obținută când algoritmul este aplicat.

Tabel 20. Rezultate experimentale de timp de execuție al aplicației **Context Switch** rulată pe sistem NUMA, prezentate comparativ atunci când algoritmul ***NUMA-BTDM*** este aplicat și atunci când nu este aplicat

Număr rundă	**Timp de execuție fără algoritmul NUMA-BTDM aplicat**	**Timp de execuție cu algoritmul NUMA-BTDM aplicat**
1	41.504	41.299
2	41.514	41.797
3	41.423	42.25
4	41.474	41.625
5	41.344	41.755
6	41.78	41.802
7	40.558	41.658
8	41.971	42.67
9	41.69	41.687
10	41.389	41.987
11	41.154	42.42
12	41.72	42.823
13	42.162	42.409
14	41.857	41.634
15	41.478	40.539
16	40.862	41.36
17	41.563	42.951
18	41.894	41.821
19	41.155	42.306
20	41.931	42.329
21	41.836	40.238
22	41.721	42.383
23	41.506	42.684
24	41.844	42.254
25	41.404	42.296
26	41.945	41.689
27	41.636	41.908
28	41.112	42.104
29	41.596	42.112
30	41.423	41.933
31	41.868	42.442
32	41.657	41.375
33	41.973	41.877
34	41.467	42.791
35	41.255	41.874
36	41.908	40.107
37	41.812	41.973
38	41.418	42.528
39	41.281	40.338
40	41.567	42.221

Figura 86 prezintă comparativ timpii de execuție obținuți în cele 40 de runde de rulare ale aplicației **Context Switch** [91] cu algoritmul ***NUMA-BTDM*** [55] aplicat, cât și fără algoritm aplicat. Din figură se observă și în cazul sistemului NUMA, o degradare a timpului de execuție al aplicației **Context Switch** [91] în urma aplicării algoritmului ***NUMA-BTDM*** [55], pentru majoritatea rundelor.

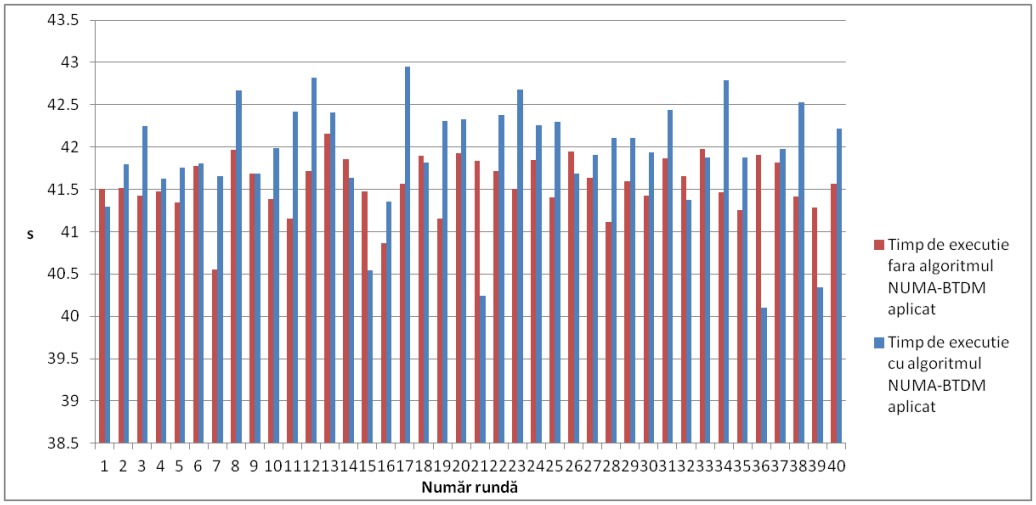

Figura 86. Timpul de execuție al aplicației **Context Switch** pe sistem UMA, atât atunci când algoritmul ***NUMA-BTDM*** este aplicat, cât și când algoritmul nu este aplicat

4.6.3.2 Consum de putere

Consumul de putere al sistemului comparativ cu cel al CPU-ului *atunci când aplicația rulează neoptimizat și atunci când rulează optimizat*

Din Fig. 87 se observă că atunci când aplicația **Context Switch** [91] rulează pe un sistem NUMA, consumul de putere al sistemului este mai mare atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat decât atunci când algoritmul nu este aplicat. În figură, abscisa reprezintă numărul rundei iar ordonata reprezintă consumul mediu de putere în W al întregului sistem, pentru fiecare rundă în parte (consumul mediu este obținut făcând media măsurătorilor obținute la fiecare 1 s). Consumul de putere al sistemului crește cu până la 2.73 W atunci când algoritmul este aplicat. Această creștere survine și în urma execuției apelurilor ***pthread_setaffinity_np*** care setează core-ul pe care să ruleze fiecare din cele două fire de execuție, apeluri care consumă putere comparabil ca valoare cu optimizarea obținută prin maparea firelor de execuție autonome pe core-uri separate.

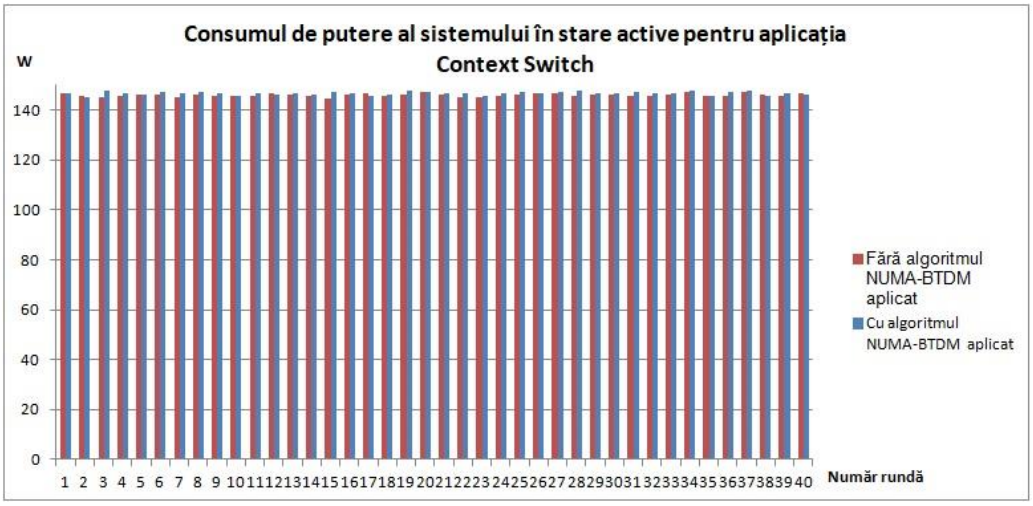

Figura 87. Consumul mediu de putere (W) al sistemului NUMA obținut în fiecare rundă la rularea aplicației **Context Switch**

Din Fig. 88 se observă că aplicația **Context Switch** [91] rulează pe sistem NUMA cu consum de putere al CPU-ului mai mic, în majoritatea rundelor, atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] decât atunci când algoritmul nu este aplicat. În figură, abscisa are aceeași semnificație ca în Fig. 87, iar ordonata se referă la consumul mediu de putere al CPU-ului din fiecare rundă.

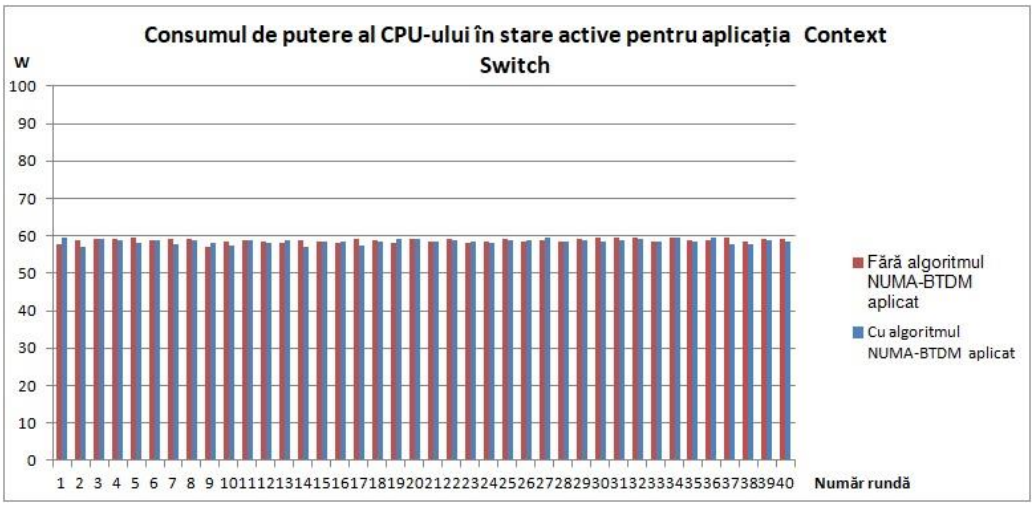

Figura 88. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA obținut în fiecare rundă la rularea aplicației **Context Switch**

Fig. 89 ilustrează consumul de putere al CPU-ului și cel al întregului sistem pentru o comparație valorică, atât atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat cât și atunci când nu este aplicat. Se observă că măsurătorile sunt corelate și

consumul CPU-ului este aproximativ 40% din cel al întregului sistem la rularea aplicației **Context Switch** [91].

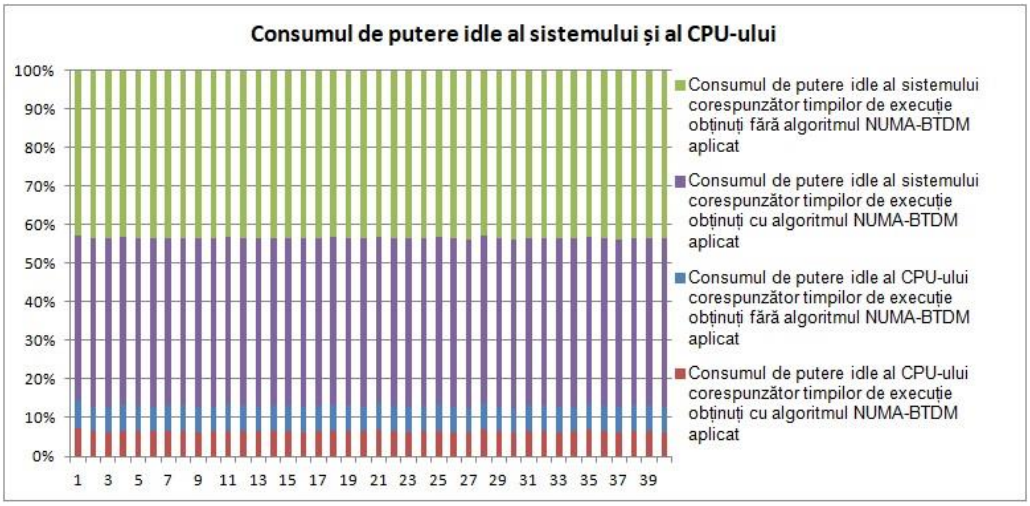

Figura 89. Consumul mediu de putere (W) al CPU-ului și al întregului sistem NUMA obținut în fiecare rundă la rularea aplicației **Context Switch**

La fel ca la sistemul UMA, și în acest caz, cu cât diferența între consumul de putere al întregului sistem NUMA și cel al CPU-ului din sistem, este mai mare, cu atât aplicația rulează mai optim. Fig. 90 ilustrează această diferență pentru aplicația **Context Switch** [91], cu și fără algoritmul ***NUMA-BTDM*** [55] aplicat. Diferența este mai mare atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, ceea ce indică o diminuare a consumului de putere al CPU-ului la rularea aplicației **Context Switch** [91] cu algoritmul ***NUMA-BTDM*** [55] aplicat.

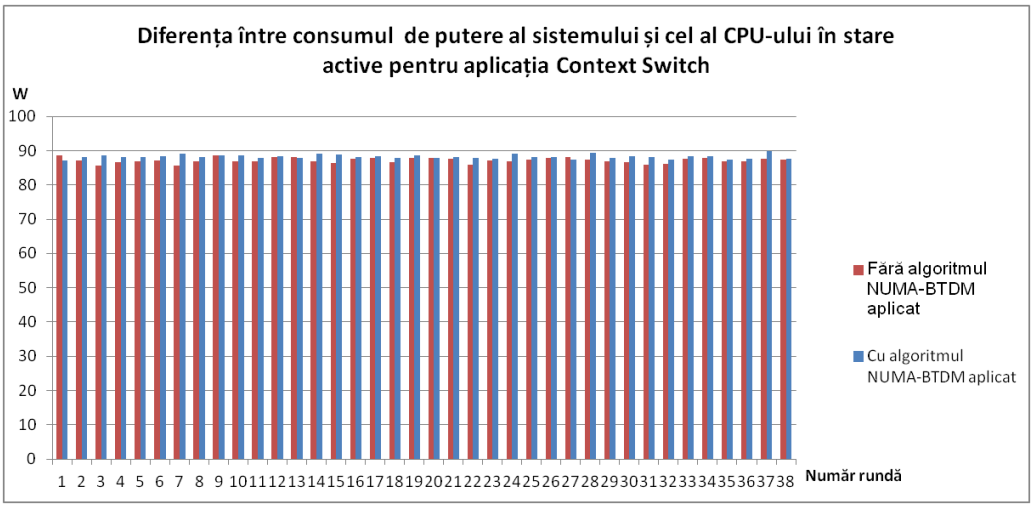

Figura 90. Diferența dintre consumul de putere (W) al întregului sistem NUMA și cel al CPU-ului obținută pentru fiecare rundă la rularea aplicației **Context Switch**

Figura 91 indică faptul că valorile medii de consum de putere obținute în fiecare rundă din cele două surse, WattsUp, respectiv ***turbostat***, sunt corelate.

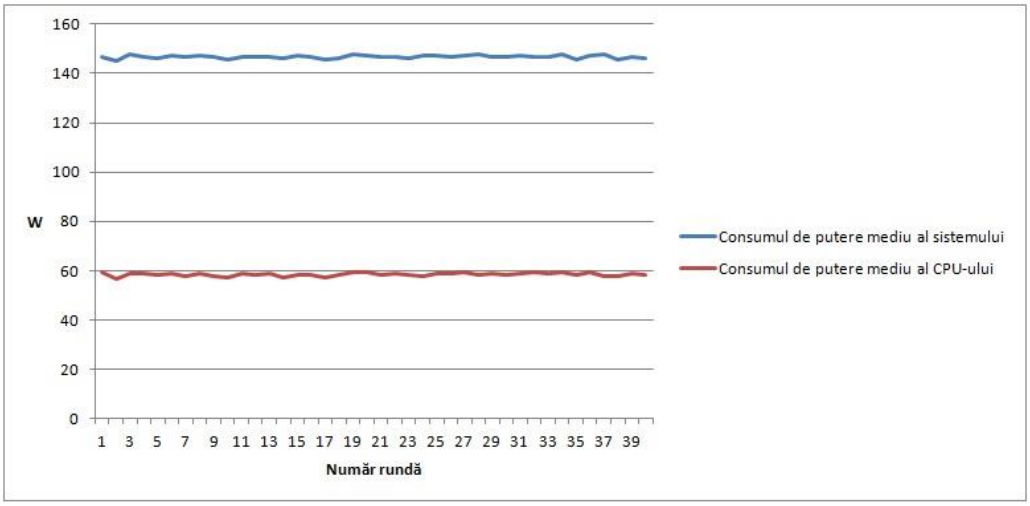

Figura 91. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA și al sistemului obținute în fiecare runda la rularea aplicației **Context Switch**

Figura 92 indică faptul că valorile de consum de putere obținute, pe sistem NUMA, în runda 1, din cele două surse, WattsUp, respectiv ***turbostat***, la fiecare 1 s de rulare a aplicației, sunt corelate. Similar, s-au întocmit grafice și pentru alte câteva runde și a fost remarcată aceeași observație.

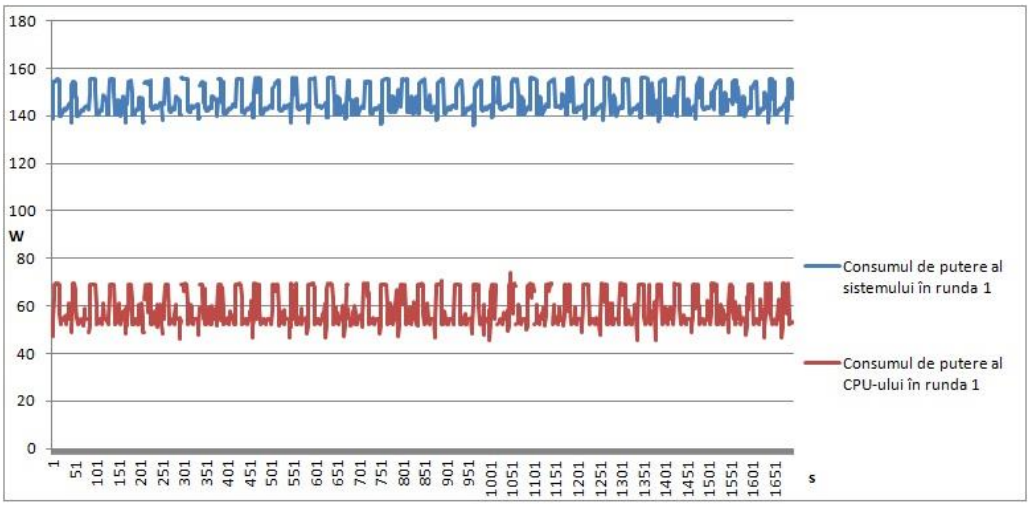

Figura 92. Valorile de consum de putere (W) al CPU-ului din sistemul NUMA și al sistemului obținute în runda 1 la rularea aplicației **Context Switch**

Consumul de putere al sistemului comparativ cu cel al CPU-ului *în stare "în așteptare"*

Fig. 93 prezintă consumul de putere al sistemului NUMA în stare "în așteptare" (fără nici o aplicație rulând). Valorile de pe abscisă reprezintă numărul rundei iar valorile de pe ordonată reprezintă consumul de putere "în așteptare" al sistemului înregistrat într-un timp egal cu timpul de execuție al aplicației atât fără algoritmul ***NUMA-BTDM*** [55] aplicat, cât și cu el aplicat, înregistrat pentru fiecare rundă în parte. De exemplu, în runda 1, prima valoare medie pentru consumul de putere "în așteptare" al sistemului este 123.28 W, obținută făcând media a 45 de valori de consum de putere înregistrate câte una la fiecare 1 s timp de 45.19 s, adică timpul de execuție obținut în prima rundă fără algoritmul ***NUMA-BTDM*** [55] aplicat, iar cea de-a doua valoare medie pentru consumul de putere "în așteptare" este 123.25 W, obținută făcând media a 48 de valori de consum de putere înregistrate câte una la fiecare 1 s timp de 48.48 s, adică timpul de execuție obținut în prima rundă cu algoritmul ***NUMA-BTDM*** [55] aplicat.

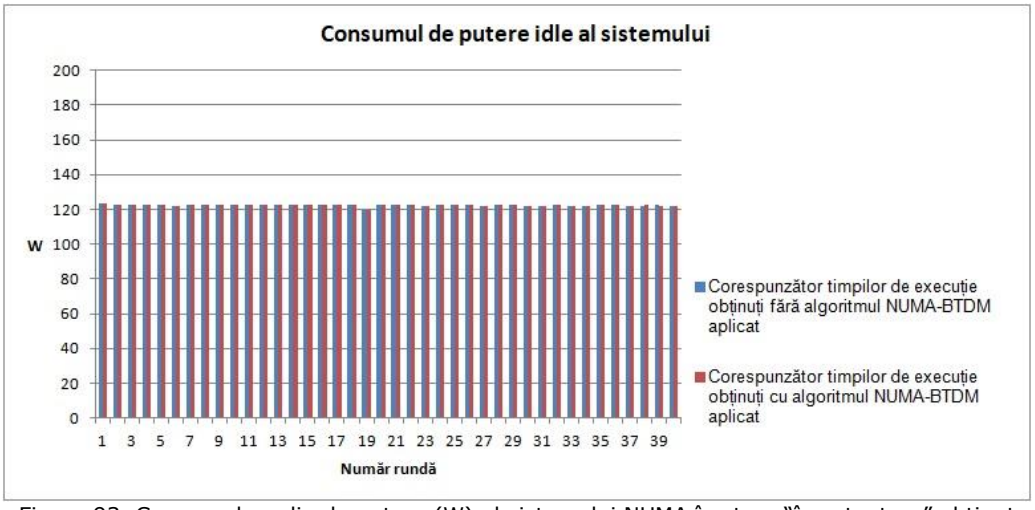

Figura 93. Consumul mediu de putere (W) al sistemului NUMA în stare "în așteptare" obținut pentru fiecare rundă

Fig. 94 prezintă consumul de putere al CPU-ului din sistemul NUMA în stare "în așteptare" (fără nici o aplicație rulând). Valorile de pe abscisă reprezintă numărul rundei iar valorile de pe ordonată reprezintă consumul de putere "în așteptare" al CPU-ului înregistrat într-un timp egal cu timpul în care ar fi rulat aplicația, pentru fiecare rundă în parte. De exemplu, în runda 1, valoarea medie pentru consumul de putere "în așteptare" al CPU-ului este aproximativ 21 W, obținută făcând media a 41 de valori de consum de putere înregistrate cu rata de 1 s, timp de 41.5 s (timpul obținut în prima rundă de rulare a aplicației **Context Switch** [91] fără algoritmul ***NUMA-BTDM*** [55] aplicat) și 20.91 W, valoare obținută făcând media a 41 de valori de consum de putere înregistrate cu rata de 1 s, timp de 41.29 s (timpul obținut la rularea aplicației **Context Switch** [91] cu algoritmul ***NUMA-BTDM*** [55] aplicat). Consumul mediu de putere "în așteptare" variază de la o rundă la alta.

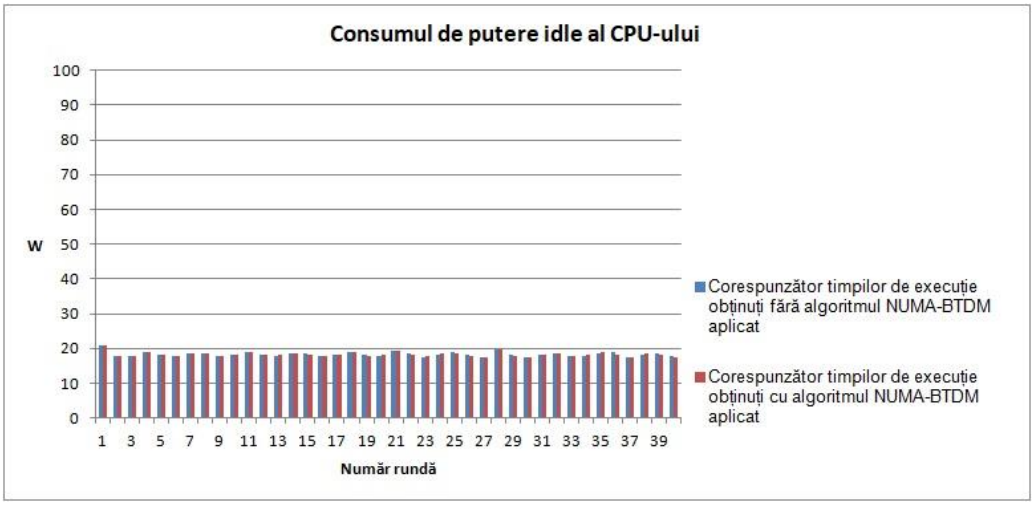

Figura 94. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA în stare "în așteptare" obținut pentru fiecare rundă

Păstrând aceeași semnificație pentru abscisă și a ordonată, Fig. 95 descrie consumul de putere al CPU-ului și al întregului sistem în corelație.

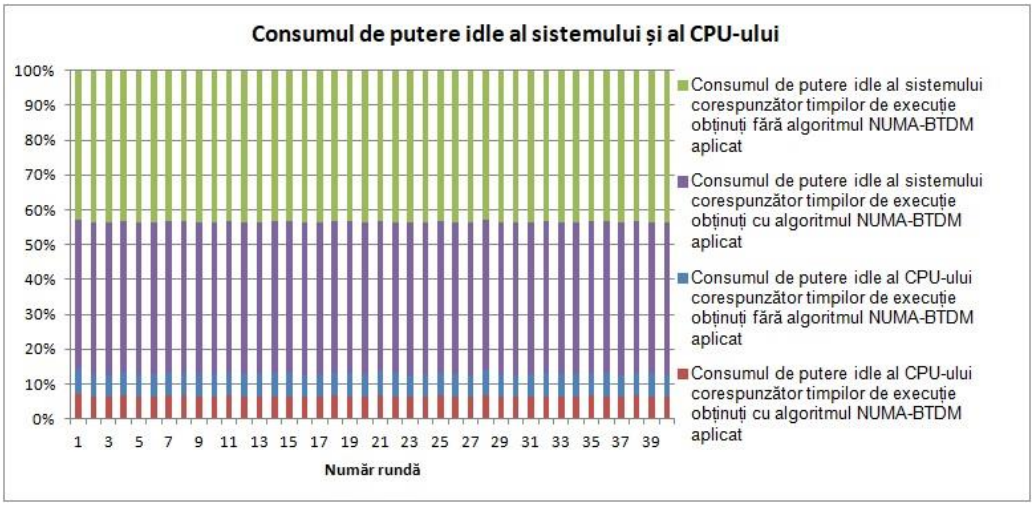

Figura 95. Consumul mediu de putere (W) al CPU-ului din sistemul NUMA și al întregului sistem, ambele în stare "în așteptare", obținut pentru fiecare rundă

Din Fig. 96 se observă că diferența între consumul de putere al întregului sistem NUMA și cel al CPU-ului în stare "în așteptare", în ambele cazuri: atât pentru timpi de execuție corespunzători rulării aplicației **Context Switch** [91] fără algoritmul ***NUMA-BTDM*** [55] aplicat (consumul de putere este marcat cu albastru), cât și pentru cei corespunzători rulării cu algoritmul ***NUMA-BTDM*** [55] aplicat (consumul de putere este marcat cu roșu), variază impredictibil în timp.

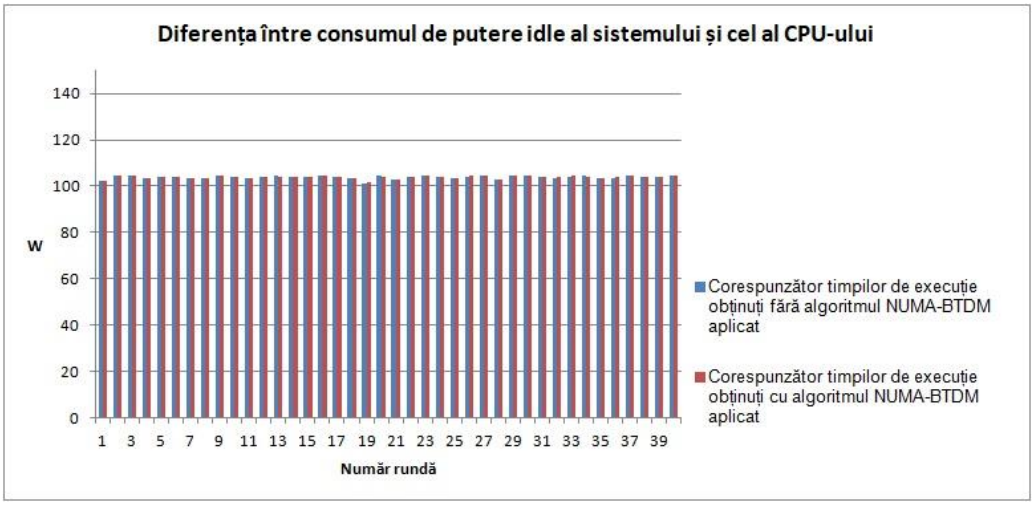

Figura 96. Diferența dintre consumul mediu de putere (W) al întregului sistem NUMA și cel al CPU-ului, ambele în stare "în așteptare"

Consumul de putere al aplicației ***Context Switch*** *obținut din diferența între consumul de putere al* CPU-ului *în stare activ și cel în stare "în așteptare", atunci când aplicația rulează neoptimizat și atunci când rulează optimizat*

Din Fig. 97 se observă că în majoritatea rundelor în care este lansată aplicația **Context Switch** [91], indicate pe abscisă, consumul de putere al aplicației atunci când aceasta este optimizată utilizând algoritmul ***NUMA-BTDM*** [55] este mai mic decât atunci când aplicația nu este optimizată. În medie, consumul este optimizat cu 0.31 W.

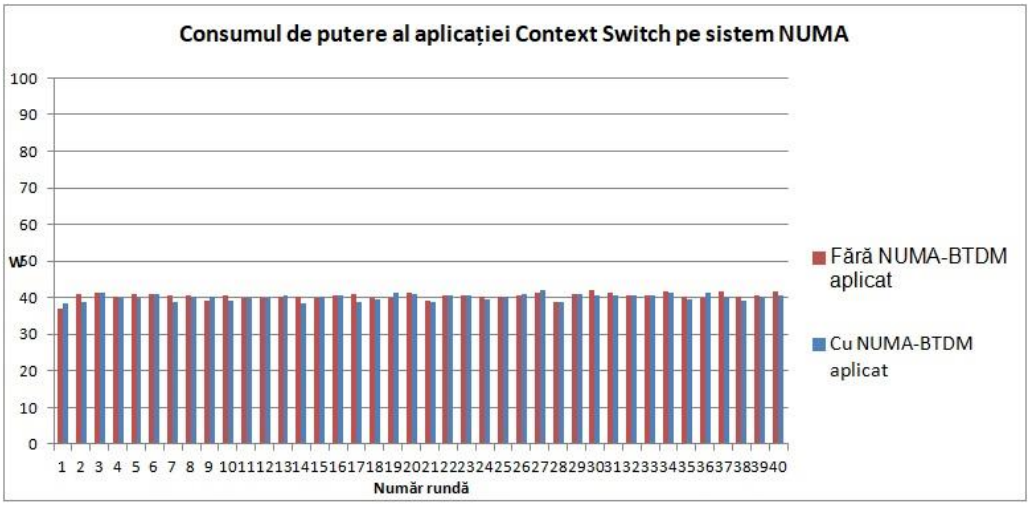

Figura 97. Consumul de putere (W) pe sistem NUMA al aplicației **Context Switch** obținut în fiecare rundă

Fig. 98 arată optimizarea de consum de putere la rularea aplicației **Context Switch** [91], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Optimizarea este cuprinsă între valorile minimă de 0.01 W dintr-un total de 40.86 W înregistrat în runda 29 și maximă de 1.91 W dintr-un total de 40.91 W înregistrat în runda 2.

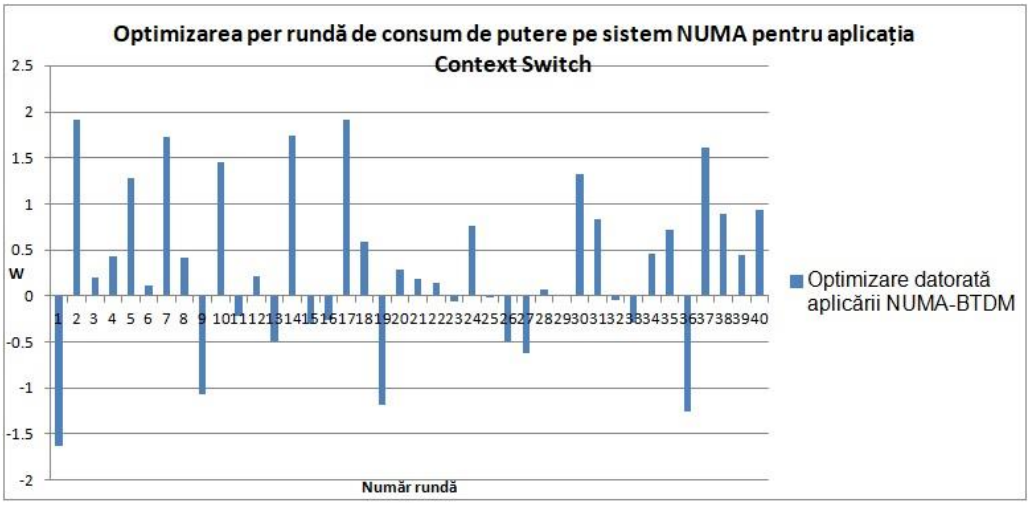

Figura 98. Optimizarea de consum de putere (W) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem NUMA

Fig. 99 arată optimizarea de consum de putere, în procente, la rularea aplicației **Context Switch** [91], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55]. Procentul maxim de optimizare de consum de putere este de 4.69% înregistrat în runda 17, iar procentul mediu este 0.76%, mai mic ca la sistem UMA.

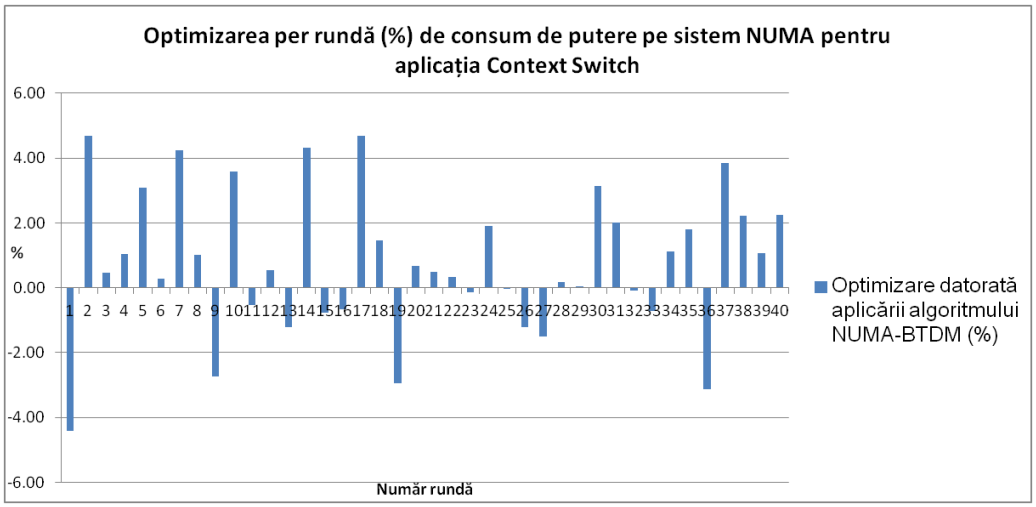

Figura 99. Optimizarea de consum de putere (%) al aplicației **Context Switch** obținută în fiecare rundă la rularea pe sistem NUMA

4.6.4 Comparație între rezultatele experimentale pe sisteme UMA și cele pe sisteme NUMA

4.6.4.1 Comparație pentru timp de execuție

Tabelul 21 prezintă comparativ rezultatele experimentale de timp de execuție, pentru aplicația **Context Switch** [91], rulată pe sistem UMA și sistem NUMA. Din tabel reies următoarele concluzii:

6. timpul de execuție pe sistem UMA este în medie mai mic decât pe sistem NUMA
7. timpul de execuție mediu pe sistem NUMA al aplicației cu algoritmul ***NUMA-BTDM*** [55] aplicat și cel al aplicației fără algoritm aplicat sunt aproximativ egale
8. pe sistem UMA, măsurătorile obținute pentru aplicația rulată fără algoritmul ***NUMA-BTDM*** [55] aplicat sunt mai dispersate față de valoarea medie decât cele obținute pentru cazul în care algoritmul este aplicat, iar la sistem NUMA, se obține reversul

Tabel 21. Comparație între rezultatele experimentale de timp de execuție obținute pe sisteme UMA și pe sisteme NUMA pentru aplicația **Context Switch**

Categorie valori	**Tip sistem**	**Valorile fiecărei categorii pentru măsurătorile obținute la rularea aplicației Context Switch**	
		Fără NUMA-BTDM aplicat	**Cu NUMA-BTDM aplicat**
Minime	UMA	39.77	44.43
	NUMA	40.56	40.11
Medii	UMA	41.62	47.23
	NUMA	41.57	41.91
Maxime	UMA	46.52	48.61
	NUMA	42.16	42.95
Varianță	UMA	2.89	0.81
	NUMA	0.10	0.45
Deviație standard	UMA	1.55	0.90
	NUMA	0.32	0.67

4.6.4.2 Comparație pentru consum de putere

Tabelul 22 prezintă comparativ rezultatele experimentale, pe sistem UMA și sistem NUMA, pentru aplicația **Context Switch** [91]. Prima coloană indică numărul rundei. A doua coloană descrie optimizarea în W, obținută atât în cazul sistemului UMA cât și în cazul sistemului NUMA, iar ultima coloană descrie această optimizare în procente. Procentele sunt calculate împărțind optimizarea în W la consumul de putere al aplicației neoptimizate și înmulțind rezultatul cu 100.

Tabel 22. Rezultate experimentale prezentând comparativ pentru sistem UMA și sistem NUMA, optimizarea obținută la aplicarea algoritmului ***NUMA-BTDM*** asupra aplicației **Context Switch**

Număr rundă	Optimizarea obținută (W)		Procent de optimizare	
	UMA	NUMA	UMA	NUMA
1	0.30	-1.63	0.69	-4.41
2	0.04	1.92	0.10	4.68
3	-0.11	0.20	-0.26	0.47
4	1.99	0.42	4.29	1.05
5	2.88	1.28	6.20	3.10
6	0.42	0.11	0.93	0.27
7	0.72	1.72	1.62	4.24
8	-0.46	0.41	-1.05	1.01
9	2.51	-1.06	5.45	-2.72
10	1.13	1.45	2.54	3.58
11	-0.73	-0.21	-1.64	-0.53
12	3.70	0.22	7.70	0.54
13	2.57	-0.49	5.54	-1.22
14	0.48	1.75	1.07	4.32
15	-1.61	-0.31	-3.66	-0.77
16	1.18	-0.27	2.67	-0.66
17	0.47	1.92	1.04	4.70
18	-1.05	0.59	-2.37	1.47
19	0.27	-1.18	0.62	-2.95
20	0.36	0.28	0.81	0.68
21	1.43	0.19	3.14	0.48
22	1.29	0.14	2.86	0.34
23	1.51	-0.06	3.33	-0.14
24	2.28	0.76	4.89	1.90
25	0.54	-0.01	1.19	-0.02
26	-1.16	-0.49	-2.62	-1.21
27	0.16	-0.62	0.36	-1.51
28	1.50	0.06	3.28	0.17
29	-0.01	0.02	-0.02	0.04
30	0.42	1.32	0.95	3.13
31	1.05	0.84	2.31	2.02
32	2.69	-0.04	5.73	-0.10
33	0.40	-0.29	0.89	-0.71
34	-0.72	0.47	-1.62	1.12
35	1.85	0.72	4.06	1.79
36	0.09	-1.25	0.19	-3.12
37	-0.64	1.61	-1.46	3.85
38	1.15	0.89	2.51	2.21
39	1.83	0.44	3.91	1.07
40	-1.38	0.93	-3.11	2.24

Fig. 100 arată optimizarea de consum de putere, în procente, la rularea aplicației **Context Switch** [91], produsă în urma aplicării algoritmului ***NUMA-BTDM*** [55], prezentată comparativ pentru sistemele UMA și NUMA. Procentul maxim de optimizare de consum de putere este 7.69 % pentru sistem UMA și 4.69% pentru sistem NUMA, iar procentul mediu este 1.57% pentru sistem UMA și 0.76% pentru sistem NUMA. Prin urmare această aplicație rulează mai optim pe sistem UMA, decât pe NUMA, după aplicarea algoritmului ***NUMA-BTDM*** [55].

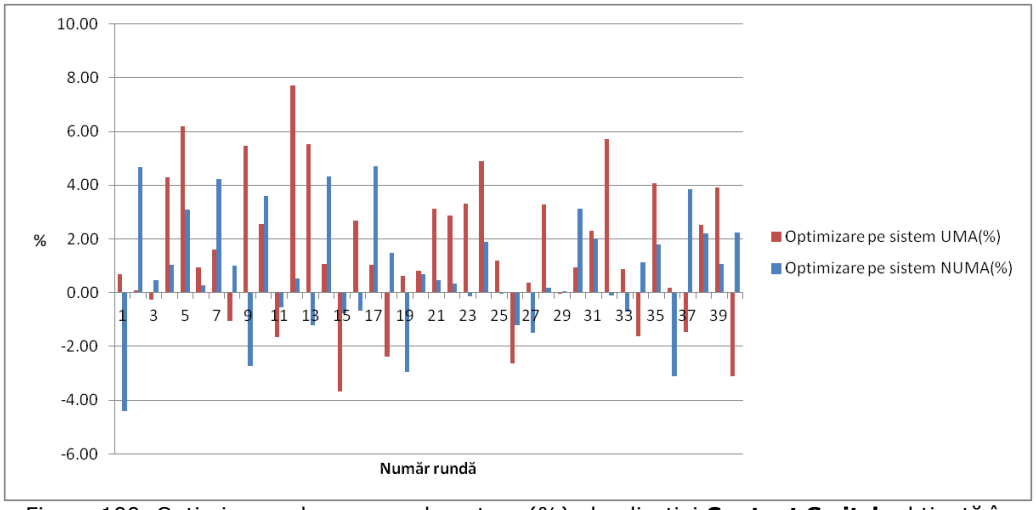

Figura 100. Optimizarea de consum de putere (%) al aplicației **Context Switch** obținută în fiecare rundă, prezentată comparativ pentru sistemele UMA și NUMA

Tabelul 23 descrie comparativ pentru sistemul UMA și sistemul NUMA, rezultatele experimentale obținute din cele două surse, utilitarul software tubostat și dispozitivul hardware specializat WattsUp, împărțite pe categorii: minime, medii, maxime, varianța, deviație standard, pentru aplicația **Context Switch** [91].

Tabel 23. Comparație între rezultatele experimentale de consum de putere obținute pe sisteme UMA și pe sistem NUMA pentru aplicația **Context Switch**

Categorie valori	**Tip sistem**	**Comparație între măsurătorile obținute din cele două surse, cu și fără aplicarea algoritmului NUMA-BTDM**	
		Utilitarul turbostat	**Dispozitivul WattsUp**
Minime	UMA	Media valorilor minime ale tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este este aplicat este 48.42 W, mai mică decât media minimelor 48.96 W, obținută atunci când algoritmul este aplicat.	Media valorilor minime ale tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este este aplicat este 138.34 W, mai mică decât media minimelor 139.22 W, obținută atunci când algoritmul este aplicat.
	NUMA	Media valorilor minime ale tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 48.14 W, iar atunci când acesta este aplicat, media minimelor este 48.79 W	Media valorilor minime ale tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este este aplicat este 139.49 W, aproximativ egală cu media minimelor 139.58 W, obținută atunci când algoritmul este aplicat.

Medii	UMA	Media valorile medii ale tuturor rundelor, obținute atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat, este 63.31 W, iar atunci când algoritmul este aplicat, media este 62.54 W, ceea ce indică o optimizare de 0.77 W	Media valorilor medii ale tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 150.32 W, fiind într-o mică măsură mai mare decât media 149.86 W, obținută atunci când algoritmul este aplicat.
	NUMA	Media valorilor medii ale tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 58.84 W, mai mare decât media 58.52 W obținută atunci când algoritmul este aplicat, rezultând o optimizare de 0.32 W.	Media valorilor medii ale tuturor rundelor, obținute atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 145.98 W, fiind mai mică decât media 146.71 W, obținută atunci când algoritmul este aplicat.
Maxime	UMA	Maximul valorilor maxime din fiecare rundă obținute atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 72.5 W iar media valorilor maxime este 69.78 W. Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, maximul este 72.59 W iar media valorilor maxime este 69.71 W, aproximativ egală cu cea obținută atunci când algoritmul nu este aplicat.	Media valorilor maxime atunci când algoritmul nu este aplicat este 156.3 W, fiind aproximativ egală cu media 156.29 W obținută atunci când algoritmul este aplicat, iar maximele valorilor maxime sunt 156.3 W, respectiv 177.1 W.
	NUMA	Media valorilor maxime obținute atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 69.39 W, fiind aproape egală cu media valorilor maxime 69.42 W obținută atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat.	Media valorilor maxime atunci când algoritmul nu este aplicat este 155.45 W, fiind aproximativ egală cu media 155.55 W obținută atunci când algoritmul este aplicat.

Varianță	UMA	Varianța medie pentru toate rundele atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 58.66 iar valoarea maximă pentru varianță este 72.4. Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, valoarea maximă a variației este 63.84 și varianța medie a tuturor rundelor este 57.67, ceea ce arată că măsurătorile obținute sunt în medie mai puțin dispersate față de valoarea medie atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat.	Varianța medie pentru toate rundele atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 40.98, iar valoarea maximă a variației este 68.11. Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, valoarea maximă pentru varianță este 56.96 și varianța medie a tuturor rundelor este 39.01, măsurătorile obținute fiind în acest caz mai puțin dispersate față de valoarea medie decât atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat.
	NUMA	Varianța medie pentru toate rundele atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 52.92 iar valoarea maximă pentru varianță este 62.4. Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, valoarea maximă pentru varianță este 54.15 și varianța medie pentru toate rundele este 46.71, măsurătorile obținute fiind în acest caz mai puțin dispersate față de valoarea medie decât atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat. În plus, măsurătorile obținute pe sistem NUMA sunt mai puțin dispersate, comparativ cu cele obținute pe sistem UMA.	Varianța medie pentru toate rundele atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 39.07 iar valoarea maximă a variației este 45.71. Atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat, valoarea maximă a variației este 40.33 și varianța medie pentru toate rundele este 31.36, ceea ce indică faptul că măsurătorile obținute sunt mai puțin dispersate față de valoarea medie atunci când algoritmul este aplicat. De asemenea, măsurătorile obținute pe sistem NUMA sunt mai puțin disipate, comparativ cu cele obținute pe sistem UMA.

Deviație standard	UMA	Media deviațiilor standard a tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 7.64 W, maximul fiind 8.5 W. Valoarea maximă a deviației standard în cazul aplicației optimizate este 7.99 W, iar valoarea medie este 7.58 W, aproximativ egală cu cea obținută în cazul aplicației neoptimizate.	Valoarea maximă pentru deviația standard atunci când algoritmul ***NUMA-BTDM*** [55] nu a fost aplicat este 8.25 W iar valoarea medie a deviației standard este 6.34 W. Media deviațiilor standard obținute atunci când este aplicat algoritmul ***NUMA-BTDM*** [55] este 6.23 W, valoarea maximă pentru deviația standard fiind 7.54 W, ambele mai mici decât cele obținute la rularea aplicației neoptimizate.
	NUMA	Media deviaților standard a tuturor rundelor atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este de 7.26 W, deviația standard maximă fiind de 7.89 W. Deviația standard atunci când algoritmul este aplicat are valoarea maximă 7.35 W și medie de 6.82 W. Valorile medii sunt în ambele cazuri, optimizat și neoptimizat, mai mici cu mai puțin de 1 W decât la sistemul UMA.	Valoarea maximă pentru deviația standard atunci când algoritmul ***NUMA-BTDM*** [55] nu este aplicat este 6.76 W iar valoarea medie a deviației standard este 6.24 W, aproximativ egală cu cea obținută la rularea pe sistem UMA. Media deviațiilor standard obținute atunci când algoritmul ***NUMA-BTDM*** [55] este aplicat este 5.57 W, fiind mai mică cu mai puțin de 1 W decât în cazul aplicației neoptimizate, cât și decât cea în cazul aplicației optimizate rulată pe sistem UMA, iar valoarea maximă pentru deviația standard este 6.35 W.

4.6.5 Concluzii ale rezultatelor experimentale pentru aplicația de referință Context Switch

Timpul de execuție al aplicației de referință **Context Switch** [91] pe sistem NUMA nu este îmbunătățit la aplicarea algoritmului ***NUMA-BTDM*** [55]. Datorată 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 de execuție cu care este redus timpul total de rulare al aplcației, în urma aplicării algoritmului ***NUMA-BTDM*** [55], 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** [91] care rulează pe sistem NUMA folosind 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 "în așteptare" și invers a firelor de execuție. Optimizarea obținută de 5% în cazul consumului de energie este o valoare considerabilă, tinându-se cont că aplicația

rulează pentru un timp de doar ~40 s, algoritmul neavând timp foarte mult să își facă simțit efectul asupra comunicării între firele de execuție și că o bună parte din energie totală necesară aplicației este consumată cu pornirea componentelor hardware pe care se execută aceasta.

4.7 Avantaje și limitări ale algoritmilor NUMA-BTLP și NUMA-BTDM

Algoritmul ***NUMA-BTLP*** [88] utilizează ca date de intrare pentru analiza realizată la compilare, representarea intermediară a compilatorului, ceea ce prezintă următorul avantaj: algoritmul este independent de setul de instrucțiuni al procesorului, ceea ce îl face aplicabil atât pentru aplicații care rulează pe arhitecturi eterogene cât și pentru acele aplicații care rulează pe arhitecturi omogene.

Un dezavantaj al algoritmului ***NUMA-BTLP*** [88] este acela că algoritmul nu ia în considerare caracteristicile dinamice ale procesoarelor cum ar fi viteza clock-ului, care poate să fie diferită de la un procesor la altul în cazul arhitecturilor eterogene, ci, în realizarea mapării firelor de execuție amânate pe cele mai puțin încărcate core-uri, algoritmul determină la compilare cel mai puțin încărcat core în funcție de numărul de operații cu întregi și de operații în virgulă flotantă realizate de firele de execuție mapate la compilare, până în acel moment, pe acel core.

Considerând că fiecare fir de execuție autonom este mapat pe un core diferit, în cazul ideal, numărul de fire de execuție autonome trebuie să fie mai mic sau egal cu numărul de core-uri fizice ale arhitecturii, așa încât fiecare fir autonom să beneficieze doar el de resursele core-ului pe care rulează (primul și/sau al doilea nivel de cache). Numărul de fire autonome poate fi extins în cazul ideal până la numărul de core-uri logice, dacă este activată opțiunea Hyper-Threading și dacă oricare două fire de execuție care rulează pe același core nu necesită mai multe resurse pentru a-și păstra aceeași performanță ca și cea înregistrată când ar rula fiecare separat pe acel core. Mai exact, două fire de execuție autonome pot rula ambele pe același core, în cazul ideal, dacă rata de eșuare a obținerii datelor din memoria cache nu crește față de rata care s-ar obține dacă firele ar rula câte unul pe același core. Prin urmare, cu cât numărul de core-uri este mai mare, cu atât performanța aplicațiilor multi-threading care utilizează un număr mare de fire de execuție autonome, este mai mare.

În cazul firelor de execuție alăturate, arhitectura nu are nici un impact asupra performanței algoritmului. Firele de execuție alăturate rulează pe aceleași core-uri, indiferent de numărul de core-uri sau de tiparul de comunicare între fire, ceea ce îmbunătățește rata de obținere a datelor din memoria cache și implicit localizarea datelor.

Firele de execuție amânate sunt mapate pe cel mai puțin încărcat core și sunt considerate alăturate relativ la firul de execuție generator și autonome relativ la firele care sunt create pe același nivel în ierarhia de creare a firelor de execuție. Maparea firelor de execuție amânate poate influența performanța algoritmului de mapare, deoarece cel mai puțin încărcat core pe care este mapat firul de execuție amânat poate fi departe în termeni de distanță NUMA de core-ul pe care firul de execuție generator al firului amânat, este mapat și care utilizează datele calculate de firul de execuție amânat.

În cel mai defavorabil caz, datele utilizate în comun de către firul de execuție amânat și de către firul de execuție generator al firului amânat, traversează toate nivelele de cache începând cu nivelul 1 de cache care deservește

core-ul pe care se execută unul dintre fire, pentru a ajunge în memoria principală, iar din memoria principală traversează înapoi toate nivelele de cache pentru a ajunge la nivelul 1 de cache care deservește core-ul pe care se execută celălalt fir. Cu cât numărul de core-uri este mai mare, cu atât cel mai defavorabil caz este mai probabil să apară.

În concluzie, un număr mare de fire de execuție amânate ar fi mapate eficient pe o arhitectură cu un număr redus de core-uri, invers decât în cazul firelor de execuție autonome.

5. CONCLUZII ȘI CONTRIBUȚII PERSONALE

NUMA-BTDM [55] 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*** [87] care clasifică firele de execuție în autonome, alăturate și amânate pe baza unei analize statice a dependențelor de date. ***NUMA-BTLP*** [87] și ***NUMA-BTDM*** [55] 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*** [45] pentru obținerea 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*** [55] folosește biblioteca ***Pthreads*** [45] 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 care utilizează ***Pthreads*** [45] 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. Am realizat acest lucru prin inserarea în codul sursă de intrare în reprezentare intermediară **LLVM IR** [2], la compilarea cu **LLVM** [2], a câte unui apel de setare a core-urilor pe care să ruleze fiecare fir de execuție
2. Eliminarea prin algoritmul ***NUMA-BTLP*** [87] a dezavantajului necunoașterii aspectelor dinamice, precum numărului 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***. Prin apelul ***pthread_setaffinity_np*** se mapează firul creat de apelul ***pthread_create***. Indiferent de numărul de ori în care este apelat ***pthread_create***, apelul ***pthread_setaffinity_np*** care-l succede, este apelat de tot atâtea ori de câte ori este apelat ***pthread_create***.
3. Definirea unor criterii statice originale de clasificare a firelor de execuție în 3 categorii și definirea acestor categorii. Dacă două fire au dependențe de date între ele (dependențe între datele transmise unui fir și datele utilizate în execuție de alt fir), am considerat că cele două fire au tipul alăturat unul în raport cu celălalt. Dacă un fir de execuție nu are dependențe de date cu nici un alt fir, atunci l-am considerat a fi de tipul denumit autonom. Dacă firul de execuție are dependențe de date doar cu firul generator, atunci acesta este de tipul amânat. Dependențele de date se identifică pe baza unei analize statice pe care am implementat-o în compilatorul **LLVM** [2], ca parte a algoritmului ***NUMA-BTLP*** [87].
4. Maparea firelor de execuție în funcție de tipul acestora. Firele de execuție autonome le distribui uniform pe core-uri, asigurând criteriul de echilibru necesar în obținerea localizării echilibrate a datelor. Un fir de execuție alăturat este distribuit pe core-urile pe care este distribuit fiecare fir în raport cu care acesta este alăturat, asigurând localizare optimă a datelor.

Firele de execuție amânate sunt distribuite pe cel mai puțin încărcat core la momentul identificării câte unui fir amânat. Încărcarea este determinată static prin însumarea numărului maxim de operații cu întregi și operații în virgulă flotantă care sunt executate de fiecare fir de execuție mapat pe acel core. Distribuirea în această manieră a firelor amânate asigură echilibrul încărcării core-urilor din sistem.

5. Integrarea algoritmilor de clasificare a firelor de execuție și de mapare a acestora în compilatorul modern **LLVM** [2]
6. Maparea firelor de execuție utilizând 2 arbori: arborele de comunicare care descrie dependențele de date între firele de execuție și arborele de generare care descrie ierarhia de generare a firelor de execuție [88] și introducerea regulilor de construire a arborelui de dependențe de date. Acestea sunt: orice fir autonom sau amânat se adaugă ca nod fiu al tuturor aparițiilor nodului care descrie firul generator, în arborele de comunicare și orice fir alăturat se adaugă ca nod fiu al tuturor nodurilor care descriu fire în raport cu care acesta este alăturat. Construirea în această manieră a arborelui de comunicare permite cunoașterea tiparului de comunicare între firele de execuție la traversarea arborelui de comunicare.
7. Maparea în fiecare nod NUMA a fiecărui fir autonom permite anticiparea cazurilor când, pe lângă aplicația optimizată de algoritmii din teză, mai rulează și o altă aplicație în sistem, pe anumite noduri NUMA, care nu mai pot fi utilizate la capacitate maximă de aplicația optimizată prin cei doi algoritmi. Astfel, execuția firelor autonome ale aplicației optimizate se mută pe alte noduri NUMA libere la acel moment, conform politicilor de planificare Linux ale planificatorului *Completely Fair Scheduler* (CFS) [80].

NUMA-BTDM [55] 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 localizare 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.

Deși ***NUMA-BTLP*** [87] inserează în cod, la compilare, apeluri adiționale de funcții care setează afinitatea CPU a fiecărui fir de execuție, la rulare, algoritmul nu degradează performanța execuției aplicațiilor testate, ci *îmbunătățește timpul de execuție și respectiv, consumul de putere pentru număr mic de fire de execuție autonome cu 2% și alăturate cu 15% și doar consumul de putere pentru un număr mediu de fire de execuție autonome cu 1%, păstrând neschimbat timpul de execuție. Prin utilizarea datelor în comun și repartizarea uniformă pe core-uri se obține localizarea echilibrată a datelor, care reduce timpul de execuție. Plasarea firelor pe aceleași core-uri determină eliminarea energiei consumate cu parcurgerea interconexiunilor, reducându-se consumul de energie (la număr mediu/mare de fire), care se reduce în plus, la scaderea timpului de execuție (la număr mic de fire).*

În încheiere, în Anexa A3 – *Metodica activităților de laborator*, am realizat o sinteză a metodelor și procedeelor didactice pe care le-am aplicat la laborator ca asistent de cercetare, în decursul studiilor doctorale și care au dat rezultate optime, conducând studenții la performanțe în domeniu. Această parte metodică exprimă sumarizat experiența acumulată în decursul practicii pedagogice și modalități de implementare pratică a noțiunilor teoretice studiate în timpul *licenței și masterului pedagogic*.

Anexe

Anexa A1 - Aplicarea algoritmilor ***NUMA-BTDM*** în mod automat și ***NUMA-BTLP*** în mod neautomat pe un exemplu de cod cu un anumit tipar de comunicare între firele de execuție

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <stdint.h>
#include <math.h>

#define NUM_THREADS 100
#define PU_PER_CPU 2

pthread_t *thr;
unsigned int *thr_type; // vectorul cu tipul fiecarui fir de executie
double * load_indicator; // numarul de operatii in virgula flotanta a fiecarui fir de
                                                                    // executie
static unsigned int current_thread_slot = 0; // indexul firului de executie curent
cpu_set_t *CPU_affinity_mask; // afinitatea CPU a fiecarui fie de executie

// Structura care pastreaza datele firelor de executie
struct thread_data_t {
    unsigned int thread_slot;
    unsigned int *pu_list;
    unsigned int pu_count;
    unsigned int is_worker;
};

struct thread_data_t *thr_data;

// Se returneaza numarul total de core-uri logice
unsigned int pthread_getnumber_pu() {
    SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    return sysinfo.dwNumberOfProcessors;
}

// Se initializeaza afinitatea CPU a fiecarui fir de executie independent
void pthread_numa_initaffinity_autonomoustpool()
{
    int i;
    for(i = 0; i < NUM_THREADS; i++)
    {
        if(thr_type[i] == 0)
        {
            if(thr_data[i].is_worker == 1) free(thr_data[i].pu_list);
            thr_data[i].pu_list = (unsigned int *)malloc(pthread_getnumber_pu() *
sizeof(unsigned int));
```

```
            thr_data[i].pu_count = 0;
            thr_data[i].is_worker = 1;
            CPU_ZERO(&CPU_affinity_mask[i]);
        }
    }
}

// Se initializeaza afinitatea CPU a tuturor firelor de executie
void pthread_numa_initaffinity_tpool()
{
    int i;
    for(i = 0; i < NUM_THREADS; i++)
    {
        thr_data[i].pu_list = (unsigned int *)malloc(pthread_getnumber_pu() *
sizeof(unsigned int));
        thr_data[i].pu_count = 0;
        thr_data[i].is_worker = 1;
        CPU_ZERO(&CPU_affinity_mask[i]);
    }
}

// Se returneaza adevarat daca unitatea de procesare este deja setata pentru firul
// respectiv indicat prin indexul current_slot, fals in caz contrar
unsigned int is_pu_set(unsigned int pu, unsigned int current_slot)
{
    int i;
    for(i = 0; i < thr_data[current_slot].pu_count; i++)
    {
        if(thr_data[current_slot].pu_list[i] == pu)
        {
            return 1;
        }
    }
    return 0;
}

// Se recalculeaza afinitatea CPU a firelor de executie independente indentificate
// pana in acel moment (numarul lor este dat de slots_so_far) conform descrierii din
// Codului 3
void pthread_numa_recalculateaffinity_autonomoustpool(unsigned int slots_so_far)
{
    pthread_numa_initaffinity_autonomoustpool();

    int k = 0, processing_unit_count = pthread_getnumber_pu();
    int current_slot = 0, pu;
    double i = 0.0, interval = (double) slots_so_far / processing_unit_count;
    if(interval < 1) interval = 1.0 / interval;
    while(k < slots_so_far && current_slot <= slots_so_far)
    {
        if(thr_type[current_slot] == 0)
        {
```

```
            pu = (int)floor(i) \% processing_unit_count;
            while(thr_data[current_slot].pu_count < processing_unit_count /
PU_PER_CPU)
            {
                if(!is_pu_set(pu, current_slot))
                {
                    thr_data[current_slot].pu_list[thr_data[current_slot].pu_count] =
                                                                                pu;
                    CPU_SET(pu, &CPU_affinity_mask[current_slot]);
                    (thr_data[current_slot].pu_count)++;
                }
                pu = (pu + PU_PER_CPU) \% processing_unit_count;
            }
            i+=interval;
            k++;
        }
        current_slot++;
    }
}

// Se afiseaza pentru fiecare fir de executie core-urile pe care ruleaza
void pthread_numa_diplayaffinity()
{
    int i, j;
    for(i = 0; i < NUM_THREADS; i++)
    {
        printf("Firul de executie \%d:", i);
        for(j = 0; thr_data[i].is_worker && j < thr_data[i].pu_count; j++)
        {
            printf("\%d ", thr_data[i].pu_list[j]);
        }
        printf("\n");
    }
}

// Se obtine unitatea de procesare cu cea mai mica incarcare
unsigned int pthread_numa_getlessloaded_pu()
{
    unsigned int processing_unit_count = pthread_getnumber_pu();
    unsigned int min_pu = 0;
    double load_indicator_per_pu, min_load_indicator_per_pu = NUM_THREADS;
    int i, j, k;

    for(i = 0; i < processing_unit_count; i++)
    {
        load_indicator_per_pu = 0.0;
        for(j = 0; j < NUM_THREADS; j++)
        {
            for(k = 0; k < thr_data[j].pu_count; k++)
            {
                if(thr_data[j].pu_list[k] == i)
```

```
            {
                load_indicator_per_pu += load_indicator[j];
            }
        }
        if(load_indicator_per_pu < min_load_indicator_per_pu)
        {
            min_load_indicator_per_pu = load_indicator_per_pu;
            min_pu = i;
        }
    }
}
return min_pu;
}

// Se seteaza afinitate CPU a firului de executie alaturat identificat prin indexul slot
void pthread_numa_setaffinity_sidebysidet(unsigned int thread_type, unsigned int
                                                    slot, unsigned int generating_slot)
{
    int processing_unit_count = pthread_getnumber_pu();
    int i,j = 0;
    if(slot == generating_slot)
    {
        thr_data[slot].pu_count = processing_unit_count / PU_PER_CPU;
        for(i = 0; i < thr_data[slot].pu_count; i++)
        {
            thr_data[slot].pu_list[i] = j;
            j += PU_PER_CPU;
            CPU_SET(thr_data[slot].pu_list[i], &CPU_affinity_mask[slot]);
        }
        return;
    }
    thr_data[slot].pu_count = thr_data[generating_slot].pu_count;
    for(i = 0; i < thr_data[generating_slot].pu_count; i++)
    {
        thr_data[slot].pu_list[i] = thr_data[generating_slot].pu_list[i];
        CPU_SET(thr_data[generating_slot].pu_list[i],&CPU_affinity_mask[slot]);
    }
}

// Se seteaza afinitate CPU a firului de executie amanat identificat prin indexul slot
void pthread_numa_setaffinity_postponedt(unsigned int thread_type, unsigned int
                                                        slot, double load_ind)
{
    unsigned int less_loaded_pu = pthread_numa_getlessloaded_pu(),
processing_unit_count = pthread_getnumber_pu();
    int i;
    for(i = 0; i < processing_unit_count / PU_PER_CPU; i++)
    {
        thr_data[slot].pu_list[i] = less_loaded_pu;
        CPU_SET(less_loaded_pu, &CPU_affinity_mask[slot]);
        less_loaded_pu += PU_PER_CPU;
```

```
    }
    thr_data[slot].pu_count = processing_unit_count / PU_PER_CPU;
}

// Se decide afinitatea CPU pentru un fir alaturat sau unul amanat identificat prin
// indexul slot si se memoreaza date legate de firele autonome
void pthread_numa_decideaffinity(unsigned int thread_type, unsigned int slot,
                                 unsigned int generating_slot, double load_ind)
{
    thr_type[slot] = thread_type;
    load_indicator[slot] = load_ind;
    switch(thread_type)
    {
        case 0: break;
        case 1: pthread_numa_setaffinity_sidebysidet(thread_type, slot,
generating_slot); break;
        case 2: pthread_numa_setaffinity_postponedt(thread_type, slot, load_ind);
break;
    }
}

// Se returneaza cel mai mic index disponibil (adica inca neasignat unui fir de
                                                                    // executie)
unsigned int getcurrentthreadslot()
{
    return current_thread_slot++;
}

// Se creaza firele de executie si se mapeaza pe core-uri imediat dupa creare
int pthread_numa_create(void *(*start)(void *), void *args)
{
    int result;
    unsigned int current_slot = getcurrentthreadslot();
    if((result = pthread_create(&thr[current_slot], NULL, *start, args)) ) {
        fprintf(stderr, "eroare la pthread_create cu codul \%d", result);
        return EXIT_FAILURE;
    }
    else // maparea firelor atunci cand crearea nu esueaza
    {
        if ((result = pthread_setaffinity_np(thr[current_slot], sizeof(cpu_set_t),
&CPU_affinity_mask[current_slot]))) {
            fprintf(stderr, "eroare la pthread_setaffinity_np cu codul \%d", result);
            return EXIT_FAILURE;
        }
    }
    return result;
}

// Calculele executate de firele de executie, fiecare asupra anumitor intervale de
// elemente din totalul celor din tabloul unidimensional args, asa incat numai firul
// generator si cel generat sa utilizeze aceleasi elemente ale matricei
```

```
void *thr_func0(void *args) {
    int i;
    for(i = *((int *)args); i < 100000/NUM_THREADS + *((int *)args); i++) {
        *((int *)args + i) = i + i;
    }
    return (void *)0;
}

int main(int argc, char **argv) {
    int i, rc;
    thr_type = (unsigned int *)malloc(NUM_THREADS*sizeof(int));
    thr = (pthread_t *)malloc(NUM_THREADS*sizeof(pthread_t));
    CPU_affinity_mask = (cpu_set_t *)malloc(NUM_THREADS*sizeof(cpu_set_t));
    thr_data = (struct thread_data_t *)malloc(NUM_THREADS*sizeof(struct
thread_data_t));
    load_indicator = (double *)malloc(NUM_THREADS*sizeof(double));

    pthread_numa_initaffinity_tpool();

    for(i = 0; i < NUM_THREADS; i++)
    {
        if(i\%6 < 2) // firele de executie a caror index indeplinesc aceasta conditie
sunt autonome
        {
            pthread_numa_decideaffinity(0,i,i,i/(NUM_THREADS-1));
            pthread_numa_recalculateaffinity_autonomoustpool(i);
        }
    }

    for(i = 0; i < NUM_THREADS; i++)
    {
        if(i\%6 == 2) pthread_numa_decideaffinity(1,i,i,i/(NUM_THREADS-1)); // fir
de executie alaturat generat de un fir autonom
        if(i\%6 == 3) pthread_numa_decideaffinity(1,i,i-1,i/(NUM_THREADS-1)); //
fir de executie alaturat generat de un alt fir de executie alaturat
        if(i\%6 == 4) pthread_numa_decideaffinity(2,i,i-2,i/(NUM_THREADS-1)); //
fir de executie amanat generat de un fir alaturat
        if(i\%6 == 5) pthread_numa_decideaffinity(2,i,i-4,i/(NUM_THREADS-1)); //
fir de executie amanat generat de un fir autonom
    }

    pthread_numa_diplayaffinity();

    // Se creeaza firele de executie si se seteaza afinitatea CPU a fiecarei fir
    void *args[100000];
    *((int *)args) = 100000;
    for (i = 0; i < NUM_THREADS; ++i) {
        if(i\%6 < 2) {
            *((int *)args) = 100000/NUM_THREADS*i;
        }
        else if(i\%6 == 2) {
```

```
            *((int *)args) = 100000/NUM_THREADS*(i-2);
        }
        else if(i\%6 == 3) {
             *((int *)args) = 100000/NUM_THREADS*(i-3);
        }
        else if(i\%6 == 4 || i\%6 == 5) {
            *((int *)args) = 100000/NUM_THREADS*(i-4);
        }
        pthread_numa_create(thr_func0, args);
    }

    // Se asteapta incheierea executiei tuturor firelor
    void *status;
    for (i = 0; i < NUM_THREADS; ++i) {
        pthread_join(thr[i], &status);
    }
    free(thr);
    free(CPU_affinity_mask);
    return EXIT_SUCCESS;
}
```

Anexa A2 - Implementarea algoritmilor ***NUMA-BTLP*** și ***NUMA-BTDM*** în **LLVM**

```
#include <vector>
#include <algorithm>
#include <pthread.h>
#include <linux/sched.h>

#include "LLVM/Transforms/Scalar/NUMABTLP.h"
#include "LLVM/Pass.h"
#include "LLVM/IR/IRBuilder.h"
#include "LLVM/InitializePasses.h"
#include "LLVM/IR/Function.h"
#include "LLVM/IR/Type.h"
#include "LLVM/IR/Constants.h"
#include "LLVM/Analysis/AliasAnalysis.h"
#include "LLVM/IR/GlobalVariable.h"
#include "LLVM/IR/Type.h"
#include "LLVM/IR/DerivedTypes.h"

using namespace LLVM;

namespace {

class ArrayType;

class NUMABTLPPass : public ModulePass {
  public:
  static char ID; // Pass ID, replacement for typeid
  NUMABTLPPass() : ModulePass(ID) {
    initializeNUMABTLPPassPass(*PassRegistry::getPassRegistry());
  }

  bool runOnModule(Module &M) override;

  void getAnalysisUsage(AnalysisUsage &AU) const override {
    getAnalysisUsage(AU);
  }
  enum ThreadType {
    AUTONOMOUS = 0, // the thread does not have data dependencies
with any other thread at the same level
    SIDEBYSIDE, // the thread has data dependencies with other
threads at the same level
    POSTPONED // the thread has data dependencies only with the
generating thread
  };

  struct Thread {
  Value * threadID;
  Value * generatingThreadID;
  unsigned char threadType;
```

```
   Function * function;
   };

   struct ThreadHierarchy {
      Thread * threadInfo;
      struct ThreadHierarchy * parentThread;
      struct ThreadHierarchy * nextBrotherThread;
      struct ThreadHierarchy * firstSonThread;
   };
   typedef struct ThreadHierarchy * ThreadHierarchyPtr;

   ThreadHierarchyPtr communicationHierarchyRoot = nullptr,
generationHierarchyRoot = nullptr;
private:
   class NUMABTDM {
   public:
      static unsigned long int logicalPUCount;
      static unsigned long int physicalPUCount;
      static unsigned long int logicalPUPerCPUCoreCount;
      struct ThreadData {
         Thread * thread;
         std::vector<unsigned int> processingUnitsList;
         unsigned int processingUnitsCount;
         cpu_set_t CPUAffinity;
         bool is_worker;
      };

      typedef struct ThreadData * ThreadDataPtr;
      static std::vector<ThreadDataPtr> threadsData;
      static std::vector<int> loadIndicator;

      static bool
recalculateAffinityForAutonomousPoolIfThreadAutonomous(ThreadHierar
chyPtr thread,unsigned long int slotsSoFar) {
         if (thread->threadInfo->threadType ==
ThreadType::AUTONOMOUS) {
            initAffinityOfAutonomousPool();
            loadIndicator.resize(logicalPUCount);
            unsigned long int autonomousSlotCount = 0, currentSlot =
0;
            unsigned int PU;
            int logicalPUPerCoreCount =
getLogicalProcessingUnitsPerCPUCoreNumber();
            double puPosition = 0.0;
            double interval = (double) slotsSoFar / physicalPUCount;
            while (autonomousSlotCount < slotsSoFar && currentSlot <=
slotsSoFar) {
               if(threadsData.at(currentSlot)->thread->threadType ==
ThreadType::AUTONOMOUS) {
                  PU = (int)floor(puPosition) % physicalPUCount;
```

```
                while (threadsData.at(currentSlot)-
>processingUnitsCount < physicalPUCount) {
                  if(!isProcessingUnitSet(PU, currentSlot)) {
                     threadsData.at(currentSlot)-
>processingUnitsList.emplace_back(PU);
                     CPU_SET(PU, &threadsData.at(currentSlot)-
>CPUAffinity);
                     loadIndicator.at(PU) ++;
                     threadsData.at(currentSlot)-
>processingUnitsCount ++;
                  }
                  PU = (PU + logicalPUPerCoreCount) %
physicalPUCount;
               }
               puPosition += interval;
               autonomousSlotCount++;
            }
            currentSlot++;
         }
         return true;
      }
      return false;
   }

   static void initAffinityOfAutonomousPool() {
      for (unsigned long int index = 0; index <
threadsData.size(); index++) {
         if (threadsData.at(index)->thread->threadType ==
ThreadType::AUTONOMOUS) {
            auto threadData = threadsData.at(index);
            if (threadData->is_worker) {
               threadData->processingUnitsList.clear();
            }
            threadData->processingUnitsCount = 0;
            threadData->is_worker = true;
            CPU_ZERO(&threadsData.at(index)->CPUAffinity);
         }
      }
   }

   static unsigned long int
getLogicalProcessingUnitsNumber(Instruction & I) {
      GlobalVariable* logicalPUCountGlobalVariable = new
GlobalVariable(*I.getModule(),
         Type::getInt32Ty(I.getContext()),
         true,
         GlobalValue::PrivateLinkage,
         0,
         ".str");
      logicalPUCountGlobalVariable->setAlignment(1);
```

```
        Constant *command =
ConstantDataArray::getString(I.getContext(), "cat /proc/cpuinfo |
awk '/^processor/{print $3}' | wc -l; \
     grep -c ^processor /proc/cpuinfo", true);
        IRBuilder<> IRBB(I.getContext());
        Value * systemCall =
IRBB.CreateSystemCallToGetProcessingUnitsNumber(command);
        Function * systemCallFunction =
dyn_cast<Function>(systemCall);
        logicalPUCountGlobalVariable-
>setInitializer(systemCallFunction);
        Instruction * logicalPUCountInstruction =
dyn_cast<Instruction>(logicalPUCountGlobalVariable);
        logicalPUCountInstruction->insertBefore(&I);
        logicalPUCount = system("cat /proc/cpuinfo | awk
'/^processor/{print $3}' | wc -l; grep -c ^processor
/proc/cpuinfo");
        return logicalPUCount;
     }

     static unsigned long int
getPhysicalProcessingUnitsNumber(Instruction & I) {
        GlobalVariable* physicalPUCountGlobalVariable = new
GlobalVariable(*I.getModule(),
        Type::getInt32Ty(I.getContext()),
          true,
          GlobalValue::PrivateLinkage,
          0,
          ".str");
        physicalPUCountGlobalVariable->setAlignment(1);
        Constant *command =
ConstantDataArray::getString(I.getContext(), "grep ^cpu\\scores
/proc/cpuinfo | uniq | awk '{print $4}'", true);
        IRBuilder<> IRBB(I.getContext());
        Value * systemCall =
IRBB.CreateSystemCallToGetProcessingUnitsNumber(command);
        Function * systemCallFunction =
dyn_cast<Function>(systemCall);
        physicalPUCountGlobalVariable-
>setInitializer(systemCallFunction);
        Instruction * physicalPUCountInstruction =
dyn_cast<Instruction>(physicalPUCountGlobalVariable);
        physicalPUCountInstruction->insertBefore(&I);
        physicalPUCount = system("grep ^cpu\\scores /proc/cpuinfo |
uniq | awk '{print $4}'");
        return physicalPUCount;
     }

     static int getLogicalProcessingUnitsPerCPUCoreNumber() {
        return (logicalPUPerCPUCoreCount = logicalPUCount /
physicalPUCount);
```

```
    }

    static bool isProcessingUnitSet(unsigned int processingUnit,
    unsigned long int currentSlot) {
      for(auto PU : threadsData.at(currentSlot)-
>processingUnitsList) {
        if (PU == processingUnit) {
          return true;
        }
      }
      return false;
    }

    static bool setAffinityOfThreadIfSideBySide(ThreadHierarchy *
thread) {
      if (thread->threadInfo->threadType ==
ThreadType::SIDEBYSIDE) {
        long int logicalPUPerCoreCount =
getLogicalProcessingUnitsPerCPUCoreNumber();
        long int slot = threadsData.size();
        if (thread->parentThread &&
          thread->parentThread->threadInfo->threadType ==
ThreadType::AUTONOMOUS &&
          thread->threadInfo->threadType ==
ThreadType::SIDEBYSIDE) {
          threadsData.at(slot)->processingUnitsCount =
logicalPUPerCoreCount;
          int j = 0;
          for (auto & PU : threadsData.at(slot)-
>processingUnitsList) {
            PU = j;
            j += logicalPUPerCoreCount;
            CPU_SET(PU, &threadsData.at(slot)->CPUAffinity);
            loadIndicator.at(PU) ++;
          }
          return true;
        }
        long int parentSlot = getIndexOfThread(thread-
>parentThread);
        threadsData.at(slot)->processingUnitsCount =
threadsData.at(parentSlot)->processingUnitsCount;
        std::copy(threadsData.at(parentSlot)-
>processingUnitsList.begin(),
        threadsData.at(parentSlot)->processingUnitsList.end(),
        threadsData.at(slot)->processingUnitsList.begin());
        for (auto PU : threadsData.at(parentSlot)-
>processingUnitsList) {
          CPU_SET(PU, &threadsData.at(slot)->CPUAffinity);
          loadIndicator.at(PU) ++;
        }
        return true;
```

```
        }
        return false;
     }

     static long int getIndexOfThread(ThreadHierarchyPtr
threadToSearch) {
        for (unsigned long int index = 0; index <
threadsData.size(); ++index) {
           Thread * T1 = dyn_cast<Thread>(threadsData.at(index)-
>thread);
           Thread * T2 = dyn_cast<Thread>(threadToSearch-
>threadInfo);
           assert(T1 && T2);
           if (T1->threadID == T2->threadID) {
              return index;
           }
        }
        return -1;
     }

     static bool setAffinityOfThreadIfPostponed(ThreadHierarchyPtr
thread) {
        if (thread->threadInfo->threadType == ThreadType::POSTPONED)
{
           int logicalPUPerCoreCount =
getLogicalProcessingUnitsPerCPUCoreNumber();
           long int lessLoadedPU = getLessLoadedPU();
           long int slot = getIndexOfThread(thread);
           for (auto & PU : threadsData.at(slot)-
>processingUnitsList) {
              PU = lessLoadedPU;
              CPU_SET(PU, &threadsData.at(slot)->CPUAffinity);
              loadIndicator.at(PU) ++;
              lessLoadedPU += logicalPUPerCoreCount;
           }
           threadsData.at(slot)->processingUnitsCount =
getLogicalProcessingUnitsPerCPUCoreNumber();
           return true;
        }
        return false;
     }

     static long int getLessLoadedPU() {
        int loadIndicatorPerPU = 0, minLoadIndicatorPerPU =
threadsData.size();
        int PUWithMinimumLoad = 0;
        for (unsigned int i = 0; i < logicalPUCount; i++) {
           loadIndicatorPerPU = 0.0;
           for(auto threadData : threadsData) {
              for(auto & PU : threadData->processingUnitsList) {
                 if (PU == i) {
```

```
                    loadIndicatorPerPU +=
loadIndicator.at(getIndexOfThreadData(threadData));
                }
            }
            if(loadIndicatorPerPU < minLoadIndicatorPerPU) {
            minLoadIndicatorPerPU = loadIndicatorPerPU;
            PUWithMinimumLoad = i;
            }
          }
        }
        return PUWithMinimumLoad;
      }

      static long int getIndexOfThreadData(ThreadDataPtr threadData)
{
      auto it = std::find(threadsData.begin(), threadsData.end(),
threadData);
      if (it != threadsData.end()) {
        return std::distance(threadsData.begin(), it);
      }
      return -1;
      }
   };

   void applyAnalysisNUMABTLPAlgoritm(ThreadHierarchyPtr thread) {
      std::vector<ThreadHierarchyPtr> dependantThreads;
      traverseInPreorderTheSubTreeWithRoot(thread->parentThread,
thread, dependantThreads);
      setThreadTypeIfPostponed(dependantThreads, thread, thread-
>parentThread);
      setThreadsTypeIfSideBySide(dependantThreads, thread);
      setThreadTypeIfAutonomous(dependantThreads, thread);
   }

   void traverseInPreorderTheSubTreeWithRoot(ThreadHierarchyPtr
currentThread, ThreadHierarchyPtr thread,
      std::vector<ThreadHierarchyPtr> dependantThreads) {
      bool isNotDependant =
applyAnalysisNUMABTLPAlgoritmForFunction(*currentThread-
>threadInfo->function, *thread->threadInfo->function);
      if (!isNotDependant) { // if thread and generatingThread are
side-by-side
        dependantThreads.emplace_back(currentThread);
      }
      for(auto sonThread = currentThread->firstSonThread;
      sonThread;
      sonThread = sonThread->nextBrotherThread) {
        traverseInPreorderTheSubTreeWithRoot(sonThread, thread,
dependantThreads);
      }
   }
```

```
   void getThreadsExecutingFunction(Function & function,
   ThreadHierarchyPtr currentThread,
   std::vector<ThreadHierarchyPtr> & threadsExecutingFunction) {
      if (currentThread) {
         if (currentThread->threadInfo->function == &function) {
            threadsExecutingFunction.push_back(currentThread);
         }
         else {
            auto sonThread = currentThread->firstSonThread;
            while(sonThread) {
               getThreadsExecutingFunction(function, sonThread,
threadsExecutingFunction);
               sonThread = sonThread->nextBrotherThread;
            }
         }
      }
   }

   Function * getFunctionExecutedByThread(ThreadHierarchyPtr thread,
   ThreadHierarchyPtr threadInCommunicationHierarchy) {
      if (threadInCommunicationHierarchy) {
         if (threadInCommunicationHierarchy->threadInfo->threadID ==
thread->threadInfo->threadID) {
            return threadInCommunicationHierarchy->threadInfo-
>function;
         }
         else {
            auto sonThread = threadInCommunicationHierarchy-
>firstSonThread;
            Function * foundFunction = nullptr;
            while(sonThread) {
               auto function = getFunctionExecutedByThread(thread,
sonThread);
               sonThread = sonThread->nextBrotherThread;
               if(function) {
               foundFunction = function;
            }
         }
         return foundFunction;
         }
      }
      return nullptr;
   }

   bool applyAnalysisNUMABTLPAlgoritmForFunction(Function &
function, Function & functionToCompareWith) {
      // if the parameter passed by reference is changed by a store
instruction,
      // which can be an innner instruction inside another
```

```
     bool isNotDependant =
isNotFunctionDataDependantWithAnyParameterOfFunction(function,
functionToCompareWith) &&
     isNotFunctionDataDependantWithAnyGlobalDataWrittenByFunction(f
unction, functionToCompareWith);
     if (!isNotDependant) {
        return false;
     }
     return true;
  }

  bool
isNotFunctionDataDependantWithAnyParameterOfFunction(Function &
function, Function & functionToCompareWith) {
     bool isParameterNotDependant = true;
     for(auto args_interator = functionToCompareWith.arg_begin();
     args_interator != functionToCompareWith.arg_end();
     ++args_interator) {
        Instruction * A = dyn_cast<Instruction>(args_interator);
        for (BasicBlock &BB : function) {
           for (Instruction &I : BB) {
              isParameterNotDependant = isParameterNotDependant &&
              isNotParameterAliasDependantWithInstruction(I,*A) &&
              isNotParameterDependantWithInnerInstructionsOfInstructi
on(I,*A, false);
              if (!isParameterNotDependant) {
                 return false;
              }
           }
        }
     }
     return true;
  }

  bool isNotParameterAliasDependantWithInstruction(Instruction & I,
Instruction & A) {
     AliasAnalysis * AA =
&getAnalysis<AAResultsWrapperPass>().getAAResults();
     AliasResult aliasResult = AA->alias(&I,&A);
     // if there is an alias between the instruction and the
parameter
     if(aliasResult != AliasResult::NoAlias) {
        GetElementPtrInst *IPtr = dyn_cast<GetElementPtrInst>(&I);
        GetElementPtrInst *APtr = dyn_cast<GetElementPtrInst>(&A);
        // if the stored value of the instruction and the parameter
are pointers to the same value
        if(IPtr && APtr && IPtr->getPointerOperand() == APtr-
>getPointerOperand()) {
           StoreInst *SI = dyn_cast<StoreInst>(&I);
           if (SI) {
              return false; // the instruction stores the parameter
```

```
            }
            for(Value::use_iterator inst_use_iterator =
I.use_begin();
            inst_use_iterator != I.use_end();
            inst_use_iterator++) {
               StoreInst *SI =
dyn_cast<StoreInst>(*inst_use_iterator);
               if (SI) {
                  return false; // the instruction stores the
parameter
               }
            }
         }
      }
      return true;
   }

   bool
isNotParameterDependantWithInnerInstructionsOfInstruction(Instructi
on & I, Instruction & arg, bool isNotDependant) {
      Value * A = dyn_cast<Value>(&arg);
      if(isNotDependant) {
         for(unsigned int opIndex = 0; opIndex < I.getNumOperands();
opIndex++) {
            Value *operand = I.getOperand(opIndex);
            Instruction *inst = dyn_cast<Instruction>(operand);
            // if there is an instruction that assigns the operand a
certain value
            if(inst) {
               StoreInst *SI = dyn_cast<StoreInst>(inst);
               if(SI && SI->getValueOperand() == A) {
                  return false;
               }
               else {
                  return (isNotDependant =
                  isNotDependant &&
                  isNotParameterAliasDependantWithInstruction(*inst,ar
g) &&
                  isNotParameterDependantWithInnerInstructionsOfInstru
ction(*inst,arg, isNotDependant));
               }
            }
         }
      }
      return false;
   }

   bool
isNotFunctionDataDependantWithAnyGlobalDataWrittenByFunction(Functi
on & function, Function & functionToCompareWith) {
      for (BasicBlock &BB : functionToCompareWith) {
```

```
        for (Instruction &I : BB) {
          auto firstOperand =
getFirstOperandOfStoreInstructionIfGlobalData(I);
          if (firstOperand) {
            if(isOperandInAnyInstructionOfFunction(function,
*firstOperand)) {
              return false;
            }
          }
      }
   }
   return true;
   }

   auto getFirstOperandOfStoreInstructionIfGlobalData(Instruction &
I) -> decltype(I.getOperand(0)) {
      StoreInst *SI = dyn_cast<StoreInst>(&I);
      if(SI) {
        const auto firstOperand = SI->getOperand(0);
        bool isOperandGlobalData = isa_impl<GlobalVariable,
decltype(*firstOperand)>::doit(*firstOperand);
        if (isOperandGlobalData) {
          return firstOperand;
        }
      }
      return nullptr;
   }

   bool isOperandInAnyInstructionOfFunction(Function & function,
Value & searchedOperand) {
      for (BasicBlock &BB : function) {
        for (Instruction &I : BB) {
          for(unsigned int opIndex = 0; opIndex <
I.getNumOperands(); opIndex++) {
            if(I.getOperand(opIndex) == &searchedOperand) {
              return true;
            }
          }
        }
      }
      return false;
   }

   void addThread(ThreadHierarchyPtr thread, ThreadHierarchyPtr
generatingThread) {
      if(!generatingThread->firstSonThread) {
        generatingThread->firstSonThread = thread;
      }
      else {
        auto sonThread = generatingThread->firstSonThread;
        while(sonThread->nextBrotherThread) {
```

```
            sonThread = sonThread->nextBrotherThread;
         }
         auto threadInCommunicationHierarchy = new struct
ThreadHierarchy;
         threadInCommunicationHierarchy->threadInfo = thread-
>threadInfo;
         threadInCommunicationHierarchy->firstSonThread = nullptr;
         threadInCommunicationHierarchy->nextBrotherThread = nullptr;
         threadInCommunicationHierarchy->parentThread =
generatingThread;
         sonThread->nextBrotherThread =
threadInCommunicationHierarchy;
      }
   }

   bool setThreadTypeIfPostponed(std::vector<ThreadHierarchyPtr>
parentsListOfThread,
   ThreadHierarchyPtr thread,
   ThreadHierarchyPtr subTreeCommunicationHierarchyRoot) {
      if (parentsListOfThread.size() == 1 &&
      parentsListOfThread.at(0)->threadInfo->threadID ==
      subTreeCommunicationHierarchyRoot->threadInfo->threadID) {
         thread->threadInfo->threadType = ThreadType::POSTPONED;
         auto subTreeRootInCommunicationHierarchy =
getThreadFromCommunicationHierarchy(subTreeCommunicationHierarchyRo
ot,
         communicationHierarchyRoot);
         addThread(thread, subTreeRootInCommunicationHierarchy);
         return true;
      }
      return false;
   }

   bool setThreadsTypeIfSideBySide(std::vector<ThreadHierarchy *>
parentsListOfThread, ThreadHierarchy * thread) {
      if (parentsListOfThread.size() > 1 && thread->threadInfo-
>threadType != ThreadType::SIDEBYSIDE) {
         thread->threadInfo->threadType = ThreadType::SIDEBYSIDE;
         for (auto parentThread : parentsListOfThread) {
            auto parentThreadInCommunicationHierarchy =
            getThreadFromCommunicationHierarchy(parentThread,
communicationHierarchyRoot);
            parentThreadInCommunicationHierarchy->threadInfo-
>threadType = ThreadType::SIDEBYSIDE;
            addThread(thread, parentThreadInCommunicationHierarchy);
         }
         return true;
      }
      return false;
   }
```

```
  bool setThreadTypeIfAutonomous(std::vector<ThreadHierarchy *>
parentsListOfThread, ThreadHierarchy * thread) {
     if (parentsListOfThread.size() == 1 ||
parentsListOfThread.at(0) == communicationHierarchyRoot) {
        thread->threadInfo->threadType = ThreadType::AUTONOMOUS;
        addThread(thread, communicationHierarchyRoot);
        return true;
     }
     return false;
  }

  ThreadHierarchyPtr getThreadFromCommunicationHierarchy(
  ThreadHierarchyPtr threadToSearch,
  ThreadHierarchyPtr threadInCommunicationHierarchy) {
     if(threadInCommunicationHierarchy) {
        if (threadInCommunicationHierarchy->threadInfo->threadID ==
threadToSearch->threadInfo->threadID &&
        threadInCommunicationHierarchy->threadInfo->function ==
threadToSearch->threadInfo->function &&
        threadInCommunicationHierarchy->parentThread ==
threadToSearch->parentThread) {
           return threadInCommunicationHierarchy;
        }
        else {
           auto threadSon = threadInCommunicationHierarchy-
>firstSonThread;
           ThreadHierarchyPtr threadToReturn = nullptr;
           while(threadSon) {
              ThreadHierarchyPtr foundThread =
getThreadFromCommunicationHierarchy(threadToSearch,threadSon);
              threadSon = threadSon->nextBrotherThread;
              if(foundThread) {
                 threadToReturn = foundThread;
              }
           }
           return threadToReturn;
        }
     }
     else {
        return nullptr;
     }
  }

  ThreadHierarchyPtr createThread(Value & threadID, Function & F) {
     ThreadHierarchyPtr thread = new struct ThreadHierarchy;
     thread->threadInfo->threadID = &threadID;
     thread->threadInfo->generatingThreadID = nullptr;
     thread->threadInfo->threadType = ThreadType::SIDEBYSIDE;
     thread->threadInfo->function = &F;
     thread->parentThread = nullptr;
```

```
    thread->nextBrotherThread = nullptr;
    thread->firstSonThread = nullptr;
    if (!communicationHierarchyRoot) {
      communicationHierarchyRoot = thread;
    }
    if (!generationHierarchyRoot) {
      generationHierarchyRoot = thread;
    }
    return thread;
  }

  void insertHwArchDetailsIfMainFunction(Module & M) {
    for (auto & F: M) {
      if (F.getName() == "main") {
        for (auto & BB : F) {
        Instruction * I =
dyn_cast<Instruction>(BB.getFirstInsertionPt());
          if (I) {
            createThread(*(dyn_cast<ConstantInt>(ConstantInt::ge
t(Type::getInt32Ty(I->getContext()), 0, false))), F);
            NUMABTLPPass::NUMABTDM::getLogicalProcessingUnitsNum
ber(*I);
            NUMABTLPPass::NUMABTDM::getPhysicalProcessingUnitsNu
mber(*I);
          }
        }
        break;
      }
    }
  }

  void insertMainThreadInCommunicationHierarchy() {
    auto threadData = NUMABTDM::ThreadDataPtr();
    threadData->thread = generationHierarchyRoot->threadInfo;
    NUMABTDM::threadsData.emplace_back(threadData);
  }

  unsigned int getNumberOfPthreadCreateCallsOfFunction(Function &
F) {
    auto numberOfSonThreads = 0;
    for(auto && BB : F) {
      for(auto & I : BB) {
        CallInst *CI = dyn_cast<CallInst>(&I);
        if (CI && CI->getName() == "pthread_create") {
          numberOfSonThreads ++;
        }
      }
    }
    return numberOfSonThreads;
  }
```

```
   void populateGenerationHierarchy(Module & M) {
      std::vector<ThreadHierarchyPtr> partitions = {};
      for (auto & F : M) {
         for (auto & BB : F) {
            for (auto & I : BB) {
               CallInst *CI = dyn_cast<CallInst>(&I);
               if (CI && CI->getName() == "pthread_create") {
                  auto FPC = getFunctionFromCallInst(*CI,M);

                  bool isSon = isFunctionParsed(*FPC, partitions);
                  bool isGenerating = isFunctionParsed(F, partitions);

                  if(!isSon && !isGenerating) {
                     // create son thread
                     Value * threadIDPC = dyn_cast<Value>(CI-
>arg_begin());
                     auto sonThread = createThread(*threadIDPC,*FPC);

                     // create generating threads
                     std::vector<Value*> threadIDs;
                     getThreadsExecutingFunction(F,M,threadIDs);
                     for (auto threadID : threadIDs) {
                        auto generatingThread =
createThread(*threadID,F);
                        addThread(sonThread,generatingThread);
                        partitions.emplace_back(generatingThread);
                     }
                  }
                  else if(!isSon && isGenerating) {
                     // create son thread
                     Value * threadIDPC = dyn_cast<Value>(CI-
>arg_begin());
                     auto sonThread = createThread(*threadIDPC,*FPC);

                     // add son to generating threads
                     std::vector<ThreadHierarchyPtr> generatingThreads;
                     getThreadsExecutingFunction(F,generationHierarchyR
oot,generatingThreads);
                     for (auto generatingThread : generatingThreads) {
                        addThread(sonThread,generatingThread);
                     }
                  }
                  else if(isSon && !isGenerating) {
                     // get son thread
                     std::vector<ThreadHierarchyPtr> sonThreads;
                     getThreadsExecutingFunction(*FPC,generationHierarc
hyRoot,sonThreads);

                     // create generating threads
```

```
                    std::vector<Value*> threadIDs;
                    getThreadsExecutingFunction(F,M,threadIDs);
                    for (auto threadID : threadIDs) {
                       auto generatingThread =
createThread(*threadID,F);
                       addThread(sonThreads.front(),generatingThread);
                       partitions.emplace_back(generatingThread);
                    }
                 }
              }
           }
        }
     }
  }

  Function * getFunctionFromCallInst(CallInst & CI, Module & M) {
     auto pthread_create_args_interator = CI.arg_begin(); // thread
ID
     assert(pthread_create_args_interator && "NUMA-BTLP algorithm
cannot be applied. Please fix argument list of pthread_create call.
Your thread ID parameter might my wrong.");
     pthread_create_args_interator += 2; // function attached
     assert(pthread_create_args_interator && "NUMA-BTLP algorithm
cannot be applied. Please fix argument list of pthread_create call.
Your function attached parameter might my wrong.");
     for (auto & F : M) {
        if (F.getName() == (*pthread_create_args_interator)-
>getName()) {
           auto argItF = F.arg_begin();
           pthread_create_args_interator ++; // args
           assert(pthread_create_args_interator && "NUMA-BTLP
algorithm cannot be applied. Please fix argument list of
pthread_create call. Your function arguments parameter might my
wrong.");
           ConstantArray * args =
dyn_cast<ConstantArray>(pthread_create_args_interator->get());
           auto argIt = args->op_begin();
           for (; argIt != args->op_end(); ++argIt) {
              if(argItF->getType() != (*argIt)->getType()) {
                 break;
              }
           }
           if (argIt == args->op_end()) {
              return &F;
           }
        }
     }
     assert(false && "NUMA-BTLP algorithm cannot be applied. Please
fix argument list of pthread_create call. Your function attached
parameter might my wrong.");
     return nullptr;
  }
```

```
  bool isFunctionParsed(Function & F,
std::vector<ThreadHierarchyPtr> & partitions) {
     bool isParsed = false;
     for(auto subtree : partitions) {
        isParsed = isParsed ||
isFunctionParsedInSubtree(subtree,F,partitions);
        if (isParsed) {
           return true;
        }
     }
     return false;
  }

  bool isFunctionParsedInSubtree(ThreadHierarchyPtr thread,
Function & F, std::vector<ThreadHierarchyPtr> & partitions) {
     if (thread) {
        if (thread->threadInfo->function == &F) {
           return true;
        }
        else {
           bool isParsed = false;
           auto son = thread->firstSonThread;
           while(son) {
              isParsed = isParsed ||
isFunctionParsedInSubtree(son,F,partitions);
              son = son->nextBrotherThread;
           }
           return isParsed;
        }
     }
     return false;
  }

  void getThreadsExecutingFunction(Function & function, Module & M,
std::vector<Value*> & threadIDs) {
     for (auto & F : M) {
          for (auto & BB : F) {
             for (auto & I : BB) {
                CallInst *CI = dyn_cast<CallInst>(&I);
                if (CI && CI->getName() == "pthread_create") {
                   auto functionPC = getFunctionFromCallInst(*CI, M);
                   if (functionPC == &function) {
                   auto threadID = CI->arg_begin();
                   Value * threadIDValue = dyn_cast<Value>(threadID);
                   threadIDs.emplace_back(threadIDValue);
                }
             }
          }
        }
     }
```

```
   }
};
}

char NUMABTLPPass::ID = 0;
std::vector<NUMABTLPPass::NUMABTDM::ThreadDataPtr>
NUMABTLPPass::NUMABTDM::threadsData = {};
std::vector<int> NUMABTLPPass::NUMABTDM::loadIndicator {0};
unsigned long int NUMABTLPPass::NUMABTDM::logicalPUCount = 1;
unsigned long int NUMABTLPPass::NUMABTDM::physicalPUCount = 1;
unsigned long int NUMABTLPPass::NUMABTDM::logicalPUPerCPUCoreCount
= 1;

Pass *LLVM::createNUMABTLPPass() { return new NUMABTLPPass(); }

INITIALIZE_PASS_BEGIN(NUMABTLPPass, "NUMABTLP", "NUMA-BTLP", false,
false)
INITIALIZE_PASS_END(NUMABTLPPass, "NUMABTLP", "NUMA-BTLP", false,
false)

bool NUMABTLPPass::runOnModule(Module &M) {
   populateGenerationHierarchy(M);
   insertHwArchDetailsIfMainFunction(M);
   insertMainThreadInCommunicationHierarchy();
   for (auto & F: M) {
      for (auto & BB : F) {
         for (auto & I : BB) {
            CallInst *CI = dyn_cast<CallInst>(&I);
            if (CI && CI->getName() == "pthread_create") {
            // thread id
            auto args_interator = CI->arg_begin();
            Value * threadID = dyn_cast<ConstantInt>(args_interator);
            assert(threadID && "NUMA-BTLP algorithm cannot be
applied. Please fix argument list of pthread_create call. Your
thread ID parameter might my wrong.");
            // function attached
            args_interator += 2;
            CallInst * CIA = dyn_cast<CallInst>(args_interator);
            Function * FA = getFunctionFromCallInst(*CIA,M);
            assert(FA && "NUMA-BTLP algorithm cannot be applied.
Please fix argument list of pthread_create call. Your function
attached parameter might my wrong.");

            std::vector<ThreadHierarchyPtr> sonThreads;
            getThreadsExecutingFunction(*FA,generationHierarchyRoot,s
onThreads);

            for (auto sonThread : sonThreads) {
               // NUMA-BTLP
               applyAnalysisNUMABTLPAlgoritm(sonThread->parentThread);
```

```
            // add thread to NUMA-BTDM
            auto threadData = NUMABTDM::ThreadDataPtr();
            threadData->thread = sonThread->threadInfo;
            NUMABTDM::threadsData.emplace_back(threadData);

            // NUMA-BTDM
            bool isAutonomous =
NUMABTDM::recalculateAffinityForAutonomousPoolIfThreadAutonomous(so
nThread,NUMABTDM::threadsData.size());
            bool isSideBySide =
NUMABTDM::setAffinityOfThreadIfSideBySide(
            getThreadFromCommunicationHierarchy(sonThread,communica
tionHierarchyRoot));
            bool isPostponed =
NUMABTDM::setAffinityOfThreadIfPostponed(
            getThreadFromCommunicationHierarchy(sonThread,communica
tionHierarchyRoot));
            cpu_set_t CPUAffinityThread;

            if (isAutonomous || isPostponed) {
               auto indexThread =
NUMABTDM::getIndexOfThread(sonThread);
               CPUAffinityThread =
NUMABTDM::threadsData.at(indexThread)->CPUAffinity;
            }

            if (isSideBySide) {
               auto indexGeneratingThread =
NUMABTDM::getIndexOfThread(sonThread->parentThread);
               auto indexThread =
NUMABTDM::getIndexOfThread(sonThread);
               CPU_AND(&NUMABTDM::threadsData.at(indexGeneratingThr
ead)->CPUAffinity,
                  &NUMABTDM::threadsData.at(indexGeneratingThread)-
>CPUAffinity,
                  &NUMABTDM::threadsData.at(indexThread)-
>CPUAffinity);
               CPUAffinityThread =
NUMABTDM::threadsData.at(indexGeneratingThread)->CPUAffinity;
            }

            std::vector<Constant *> CPUAffinities;
            for (long unsigned int i = 0; i < CPU_SETSIZE / (8 *
sizeof(unsigned long)); i++) {
               // CPUAffinity[i]
               ConstantInt * CI =
ConstantInt::get(Type::getInt64Ty(I.getContext()),CPUAffinityThread
.__bits[i],false);
               CPUAffinities.emplace_back(dyn_cast<Constant>(CI));
            }
```

```
            // types of CPU affinities
            std::vector<Type *> CPUAffinitiesType = {};
            std::transform(CPUAffinities.begin(),CPUAffinities.end(
),CPUAffinitiesType.begin(),[&](Constant * CPUAffinity) {
               return (Type *)CPUAffinity;
            });

            ArrayRef<Type *> ART(CPUAffinitiesType);
            ArrayRef<Constant *> ARC(CPUAffinities);
            Value * CPUset =
dyn_cast<ConstantStruct>(ConstantStruct::get(StructType::get(I.getC
ontext(),ART,false),ARC));
            Value * CPUsetsize =
dyn_cast<ConstantInt>(ConstantInt::get(IntegerType::getInt32Ty(I.ge
tContext()),sizeof(NUMABTDM::threadsData.back()-
>CPUAffinity),false));

            IRBuilder<> IRBB(I.getContext());
            Instruction * setaffinityI = dyn_cast<Instruction>(
            IRBB.CreatePThreadSetAffinityNpCall(threadID,CPUsetsize
,CPUset));
            setaffinityI->insertAfter(&I);
            }
         }
      }
   }
  }
  return false;
}
```

Anexa A3 - Metodica activităților de laborator

Cursul de Programarea Calculatoarelor
Anul I, Calculatoare și Tehnologia Informației

Sistemul de învățământ universitar este oportunitatea prin care tinerii absolvenți de liceu fac trecerea spre o nouă formă de învățământ diferită de cea din mediul preuniversitar. Studenții din anul I de la Calculatoare și Tehnologia Informației se deprind cu forme noi de a asimila cunostiintele precum: învățarea prin studiu individual, învățarea prin descoperire și rezolvarea problemelor prin studii de caz.

Activitatea prin lucrări de laborator permite înțelegerea mai bune a noțiunilor prezentate la curs, dar și aprofundarea aspectele pragmatice ale disciplinei studiate.

Predarea-învățarea respectă următorii pași care se reiau ciclic:

1. Stabilirea obiectivelor cognitive și a etapelor de lucru, de către cadrul didactic
2. Trasnmiterea cunoștiințelor si asimilarea acestora de către student
3. Oferirea de răspunsuri clarificatoare studenților prin autosesizarea cadrului didactic sau semnalarea de către student a neînțelegerii unor informații
4. Sesizarea abaterilor între rezultatele așteptate cu privire la informațiile asimilate de studenți și rezultatele obținute de aceștia

Competențele pe care asistentul dorește să le formeze la studenți prin intermediul obiectivelor cognitive, sunt formulate la începutul activității. În primă fază, în vederea realizării lucrării de laborator, asistentul prezintă studenților etapele de lucru, fără a fi nevoit încă să-și adapteze strategia didactică de transmitere a informațiilor. Din momentul în care studenții recepționează informațiile și implementează lucrarea pe calculator, intervine rolul corectiv al asistentului, care pentru fiecare etapă de lucru, compară rezultatele preconizate a fi obținute, cu rezultatele reale ale studenților, așa încât prin diferența lor se obțin abaterile sau erorile de la prescrierea inițială.

Acesta este momentul în care asistentul îndrumător de laborator trebuie să modifice strategia didactică, respectiv să utilizeze o gamă diversificată de metode și procedee didactice, pliate pe fiecare student în parte, în funcție de particularitățile lui cognitive, dar și strategia didactică la modul general, dacă studenții nu se aliniază standardelor de realizare a lucrării de laborator.

O lucrare de laborator și-a atins obiectivele atunci când diferența între ceea ce trebuie să cunoască studentul și rezultatele reale ale acestuia tinde spre zero.

Ca și metodologie didactică consider adecvate folosirea de către asistent a metodelor de învățământ activ-participative de tipul stimul-reacție (de exemplu: observă rezultatul dacă modifici ...), metoda problematizării (cu întrebări de tipul: ce se obține în urma adăugării ... ?), a conversației euristice (cu întrebări de tipul: cum ai putea să exprimi acestea în termenii introduși astăzi…?) și metoda descoperirii (cu întrebări de tipul: ce alte metode de rezolvare mai sunt?).

Eșalonarea informațiilor în pași logici, prezentate concis, este una din cerințele reușitei transmiterii și asimilării informațiilor de către studenți.

O astfel de abordare este foarte complexă, îndeosebi în faza de pregătire a lucrării de laborator, întrucât necesită studiul metodologiei didactice adică a metodelor și procedeelor de care dispune un cadru didactic, în funcție de specificul activității desfășurate: teoretică sau practică, în funcție de resursele materiale de

care dispune, de resursa de timp și de resursa umană adică de nivelul de cunoștințe cu care studentul începe activitatea și gradul de atenție al acestuia.

O analiză atentă trebuie acordată perturbațiilor care intervin în activitatea studentului în timpul realizării lucrării de laborator. Aceste perturbații pot fi externe activității de predare-învățare precum nevoi personale, dar și de natură intrinsecă, în special motivația, implicarea, emoția, în general starea interioară.

Din acest punct de vedere, rolul asistentului este de crea o atmosferă propice studiului, caldă, apropiată de nevoile studentului, un raport bazat pe încredere și respect reciproc în același timp.

Bibliografie

[1] Iulia Știrb. Îmbunătățirea performanței și a consumului de energie prin localizare echilibrată a datelor la execuția programelor paralele pe sisteme NUMA. Raport de cercetare, pag. 40, 2017.
[2] The LLVM compiler infrastructure project. https://LLVM.org/, 2019. Accessed: 9.10.2019.
[3] Nakul Manchanda and Karan Anand. Non-uniform memory access (NUMA). New York University, 4, 2010.
[4] Manu Awasthi, David Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. În Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on, pages 319–330. IEEE, 2010.
[5] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Philippe OA Navaux, and Israel Koren. Affinity-based thread and data mapping in shared memory systems. ACM Computing Surveys (CSUR), 49(4):64, 2017.
[6] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Mohammad S Alhakeem, Philippe OA Navaux, and Hans-Ulrich Heiß. Locality and balance for communication-aware thread mapping in multicore systems. În European Conference on Parallel Processing, p. 196–208. Springer, 2015.
[7] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. În Proceedings of the 19th international conference on Parallel architectures and compilation techniques, p. 557–558. ACM, 2010.
[8] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. În ACM SIGOPS Operating Systems Review, volume 41, p. 47–58. ACM, 2007.
[9] Matthias Diener, Eduardo HM Cruz, Laércio L Pilla, Fabrice Dupros, and Philippe OA Navaux. Characterizing communication and page usage of parallel applications for thread and data mapping. Performance Evaluation, 88:18–36, 2015.
[10] Michael Frasca, Kamesh Madduri, and Padma Raghavan. NUMA-aware graph mining techniques for performance and energy efficiency. În Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 95. IEEE Computer Society Press, 2012.
[11] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. MODESTO: data-centric analytic optimization of complex stencil programs on heterogeneous architectures. În Proceedings of the 29th ACM on International Conference on Supercomputing, p. 177–186. ACM, 2015.
[12] Michael Dennis. Static performance prediction of compiler optimizations. Machine Learning: Theory and Applications, p. 34, 2015.
[13] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent C/C++ programs. În 2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), p. 216–226. IEEE, 2016.
[14] VK Gautama and PK Singh. An overview of compiler based cache optimization techniques. Int. J. of Scientific Research And Education, 4(3), 2016.
[15] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. Predictive modeling methodology for compiler phase-ordering. În Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms, p. 7–12. ACM, 2016.

[16] Cédric Bastoul. Mapping deviation: a technique to adapt or to guard loop transformation intuitions for legality. În Proceedings of the 25th International Conference on Compiler Construction, p. 229–239. ACM, 2016.
[17] Eun Jung Park. Automatic selection of compiler optimizations using program characterization and machine learning. PhD thesis, University of Delaware, 2015.
[18] Hao Zhou and Jingling Xue. A compiler approach for exploiting partial SIMD parallelism. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):11, 2016.
[19] Polyhedral optimizations for LLVM. 2019.
[20] GCC, the GNU compiler collection. 2019.
[21] Aditya Kumar and Sebastian Pop. SCoP detection: A fast algorithm for industrial compilers. În 6th International Workshop on Polyhedral Compilation Techniques on IMPACT, 2016.
[22] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4):451–490, 1991.
[23] Sebastian Buchwald, Denis Lohner, and Sebastian Ullrich. Verified construction of static single assignment form. În Proceedings of the 25th International Conference on Compiler Construction, p. 67–76. ACM, 2016.
[24] Hao Zhou and Jingling Xue. A compiler approach for exploiting partial SIMD parallelism. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):11, 2016.
[25] Masatomo Hashimoto, Masaaki Terai, Toshiyuki Maeda, and Kazuo Minami. Extracting facts from performance tuning history of scientific applications for predicting effective optimization patterns. În Proceedings of the 12th Working Conference on Mining Software Repositories, p. 13–23. IEEE Press, 2015.
[26] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J Harrison, and Ponnuswamy Sadayappan. On fusing recursive traversals of Kd trees. În Proceedings of the 25th International Conference on Compiler Construction, p. 152–162. ACM, 2016.
[27] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The pluto+ algorithm: a practical approach for parallelization and locality optimization of affine loop nests. ACM Transactions on Programming Languages and Systems (TOPLAS), 38(3):12, 2016.
[28] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J Ramanujam, Atanas Rountev, and P Sadayappan. Distributed memory code generation for mixed irregular/regular computations. În ACM SIGPLAN Notices, vol. 50, p. 65-75. ACM, 2015.
[29] Iulia Știrb and Horia Ciocârlie. Improving performance and energy consumption with loop fusion optimization and parallelization. În Computational Intelligence and Informatics (CINTI), 2016 IEEE 17th International Symposium on, p. 000099-000104. IEEE, 2016.
[30] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. ACM Transactions on Computer Systems (TOCS), 28(4):8, 2010.
[31] Lukasz Szustak, Krzysztof Rojek, Tomasz Olas, Lukasz Kuczynski, Kamil Halbiniak, and Pawel Gepner. Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. Scientific Programming, 2015:10, 2015.
[32] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: A high-productivity programming language for HPC with logical regions. În

Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 81. ACM, 2015.
[33] Karthik Murthy and John Mellor-Crummey. A compiler transformation to overlap communication with dependent computation. În 2015 9th International Conference on Partitioned Global Address Space Programming Models (PGAS), p. 90–92. IEEE, 2015.
[34] Antoine Fraboulet, Guillaume Huard, and Anne Mignotte. Loop alignment for memory accesses optimization. În Proceedings of the 12th international symposium on System synthesis, p. 71. IEEE Computer Society, 1999.
[35] Imen Fassi and 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.
[36] Matthias Diener, Eduardo HM Cruz, Laércio L Pilla, Fabrice Dupros, and Philippe OA Navaux. Characterizing communication and p. usage of parallel applications for thread and data mapping. Performance Evaluation, 88:18–36, 2015.
[37] Luiz GA Martins, Ricardo Nobre, João MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):8, 2016.
[38] LizheWang, Samee U Khan, Dan Chen, Joanna KołOdziej, Rajiv Ranjan, Cheng-Zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. Future Generation Computer Systems, 29(7):1661–1670, 2013.
[39] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and KirkWCameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. IEEE Transactions on Parallel and Distributed Systems, 21(5):658-671, 2010.
[40] Susanne Albers, Fabian Müller, and Swen Schmelzer. Speed scaling on parallel processors. Algorithmica, 68(2):404–425, 2014.
[41] Message passing interface. 2017.
[42] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5):720–748, 1999.
[43] Robert Armstrong, Debra Hensgen, and Taylor Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. În Proceedings of the 1998 7th Heterogeneous Computing Workshop (HCW 98), p. 79–87. IEEE, 1998.
[44] Types of parallel jobs. https://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/types-of-jobs/, 2019. Accessed: 16.08.2019.
[45] Posix Threads. http://man7.org/linux/man-p./man7/pthreads.7.html, 2019. Accessed: 16.08.2019.
[46] Luiz GA Martins, Ricardo Nobre, João MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):8, 2016.
[47] LizheWang, Samee U Khan, Dan Chen, Joanna Koªodziej, Rajiv Ranjan, Cheng-Zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. Future Generation Computer Systems, 29(7):1661-1670, 2013.
[48] Susanne Albers, Fabian Müller, and Swen Schmelzer. Speed scaling on parallel processors. Algorithmica, 68(2):404-425, 2014.

[49] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe OA Navaux. Tabarnac: Visualizing and resolving memory access issues on numa architectures. În Proceedings of the 2nd Workshop on Visual Performance Analysis, p. 1. ACM, 2015.
[50] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. ACM Computing Surveys (CSUR), 46(4):47, 2014.
[51] The OpenMP API specification for parallel programming. 2016.
[52] Posix Threads. 2017.
[53] MPICH. high-performance portable MPI. 2017.
[54] Open MPI: Open source high performance computing. 2017.
[55] 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.
[56] Goran Velkoski, Sasko Ristov, and Marjan Gusev. Loosely or tightly coupled affinity for matrix-vector multiplication. În 2013 36th International Convention on Information & Communication Technology Electronics & Microelectronics (MIPRO), p. 228–233. IEEE, 2013.
[57] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. ACM SIGARCH Computer Architecture News, 33(4):80–91, 2005.
[58] François Pellegrini and 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, 1996.
[59] Haris Ribic and Yu David Liu. Energy-efficient work-stealing language runtimes. În ACM SIGARCH Computer Architecture News, vol. 42, p. 513_528. ACM, 2014.
[60] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. În 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p. 180–186. IEEE, 2010.
[61] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. IEEE Transactions on Parallel and Distributed Systems, 25(4):993–1002, 2014.
[62] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. În Proceedings of the 10th International Parallel Processing Symposium (IPPS'96), p. 314–319. IEEE, 1996.
[63] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing, 20(1):359–392, 1998.
[64] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. Parallel hypergraph partitioning for scientific computing. În 20th International Parallel and Distributed Processing Symposium (IPDPS 2006) p. 10–pp. IEEE, 2006.
[65] Eduardo HM Cruz, Matthias Diener, and Philippe OA Navaux. Using the translation lookaside buffer to map threads in parallel applications based on shared memory. În 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), p. 532–543. IEEE, 2012.

[66] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of mpi processes on hierarchical NUMA architectures. Euro-Par 2010-Parallel Processing, p. 199–210, 2010.
[67] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, and Philippe OA Navaux. An efficient algorithm for communication-based task mapping. În 2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p. 207–214. IEEE, 2015.
[68] Petar Radojkovic, Vladimir Cakarevic, Javier Verdú, Alex Pajuelo, Francisco J Cazorla, Mario Nemirovsky, and Mateo Valero. Thread assignment of multithreaded network applications in multicore/multithreaded processors. IEEE Transactions on Parallel and Distributed Systems, 24(12):2513–2525, 2013.
[69] Jesper Larsson Traff. Implementing the MPI process topology mechanism. În Supercomputing, ACM/IEEE 2002 Conference, p. 28–28. IEEE, 2002.
[70] D Solt. A profile-based approach for topology aware MPI rank placement. În High Performance Computation Conference (HPCC), Springer Lecture Notes in Computer Sciences, 2007.
[71] Gheorghe Cascaval, Evelyn Duesterwald, Stephen E Smith, Peter F Sweeney, and Robert W Wisniewski. Method and system for optimizing communication in MPI programs for an execution environment, 20 Noiembrie 2008. US Patent App. 11/750,912.
[72] François Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs. În 1994 Proceedings of the Scalable High-Performance Computing Conference, p. 486–493. IEEE, 1994.
[73] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. În USENIX Annual Technical Conference, p. 277–289, 2015.
[74] Timothy Brecht. On the importance of parallel application placement in NUMA multiprocessors. În Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), p. 1–18, 1993.
[75] Manuel F Dolz, Francisco D Igual, Thomas Ludwig, Luis Piñuel, and Enrique S Quintana-Ortí. Balancing task-and data-level parallelism to improve performance and energy consumption of matrix computations on the Intel Xeon Phi. Computers & Electrical Engineering, 46:95-111, 2015.
[76] Zoltan Majo and Thomas R Gross. A library for portable and composable data locality optimizations for numa systems, vol. 50. ACM, 2015.
[77] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. În IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), p. 1-8. IEEE, 2008.
[78] sched - overview of CPU scheduling. http://man7.org/linux/man-p./man7/sched.7.html, 2019. Accessed: 16.08.2019.
[79] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over numa architectures: an openmp runtime perspective. În International Workshop on OpenMP, p. 79–92. Springer, 2009.
[80] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the linux scheduler. ACM SIGOPS Operating Systems Review, 42(5):34–43, 2008.
[81] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. În ACM Sigplan Notices, vol. 44, p. 65–74. ACM, 2009.

[82] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. În International Conference on High Performance Computing for Computational Science, p. 1–25. Springer, 2010.
[83] B Dally. Gpu computing: To exascale and beyond. Lecture slides–http://www. nvidia.com/content/PDF/sc, 2010.
[84] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. What do programmers know about software energy consumption? IEEE Software, 33(3):83–89, 2016.
[85] Rajiv Nishtala, Daniel Mossé, and Vinicius Petrucci. Energy-aware thread co-location in heterogeneous multicore processors. În Proceedings of the 11th ACM International Conference on Embedded Software, p. 21. IEEE Press, 2013.
[86] Yingxin Wang, Yan Cui, Pin Tao, Haining Fan, Yu Chen, and Yuanchun Shi. Reducing shared cache contention by scheduling order adjustment on commodity multi-cores. În 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), p. 984–992. IEEE, 2011.
[87] Iulia Știrb. NUMA-BTLP: A static algorithm for thread classification. În 2018 5th International Conference on Control, Decision and Information Technologies (CoDIT), p. 882–887. IEEE, 2018.
[88] Iulia Știrb. Extending NUMA-BTLP algorithm with thread mapping based on a communication tree. Computers, 7(4):66, 2018.
[89] CPU-X. https://github.com/X0rg/CPU-X, 2019. Accessed: 16.08.2019.
[90] CPU-benchmarking. https://github.com/pdpriyanka/CPU-Benchmarking, 2016. Accessed: 16.08.2019.
[91] contextswitch. https://github.com/tsuna/contextswitch, 2016. Accessed: 16.08.2019.