Algoritmi Paraleli și Distribuiți
Introducere

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Despre curs

- Interactivitate – dialog
- Open-office: ????, PRECIS 605
- Regulament afișat pe pagina cursului
  - Forma de evaluare a examenului final poate fi decisă/schimbată de către titularul de curs, în funcție de un context specific. Forma de evaluare poate fi: scris sau oral.
- Punctajele obținute pe parcurs sau examen se pot păstra pentru un an universitar (nu acumulare)
- Temele se pot trimite doar pe parcursul primului semestru universitar

Vineri, 9-11 AM

Despre curs

- Nota:
  10 = **6** (Parcurs) + **4** (Examen)

Laborator+Curs **>= 3** Examen **>= 2**

Parcurs: 3 teme(3p) + laborator (2p) +
Evaluare (1p)

Întrebări?

Despre mine

Distributed
Systems
Networks
Simulation
Opportunistic
Mobile
Data
Large
Cloud
Computing
Analysis
Scale
Environments
Monitoring
based
using
Intelligent
Platform
Applications
System
Services
Grid
Framework
Scheduling
Transportation
distributed
Context-Aware
Dissemination
Model
Management
networks
data
Routing
Security
Algorithms
Modeling
Dynamic
Using
Integrated
Based
Control
Approach
Towards
Social
Living
Smart
Peer-to-Peer
Collaborative
Architecture
Detection
Vehicular
Optimization
Internet
Network
Public
Activities
MONARC
Resource
Dependable
Replication
Replication
Urban
Algorithm
Grid
Big
Agent
Assisted
Mobile
Cloud
Service
Systems
Data
Management
Control
Approach
Towards
Dissemination
Environments
Scale
Computing
Analysis
Context-Aware
Mobile
Data
Intelligent
Platform
using
based
Monitoring
Systems
Architecture
Peer-to-Peer
Collaborative
Monitoring
management
Simulation
Opportunistic
Networks
Distributed
Large
Cloud
Applications
System
Services
Transportation
Framework
Scheduling
Grid
Routing
Big
Using
Agent
Model
Management
networks

Bibliografia

G.R.Andrews
Concurrent Programming. Principles and Practice

C.A.R. Hoare
Communicating Sequential Processes

S.G. Akl
The Design and Analysis of Parallel Algorithms

G.R. Andrews, R.A. Olsson
The SR Programming Language. Concurrency in Practice

Ian Foster
Designing and Building Parallel Programs

A.S. Tanenbaum
Structured Computer Organization (Fourth Edition)

Allen B. Downey
The Little Book of Semaphores

Wan Fokkink
Distributed Algorithms: An Intuitive Approach (2nd edition)

Nancy A. Lynch
Distributed Algorithms

De ce calcul paralel și distribuit?

- Timp de execuție mai scurt
- Permite abordarea problemelor de dimensiuni mari
- Accesul resurselor aflate la distanță
- Reducerea costurilor
- Toleranță la defecte
- Ascunderea timpilor de așteptare
- Redundanța
- Scalabilitatea
- Scăderea timpului de răspuns
- Securitate

Limitele programării secvențiale?

Cramming More Components onto Integrated Circuits

GORDON E. MOORE, LIFE FELLOW, IEEE

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65 000 components on a single silicon chip.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits

Each approach
each borrowed te
believe the way
various approach

The advocates
already using th
resistors by apply
conductor substra
upon films are de
attachment of active semiconductor devices to the passive film arrays.

Both approaches have worked well and are being used in equipment today.

Limitele programării secvențiale?

cofounded Intel

Limitele programării secvențiale

- Viteza de transmisie
  - Maxim c – viteza luminii
- Miniaturizare
  - Tranzistor de mărimea unui atom
- Economic
  - Costuri enorme pentru cercetare și proiectarea unui nou timp de procesor

Limitele programării secvențiale

- CMOS

Importanța cursului

- Apar tot mai multe tehnologii distribuite
  - Blockchain; Peer-to-Peer
- Chiar și un procesor de ceas are mai multe core-uri
  - LG Watch Sport - MSM8909w Processor
    - Quad-Core
- Suport în noile IDE-uri
  - Eclipse; Visual Studio
- Număr mare de aplicații distribuite
  - Dropbox; Spark; Boinc

Importanța cursului

- Aproape toate aplicațiile folosesc mai multe thread-uri

Importanța cursului

Task Manager

File Options View

Processes | Performance | App history | Start-up | Users | Details | Services

Name	PID	Status	Username	CPU	Memory (p...	Threads	Description
explorer.exe	4948	Running	cristian.chi...	00	154,348 K	260	Windows Explorer
System	4	Running	SYSTEM	00	20 K	240	NT Kernel & System
Dropbox.exe	7900	Running	cristian.chi...	00	161,068 K	148	Dropbox
NVIDIA Web Helper.	21388	Running	cristian.chi...	00	29,024 K	96	NVIDIA Web Helper Service
Origin.exe	13304	Running	cristian.chi...	00	100,008 K	96	Origin
nvcontainer.exe	5752	Running	SYSTEM	00	30,000 K	86	NVIDIA Container
firefox.exe	28740	Running	cristian.chi...	00	424,460 K	85	Firefox
nvcontainer.exe	8964	Running	NETWORK...	00	11,180 K	83	NVIDIA Container
firefox.exe	10140	Running	cristian.chi...	00	770,132 K	73	Firefox
Skype.exe	9156	Running	cristian.chi...	00	225,100 K	66	Skype
POWERPNT.EXE	27828	Running	cristian.chi...	10	183,364 K	64	Microsoft PowerPoint
firefox.exe	28840	Running	cristian.chi...	00	890,532 K	64	Firefox
CorsairLink4.Service.	15076	Running	SYSTEM	01	40,780 K	58	Corsair LINK 4 Service
firefox.exe	23840	Running	cristian.chi...	00	506,796 K	57	Firefox
firefox.exe	22076	Running	cristian.chi...	00	602,072 K	56	Firefox
BitTorrent.exe	1168	Running	cristian.chi...	00	67,916 K	54	BitTorrent
MsMpEng.exe	29124	Running	SYSTEM	00	114,688 K	50	Antimalware Service Executable
SearchUI.exe	10212	Suspended	cristian.chi...	00	102,652 K	49	Search and Cortana application
FortiTray.exe	20760	Running	cristian.chi...	00	5,240 K	48	FortiClient System Tray Controller
OVRServer_x64.exe	7968	Running	cristian.chi...	00	53,356 K	46	OVRServer_x64.exe 676007-public SC:67765493906
Steam.exe	13576	Running	cristian.chi...	00	309,492 K	44	Steam Client Bootstrapper
googledrivesync.exe	10116	Running	cristian.chi...	00	178,840 K	42	googledrivesync.exe

Supercomputers

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR , Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9	

Summit

Sierra

Sunway TaihuLight

Computing power 2018

24-hour average: 28.019 PetaFLOPS.
Active: 156,141 volunteers, 648,324 computers.

Computing power 2019

24-hour average: 23.702 PetaFLOPS.
Active: 135,048 volunteers, 473,747 computers.

Algoritmi Paraleli/Distribuiți vs Secvențiali

Resurse fizice

- Procesor – multi-core – 28 core-uri

- Cluster

- Grid/Cloud

Taxonomia Flynn

Some Computer Organizations and Their Effectiveness

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—A hierarchical model of computer organizations is developed, based on a tree model using request/service type resources as nodes. Two aspects of the model are distinguished: logical and physical.

General parallel- or multiple-stream organizations are examined as to type and effectiveness—especially regarding intrinsic logical difficulties.

The overlapped simplex processor (SISD) is limited by data dependencies. Branching has a particularly degenerative effect.

The parallel processors [single-instruction stream–multiple-data stream (SIMD)] are analyzed. In particular, a nesting type explanation is offered for Minsky's conjecture—the performance of a parallel processor increases as log M instead of M (the number of data stream processors).

Multiprocessors (MIMD) are subjected to a saturation syndrome based on general communications lockout. Simplified queuing models indicate that saturation develops when the fraction of task time spent locked out (L/E) approaches $1/n$, where n is the number of processors. Resources sharing in multiprocessors can be used to avoid

more "macr
particular us
must be sha
more signific
1) There i
limiting reso
est will eithe
tions will ap
figurations.
computer wi
cerned with
potential, wh
siderations.
2) We ma
sets. It is ass
set of instru

Taxonomia Flynn

- SISD
  - Single Instruction Stream, Single Data Stream
  - Calculatorul Clasic one-core
- SIMD
  - Single Instruction Stream, Multiple Data Streams
  - Suportul SSE; procesoare GPU
- MISD
  - Multiple Instruction Streams, Single Data Stream
  - Sisteme specializate
- MIMD
  - Multiple Instruction Streams, Multiple Data Streams
  - Procesoare actuale (ce aveți acasă și în buzunar)

SISD

- Model clasic Arhitectura *von Neumann*

SISD

- Model clasic Arhitectura *von Neumann*

First Draft of a Report
on the EDVAC

by

John von Neumann

SIMD

C
P_1
P_2
P_n
M
M
M
prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
load A(2)
load B(2)
C(2)=A(2)*B(2)
store C(2)
next instruct
P2
prev instruct
load A(n)
load B(n)
C(n)=A(n)*B(n)
store C(n)
next instruct
Pn
time

MISD

C_1
C_2
C_n
P_1
P_2
P_n
M

prev instruct
load A(1)
C(1)=A(1)*1
store C(1)
next instruct
P1
prev instruct
load A(1)
C(2)=A(1)*2
store C(2)
next instruct
P2
prev instruct
load A(1)
C(n)=A(1)*n
store C(n)
next instruct
Pn
time

MIMD

C_1
P_1
M_1
C_2
P_2
M_2
C_n
P_n
M_n

prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
call funcD
x=y*z
sum=x*2
call sub1(i,j)
next instruct
P2
prev instruct
do 10 i=1,N
alpha=w**3
zeta=C(i)
10 continue
next instruct
Pn
time

Memorie Partajată

- Shared memory
  - Parallel Random Access Memory - **PRAM**
  - **EREW** – Exclusive Read Exclusive Write
  - **CREW** – **Concurrent Read Exclusive Write --** cel mai des întâlnit
  - **ERCW** – Exclusive Read Concurrent Write
  - **CRCW** – Concurrent Read Concurrent Write
  - O variabilă poate fi citită într-un pas în model **CR** dar în log(N) în model **ER**.

Memorie Partajată

- Uniform Memory Access
  - **UMA**

Memorie Partajată - Acces

Memorie Distribuită

- Massively Parallel Processors
- Network of workstations
- Non-Uniform Memory Access
  - **NUMA**

Memorie Distribuită - Acces

Rețele de configurare

- Topologii
  - Tablou
  - Arbore
  - Cub
  - Hipercub

- Depinde de
  - Aplicație
  - Performanțe dorite
  - Număr procesoare disponibile
- Exemple: IBM 9000, Cray C90, Fujitsu VP

Tipuri de paralelism

- La nivel de bit
- La nivel de instrucțiune
- La nivel de task

Tipuri de paralelism

- La nivel de bit
- La nivel de instrucțiune
- La nivel de task

Tipuri de paralelism

- La nivel de bit
- La nivel de instrucțiune
- La nivel de task

C = A + B

Adunarea a doi vectori

Tipuri de paralelism

- La nivel de bit
- La nivel de instrucțiune
- La nivel de task
  - **Multi-Tasking (pot comunica și procesele)**
  - **Multi-Threading**

Tipuri de paralelism

- La nivel de bit
- La nivel de instrucțiune
- La nivel de task
  - **Multi-Tasking (pot comunica și procesele)**
  - **Multi-Threading**

Cum pot comunica două procese?

Notații pseudocod

```
co S1 || S2 || ... || Sn oc
```

Ex.1:

```
x=0; y=0;
co x=x+1 || y=y+1 oc
z=x+y;
```

Notații pseudocod

```
co [cuantificator]{Sj}
```

Ex. 2:

```
co [j=1 to n] {a[j]=0; b[j]=0;}
```

```
process Name[cuatificatori] { Sj }
```

Threads vs cores

New 8-Core Intel® Core™ i7 Processor Extreme Edition

Intel® Core™ i7-5960X Processor Extreme Edition
Transistor count: 2.6 Billion
Die size: 17.6mm x 20.2mm

* 20MB of cache is shared across all 8 cores

Procese și thread-uri

- Proces
  - Instanță a unui program în execuție
  - Pentru un proces, SO alocă:
    - Un spațiu în memorie (codul programului, zona de date, stiva)
    - Controlul anumitor resurse (fișiere, dispozitive I/O, …)

Thread (fir de execuție)

- Un proces poate avea mai multe fire de execuție
- Fir de execuție (thread):
  - Flux de control secvențial în cadrul unui proces
- Firele de execuție
  - împart același spațiu de adrese;
  - au în comun: zonele de cod și date din memorie, resursele procesului
  - zona de stivă – reprezintă starea curentă a thread-ului și este proprie thread-ului

Multi-proces vs. multi-thread

Execuția thread-urilor

→ Putem considera execuție în paralel

Defapt:

→ planificarea este realizată de către sistemul de operare

→ politici: divizarea timpului (*time sharing*), priorități

Hyperthreading – the confusion

Thank you

Hyperthreading – the confusion

New 8-Core Intel® Core™ i7 Processor Extreme Edition

Intel® Core™ i7-5960X Processor Extreme Edition
Transistor count: 2.6 Billion
Die size: 17.6mm x 20.2mm

* 20MB of cache is shared across all 8 cores

Multi-tasking vs Multi-threading

Multi-tasking

Task 1

Task 2

Multi-tasking

Task 2

Task 1

Multi-tasking

Task 1

Task 2

Multi-tasking

Task 2

Task 1

Multi-threading

Thread 2

Thread 3

Thread 4

Thread 1

Multi-threading

Thread 1

Thread 2

Thread 4

Thread 3

Multi-threading

Thread 1

Thread 2

Thread 4 Thread 3

Multi-threading

Thread 2

Thread 3

Thread 4

Thread 1

Multi-threading

Thread 1

Thread 4

Thread 2 Thread 3

Multi-tasking vs Multi-threading

Care sunt diferențele?

Un proces are mai multe thread-uri

Thread-urile unui proces împart memoria

Multi-tasking vs Multi-threading

Poți avea multi-tasking pe mai multe core-uri?

Poți avea mai multe thread-uri pe un singur core?

Multi-tasking vs Multi-threading

Poți avea multi-tasking pe mai multe core-uri? **DA**

Poți avea mai multe thread-uri pe un singur core? **DA**

Notiuni cu care vom lucra

Race condition

Thread 1	Thread 2
a = a + 2	**a = a + 2**

Care este valoare lui a în acest moment?

Race condition

a = 0

Thread 1	Thread 2
a = a + 2	**a = a + 2**

Care este valoare lui a în acest moment?

Race condition

a = 0

Thread 1	Thread 2
a = a + 2	**a = a + 2**

What is the value of a?

4

Race condition

a = 0

Thread 1	Thread 2
a = a + 2	**a = a + 2**

What is the value of a?

4 **AND** 2

Un alt exemplu

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax =	**eax =**

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 0	**eax =**

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 0	**eax = 0**

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 0**

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**

eax = 2 **eax = 2**

Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 2**

Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 2**

Un pas si mai departe...

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax =	**eax =**

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 0	**eax =**

Race condition

a = 0

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**

eax = 2

eax =

Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**

eax = 2

eax =

Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 2**

Race condition

a = 2

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
eax = 2	**eax = 4**

Race condition

a = 4

Thread 1	Thread 2
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**

eax = 2

eax = 4

Primitive de sincronizare

Synchronization primitives

- Atomics
- Semaphore
  - Binary semaphore (Mutex)
  - Critical section
- Barrier

Atomics

- Fie operația de 64 biți pe un procesor de 64 biți **C = A + B**

load(A, eax)

load(B, ebx)

eax = eax + ebx

write(C, eax)

Atomics

- Fie operația de 64 biți pe un procesor de 32 biți **C = A + B**

```
load(A[0], eax)
load(B[0], ebx)
eax = eax + ebx
write(C[0], eax)
load(A[1], eax)
load(B[1], ebx)
eax = eax + ebx
write(C[1], eax)
```

Atomics

- Fie operația de 64 biți pe un procesor 32 biți **C = A + B**

```
load(A[0], eax)
load(B[0], ebx)
eax = eax + ebx
write(C[0], eax)
load(A[1], eax)
load(B[1], ebx)
eax = eax + ebx
write(C[1], eax)
```

Putem avea doar jumătate de **C** modificat

Atomics

- Fie operația de 64 biți pe un procesor 32 biți **C = A + B**

```
load(A[0], eax)
load(B[0], ebx)
eax = eax + ebx
write(C[0], eax)
load(A[1], eax)
load(B[1], ebx)
eax = eax + ebx
write(C[1], eax)
```

Atomicitatea asigură că **C** va fi vizibil doar complet modificat, sau complet nemodificat.

Excludere mutuală - mutex

Mutual exclusion – soluția lui Dekker

EWD35 – 1

EWD35.htm

About the sequentiality of process descriptions.

Over de sequentialiteit van procesbeschrijvingen.

Het is niet ongebruikelijk, wanneer een spreker zijn
een inleiding. Omdat ik mij hier mogelijk richt tot een g
minder vertrouwd is met de problematiek, die ik wil aansn ,
die ik zal moeten gebruiken, wilde ik in dit geval ter in
inleidingen houden, nl. een om de achtergrond van de prob
een tweede, om U een gevoel te geven voor de aard van de
wij tegen het lijf zullen lopen.

Theodorus (Dirk) J. Dekker

Mutual exclusion – soluția lui Dijsktra

Solution of a Problem in Concurrent Programming Control

E. W. Dijkstra
Technological University, Eindhoven, The Netherlands

A number of mainly independent sequential-cyclic processes with restricted means of communication with each other can be made in such a way that at any moment one and only one of them is engaged in the "critical section" of its cycle.

Introduction

Given in this paper is a solution to a problem for which, to the knowledge of the author, has been an open question since at least 1962, irrespective of the solvability. The paper consists of three parts: the problem, the solution, and the proof. Although the setting of the problem might seem somewhat academic at first, the author trusts that anyone familiar with the logical problems that arise in

computer can only request one one-way message at a time.
And only this w
this problem is f

The Solution

The common
"Boo
The integer
will only be set
by the others. It
well outside the
mentioned set to
The program

```
"integer j;
Li0:  b[i] := false
Li1:  if k ≠ i the
Li2:  begin c[i] :
Li3:  if b[k] then
      go to Li1
      end
        else
```

Soluția lui Dekker

```
wants_to_enter[0] = true
while (wants_to_enter[1]) {
        if (turn != 0) {
                wants_to_enter[0] = false
                while (turn != 0) { // busy wait }
                wants_to_enter[0] = true
        }
}
// critical section ...
turn = 1
wants_to_enter[0] = false
```

Soluția lui Dijsktra

```
b[i] = fals
while(sw[i]) {
        sw[i] = F
        if (k!=i) {
                c[i] = true
                if(b[k])
                        k = i
                sw[i] = T
        } else {
        c[i] = false
        for(j=0;j<N;j++)
                if(j!=i && !c[j])
                        sw[i] = T
        }
}
//critical
b[i] = true
c[i] = true
```

lock() - P

unlock() - V

flag[0] =
flag[0] =

Soluția lui Peterson

```
flag[0] = true;
P0_gate: turn = 1;
while (flag[1] && turn == 1) { // busy wait }
// critical section ...
flag[0] = false;
```

Soluție cu asistare hardware

```
while (test_and_set(lock));
// critical section
lock = 0
```

Race condition – soluție

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 0**

Race condition – soluție

?

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition – soluție

OK

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition – soluție

?

a = 2

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition – soluție

OK

a = 2

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition – soluție

?

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 0**

Race condition – soluție

OK

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 0**

Race condition – soluție

?

a = 2

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 0**

Race condition – soluție

OK

a = 2

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 0**

Race condition – soluție

?

a = 4

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 2**

Race condition – soluție

OK

a = 4

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 2	**eax = 2**

Race condition – soluție

?

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

Race condition – soluție

a = 0

Thread 1	Thread 2
lock(locka)	**lock(locka)**
load(a, eax)	**load(a, eax)**
eax = eax + 2	**eax = eax + 2**
write(a, eax)	**write(a, eax)**
unlock(locka)	**unlock(locka)**
eax = 0	**eax = 2**

IMPOSIBIL

Algoritmi Paraleli și Distribuiți
Complexitate algoritmi paraleli

Prof. Ciprian DOBRE
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Performanța

- Timp de execuție
- Memorie ocupată
- Număr de procese (thread-uri)
- Scalabilitate
- Toleranță la defecte
- Cost
- Eficiență
- Fiabilitate
- Portabilitate

Măsuri

- T - Timpul total necesar execuției programului paralel
- P - Numărul de procesoare utilizate
- G – Timp execuție **cel mai rapid algoritm secvențial**
- S – Speedup
  - $S = \frac{G}{T}$

- Costul C = T * P

- Eficienţa $E = \frac{G}{C} = \frac{G}{TP} = \frac{S}{P}$

Complexitate

9	6	9	4	2	7	6	5	6	…	1

* 3

27	18	27	12	6	21	18	15	18	…	3

Complexitate secvențială?

Complexitate

* 3

Complexitate secvențială?

Complexitate

* 3

Complexitate secvențială?

Complexitate

9	6	9	4	2	7	6	5	6	…	1

* 3

27	18	27

Complexitate secvențială?

Complexitate

9	6	9	4	2	7	6	5	6	…	1

* 3

27	18	27	12

Complexitate secvențială?

Complexitate

9	6	9	4	2	7	6	5	6	…	1

* 3

27	18	27	12	6	21	18	15	18	…	3

Complexitate secvențială? O(**N**)

Complexitate

* 3

Complexitate paralelă?

Complexitate

Complexitate paralelă? **O(1)** ?

Complexitate

9	6	9	4	2	7	6	5	6	…	1

* 3

27	18	27	12	6	21	18	15	18	…	3

Complexitate paralelă? **O(1)** ? **P** = **N**

Complexitate

9	6	9	4	2	7	6	5	6	…	1

* 3

27	18	27	12	6	21	18	15	18	…	3

Complexitate paralelă? $\mathbf{O}(\frac{N}{P})$

Complexitate

Speedup?

Complexitate

Speedup?

$$T = O\left(\frac{N}{P}\right)$$

Complexitate

Speedup?

T = O($\frac{N}{P}$)

G = O(N)

Complexitate

Speedup?

T = O($\frac{N}{P}$)

G = O(N)

$$S = \frac{O(N)}{O(\frac{N}{P})}$$

Complexitate

Speedup?

T = O($\frac{N}{P}$)

G = O(N)

$$S = \frac{O(N)}{O(\frac{N}{P})} = O(P)$$

Complexitate

$$S = \frac{O(N)}{O(\frac{N}{P})} = O(P)$$

Eficiența?

T = O($\frac{N}{P}$)

G = O(N)

Complexitate

$$S = \frac{O(N)}{O(\frac{N}{P})} = O(P)$$

Eficiența?

T = O($\frac{N}{P}$)

G = O(N)

$$E = \frac{S}{P} = \frac{O(P)}{P} = 1$$

Legea lui Amdhal

Amdahl's law

Validity of the single processor approach to achieving large scale computing capabilities[1]

Gene M. Amdahl

IBM Sunnyvale, California

1 INTRODUCTION

For over a decade prophets have voiced the contention that the o
has reached its limits and that truly significant advences can be m
multiplicity of computers in such a manner as to permit cooperati
direction has been pointed out as general purpose computers wi
of memories, or as specialized computers with geometrically relat
controlled by one or more instruction streams.

Legea lui Amdahl

f - Procent de operații din algoritmul secvențial care **NU** se pot executa paralel.

$$T = fG + (1 - f)\frac{G}{P}$$

$$S = \frac{G}{T}$$

Legea lui Amdahl

f - Procent de operații din algoritmul secvențial care **NU** se pot executa paralel.

$$T = fG + (1 - f)\frac{G}{P}$$

$$S = \frac{G}{T} = \frac{G}{fG + \frac{(1-f)G}{P}} = \frac{1}{f + \frac{(1-f)}{P}}$$

Legea lui Amdahl

$$S = \frac{1}{f + \frac{(1-f)}{P}}$$

Ce se întâmplă dacă P e foarte mare (chiar mai mare decât N)?

Legea lui Amdahl

$$S = \frac{1}{f + \frac{(1-f)}{P}}$$

Ce se întâmplă dacă P e foarte mare (chiar mai mare decât N)?

$$\boldsymbol{S \leq \frac{1}{f}}$$

Legea lui Amdahl

f - Procent de operații din algoritmul secvențial care **NU** se pot executa paralel.

$$T = fG + (1 - f)\frac{G}{P}$$

Ce se întâmplă dacă P e foarte mare (chiar mai mare decât N)?

Legea lui Amdahl

f - Procent de operații din algoritmul secvențial care **NU** se pot executa paralel.

$$T = fG + (1 - f)\frac{G}{P}$$

Ce se întâmplă dacă P e foarte mare (chiar mai mare decât N)?

$$\boldsymbol{T \geq fG}$$

Legea lui Amdahl

Legea lui Amdahl

Exemplu: sortarea unui vector

Bubble sort

9	6	9	4	2	7	6	5	6	1

6	9	9	4	2	7	6	5	6	1

6	9	9	4	2	7	6	5	6	1

6	9	4	9	2	7	6	5	6	1

6	9	4	2	9	7	6	5	6	1

Bubble sort

Repetă
până e
sortat

Bubble sort

6 9 4 2 7 6 5 6 1 9
6 4 9 2 7 6 5 6 1 9
6 4 2 9 7 6 5 6 1 9
6 4 2 7 9 6 5 6 1 9
6 4 2 7 6 9 5 6 1 9

Bubble sort

6	4	2	7	6	5	9	6	1	9

6	4	2	7	6	5	6	9	1	9

6	4	2	7	6	5	6	1	9	9

6	4	2	7	6	5	6	1	9	9

.....

Bubble sort

.....

Complexitate?

Bubble sort

4	6	2	7	6	5	6	1	9	9

.....

Complexitate: **O(n^2)**
Un pas trece prin toate elementele
Garantat să termine după n repetiții

1	2	4	5	6	6	6	7	9	9

Parallel bubble sort

Cum paralelizăm?

Parallel bubble sort

Parallel bubble sort

Aceste două operații (și toate perechile similare)
NU pot fi executate în același timp

Solution hint: Nu e necesară execuția
operațiilor într-o anumită ordine

OETS: Odd-Even Transposition Sort

OETS: Odd-Even Transposition Sort

Pot fi executate în paralel

.....

Pot fi executate în paralel

.....

.....

OETS: Odd-Even Transposition Sort

OETS: Odd-Even Transposition Sort

NU pot fi executate în paralel

Ce facem? Ce folosim?

.....

OETS: Odd-Even Transposition Sort

Soluția:
Barrier
între
fiecare
repetiție

.....

OETS: Odd-Even Transposition Sort

OETS: Odd-Even Transposition Sort

OETS: Odd-Even Transposition Sort

OETS: Odd-Even Transposition Sort

Complexitate a soluției paralele?

OETS: Odd-Even Transposition Sort

Complexitate a soluției paralele?

$$T = O\left(\frac{N}{P} * N\right) (= O(N) \text{ pentru } P = N)$$

.....

OETS: Odd-Even Transposition Sort

Speedup?

$$T = O\left(\frac{N}{P} * N\right) (= O(N) \text{ pentru } P = N)$$

.....

OETS: Odd-Even Transposition Sort

Speedup?

$$S = \frac{N^2}{\frac{N^2}{P}} = P$$

.....

OETS: Odd-Even Transposition Sort

Speedup? **Sigur**?

$$S = \frac{N^2}{\frac{N^2}{P}} = P$$

.....

OETS: Odd-Even Transposition Sort

Speedup?

$$S = \frac{N log_2 N}{\frac{N^2}{P}}$$

.....

OETS: Odd-Even Transposition Sort

Speedup?

$$S = \frac{Plog_2N}{N}$$

.....

OETS: Odd-Even Transposition Sort

Nu uitați: Așteptatul la barieră poate introduce timpi mari!

$$S = \frac{Plog_2N}{N}$$

.....

Shear sort
(Row-column sort)
(Snake sort)

9	6	9	4
2	7	6	5
9	3	6	2
5	4	1	5

Sortează fiecare linie **pară** în mod **ascendent**
Sortează fiecare linie **impară** în mod **descendent**

Shear sort

9	6	9	4
2	7	6	5
9	3	6	2
5	4	1	5

Sortează crescător coloanele

Shear sort

Repetă tot de $log_2 n$ ori

Shear sort

De ce sortăm liniile pare crescător și cele impare descrescător?

Shear sort

De ce liniile pare crescător și celelalte descrescător?
Dorim compararea celui mai mare element de pe linia i cu cel mai mic de pe linia i+1

Shear sort

Shear sort

2	3	4	1
4	5	5	2
5	6	6	9
7	6	9	9

Shear sort

Shear sort

1	2	3	2
5	5	4	4
5	6	6	6
9	9	7	9

Shear sort

1	2	2	3
5	5	4	4
5	6	6	6
9	9	9	7

Shear sort

1	2	2	3
5	5	4	4
5	6	6	6
9	9	9	7

Shear sort

Lista finală se obține citind în formă de șerpuită (snake sort)

Shear sort

1	2	2	3
5	5	4	4
5	6	6	6
9	9	9	7

1	2	2	3	4	4	5	5	5	6	6	6	7	9	9	9

Shear sort

9	6	9	4
2	7	6	5
9	3	6	2
5	4	1	5

Complexitate? $G =$

Shear sort

Complexitate? $G = log_2 N \ *$
$log_2 N$ repetiții

Shear sort

Complexitate? $G = log_2 N \ * \ 2 \ * \ \sqrt{N}$

$\sqrt{N}$ linii/coloane

Shear sort

Complexitate? $G = log_2 N \ * \ 2 \ * \ \sqrt{N} \ * \ \sqrt{N} * \ log_2 \sqrt{N}$

$\sqrt{N} * \ log_2 \sqrt{N}$ cel mai bun algoritm secvențial de sortare

Shear sort

Complexitate? $G = log_2 N \ * N \ * \ log_2 \sqrt{N}$

Shear sort

Complexitate? $G = Nlog_2N \ * \ log_2\sqrt{N}$

Cel mai bun algoritm secvențial rămâne $Nlog_2N$

Shear sort

Cum paralelizăm?

Shear sort

Cum paralelizăm?
Toate liniile în paralel, barieră, apoi toate coloanele, apoi barieră, și repetăm.

Shear sort

Cum paralelizăm?
Toate liniile în paralel, barieră, apoi toate coloanele, apoi barieră, și repetăm.

Shear sort

Complexitate versiune paralelă?

$$G = log_2 N \ * \ 2 \ * \ \sqrt{N} \ * \ \sqrt{N} * \ log_2 \sqrt{N}$$

Shear sort

Complexitate versiune paralelă? N = P

$$T = log_2 N \ * \ 2 \ * \ \sqrt{N} * log_2 \sqrt{N}$$

Shear sort

Complexitate versiune paralelă? N = P

$$T = \sqrt{N}\, log_2 \sqrt{N} * log_2 N$$

Shear sort

Complexitate versiune paralelă?

$$T = log_2 N \ * \ 2 \ * \ \frac{\sqrt{N}}{P} * \ \sqrt{N} * \ log_2 \sqrt{N}$$

Shear sort

Complexitate versiune paralelă?

$$T = \frac{N}{P} log_2 N \ * \ log_2 \sqrt{N}$$

Shear sort

Speedup?

$$T = \frac{N}{P} log_2 N \ * \ log_2 \sqrt{N}$$

$$G = \ N log_2 N$$

Shear sort

$$S = \frac{Nlog_2N}{\frac{N}{P}log_2N \ * \ log_2\sqrt{N}}$$

Speedup?

$$T = \frac{N}{P}log_2N \ * \ log_2\sqrt{N}$$

$$G = \ Nlog_2N$$

Shear sort

$$S = \frac{P}{log_2\sqrt{N}}$$

Speedup?

$$T = \frac{N}{P} log_2 N \ * \ log_2\sqrt{N}$$

$$G = \ Nlog_2 N$$

Rank Sort

9	4	2	7	6	5	6	1

1	2	4	5	6	6	7	9

Rank Sort

9 4 2 7 3 5 6 1
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?

Rank Sort

9 4 2 7 3 5 6 1
0
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?

Rank Sort

9 4 2 7 3 5 6 1
0
5
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?

Rank Sort

9 4 2 7 3 5 6 1
0
5
4
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?

Rank Sort

9 4 2 7 3 5 6 1
0
5
4
2
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?

Rank Sort

9 4 2 7 3 5 6 1
0
5
4
2
6
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?
Câte numere sunt mai mici ca mine?

Rank Sort

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Rank Sort

Câte numere sunt mai mici ca mine?

Rank Sort

Rank Sort

Poziții în vectorul final

Rank Sort

Rank Sort

Rank Sort

Rank Sort

7 3 1 6 2 4 5 0
Poziție în vectorul final
9 4 2 7 3 5 6 1
1 2 3 4
0 1 2 3 4 5 6 7

Rank Sort

7 3 1 6 2 4 5 0
Poziție în vectorul final
9 4 2 7 3 5 6 1
1 2 3 4 5
0 1 2 3 4 5 6 7

Rank Sort

7 3 1 6 2 4 5 0
Poziție în vectorul final
9 4 2 7 3 5 6 1
1 2 3 4 5 6
0 1 2 3 4 5 6 7

Rank Sort

7 3 1 6 2 4 5 0
Poziție în vectorul final
9 4 2 7 3 5 6 1
1 2 3 4 5 6 7
0 1 2 3 4 5 6 7

Rank Sort

7 3 1 6 2 4 5 0
Poziție în vectorul final
9 4 2 7 3 5 6 1
1 2 3 4 5 6 7 9
0 1 2 3 4 5 6 7

Parallel Rank Sort

9	4	2	7	6	5	6	1

1	2	4	5	6	6	7	9

Parallel Rank Sort

Toate întrebările pot fi răspunse în paralel.

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Câte numere sunt mai mici ca mine?

Parallel Rank Sort

Toate întrebările pot fi răspunse în paralel.

Parallel Rank Sort

Parallel Rank Sort

Toate întrebările pot fi răspunse în paralel.

Inmultiri de matrici

Matrix multiply

```
for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
                for(k = 0; k < N; k++)
                        c[i][j] += a[i][k] * b[k][j]
```

Matrix multiply

```
for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
                for(k = 0; k < N; k++)
                        c[i][j] += a[i][k] * b[k][j]
```

Complexitate?

Matrix multiply

```
for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
                for(k = 0; k < N; k++)
                        c[i][j] += a[i][k] * b[k][j]
```

Complexitate?

$$G = N^3$$

Matrix multiply

Atenție avem N^2 elemente

```
for(i = 0; i < N; i++)
        for(j = 0; j < N; j++)
                for(k = 0; k < N; k++)
                        c[i][j] += a[i][k] * b[k][j]
```

Complexitate?

$$G = N^3$$

Matrix multiply

```
co[Tid = 1 to P]
{
        start = Tid * ceil(N/P)
        end = min((Tid+1) * ceil(N/P),N)
        for(i = start; i < end; i++)
                for(j = 0; j < N; j++)
                        for(k = 0; k < N; k++)
                                c[i][j] += a[i][k] * b[k][j]
}
```

Matrix multiply

Complexitate?

```
co[Tid = 1 to P]
{
        start = Tid * ceil(N/P)
        end = min((Tid+1) * ceil(N/P),N)
        for(i = start; i < end; i++)
                for(j = 0; j < N; j++)
                        for(k = 0; k < N; k++)
                                c[i][j] += a[i][k] * b[k][j]
}
```

Matrix multiply

Complexitate?

$$T = \frac{N^3}{P}$$

```
co[Tid = 1 to P]
{
        start = Tid * ceil(N/P)
        end = min((Tid+1) * ceil(N/P),N)
        for(i = start; i < end; i++)
                for(j = 0; j < N; j++)
                        for(k = 0; k < N; k++)
                                c[i][j] += a[i][k] * b[k][j]
}
```

Matrix multiply

Speedup?

$$T = \frac{N^3}{P}$$

```
co[Tid = 1 to P]
{
        start = Tid * ceil(N/P)
        end = min((Tid+1) * ceil(N/P),N)
        for(i = start; i < end; i++)
                for(j = 0; j < N; j++)
                        for(k = 0; k < N; k++)
                                c[i][j] += a[i][k] * b[k][j]
}
```

Matrix multiply

Speedup?

```
co[Tid = 1 to P]
{
        start = Tid * ceil(N/P)
        end = min((Tid+1) * ceil(N/P),N)
        for(i = start; i < end; i++)
                for(j = 0; j < N; j++)
                        for(k = 0; k < N; k++)
                                c[i][j] += a[i][k] * b[k][j]
}
```

$$S = \frac{N^3}{\frac{N^3}{P}}$$

Matrix multiply

Speedup?

$$S = P$$

```
co[Tid = 1 to P]
{
        start = Tid * ceil(N/P)
        end = min((Tid+1) * ceil(N/P),N)
        for(i = start; i < end; i++)
                for(j = 0; j < N; j++)
                        for(k = 0; k < N; k++)
                                c[i][j] += a[i][k] * b[k][j]
}
```

Super-linear speedup?

$S > P$?

- *"... the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."*

Amdhal, 1967

- *"... speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size."*

Gustafson, 1988

Algoritmi Paraleli și Distribuiți
Lucru cu semafoare

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

Sincronizare: Notiuni introductive

Dezvoltarea algoritmilor folosind variabile partajate (MIMD)

- Sincronizarea: excluderea mutuală şi sincronizarea condiţionată.
- Câteva rezultate notabile:
  - mutex
  - semafoare
    - regiuni critice
  - bariere
- Câteva dintre problemele *clasice*:
  - producători-consumatori
  - problema filozofilor
  - cititori-scriitori
  - problema bărbierului

Sectiuni critice

- Problema: fiecare proces P(i) al unei colectii de procese P(i:1..n) executa ciclic o sectiune critica, in care are acces exclusiv la anumite resurse partajate, urmata de o sectiune necritica in care foloseste doar resurse locale.

- Semafor = tip special de variabile partajate manipulate prin 2 operații atomice: P și V *(Dijkstra, 1965)*
  - Asigura excluderea mutuala intre procesele care acceseaza sectiunea critica.

Secțiuni critice

```
sem mutex= 1;
process P[i=1 to n] {
    while (true) {
      P(mutex); /* acaparare secțiune critică */
     Secțiune critică;
      V(mutex); /* eliberare secțiune critică */
     Secțiune necritică
    }
}
```

Exemplu executie…

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

```
P() sau Proberen
V() sau Verhogen
 - Dijsktra
```

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

```
P() sau Proberen
V() sau Verhogen
 - Dijsktra
```

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

```
P() sau Proberen
V() sau Verhogen
 - Dijsktra
```

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

```
P() sau Proberen
V() sau Verhogen
 - Dijsktra
```

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

```
P() sau Proberen
V() sau Verhogen
 - Dijsktra
```

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Zonă Critică

P() sau Proberen
V() sau Verhogen
- Dijsktra

Doar un thread poate fi în zona critică la un moment dat

Producator-consumator

Producer - Consumer

EWD209 - 0

EWD209.html

A Constructive Approach to the Problem of Program Correctness.

Summary. As an alternative to methods by which the correctness of given programs can be established a posteriori, this paper propo of program generation such as to produce a priori is treated to show the form that such a control t comes from the field of parallel programming; the is representative for the way in which a whole mu actually been constructed.

Introduction.

The more ambitious we become in our machine becomes the problem of program correctness. The c

Producători și consumatori

- Problema
  - Se consideră M producători și N consumatori care comunică printr-un singur buffer partajat.

Producători și consumatori

```
typeT buf; /* T - tipul datelor */
sem gol= 1, plin= 0; /* sem binare */
```

```
process Producător[i=1 to M]{
    typeT v;
    while(true) {
      v = produce();
      P(gol);
      buf = v;
      V(plin);
    }
}
```

```
process Consumator[i=1 to N]{
    typeT w;
    while(true) {
      P(plin);
      w = buf;
      V(gol);
      consumă(w);
    }
}
```

semnalizare

Comunicarea producător și consumator prin tampon limitat

```
typeT buf[1:k];
sem gol = k, plin = 0;  /* semafoare generale */
process Producător {
      typeT v;
      int ultim= 1;
      while (true) {
        v = produce();
        P(gol);                        /* există locuri goale? */
        buf[ultim] = v;
        ultim = ultim mod k + 1; /* circular */
        V(plin);
      } }
process Consumator {
      typeT w;
      int prim= 1;
      while (true) {
        P(plin);                       /* există valori în buffer? */
        w = buf[prim]; prim = prim mod k + 1;
        V(gol);
        consumă(w);
      }
}
```

Comunicarea producător și consumator prin tampon limitat

Mai mulți producători și mai mulți consumatori

```
typeT buf[1:k];
int prim = 1, ultim = 1;
sem gol = k, plin = 0;
sem mutexP = 1, mutexC = 1;
```

```
process Producător[i=1 to M]{
    typeT v;
    while (true) {
      v = produce();
      P(gol);
        P(mutexP);
        buf[ultim] = v;
        ultim = ultim mod k + 1;
        V(mutexP);
      V(plin);
    }
}
```

```
process Consumator[i=1 to N]{
    typeT w;
    while (true) {
      P(plin);
        P(mutexC);
        w = buf[prim];
        prim = prim mod k + 1;
        V(mutexC);
      V(gol);
      consumă(w);
    }
}
```

sectiune critica

semnalizare

Problema filozofilor

Problema filozofilor

```
process Filozof[i=1 to 5]{
    while (true) {
        ia bețișoare;
        mănâncă;
        eliberează bețișoare;
        gândește;
    }
}
```

DEADLOCK

Fiecare mai are nevoie de un bețișor

Problema filozofilor (2)

```
sem f[1:5] = ([5]1);
process Filozof[i=1 to 4]{
      while (true) {
        P(f[i]); P(f[i+1]);
        mănâncă;
        V(f[i]); V(f[i+1]);
        gândește;
} }
process Filozof[i=5]{
      while (true) {
        P(f[1]); P(f[5]);
        mănâncă;
        V(f[1]); V(f[5]);
        gândește;
} }
```

Problema cititori-scriitori

Problema cititorilor și scriitorilor
Excludere mutuală

zona de memorie

(resursa critica)

1) Un singur scriitor are dreptul sa scrie la un moment dat,
2) Doar daca nu exista un cititor curent care citeste

1) Un cititor poate citi din memorie, indiferent daca deja exista alti cititori ce deja citesc
2) Doar daca nu exista un scriitor care scrie

Problema cititorilor și scriitorilor
Excludere mutuală

```
sem rw = 1;
process Cititor[i=1 to m]{
     while (true){
        P(rw);
        citește din resursa comună;
        V(rw);
}    }
process Scriitor[j=1 to n]{
     while (true){
        P(rw);
        scrie în resursa comună;
        V(rw);
}    }
```

Excludere mutuala strica !
Un singur proces are voie sa execute opeartii pe resursa critica…

R
Stop
zona de memorie
(resursa critica)
G
Clear

Problema cititorilor si scriitorilor
Excludere mutuală (2)

```
int nr = 0; sem mutexR = 1, rw = 1;
process Cititor[i=1 to m]{
      while (true){
        P(mutexR);
          nr = nr + 1;
          if (nr == 1) P(rw);    /* dacă primul cititor */
        V(mutexR);
        citește din resursa comună;
        P(mutexR);
          nr = nr - 1;
          if (nr == 0) V(rw);    /* dacă ultimul cititor */
        V(mutexR);
}     }
process Scriitor[j=1 to n]{
      while (true){
        P(rw);
        scrie în resursa comună;
        V(rw);
}     }
```

instructiune cu garda

sectiune critica

R
Stop
zona de memorie
(resursa critica)
G
Clear
nr = 0
nr = 1
nr = 2

Problema cititorilor și scriitorilor
Sincronizare condiționată

- Invariant global:

```
RW: ( nr == 0 || nw == 0 ) && nw <= 1
```

- Proces Reader:
  - Incrementarea `nr` e condiționată de `(nw == 0)`
- Proces Writer:
  - Incrementarea `nw` e condiționată de `(nr == 0 && nw == 0)`
- Decrementarea nu trebuie condiționată

Problema cititorilor și scriitorilor
Sincronizare condiționată (2)

```
int nr = 0, nw = 0;
process Cititor[i=1 to m]{
     while (true){
       <await (nw == 0) nr = nr + 1>
       citește din resursa comună;
       <nr = nr - 1>
}    }
process Scriitor[j=1 to n]{
     while (true){
       <await (nr == 0 && nw == 0) nw = nw + 1>
       scrie în resursa comună;
       <nw = nw - 1>
}    }
```

Problema cititorilor și scriitorilor
Sincronizare condiționată (3)

Politici:

- noile cereri de la cititori sunt întârziate dacă un scriitor așteaptă
- un cititor întârziat este trezit doar dacă nu există un scriitor în așteptare

```
int nr = 0;   /* nr. cititori care folosesc resursa */
int nw = 0;   /* nr. scriitori care folosesc resursa */
sem e = 1;    /* intrare secțiune atomică */
sem r = 0;    /* intarzie cititori daca nw>0 sau dw>0 */
sem w = 0;    /* intarzie scriitori daca nw>0 sau nr>0 */
int dr = 0;   /* nr. cititori întârziați */
int dw = 0;   /* nr. scriitori întârziați */
```

Problema cititorilor și scriitorilor Sincronizare condiționată (4)

Split binary semaphore

- Folosit pentru a implementa atât excluderea mutuală cât și sincronizarea condiționată.
- Semafoarele e, r și w formează împreună un semafor ***splitat*** *(split binary semaphore)*:
  - cel mult un semafor este 1 la un moment dat $0 \leq e + r + w \leq 1$
  - fiecare cale de execuție începe cu un P și se termină cu un singur V
  - instrucțiunile între P și V se execută în excludere mutuală.
- Tehnica se numește ***pasarea ștafetei***:
  - Inițial un semafor este 1 și un proces poate prelua ștafeta printr-o operație P asupra semaforului
  - când un proces deține ștafeta (se execută într-o secțiune critică și toate semafoarele sunt 0), el poate pasa ștafeta altui proces printr-o operație V asupra unuia din cele trei semafoare.

Problema cititorilor și scriitorilor
Sincronizare condiționată (5)

```
process Cititor[i=1 to m]{
   while (true){
     P(e);
     if (nw > 0 or dw > 0){
       dr = dr + 1; V(e); P(r);
     }
     nr = nr + 1;
     if (dr > 0) { dr = dr - 1; V(r); }
     else if (dr == 0) V(e);
     citește din resursa comună;
     P(e);
     nr = nr - 1;
     if (nr == 0 and dw > 0) { dw = dw - 1; V(w); }
     else if (nr > 0 or dw == 0) V(e);
   }
}
```

Problema cititorilor și scriitorilor
Sincronizare condiționată (6)

```
process Scriitor[j=1 to n]{
  while (true){
    P(e);
    if (nr > 0 or nw > 0) {
      dw = dw + 1; V(e); P(w)
    }
    nw = nw + 1;
    V(e);
    scrie în resursa comună;
    P(e);
    nw = nw - 1;
    if (dr > 0 and dw == 0) { dr = dr - 1; V(r); }
    else if (dw > 0) { dw = dw-1; V(w); }
    else if (dr == 0 and dw == 0) V(e);
  }
}
```

PRIO SCRIITORI

Problema barbierului

Problema bărbierului

- Problema:
  - O frizerie cu un bărbier, un scaun de bărbier, n scaune de așteptare.
  - Când nu sunt clienți, bărbierul doarme.
  - Când sosește un client fie trezește bărbierul, fie așteaptă dacă acesta e ocupat.
  - Dacă toate scaunele sunt ocupate, clientul pleacă.

Pas 1

clienti

BarbierGata

Pas 2

Pas 3

clienti

BarbierGata

Pas 4

clienti

BarbierGata

Problema bărbierului (1)

```
int NumărScauneLibere = n;
sem Clienți = 0;
sem BărbierGata = 0;
sem Scaune = 1;

process Bărbier{
   while (true){
      P(Clienți);                /* se caută un client;
                                    dacă există, este chemat */
      P(Scaune);                 /* are client, va modifica
                                    NumărScauneLibere */
      NumărScauneLibere++;       /* se eliberează un scaun */
      V(BărbierGata);            /* bărbierul e gata să tundă */
      V(Scaune);                 /* a terminat de modificat
                                    NumărScauneLibere */
      /* Bărbierul tunde... */
   }
}
```

sincronizare cu Client

atomicitate

Problema bărbierului (2)

sincronizare cu Barbier

atomicitate

```
process Client[i=1 to m]{
   while (true) {
     P(Scaune);                          /* clientul încearcă să obțină
                                            un scaun liber */
     if (NumărScauneLibere > 0){ /* există un scaun disponibil? */
       NumărScauneLibere--;              /* clientul se așază */
      V(Clienți);                       /* se anunță bărbierul că s-a
                                            ocupat un scaun */
      V(Scaune);                        /* se renunță la accesul asupra
                                            NumărScauneLibere */
      P(BărbierGata);                   /* clientul așteaptă pentru a
                                            fi tuns */
       /* Clientul e tuns... */
     } else {                         /* nu sunt scaune libere */
       V(Scaune);                          /* eliberează mutexul */
       /* Clientul pleacă netuns... */
     }
   }
}
```

Problema bărbierului (var. mutex)

emptyChairs = N
Clients = 0
BarberReady = 0
Chairs = 1

Bărbier

```
while(true) {
    Clients.lock();
    Chairs.lock();
    emptyChairs++;
    BarberReady.unlock();
    Chairs.unlock();
    cutHair();
}
```

Client

```
Chairs.lock();
if(emptyChairs>0) {
    emptyChairs--;
    Clients.unlock();
    Chairs.unlock();
    BarberReady.lock();
    getHairCut();
} else {
    Chairs.unlock();
}
```

Sumar

- Dezvoltarea algoritmilor folosind variabile partajate (MIMD)
- Semafoare
- Secțiuni critice
- Probleme:
  - Producători și consumatori
  - Problema filozofilor
  - Problema cititorilor și scriitorilor
  - Problema bărbierului

Arhitecturi Paralele
Abordări probleme paralele

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Paralelism: modele Pipeline

Pipeline

- CPU Instruction pipeline
- Graphics pipeline
- Various algorithms

Un pas poate fi executat de:

- thread
- process
- element hardware

Without Pipeline

Without Pipeline

Without Pipeline

Without Pipeline

Without Pipeline

Without Pipeline

$$total_execution_time = task_execution_time \ * number_of_tasks$$

Pipeline

Task 6
Task 5
Task 4
Task 3
Task 2
Task 1
Step1
Step2
Step3
Step4

Pipeline

Task 6
Task 5
Task 4
Task 3
Task 2
Step1
Step2
Step3
Step4
Task 1

Pipeline

Task 6
Task 5
Task 4
Task 3
Step1
Step2
Step3
Step4
Task 2
Task 1

Pipeline

Task 6
Task 5
Task 4
Step1
Step2
Step3
Step4
Task 3
Task 2
Task 1

Pipeline

Task 6
Task 5
Step1
Step2
Step3
Step4
Task 4
Task 3
Task 2
Task 1

Pipeline

Task 6
Step1
Task 5
Step2
Task 4
Step3
Task 3
Step4
Task 2
Task 1

Pipeline

Step1
Step2
Step3
Step4
Task 6
Task 5
Task 4
Task 3
Task 2
Task 1

Pipeline

Step1
Step2
Step3
Step4
Task 6
Task 5
Task 4
Task 3
Task 2
Task 1

Pipeline

Step1
Step2
Step3
Step4
Task 6
Task 5
Task 4
Task 3
Task 2
Task 1

Pipeline

Ideal: $$step_execution_time = \frac{task_execution_time}{number_of_steps}$$

După $number_of_steps$ tascuri:

$$total_execution_time = number_of_tasks * step_execution_time$$

Un task se termină la fiecare “step tick”

Sortare...

Sorting

9	4	2	7	6	5	6	1

1	2	4	5	6	6	7	9

Sorting with pipeline

9	4	2	7	6	5	6	1

Sorting with pipeline

9	4	2	7	6	5	6

Sorting with pipeline

9	4	2	7	6	5

Sorting with pipeline

9	4	2	7	6

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Sorting with pipeline

Calcule polinomiale...

Polynomial

$$P(x) = \sum_{i=0}^{n} a_i x^i = a_0 x^0 + a_1 x^1 + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n, n \geq 0$$

Pipeline

Polynomial calculation + Pipeline

$$P(x) = 1 + 8x + (-4)x^3 + x^4$$

	0	1	2	3	4
a	1	8	0	-4	1

Polynomial calculation + Pipeline

$$P(x) = 1 + 8x + (-4)x^3 + x^4$$

Polynomial calculation + Pipeline

x_1 x_2 x_3 x_4 x_5

Polynomial calculation + Pipeline

x_1 x_2 x_3 x_4

Polynomial calculation + Pipeline

x_1 x_2 x_3

Polynomial calculation + Pipeline

Polynomial calculation + Pipeline

Polynomial calculation + Pipeline

Polynomial calculation + Pipeline

$$\mathbf{1}+\mathbf{8}x_5+(-\mathbf{4})x_5^3+x_5^4$$

Polynomial calculation + Pipeline

$$\mathbf{1}+\mathbf{8}x_4+(-\mathbf{4})x_4^3+x_4^4$$

$$\mathbf{1}+\mathbf{8}x_5+(-\mathbf{4})x_5^3+x_5^4$$

Polynomial calculation + Pipeline

$$\mathbf{1} + \mathbf{8}x_3 + (-\mathbf{4})x_3^3 + x_3^4$$

$$\mathbf{1} + \mathbf{8}x_4 + (-\mathbf{4})x_4^3 + x_4^4$$

$$\mathbf{1} + \mathbf{8}x_5 + (-\mathbf{4})x_5^3 + x_5^4$$

Polynomial calculation + Pipeline

$$\mathbf{1 + 8}x_3 + (-\mathbf{4})x_3^3 + x_3^4$$

$$\mathbf{1 + 8}x_4 + (-\mathbf{4})x_4^3 + x_4^4$$

$$\mathbf{1 + 8}x_2 + (-\mathbf{4})x_2^3 + x_2^4 \qquad \mathbf{1 + 8}x_5 + (-\mathbf{4})x_5^3 + x_5^4$$

Polynomial calculation + Pipeline

$$\mathbf{1} + \mathbf{8}x_3 + (-\mathbf{4})x_3^3 + x_3^4$$

$$\mathbf{1} + \mathbf{8}x_1 + (-\mathbf{4})x_1^3 + x_1^4 \qquad \mathbf{1} + \mathbf{8}x_4 + (-\mathbf{4})x_4^3 + x_4^4$$

$$\mathbf{1} + \mathbf{8}x_2 + (-\mathbf{4})x_2^3 + x_2^4 \qquad \mathbf{1} + \mathbf{8}x_5 + (-\mathbf{4})x_5^3 + x_5^4$$

Aplicație: problema Damelor

N Queens Problem

N Queens Problem

No more then one queen per line

N Queens Problem

No more then one queen per column

N Queens Problem

No more then one queen per diagonals

N Queens Problem

N Queens Problem - Solution

	0	1	2	3
0	Q			
1				
2				
3				

0	1	2	3
0			

N Queens Problem - Solution

	0	1	2	3
0	Q	Q		
1				
2				
3				

0	1	2	3
0	0		

Line conflict

N Queens Problem - Solution

	0	1	2	3
0	Q			
1		Q		
2				
3				

0	1	2	3
0	1		

Diagonal conflict

N Queens Problem - Solution

	0	1	2	3
0	Q			
1				
2		Q		
3				

0	1	2	3
0	2		

OK.

And so on…

N Queens Problem – Parallel Solution

	0	1	2	3
0				
1				
2				
3				

N Queens Problem – Parallel Solution

0	1	2	3	
0	0			x
0	1			x
0	2			
0	3			

0	1	2	3	
1	0			x
1	1			x
1	2			x
1	3			

And so on…

0	1	2	3	
2	0			
2	1			x
2	2			x
2	3			x

0	1	2	3	
3	0			
3	1			
3	2			x
3	3			x

Cautare binara

Binary Search

Searching for $\boxed{3}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

Binary Search

Searching for 3

Interest area 0 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 < 7

Binary Search

Searching for 3

Interest area 0 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 < 5

Binary Search

Searching for 3

Interest area 0 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 > 2

Binary Search

Searching for | 3 |

Interest area | 3 | 3 |

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 = 3

end

O($log_2(n)$) time

Parallel Search

Searching for $\boxed{3}$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

Parallel Search

Searching for 3

Interest area 0 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

The element is in my area

Parallel Search

Searching for 3

Interest area 1 4

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

I found the element

I found the element

Parallel Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

Operations can**NOT** be executed in parallel

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

Parallel Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

Parallel Search – solutia 2 (cu mai putine threaduri)

Parallel Search – solution 2 (fewer threads)

Searching for 3

Interest area 0 15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 < 5 **3 < 7** **3 < 8**

Operations can be executed in parallel

Parallel Search

Searching for 3

Interest area 0 3

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 > 1

3 > 2

3 = 3 end

O($log_p(n)$) time

Parallel Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 < 5 **3 < 7** **3 < 8**

Operations can**NOT** be executed in parallel

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 > 1 **3 > 2** **3 = 3** **end**

Parallel Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 < 5 **3 < 7** **3 < 8**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	2	3	5	5	6	6	7	7	7	8	8	9	9	9

3 > 1 **3 = 3**

3 > 2

Arhitecturi Paralele
Abordări probleme paralele log(N)

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Sume prefix

Aplicații folosind paralelismul de date
Calculul sumelor prefix

Problemă:

Se dă tabloul **a[1:n]**, se cere **s[1:n]**, unde:

$$s[i] = \sum_{k=1}^{i} a[k]$$

- Algoritm secvențial:

```
s[1] = a[1];
for [i = 2 to n]
    s[i] = a[i] + s[i-1];
```

- Algoritm paralel:

  - derivat din algoritmul ***sumei elementelor unui vector***

Suma elementelor unui vector

$$\sup\left(\frac{n}{2}\right) \text{ procesoare, } \sup(\log_2 n) \text{ pași}$$

Suma elementelor unui vector

Suma elementelor unui vector

```
int a[1:n];
process suma[k=1 to n] { // suma proceselor p din figur
      for [j = 1 to sup(log2 n)] {
            if (k mod 2^j == 0)
                  a[k] = a[k-2^(j-1)] + a[k];
            barrier;
      }
}
```

Sume prefix (varianta 1)

Sume prefix – varianta 1

```
int a[1:n];
process suma[k=1 to n] {
      for [j = 1 to sup(log2 n)] {
            if (k - 2^(j-1) >= 1)
                  a[k] = a[k-2^(j-1)] + a[k];
            barrier;
      }
}
```

Eroare de sincronizare...

Sume prefix – varianta 1 – probleme

- Presupunem că procesorul numărul 3 este mai lent decât restul.
  - Suprascriere a locației de memorie

$$s_2^3 = a_3 + s_1^2$$

Sume prefix – varianta 2

```
int a[1:n], temp[1:n];
process suma[k=1 to n] {
      for [j = 1 to sup(log2 n)] {
            temp[k] = a[k];
            barrier;
            if (k - 2^(j-1) >= 1)
                        a[k] = temp[k-2^(j-1)] + a[k];
            barrier;
      }
}
```

Sume prefix – varianta 3

```
int a[1:n], temp[1:n];
process suma[k:1 to n] {
    int d = 1;
    while (d < n) {
        temp[k] = a[k];
        barrier;
        if (k - d >= 1) a[k] = temp[k-d] + a[k];
        barrier;
        d = 2 * d;
    }
}
```

Parallel Scan

Parallel Scan

scan (o) < x1, ..., xn >

==

< x1, x1 o x2, ..., x1 o ... xn >

Exemplu: Filtru

Considerați la **intrare** un vector, trebuie găsit la ieșire un alt **vector** care să conțină doar elementele pentru care **(f elt)** este **true**

Fie f x = x > 10

filter f < 17, 4, 6, 8, 11, 5, 13, 19, 0, 24 >
= < 17, 11, 13, 19, 24 >

Paralelizabil?

- Găsirea elementelor este o operație ușoară
- Dar punerea lor în secvența corectă pare mai dificilă

Prefixe paralele - to the rescue

1. Folosim mapare paralelă pentru a calcula un bit-vector pentru elementele ce se potrivesc operației

   input: < 17, 4, 6, 8, 11, 5, 13, 19, 0, 24 >
   bits: < 1, 0, 0, 0, 1, 0, 1, 1, 0, 1 >

2. Aplicăm sume prefix pe elementele bit-vectorului

   bitsum < 1, 1, 1, 1, 2, 2, 3, 4, 4, 5 >

3. Aplicăm parallel map pentru a produce ieșirea

   output < 17, 11, 13, 19, 24 >

Generic: Scan

Se aplică aceeași operație între elementele unui vector.
Se colectează toate rezultatele parțiale

Poate fi executat în log(n) pași folosind un arbore

Operația poate lua orice formă (+, *, min, max, and, etc.)

Scan - sum

Scan - sum

Pot fi executate în paralel

Scan - sum

Pot fi toate executate în paralel

Scan - sum

Pot fi toate executate în paralel

Scan – Cum funcționează?

Scan – Cum funcționează

Scan – Cum funcționează?

Scan – Cum funcționează?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

Scan – How it works?

9	4	2	7	3	5	6	1
9	13	6	9	10	8	11	7
9	13	15	22	16	17	21	15
9	13	15	22	25	30	36	37

Difuzarea unei valori

Difuzarea unei valori – justificare O(log P)

- D – celula din memoria comună ce trebuie difuzată
- Folosim un tablou A[1:N]

procedure BROADCAST (D, N, A)

Pas 1: Procesorul P1

1.1. citește valoarea din D

1.2. o memorează în propria memorie

1.3. o scrie în A[1]

Pas 2:

fa i := 0 to (log N -1) ->

fa j := 2^i+1 to $2^{(i+1)}$ do in parallel

Procesor Pj

2.1. citește valoarea din A[j-2^i]

2.2. o memorează în propria memorie

2.3. o scrie în A[j]

af

af

Difuzarea unei valori

Operații cu vectori – broadcast

Pas 1: (Procesorul P_1)

```
int t;/* t locala P1 */
t = D;
A[1] = t;
```

Pas 2:

```
for [i = 0 to (log N -1)] {
     process Pj[j = 2^i+1 to 2^(i+1)]{ /*(Procesor Pj)*/
          int t; /* t locală Pj */
          t = A[j-2^i];
          A[j] = t;
     }
}
```

Value Broadcast

Start

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
7																…

Value Broadcast

End

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	…

Inefficient Value Broadcast

O(n) time

Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

i = 1

Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

i = 2

These operations can be executed in parallel

Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

i = 4

These operations can be executed in parallel

Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

i = 8

These operations can be executed in parallel

Efficient Value Broadcast

O(log2(n)) time with p = n/2 processors

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	…

Efficient Value Broadcast

i = 2

These operations can**NOT** be executed in parallel

i = 4

Efficient Value Broadcast

i = 2

i = 4

Operații cu liste

Operații cu liste

- Listă *n* elemente: `data[1:n]`
- Legăturile între elemente: `leg[1:n]`
- Capul listei: `cap`
- Elementul `i` face parte din listă ⇔ fie
  - `cap = i`
  - există un element `j`, între `1` și `n`, a.î. `leg[j]=i`

- Problemă: vrem ca fiecare procesor să afle capătul listei

Operații cu liste

```
int leg[1:n], end[1:n];
process Află[i=1 to n] {
      int nou, d=1;
      end[i] = leg[i];
      barrier;
      while (d<n) {
            nou = 0;
            if (end[i]<>0 and end[end[i]]<>0)
                  nou=end[end[i]]
            barrier;
            if (nou<>0) end[i]=nou;
            barrier;
            d = 2*d;
      }
}
```

Operații cu liste

Procesor	End[i]	End[i]	End[i]	End[i]
1	3	2	5	5
2	4	5	5	5
3	2	4	5	5
4	5	5	5	5
5	0	0	0	0
	init	*d = 1* *d < 5*	*d = 2* *d < 5*	*d = 4* *d < 5*

```
int leg, end[1:n];
process Află[i=1 to n] {
    int nou, d=1;
    end[i] = leg[i];
    barrier;
    while (d < n) {
        nou = 0;
        if (end[i]<>0 and end[end[i]]<>0)
            nou = end[end[i]]
        barrier;
        if (nou <> 0)
            end[i] = nou;
        barrier;
        d = 2 * d;
    od
oc
```

Algoritmi Paraleli și Distribuiți
Algoritmi genetici paraleli - optional

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Introducere

Algoritmii genetici

- reprezintă o soluţie a problemelor de optimizare,
- bazată pe mecanisme împrumutate din genetică.

Un AG

- menţine o populaţie de indivizi,
- fiecare individ reprezinta o soluţie potenţială a unei probleme

AG realizeaza, în fiecare etapă, următoare operaţii:

- evaluarea populaţiei curente
- selecţia celor mai buni indivizi
- transformarea populaţiei folosind operatori genetici de încrucişare şi mutaţie.

Domeniile de aplicare

- optimizarea parametrilor
- control optim
- transport
- optimizare combinatorială
- desenare de grafuri
- învăţare inductivă a regulilor de decizie
- stabilirea cablajelor
- planificarea
- jocuri, modelarea cognitivă
- optimizarea interogării bazelor de date.

Metode clasice de optimizare

Problema:

- cautarea intr-un spatiu de siruri binare de 30 de biti, cu functia obiectiv

$$f(v) = | 11*one(v)-150|$$

- unde **one(v)** este numarul de unitati din vectorul binar v.
- Functia f(v) are
  - un maxim global pentru v_g = (1 1 1 1 …1), pentru care $f(v_g)$ = 180 si
  - un maxim local pentru v_l = (0 0 0 …0), pentru care $f(v_l)$ = 150.

Algoritmul hill climbing

```
procedure hillclimber
begin
    t := 0
    do t < MAX ->
        local := false
        selectează aleator sirul curent Vc
        evalueaza Vc
        do not local ->
            gaseste Vn dintre vecinii cu cea mai mare valoare a
                functiei obiectiv F
            if F(Vc) < F(Vn) ->        Vc := Vn
            [] F(Vc) >= F(Vn) -> local := true
            fi
        od
        t := t+1
    od
end
```

Obs: daca sirul de pornire are <= 13 unitati atunci se gaseste intotdeauna maximul local

Simulated annealing

```
procedure simulated-annealing
begin
    t := 0
    initializeaza temperatura T
    selecteaza aleator sirul curent Vc
    evalueaza Vc
    do not cond-stop ->
        do not cond-terminare ->
                selecteaza un nou sir Vn vecin cu Vc
                if F(Vc) < F(Vn)   -> Vc = Vn
                [] F(Vc) >= F(Vn) ->
                        if random[0,1) < exp((F(Vn)-F(Vc))/T) ->
                                Vc = Vn
                        fi
                fi
        od
        T = g(T,t)
        t = t+1
    od
end
```

cand T atinge o anumita limită inferioară.

verifică atingerea unui “echilibru termic” de ex cand distribuţia probabilităţilor noilor şiruri selectate se apropie de Bolzmann.

- Pentru problema sirurilor binare:
- daca v12 are 12 unitati:
  - $f(v12) = |11*12 – 150| = 18$
  - $f(v13) = |11*13 – 150| = 7$
- Algoritmul accepta o solutie cu 13 unitati cu probabilitatea
  - $p = \exp((f(Vn)-f(Vc))/T) = \exp((7-18)/T)$
- pentru T=20:
  - $p = 0.576 > 0.5$

Algoritmi genetici

```
procedure algoritm_genetic
begin
    t := 0;
    creaza o populatie initiala de cromozomi P(t);
    evalueaza fiecare cromozom al populatiei initiale.
    do t < not conditie de terminare ->
        t := t+1;
        selecteaza parintii pentru reproducere;
        creaza noi cromozomi prin imperechere si mutatie;
        inlocuieste anumiti membri ai populatiei cu cei noi;
        evalueaza cromozomii din noua populatie
    od
end
```

Problema:

Dat fiind graful G = (V, E) să se găsească o partiţionare a sa în două subgrafuri având acelaşi număr de noduri, prin eliminarea unui număr minim de muchii.

Problema este NP completă.

Rezolvare cu alg genetici

- soluţie reprezentată ca un vector cu elemente binare, de dimensiune egală cu numărul nodurilor din graf
- o gena corespunde unui nod
  - 1 = nod in prima partitie
  - 0 = nod in a doua partitie

Evaluare solutie

- eval(x) = cutsize(x)

Generare populatie initiala

- aleator cu corectii
  - orice cromozom sa aiba acelasi numar de 0 si 1 (±1)

Selecţia parintilor pentru reproducere

Metoda ruletei

calculează eval(xi) pentru fiecare cromozom i = 1..pop-size

asociază cu fiecare cromozom o valoare de "potrivire"

f(xi) = max-fit - eval(xi)

astfel că f(xi) > 0 pentru orice i = 1..pop-size

calculează valorile de potrivire cumulative

cf(xi) = SUMA f(xj), j = 1..i

Selecţia

generează un număr aleator ***rand*** în intervalul [o, cf-max]

if ***rand*** <= cf(x_1) -> alege primul cromozom x_1 **fi**

if cf(x_{i-1}) < *rand* <= cf (x_i) -> alege x_i **fi**

Exemplu selectie cromozomi										
Cromozom	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Potrivire	**8**	**2**	**17**	**7**	**2**	**12**	**11**	**7**	**3**	**7**
Potrivire cumulata	**8**	**10**	**27**	**34**	**36**	**48**	**59**	**66**	**69**	**76**

Numar aleator	**23**	**49**	**76**	**13**	**1**	**27**	**57**
Cromozom ales	**3**	**7**	**10**	**3**	**1**	**3**	**7**

- Operatorul *mutation*

Aplicarea operatorului *mutation* pentru 0.08								
Cromozom vechi	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**
Probabilitate	**0.80**	**0.11**	**0.27**	**0.45**	**0.12**	**0.05**	**0.15**	**0.03**
Cromozom nou	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**

Operatorul de încrucişare (*crossover*)

Aplicarea operatorului *crossover* pentru pozitia 6								
Primul parinte	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
Al doilea parinte	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Primul fiu	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**
Al doilea fiu	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**

Aplicarea operatorului *crossover* pentru pozitiile 3 si 6								
Primul parinte	1	1	1	1	1	1	1	1
Al doilea parinte	0	0	0	0	0	0	0	0
Primul fiu	0	0	0	1	1	1	0	0
Al doilea fiu	1	1	1	0	0	0	1	1

Aplicarea operatorului *crossover uniform*								
Primul parinte	1	1	1	1	1	1	1	1
Al doilea parinte	0	0	0	0	0	0	0	0
Masca	1	0	0	1	0	0	0	1
Primul fiu	1	0	0	1	0	0	0	1
Al doilea fiu	0	1	1	0	1	1	1	0

Justificarea funcţionării algoritmilor genetici

Schema

- este *un şir construit cu simbolurile 0, 1 şi * (don't care)*
- sunt posibile 3^m scheme de lungime **m**
- o schemă reprezintă toate şirurile care coincid cu ea în poziţiile diferite de *
- o schemă cu **r** simboluri * are drept corespondente 2^r şiruri
- un şir de lungime **m** poate fi pus în corespondenţă cu 2^m scheme
- intr-o populaţie de dimensiune **n** pot fi reprezentate între 2^m şi $n*2^m$ scheme

Proprietăţile schemelor

- Ordinul o(S) = numarul de pozitii fixe

  S1 = (010***11***1) are o(S1) = 6
- Lungimea caracteristică δ(S) = distanţa între prima şi ultima poziţie fixă

  δ(S1) = 11

Evoluţia schemelor - selectia

pop_size dimensiunea populaţiei eval(v_i) = potrivirea cromozomului i

m lungimea unui cromozom

Şirul v_i are probabilitatea de selectie conform ruletei $p_i = \frac{eval(v_i)}{F(t)}$

Potrivirea totală a populatiei $F(t) = \sum_{i=1}^{pop-size} eval(v_i)$

p = $\xi(S,t)$ nr. cromozomi v_{ij} care corespund la momentul *t* cu schema *S*.

Potrivirea schemei *S* la momentul *t* $eval(S,t) = \sum_{j=1}^{p} \frac{eval(v_{ij})}{p}$

Probabilitatea de selecţie a unui şir care corespunde schemei *S* = $\frac{eval(S,t)}{F(t)}$

Numar de selectii este *pop_size*

Rezulta

$$\xi(S,t+1) = \xi(S,t) * pop_size * \frac{eval(S,t)}{F(t)}$$

Fie $\overline{F(t)} = \dfrac{F(t)}{pop_size}$ potrivirea medie a populatiei

Atunci

$$\xi(S,t+1) = \xi(S,t) * \frac{eval(S,t)}{\overline{F(t)}}$$

O schemă "peste medie" are $\dfrac{eval(S,t)}{\overline{F(t)}} > 1$

Rescriem relatia $eval(S,t) = \overline{F(t)}(1+\varepsilon)$

deci

$$\xi(S,t+1) = \xi(S,t) * (1+\varepsilon)$$

si S primeşte un număr mai mare de indivizi în noua populaţie.

Raportat la populatia initiala

$$\xi(S,t) = \xi(S,0) * (1+\varepsilon)^t$$

numărul de indivizi corespunzatori schemei S creşte in progresie geometrica.

Încrucişarea

Fie şirul (1110111101001)

şi schemele:

S0=(****111******), δ(S0)=2

S1=(111********01), δ(S1)=12

Dacă poziţia de tăiere = 10, schema S0 se regăseşte într-unul din fii, în timp ce schema S1 are şanse foarte reduse de reproducere.

➔lungimea caracteristică este importanta în reproductibilitatea schemei

m-1 posibilităţi selectie loc de încrucişare

probabilitatea de distrugere a schemei *S* este

probabilitatea de supravieţuire

cu prob incrucisarii p_c

$$p_s(S) = 1 - p_c * \frac{\delta(S)}{m-1}$$

si deoarece se pot combina indivizi aparţinând unor scheme comune

$$p_s(S) \geq 1 - p_c * \frac{\delta(S)}{m-1}$$

efect combinat selectie & incrucisare

$$\xi(S,t+1) \geq \xi(S,t) * \frac{eval(S,t)}{\overline{F(t)}} * (1 - p_c * \frac{\delta(S)}{m-1})$$

Mutaţia

probabilitatea mutaţiei unui singur bit este p_m

probabilitatea nemodificării sale este (1- p_m).

probabilitate ca o schemă *S* să supravieţuiască unei mutaţii

$$p_s(S) = (1 - p_m)^{o(S)} \quad \rightarrow \quad p_s(S) \approx 1 - p_m * o(S)$$

efectul combinat al selecţiei, încrucişării şi mutaţiei este

$$\xi(S,t+1) \geq \xi(S,t) * \frac{eval(S,t)}{\overline{F(t)}} * (1 - p_c * \frac{\delta(S)}{m-1} - p_m * o(S))$$

Teorema schemei. Schemele **scurte**, **de ordin scăzut** şi **peste medie** cresc ***exponenţial*** de-a lungul generaţiilor unui algoritm genetic.

Ipoteza blocurilor constructive. Un algoritm genetic atinge performaţe aproape optime prin juxtapunerea unor scheme scurte, de ordin scăzut, de mare performaţă, numite **blocuri constructive**.

Considerarea constrângerilor în algoritmii genetici

Metode:

- **penalizări pentru solutii ce nu respecta constrângeri**
- **eliminarea soluţiilor necorespunzătoare**
- **aplicarea unor algoritmi de corecţie.**

Ex. problema rucsacului.

- **date fiind o mulţime de ponderi W, profiturile asociate P şi capacitatea C a rucsacului, să se găsească un vector binar $X = \langle x_1, x_2, ..., x_n \rangle$ a.i.**

$$\sum_{i=1,n} x_i * w_i <= C \text{ şi}$$

$$P(X) = \sum_{i=1,n} x_i * p_i \text{ este maxim.}$$

Functia de evaluare $eval(X) = \sum_{i=1,n} x_i * p_i - Pen(X)$

- **unde** $Pen(X) = 0$ **pentru soluţiile fezabile şi**

  $Pen(X) \neq 0$ **pentru soluţiile nefezabile (suma > C).**

Variante:

- **logaritmică,** $Pen_1(X) = \log_2 (1 + \rho (\sum_{i=1,n} x_i * w_i - C))$
- **liniară,** $Pen_2(X) = \rho (\sum_{i=1,n} x_i * w_i - C)$
- **pătratică,** $Pen_3(X) = (\rho (\sum_{i=1,n} x_i * w_i - C))^2$

unde $\rho = MAX_{i=1,n} \{ p_i / w_i \}$

Algoritmul bazat pe corecţia soluţiei

eval(X) = $\sum_{i = 1,n} x'_i * p_i$ unde X' este versiunea corectată a lui X

procedure corectie (X)
begin
    depasire = false
    X' = X
    if $\sum_{i = 1,n} x'_i * w_i > C$ **->** depasire = true **fi**
    do depasire **->**
        i = selecteaza un element din sac
        scoate elementul $x'_i = 0$
        if $\sum_{i = 1,n} x'_i * w_i <= C$ **->** depasire = false **fi**
    od
end

Algoritmul bazat pe decodificatori

Un cromozom este interpretat ca o strategie de a incorpora elemente într-o soluţie.

Fiecare cromozom este un vector X de **n** întregi.

Componenta **i** a vectorului X

- este un întreg din domeniul **1..n-i+1**
- indică poziţia unui element al unei liste **L**
- elementele selectate sunt eliminate din lista **L**
  - pas 1 – L are n elemente → valoarea X[1] in domeniul 1..n
  - pas 2 - L are n-1 elemente → valoarea X[2] in domeniul 1..n-1
  - s.a.m.d.

Efect

- la mutaţie, gena **i** poate lua orice valoare între **1** şi **n-i+1**,
- încrucişarea unor părinţi corecţi produce descendenţi corecţi.

Algoritmul bazat pe decodificatori

Exemplu

- fie L = (1,2,3,4,5,6).
- Vectorul X = <4,3,4,1,1,1> este decodificat astfel:

- X[1] = 4 → se alege L[4] = 4 – L devine L = (1,2,3,5,6)
- X[2] = 3 → se alege L[3] = 3 – L devine L = (1,2,5,6)
- X[3] = 4 → se alege L[4] = 6 – L devine L = (1,2,5)
- s.a.m.d.

Rezultat:

- Vectorul X = <4,3,4,1,1,1> este decodificat
- ca secvenţa 4, 3, 6, 1, 2, 5.
- 6 este al 4-lea element după eliminarea lui 4 şi a lui 3

```
procedure decode (X)
    construieste o lista L cu n elemente <wj, pj>
    i := 1
    suma-ponderi = 0
    suma-profit = 0
    do i <= n ->
        j = xi
        elimina elementul cu numarul de ordine j din L
        if suma-ponderi + wj <= C ->
            suma-ponderi = suma-ponderi + wj
            suma-profit = suma-profit + pj
        fi
        i = i+1
    od
end
```

Variante pentru construieste

- **aleator** – ordinea elementelor din fisierul de intrare care este aleatoare
- **greedy** - elementele în ordinea descrescătoare a rapoartelor p_i / w_i;

Algoritmi genetici paraleli

Abordari

- paralelizarea operatorilor genetici
- distribuirea populatiei.

Paralelizarea operatorilor genetici

- partitionare functionala: paralelizarea buclei care produce noua generatie
- se mentine o singura populatie
- operatorii genetici sunt aplicati in paralel
  - functia de potrivire si mutatia aplicate in paralel indivizilor
  - crossover la doi indivizi
- mai adaptat pentru memorie partajata

Abordarea populatie distribuita

- Corespunde descompunerii domeniului
- prelucrari facute in paralel pe diferite sub-populatii
- exploreaza mai bine spatiul solutiilor
- PGAs mentin sub-populatii separate care evolueaza independent

PGAs de granularitate fina

- Asigneaza un individ fiecarui task
- Fiecare task foloseste indivizi din vecinatatea sa
- Ajuta migrarea indivizilor

Intrebari:

- care este tolopogia de inter-conectare ?
- care este dimensiunea vecinatatii ?
- care este schema de inlocuire pentru includerea migratorilor in sub-populatia tinta ?

PGAs de granularitate mare

- Fiecare sub-populatie contine un numar mare de indivizi
- AG se executa independent pe fiecare sub-populatie si produce o solutie a problemei
- Rezultatul final este obtinut prin selectia celei mai bune solutii
- Indivizii migreaza periodic de la o sub-populatie la alta

Synchronous Island PGAs

- migrarea are loc dupa un anumit numar de generatii

Asynchronous Island PGAs

- migrarea intre doua sub-populatii nu este corelata cu restul migrarilor
- mai aproape de migrarea din natura
- mai potrivita cu algoritmi distribuiti

Intrebari

- care este topologia proceselor?
- este reteaua omogena sau nu?
- care este rata de migrare?
- cati migratori sunt inter-schimbati?
- cum sunt selectati migratorii?
- cum sunt combinati cu populatia tinta?

Asynchronous Island PGAs

Modelul "replicated workers"

Programe evolutive

Se bazează pe algoritmii genetici, dar folosesc:

- un set mai bogat de structuri de date
- un set extins de operatori genetici

Un exemplu - Problema transportului

Enuntul problemei

- se cere un plan de transport cu cost minim
  - un singur gen de marfa,
  - de la n surse la k destinatari
  - o destinatie poate primi marfa de la mai multe surse
  - costul unei rute este proportional cu cantitatea transportata (problema este lineara)

Notatii

sour[i]	cantitatea furnizata de sursa i
dest[j]	cantitatea ceruta de destinatia j
cost[i,j]	costul unitar de transport intre i si j
x[i,j]	cantitatea transportata intre i si j

Un exemplu - Problema transportului (2)

Se cere sa se minimizeze

$$\Sigma\ \Sigma\ cost[i,j]*x[i,j] \text{ pentru } i=1,n;\ j=1,k$$

n surse

k destinatari

cu constrangerile

$$\Sigma_{j=1,k}\ x[i,j] <= sour[i]$$

$$\Sigma_{i=1,n}\ x[i,j] >= dest[j]$$

$$x[i][j] >= 0 \text{ pentru } i=1,n;\ j=1,k$$

Exemplu

- n = 3; sour = [15,25,5]
- k = 4; dest = [5,15,15,10]

- Matricea de cost ➔

10	0	20	11
12	7	9	20
0	14	16	18

- Matricea X corespunzatoare planului optim → cost total = 315

-

sour \ dest	5	15	15	10
15	0	5	0	10
25	0	10	15	0
5	5	0	0	0

AG clasic

Fiecare solutie reprezentata ca un **sir agregat** de biti

$$v_1, v_2, \ldots, v_p$$

unde p = n*k

si v_i sir binar de forma $v_i = w_i^0, \ldots, w_i^s$

v_i reprezinta un element x[j,m] asociat cu linia **j** coloana **m** ale matricei X, adica

$$j = \inf [(i-1)/k+1]$$

$$m = (i-1) \bmod k + 1$$

s calculat pentru a putea reprezenta intregul maxim din problema = $\mathbf{2^{s+1}-1}$.

Obs.

- nu exista o definitie simpla si naturala a operatorilor genetici
- producerea unor solutii care sa satisfaca constrangerile necesita corectii complicate

Reprezentare vectoriala

O solutie a problemei

- reprezentata ca un sir de **p=n*k** numere intregi
- cu valori in domeniul 1..p
- interpretata ca o strategie de a incorpora elemente în soluţie

Prima operatie pentru sirul {7,....} este

sour \ dest	5	15	15 (-15=0)	10
15	0	0	0	0
25 (-15=10)	0	0	**15**	0
5	0	0	0	0

Procedura de decodificare a unei solutii

```
procedure initialization;
begin
   seteaza ca nevizitate elem. din lista cu valori de la 1 la p;
   do not toate nodurile vizitate ->
         select urmatorul numar q si marcheaza vizitat;
         {calc nr linie i si coloana j corespunzatoare}
         i = (q-1)/k+1; j=(q-1) mod k +1;
         val = min(sour[i], dest[j]);
         v[i][j] = val;
         sour[i] = sour[i] – val;
         dest[j] = dest[j] – val
   od
end
```

- Solutia optima pentru problema de transport
  - {7,9,4,2,6, *,*,*,*,*,*,*}.

sour \ dest	5	15	15	10
15	0	5	0	10
25	0	10	15	0
5	5	0	0	0

Observatii

- orice permutare de numere cu valori intre 1 si p produce o solutie ce satisface constrangerile;
- operatorii genetici sunt simplu de realizat.
  - **Mutatia** se face prin interschimbarea a doua numere din sir.
  - **Incrucisare**

Ex. incrucisare:

Pentru perechea {1,2,3,<u>4,5,6,7</u>,8,9,10,11,12}

{7,3,1,11,4,12,5,2,10,9,6,8}

  - se alege un tipar din primul parinte
  - restul elementelor se pun in ordinea din al doilea parinte

rezulta {3,1,11, <u>4,5,6,7</u>,12,2,10,9,8}

Reprezentarea matriceala

O solutie reprezentata in forma originala a problemei:

X = (x[i][j]) pentru i=1,k; j=1,n

0	0	5	0	3
0	4	0	0	0
0	0	5	7	0
3	1	0	0	2

Mutatia

- selecteaza o submatrice
- inlocuieste cu alta submatrice avand aceleasi sume pe linii si pe coloane

4	0	0
1	0	2

2	0	2
3	0	0

include in matricea originara

0	0	5	0	3
0	2	0	0	2
0	0	5	7	0
3	3	0	0	0

Incrucisarea

X1 si X2

1	0	0	7	0
0	4	0	0	0
2	1	4	0	5
0	0	6	0	0

0	0	5	0	3
0	4	0	0	0
0	0	5	7	0
3	1	0	0	2

DIV (media lor)
si REM (restul)

0	0	2	3	1
0	4	0	0	0
1	0	4	3	2
1	0	3	0	1

1	0	1	1	1
0	0	0	0	0
0	1	1	1	1
1	1	0	0	0

se imparte REM in
REM1 si REM2
(suma lor esteREM)

0	0	1	0	1
0	0	0	0	0
0	1	0	1	0
1	0	0	0	0

1	0	0	1	0
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0

V3 = DIV+REM1
si V4 = DIV+REM2

0	0	3	3	2
0	4	0	0	0
1	1	4	4	2
2	0	3	0	1

1	0	2	4	1
0	4	0	0	0
1	0	5	3	3
1	1	3	0	1

Dilema prizonierului

Doi suspecti sunt arestati de politie. Politia nu are suficiente probe pentru condamnare si separa prizonierii, vizitandu-i si facandu-le aceeasi propunere.

- Daca unul depune marturie pentru pedepsirea celuilalt (tradeaza) si celalalt tace, tradatorul este eliberat si cel care tace ia 10 ani de puscarie.
- Daca amandoi tac, ambii sunt condamnati la 2 ani pentru o vina minora.
- Daca fiecare il tradeaza pe celalalt, fiecare primeste o sentinta de 5 ani.

Fiecare prizonier trebuie sa aleaga intre tradare si tacere

Fiecare este asigurat ca celalalt nu va sti despre tradarea lui inaintea terminarii investigatiei.

Cum trebuie sa actioneze prizonierii?

Dilema prizonierului

Jucatorul 1	Jucatorul 2	P1	P2	Comentariu
Tradeaza	Tradeaza	1	1	pedeapsa pentru tradare simultana
Tradeaza	Tace	5	0	rasplata tradarii / pedeapsa tacerii
Tace	Tradeaza	0	5	pedeapsa tacerii / rasplata tradarii
Tace	Tace	3	3	rasplata tacerii

Gasirea strategiei optime

Din teoria jocurilor:

- jucatorii prefera sa tradeze (fiecare presupune ca si celalalt tradeaza si pedeapsa este mica)
- pentru un joc iterativ (doi jucatori repeta de mai multe ori jocul), cooperarea devine posibila, fiecare gandindu-se la un castig mai mare
- se cauta strategia optima de castig bazata pe ultimele trei runde
- se compara
    - o strategie clasica cu una obtinuta prin algoritmi genetici sau
    - doua strategii obtinute genetic

Proiectarea algoritmului genetic

Reprezentarea solutiei:

- **trei mutari anterioare** ➔ 4^3 = 64 cazuri posibile ➔ 6 biti
- pentru fiecare caz se alege o decizie de tradare sau tacere
- o decizie = 1 bit ➔ 64 cazuri posibile = 64 biti
- Total 70 biti / cromozom

Obtinerea strategiei optime:

Se considera o populatie initiala

- fiecare jucator reprezentat de 70 de biti generati aleator.

Se evalueaza strategia fiecarui jucator

- prin calculul scorului mediu obtinut intr-un numar de jocuri stabilit

Se face selectia jucatorilor pentru inmultire.

Se face imperecherea cu incrucisare si mutatie.

Obtine o noua populatie si reia ciclul

Algoritmi Paraleli și Distribuiți
Comunicarea prin mesaje. Complexitatea algoritmilor distribuiți

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

1. Comunicarea prin mesaje

Transmiterea de mesaje

- Îndeplinește cele două funcții:
  - **comunicarea** - datele ajung de la transmiţător la receptor
  - **sincronizarea** - un mesaj nu poate fi recepţionat înainte de a fi transmis
- **Canalele** sunt obiectele puse în comun de procese
- **Programe distribuite**

Comunicare asincrona prin mesaje

Comunicarea asincronă prin mesaje

Modelul

Canalele păstrează ordinea mesajelor

Canalele nu pierd mesajele

canalele sunt cozi FIFO de mesaje, de capacitate nelimitata

operaţia **send** nu este blocantă

operaţia **receive** este blocantă

Declaraţie – forma generală

chan nume_canal (tip1 id1,..., tipn idn)

Grup indexat de canale

chan rezultat [1:n](int)

Operaţii

send nume_canal ($expresie_1$,..., $expresie_n$)

receive nume_canal ($variabila_1$,..., $variabila_n$)

empty (nume_canal)

Exemplu - Filtru

```
chan  input (char),
      output (char [1:Maxl]);

process car_linii{
  char line[1:Maxl]; int i=1;
  while (true) {
    receive input (line[i]);
    while (line[i] <> CR and i < Maxl) {
      /* line[1:i] = ultimele i caractere introduse*/
      i = i + 1;
      receive input (line[i]);
    }
    send output (line);
    i = 1;
  }
}
```

Modelul Replicated Workers

Integrare numerica

- Stanga – varianta secventiala
  - calcul iterativ - la fiecare pas se dubleaza numarul de subintervale
  - se termina cand diferenta intre doua valori succesive este acceptabila
- Dreapta – varianta distribuita
  - un proces primeste un subinterval [a,b] si aria corespunzatoare
  - imparte intervalul in doua sub-intervale [a.m], [m,b] si calculeaza ariile
  - compara suma celor doua arii cu aria primita
  - continua calculul separat pe cele doua subintervale, daca este cazul

Replicated workers

Exemplu - Replicated Workers (2)

```
chan sac (real a, b, fa, fb, aria);
chan rezultat (real a, b, aria);

process Administrator{
  real xl, xr, yl, yr, a, b, aria, total: real;
  /* alte variabile marcare intervale terminate */

  yl = f(xl); yr = f(xr);
  aria = (yl + yr) * (xr - xl) / 2;
  send sac(xl, xr, yl, yr, aria);

  while (nu s-a calculat toata aria) {
    receive rezultat (a, b, aria);
    total = total + aria;
    marchează intervalul [a, b] terminat;
  }
}
```

Exemplu - Replicated Workers (3)

```
process Lucru[i=1 to n]{
  real a, b, m, ya, yb, ym;
  real ariaSt, ariaDr, ariaTot, dif;
  while (true) {
    receive sac (a, b, ya, yb, ariaTot);
    m = (a + b) / 2;
    ym = f(m);
    calculează ariaSt și ariaDr;
    dif = ariaTot - ariaSt - ariaDr;
    if (dif mic) send rezultat (a, b, ariaTot);
    else { send sac (a, m, ya, ym, ariaSt);
           send sac (m, b, ym, yb, ariaDr);
    }
  }
}
```

Comunicare sincrona prin mesaje

Comunicarea sincrona prin mesaje

Modelul

Caracteristici similare cu comunicarea asincrona

Diferenta: si operaţia **send** **este** blocantă – transmitatorul se blocheaza pana cand mesajul este receptionat

Operaţii

synch_send nume_canal (expresie$_1$,..., expresie$_n$)

receive nume_canal (variabila$_1$,..., variabila$_n$)

empty (nume_canal)

Neajunsuri

reduce concurenta

Exemplu – comunicare producator / consumator

```
chan prodcons (real);

process prod {
    real date[n];
    for [i = 1 to n] {
        calculeaza date[i];
        synch_send prodcons (date[i]);    /* intarzie transmitatorul */
    }                                     /* daca receptorul este mai lent */
}
process cons {
    real buf[n];
    for [i = 1 to n] {
        receive prodcons (buf[i]);
        prelucreaza datele primite;
    }
}
```

2. Complexitatea algoritmilor distribuiți

Modelul Foster

Modelul Foster

(Foster 1995)

Timpul total de execuţie

- Definiţie
  - Timpul scurs de la începerea execuţiei primului proces până la terminarea execuţiei ultimului proces.

$$T = f\,(N,\ P,\ U, \ldots)$$

$$= T^{j}{}_{comp} + T^{j}{}_{commun} + T^{j}{}_{idle}$$

$$= \left(\frac{1}{P}\right) * \left(\sum_{i=0}^{P-1} T^{i}{}_{comp} + \sum_{i=0}^{P-1} T^{i}{}_{commun} + \sum_{i=0}^{P-1} T^{i}{}_{idle}\right)$$

$$= \left(\frac{1}{P}\right) * \left(T_{comp} + T_{commun} + T_{idle}\right)$$

Timpul de calcul - T_{comp}

- Depinde de:
  - dimensiunea problemei N
  - numărul de task-uri sau procesoare
  - caracteristicile procesoarelor şi memoriei

- Exemplu:
  - scalarea dimensiunii problemei sau a numărului de procesoare poate modifica performanța cache-ului sau eficacitatea pipelining-ului procesorului

Timpul de comunicare – T_{commun} (1)

- Timpul petrecut cu transmiterea - recepţia de date
- Tipuri:
  - inter-procesor
  - intra-processor
- Presupunere: au costuri comparabile

$$T_{msg} = t_s + t_w L$$

Machine	t_s (μs)	t_w(μs)
IBM SP2	40	0.11
Intel DELTA	77	0.54
Intel Paragon	121	0.07
Meiko CS-2	87	0.08
nCUBE-2	154	2.4
Thinking Machines CM-5	82	0.44
Workstations on Ethernet	1500	5.0
Workstations on FDDI	1150	1.1

Timpul de comunicare – T_{commun} (1)

- Măsurători experimentale: timp dus-întors pentru mesaje (RTT)

Timpul idle - T_{idle}

- Datorită:
  - lipsei calculelor → *load balancing*
  - lipsei datelor → *overlapping computation and communication*

Model revizuit pentru cost comunicare

- Model comunicare simplu: $T_{msg} = t_s + t_w L$
- Competiţie pentru bandă: $T_{msg\text{-}b} = t_s + t_w SL$
- S = număr de procesoare care comunică concurent pe acelaşi canal (și în același sens)

Exemplu – Sequential Floyd

- $I[i,j]_0 = 0$, daca i == j
- $I[i,j]_0 = cost(i,j)$, dacă există muchie și i != j
- $I[i,j]_0 = MAX$, altfel

```
for [k = 0 to N - 1]
  for [i = 0 to N - 1]
    for [j = 0 to N - 1]
      I[i,j]_{k+1} = min(I[i,j]_k, I[i,k]_k + I[k,j]_k)
```

Durata unei iterații

$$T = t_c N^3$$

Exemplu – Parallel Floyd 1 (1)

- Bazat pe o descompunere uni-dimensională pe rânduri a matricei I (algoritmul poate folosi cel mult P procesoare, P<=N)
- Fiecare task are unul sau mai multe rânduri adiacente din I şi răspunde de calculul acestor rânduri

```
for [k = 0 to N-1]
   for [i = local_i_start to local_i_end]
     for [j = 0 to N-1]
     I[i,j]k+1 = min(I[i,j]k, I[i,k]k + I[k,j]k)
```

Exemplu – Parallel Floyd 1 (2)

- În pasul k, procesele au nevoie de linia k din matricea I.
- Procesorul cu această linie o difuzează tuturor în log P paşi folosind o structură arborescentă.
- Deoarece fiecare mesaj are N cuvinte, timpul de difuzare a unei linii este: $\log P\ (t_s + t_w N)$

Exemplu – Parallel Floyd 1 (3)

- Fiecare task este rădăcină pentru cel puţin o difuzare (dacă P ≤ N).
- Folosind un hipercub, fiecare nod poate difuza în log P paşi
- Pentru N difuzări, cu mesaje de lungime N:

$$T_{Floyd1} = \frac{t_c N^3}{P} + N \log P \, (t_s + t_w N)$$

Timpul de comunicare

Timpul algoritmului secvențial împărțit la numărul de procesoa

Exemplu – Parallel Floyd 2 (1)

- Descompunere bi-dimensională a matricelor.
- Foloseşte până la N^2 procesoare.

```
for [k = 0 to N - 1]
  for [i = local_i_start to local_i_end]
    for [j = local_j_start to local_j_end]
      I[i,j]k+1 = min(I[i,j]k, I[i,k]k + I[k,j]k)
```

- Fiecare task răspunde de calculul elementelor situate intr-un patrat.

Exemplu – Parallel Floyd 2 (2)

- Cerinţe de comunicare în pasul k: două operaţii de difuzare.
- În fiecare pas, fiecare task cere, în plus faţă de datele locale, $\frac{N}{\sqrt{P}}$ valori de la două task-uri din aceeaşi linie şi aceeaşi coloană.
- Fiecare din cele doua mesaje are $\frac{N}{\sqrt{P}}$ elemente.
- Difuzarea unui mesaj durează (sunt $\sqrt{P}$ procese):

$$log\sqrt{P}\left(t_s + \frac{t_w N}{\sqrt{P}}\right)$$

a) b)

Exemplu – Parallel Floyd 2 (3)

- În fiecare din N paşi, $\frac{N}{\sqrt{P}}$ valori trebuie difuzate la $\sqrt{P}$ task-uri în fiecare linie şi coloană (se poate folosi o structură hipercub); costul total este:

$$T_{\text{Floyd-2}} = t_c \frac{N^3}{P} + 2N \log\sqrt{P}\left(t_s + \frac{t_w N}{\sqrt{P}}\right)$$

$$= t_c \frac{N^3}{P} + N \log P \left(t_s + \frac{t_w N}{\sqrt{P}}\right)$$

Modelul LogP

Modelul LogP

L - latency sau întârzierea de transmitere a unui mesaj mic de la sursă la destinatar

o - overhead, durata pentru care procesorul este angajat în transmiterea sau recepţia fiecărui mesaj

g - gap, intervalul minim de timp între două transmiteri succesive sau două recepţii succesive la acelaşi modul

P - numărul de module procesor / memorie.

Modelul LogP

L - latency sau întârzierea de transmitere a unui mesaj mic de la sursă la destinatar

o - overhead, durata pentru care procesorul este angajat în transmiterea sau recepţia fiecărui mesaj

g - gap, intervalul minim de timp între două transmiteri succesive sau două recepţii succesive la acelaşi modul

P - numărul de module procesor / memorie.

Exemplu - Difuzarea unei valori (1)

- Folosind modelului PRAM*, transmiterea are loc conform tiparului:
  - P0 → P1
  - P0, P1 → P2, P3
  - P0, P1, P2, P3 → P4, P5, P6, P7

- Presupunând $g \geq 0$ şi luând P = 8, o =2, g = 2, L = 6, obţinem timp total de difuzare = 30 de unităţi.

* Parallel Random Access Machine – vezi curs 1.

P0
P4
P2
P6
P1
P5
P3
P7
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
Procesoare
Timp (unităti de timp)
Overhead T
gap
Overhead TR

Exemplu - Difuzarea unei valori (2)

- Pentru modelul LogP, fiecare proces transmite imediat ce are o valoare.
- Timpul total este de 24 de unităţi.

- Drum P0 – P1
  - Overhead transmitere: 2 unități
  - Latență: 6 unități
  - Overhead recepție: 2 unități
  - TOTAL: 10 unități

P0
P4
P2
P6
P1
P5
P3
P7
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
Procesoare
Timp (unități de timp)

Sumar

- Comunicarea prin mesaje
  - Comunicarea asincronă prin mesaje
  - Modelul Replicated Workers
  - Comunicarea sincronă prin mesaje
- Complexitatea algoritmilor distribuiți
  - Modelul Foster
  - Modelul LogP

Algoritmi Paraleli și Distribuiți
Ceasuri Logice
Ordonarea evenimentelor

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

De la ceasuri fizice la... ceasuri logice

Ceasuri fizice

- Timpul este ne-ambiguu în sisteme ne-distribuite
  - există un singur ceas și (de obicei) kernelul face apeluri pentru a-i accesa valoarea
- Folosit implicit în multe aplicații
  - *Ex:* make *(pentru a determina fișierele care trebuie recompilate)*
- Sistemele distribuite introduc ambiguitatea în legătură cu timpul

Ceasuri fizice (2)

- Computerele folosesc un cristal de quartz care oscilează la o valoare bine definită și generează întreruperi (clock ticks) la intervale regulate
- Fiecare *clock tick* incrementează o valoare din memorie – kernelul convertește această valoare într-un format standardizat
  - *Ex:* nr. de milisecunde pornind de la Thu Jan 1 12:00:00 GMT 1970.
- Diferența în timp dintre calculatoare se numește clock skew.

Ceasuri fizice (3)

- Clock skew dintre cele 2 computere confuzează make-ul:

Sincronizarea ceasurilor fizice

- 2 tipuri:
  - doar între 2 calculatoare
  - sincronizare cu un standard *(ex: UTC)*
- Este imposibil de eliminat complet clock skew și imposibil de a asigura acuratețea perfectă a unui ceas.

Sincronizarea ceasurilor fizice (2)

- Flaviu Cristian – adaptive internal clock synchronization:
  - time server, cu care se sincronizează celelalte ceasuri
  - probleme:
    - timpul nu trebuie sa curgă în sens invers
    - mesajul ajunge de la server la client într-un anumit timp

T_0 și T_1 sunt masurați cu același ceas

Sincronizarea ceasurilor fizice (3)

- Berkeley:
  - folosit în Berkeley UNIX
  - opus lui Flaviu Cristian*: un server de timp trimite mesaje periodic clienților pentru a afla timpul lor, calculează o medie și anunță clienții cum să-și actualizeze ceasurile*
  - scopul nu e de a se sincroniza cu time serverul, ci de a pune toate mașinile de acord asupra timpului (chiar dacă este greșit)

Sincronizarea ceasurilor fizice (4)

- Descentralizat: Network Time Protocol
  - Organizare ierarhică a serverelor de timp:
    - Stratul 1:
      - ceasuri de referință
        - *US Naval Observatory*
        - *Global Positioning System*
        - *Atomic Clocks*
    - Stratul 2:
      - routere, servere importante, etc.
  - Algoritmul și protocolul NTP extrem de complexe

    (RFC 1305 NTPv3 – 112 pagini , NTPv4 și mai complex)
  - Sincronizare precisă a ceasurilor: 1-50ms

Ceasuri logice (solutia Lamport)

Ceasuri logice și ordonarea evenimentelor

- Într-un algoritm distribuit, fiecare proces este caracterizat de:
  - o mulțime de stări
  - acțiuni care schimbă starea

- Eveniment *(producerea unei acțiuni)*
  - Evenimentele pot fi ordonate conform timpului fizic de producere.
  - Dificil pentru evenimente din procese diferite.

- Soluție mai simplă: ordinea relativă a evenimentelor

- Definită de relația petrecut înainte (→):
  - dacă **a** şi **b** sunt evenimente din acelaşi proces şi **a** îl precede in timp pe **b**, atunci a → b
  - când **a** reprezintă transmiterea unui mesaj de către un proces, iar **b** recepţia aceluiaşi mesaj de către un altul, atunci a → b
  - dacă a → b şi b → c, atunci a → c

Ceasuri logice și ordonarea evenimentelor (2)

- Mecanism de sincronizare bazat pe ***timp relativ***.
- ***timpul relativ*** poate să nu fie echivalent cu ***timpul real***.
  - *Exemplu:* Unix make
    - *Este important ca output.c să fie actualizat după generarea lui output.o ?*
- În aplicațiile distribuite ceea ce contează este ca procesele să *ajungă la un acord asupra ordinii de producere a* evenimentelor.
- Astfel de ***ceasuri*** se numesc ***ceasuri logice***.

Operating Systems

R. Stockton Gaines
Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems

CR Categories: 4.32, 5.29

Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

The Partial Ordering

Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21.7 (1978): 558-565.

Ceasuri logice și ordonarea evenimentelor (2)

- Ceas logic – întreg incrementat la producerea unui eveniment
- Algoritmul lui Lamport folosește relația → pentru a face ordonarea parțială a evenimentelor:
  - pentru fiecare proces o variabilă cl inițial 0
  - pentru fiecare mesaj un câmp tt
- Reguli:
  - la producerea unui eveniment intern:
    - cl este incrementat
    - valoarea ceasului logic este asociată evenimentului ca amprentă de timp
  - la transmiterea unui mesaj :
    - incrementează cl al procesului transmitator cu 1
    - actualizează tt = cl
  - la primirea unui mesaj cu amprenta de timp tt:
    - actualizează cl = maxim(cl, tt) + 1

- Ordonare parțială: $a \rightarrow b \Rightarrow cl(a) < cl(b)$

Exemplu

Aplicatie: semafoare distribuite

Semafoare distribuite

- Folosește primitive P și V
  - Când un proces execută o operaţie P sau V el difuzează mesaje m(id, tag, ts) celorlalte procese:

    broadcast ch(m) trimite m pe fiecare din ch[1:n]

    mesajul m are aceeasi amprenta de timp pe toate canalele
  - Analizează răspunsurile pentru a determina continuarea execuţiei
- Fiecare proces:
  - Stochează o coadă de mesaje mq și un ceas logic cl
  - La recepția unui mesaj P sau V îl stochează în mq, sortat crescător după ts
  - Transmite prin broadcast ACK *(folosit pentru a actualiza prefixele stabile din mq)*
  - Stochează o variabila s *(semaforul)*:
    - V – s este incrementat și mesajul șters
    - P – dacă $s > 0$, s decrementat și mesajul șters
      - mesajele P sunt procesate în ordinea în care apar în prefixul stabil deci fiecare proces ia aceeași decizie despre ordinea terminării operațiilor P

Aplicație: semafoare distribuite (2)

- Implementarea semafoarelor:
  - procesele Utiliz(i) inițiază operațiile P sau V
  - procesele Ajutor(i) implementează operațiile P și V

Aplicație: semafoare distribuite (3)

```
enum fel(V, P, ack);
chan opsem[1:n](int transm, fel op, int timp);
chan start[1:n](int timp);

process Utiliz[i= 1 to n] {
  int cl = 0;                    /* ceas logic */
  int ts;
  ...
  cl = cl + 1;
  broadcast opsem(i, V, cl);          /* operația V */
  ...
  cl = cl + 1;
  broadcast opsem(i, P, cl);          /* operația P */
  receive start[i](ts);
  cl = max(cl, ts)+1;
  ...
}
```

Aplicație: semafoare distribuite (4)

```
process Ajutor[i = 1 to n]{
    typedef struct{int transm, fel k, int ts) qelem;
    qelem qm[lmax]; /*coada ordonata dupa timestamp tm*/
    int cl = 0;
    int sem = valoare_initială;
    int transm; fel k; int ts;
    while (true) {
        receive opsem[i](transm, k, ts);
        cl = max(cl, ts) + 1;
        if (k == P or k == V) {
            inserează (transm, k, ts) în locul corespunzător în qm;
            cl = cl + 1; broadcast opsem(i, ack, cl); }
        else if (k == ack) {
            înregistrează transmiterea unui ack;
            for [mesajele V complet confirmate] {
                scoate mesaj din qm;
                sem = sem + 1; }
            for [mesajele P complet confirmate st sem > 0] {
                scoate mesaj (transm, k, ts) din qm; // atentie la coliziuni
                sem = sem - 1;
                if (tansm == i) {cl = cl + 1; send start[i](cl);} }
        }
    }
}
```

Ceasuri logice vectoriale

Ceasuri logice vectoriale

Cu soluţia Lamport:

e precede f $\Rightarrow$ amprenta_logică (e) < amprenta_logică (f)

Dar

amprenta_logică (e) < amprenta_logică (f) $\not\Rightarrow$ e precede f

Ex: se poate spune ca **e** precede **c** ?

Soluția: ceasuri logice vectoriale.

Ceasuri logice vectoriale (2)

- Fiecare P_i are asociat un tablou $V_i[1..n]$ în care:
  - $V_i[i]$ este numărul de evenimente produse în procesul P_i
  - $V_i[j]$ este numărul de evenimente despre care P_i ştie *(a aflat)* că au avut loc la P_j.

- Procesul P_i actualizează V_i la fiecare **eveniment** din P_i
  - e.g. pentru procesorul 3, (1,2,1,3) → (1,2,2,3)

- Când P_i **transmite** mesajul *m* *(eveniment send m):*
  - P_i incrementează $V_i[i]$
  - P_i adaugă V_i la *m* ca vector de amprente de timp curent $\boldsymbol{vt_m}$

- Când P_j **primeşte** *m* și $\boldsymbol{vt_m}$ *(eveniment receive m):*
  - P_j ajustează: $V_j[k] = max\{V_j[k], vt_m[k]\}$ pentru fiecare k

    e.g., P_2 primește un mesaj cu timpul (3,2,4) iar timpul curent al lui P_2 este (3,4,3), atunci P_2 ajusteaza timpul la (3,4,4)
  - P_i incrementează $V_j[i]$cu 1

Ceasuri logice vectoriale (3)

Aplicarea regulilor ceasurilor logice vectoriale

Reguli:

- $VT_1 = VT_2 \Leftrightarrow VT_1[i] = VT_2[i]$, pentru $i = 1, \dots, N$
- $VT_1 \leq VT_2 \Leftrightarrow VT_1[i] \leq VT_2[i]$, pentru $i = 1, \dots, N$
- $VT_1 < VT_2 \Leftrightarrow VT_1[i] \leq VT_2[i]$, și $VT_1 <> VT_2$ *(de exemplu (1,2,2) < (1,3,2))*

Fie vt(a) şi vt(b) vectorii de amprente de timp asociaţi ev. a şi b. Atunci:

vt(a) < vt(b) $\Rightarrow$ **evenimentul a precede cauzal b**

vt(a) $\not<$ **vt(b) and vt(a)** $\not>$ **vt(b) and vt(a)** $\neq$ **vt(b)** $\Leftrightarrow$ **evenimentele a şi b sunt concurente**

Aplicație: Ordonare Cauzală Multicast

Ordonare Cauzală Multicast

- Procesele unei colecţii P comunică între ele doar prin mesaje cu difuzare
- Se cere ca mesajele să respecte **dependenţa cauzală**

$$m \rightarrow m' \Rightarrow livrarep(m) \rightarrow livrarep(m')$$

Protocolul *(vectori de timp)*:

- fiecare proces $P_i = (i = 1..n)$ are asociat un vector $V_i[1..n]$, cu toate elementele iniţial 0
- se numara doar operatiile de transmitere de mesaje
- $V_i[i]$, este nr ev. transmitere de mesaje produse de P_i;
- $V_i[j]$ este nr ev. transmitere de mesaje despre care $\boldsymbol{P_i}$ ştie (a aflat) că au avut loc la P_j.
- Procesul P_i actualizează V_i la fiecare eveniment de trimitere sau recepţie de mesaj din P_i.

Ordonare Cauzală Multicast (2)

- Când P_S **transmite** mesajul *m*:
  - $\boldsymbol{P_S}$ **incrementează** $\boldsymbol{V_S[s]}$
  - $\boldsymbol{P_S}$ **adaugă** $\boldsymbol{V_s}$ **la** ***m*** **ca vector de amprente de timp curent** $\boldsymbol{vt_m}$
    - **Obs: Pentru ordonare cauzală, incrementarea lui** $\boldsymbol{V_s[s]}$ **se face doar la transmiterea de mesaje de către s**
    - vt_m spune receptorului câte evenimente (din alte procese) au precedat **m** şi ar putea influenţa cauzal pe **m**.

- Când P_d **primeşte** mesajul *m* împreună cu vt_m, mesajul este păstrat într-o coadă de **întârziere** şi este **livrat** doar dacă:
  - $\boldsymbol{vt_s[s] = V_d[s] + 1}$ (acesta este următorul timestamp pe care **d** îl așteaptă de la **s**)
  - $\boldsymbol{vt_m[k] \leq V_d[k]\ pentru\ k <> s}$ (**d** a văzut toate mesajele ce au fost văzute de **s** la momentul când a trimis mesajul **m**)

- Când mesajul este livrat, V_d este actualizat conform regulilor vectorilor de timp:
  $\boldsymbol{V_d[k] = max\{V_d[k],\ vt_m[k]\}}$ **pentru fiecare** $\boldsymbol{k = 1, n}$**.**

Ordonare Cauzală Multicast (3)

Alte acţiuni ale protocolului la livrarea mesajelor:

- dacă $vt_m[k] > V_d[k]$ pentru un oarecare k atunci se întârzie *m*

P_S a primit mesaje de care mesajul curent poate fi cauzal dependent, dar pe care P_d încă nu le-a primit;

- dacă $vt_m[s] > V_d[s] + 1$ atunci se întârzie *m*

mai sunt mesaje de la P_S pe care P_s nu le-a primit

(asigură ordinea FIFO pentru canale nonFIFO)

- dacă $vt_m[s] < V_d[s]$ atunci rejectează *m*

m este un duplicat al unui mesaj primit anterior

Sumar

- Ceasuri fizice
- Ceasuri logice și ordonarea evenimentelor
- Semafoare distribuite
- Ceasuri logice vectoriale
- Ordonare cauzală multicast

Algoritmi Paraleli și Distribuiți
Algoritmi undă.

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Algoritmi undă

- În dezvoltarea algoritmilor distribuiți apar frecvent anumite tipuri de **probleme generale** pentru **rețele de procese**.
- Rezolvate prin transmitere de mesaje după o **schemă predefinită**, **dependentă de topologie** care asigură participarea **tuturor proceselor**.
- Justifică tratarea lor izolată de alți algoritmi în care aceste scheme pot fi folosite.

Notiuni preliminare - configuratii

- Un **program distribuit** consta dintr-o **colectie de procese** secventiale comunicante
- **Configuratie** = Ansamblul **starilor** tuturor proceselor la un moment dat
- O **operatie** modifica starea unui proces, implicit configuratia
- O executie a programului poate fi **modelata** printr-o secventa de **configuratii**.

- Ex.1 Program distribuit = colectie a doua procese, initial a si b au valoarea
- process P1{ int a =0; a = a+1;} process P2{ int b =0; b = b+1;}
- configuratia = ansamblul valorilor celor doua variabile modificate de procese configuratia initiala este <0,0>
- daca operatia primului proces se executa mai intai, configuratia devine <1,0>
- O executie posibila = secventa de configuratii: (<0,0> , <1,0> , <1,1>)

Sisteme de tranziţii

- Un program distribuit poate fi modelat ca un **sistem de tranziții**:
  - Mulțimea tuturor stărilor (**configurațiilor**) posibile ale sistemului
  - Tranzițiile pe care sistemul le poate face între stări
  - Stările din care sistemul poate porni (inițiale)

- Formal, un **sistem de tranziții** este un triplet $S = (C, \rightarrow, I)$
  C este o mulțime de configurații
  $\rightarrow$ este relația de tranziție binară pe C
  I este setul configurațiilor inițiale (o submulțime a lui C)

- O **execuție** a lui S este o secvență maximală
  $$E = (\gamma_0, \gamma_1, \gamma_2, \dots), \text{unde } \gamma_0 \in I \text{ și } \gamma_i \rightarrow \gamma_{i+1},\ pentru\ i \geq 0.$$

- O configurație **terminală** γ nu are succesor:
  $$\nexists\ \delta \text{ astfel încât } \gamma \rightarrow \delta.$$

Sisteme de tranziţii (2)

- O secvență E este **maximală** dacă:
  - Este infinită sau
  - Sfârșește într-o configurație terminală.
- Configurația δ este **tangibilă** din γ dacă există o secvență $\gamma_0, \gamma_1, \gamma_2, \dots, \gamma_k$ a.î.:

$$\gamma = \gamma_0$$
$$\delta = \gamma_k$$
$$\gamma_i \rightarrow \gamma_{i+1}, pentru\ i = \overline{0, k-1}$$

- Un **sistem distribuit** constă dintr-o colecție de procese și un subsistem de comunicații.
- **Convenții:**
  - pentru sistem: **tranziții** și **configurații**
  - pentru proces: **evenimente (interne / send,receive)** și **stări**
- Unei **execuții** E îi corespunde o **secvență de evenimente** din diferite procese.

Sisteme de tranziţii (3)

- Pentru o execuție E, **relația de ordine cauzală** $<$ este cea mai slabă relație care satisface:
  - dacă $\boldsymbol{a}$ și $\boldsymbol{b}$ sunt două evenimente diferite ale aceluiași proces și $\boldsymbol{a}$ se produce înaintea lui $\boldsymbol{b}$ atunci $\boldsymbol{a} < \boldsymbol{b}$
  - dacă $\boldsymbol{a}$ este un eveniment send, iar $\boldsymbol{b}$ evenimentul receive corespunzător atunci $\boldsymbol{a} < \boldsymbol{b}$
  - dacă $\boldsymbol{a} < \boldsymbol{b}$ și $\boldsymbol{b} < \boldsymbol{c}$ atunci $\boldsymbol{a} < \boldsymbol{c}$ (< este tranzitivă).
- Dacă $\boldsymbol{a} < \boldsymbol{b}$ și $\boldsymbol{b} < \boldsymbol{a}$ atunci a și b sunt **concurente**.

- O execuție E este **echivalentă** cu F ($E \sim F$) dacă:
  - au aceeași colecție de evenimente (ordinea diferă)
  - evenimentele respectă aceeași relație de ordine cauzală
  - ultima configurație a lui E coincide cu ultima configurație a lui F
- **Obs**: două execuții echivalente pot să nu aibă aceleași configurații.
- Un **calcul** (computation) al unui algoritm distribuit este o clasă de echivalență (sub relația $\sim$) a execuțiilor algoritmului.

Algoritmi undă (Wave algorithms)

- **Schema** de transmitere de mesaje:
  - dependentă de topologie
  - asigură participarea tuturor proceselor
- Algoritmii undă implementează astfel de scheme pentru:
  - difuzarea informației
  - realizarea unei sincronizări globale între procese
  - declanșarea unu eveniment în fiecare proces
  - calculul unei funcții în care fiecare proces participă cu o parte a datelor de intrare

- Proprietăți:
  - **terminare** – fiecare *calcul* este finit
  - **decizie** – fiecare *calcul* conține cel puțin un eveniment de decizie (*decide*)
  - **dependență** – în fiecare *calcul*, fiecare *decide* este precedat cauzal de un eveniment în fiecare proces

Definiții (1)

Calcul (computation) ⇔ **undă** (wave)

Categorii de procese:

Inițiatori (starters) – primul eveniment este unul intern sau un send

Neinițiatori (followers) – primul eveniment este un receive

Clasificare:

- centralizare
  - algoritmi centralizați – un inițiator
  - algoritmi descentralizați – set arbitrar de inițiatori

- topologie
  - inel, arbore, clică etc.
  - fixă (nu se produc modificări topologice)
  - nedirecționată (canale bidirecționale); excepțiile menționate explicit
  - conectată (există o cale între oricare două procese)

Definiții (2)

- cunoștințe inițiale – exemple:
  - identitatea proprie (nume)
  - identitățile vecinilor
  - **setul direcției (ordinea vecinilor)**

- numărul de decizii (regulă = cel mult o decizie în fiecare proces)
  - un singur proces decide
  - toate decid
  - unele decid

- complexitate
  - <u>număr de mesaje schimbate</u>
  - număr de biți interschimbați
  - <u>timpul necesar pentru un calcul</u>

Noțiuni preliminare
Sensul legăturilor

Complexitatea comunicării într-un algoritm distribuit depinde de topologie, dar și de:

- cunoașterea topologiei la nivelul fiecărui nod (topological awareness)

- setul de direcții (sense of direction)
  - muchiile incidente unui nod sunt etichetate cu direcția spre care conduc în rețea
  - setul de etichete este același pentru fiecare nod
  - mărimea setului depinde de topologie (2 pt. inel, 4 pt. tor, etc)
  - trebuie respectată o **condiție suplimentară de consistență**

Noțiuni preliminare
Sensul legăturilor (2)

Inel:

- 2 direcții: *Prec* (precedent) și *Urm* (următor)
- condiția de consistență: **precedentul lui p este q ⇔ următorul lui q este p**

Noțiuni preliminare
Sensul legăturilor (3)

Clică:

- N noduri de grad $N-1$
- set $direcții = \{1, 2, \ldots, \mathrm{N}-1\}$
- condiția de consistență: **direcția de la nodul i la j are sensul $(j-i)\ mod\ N$**

Noțiuni preliminare
Sensul legăturilor (4)

Hipercub:

- pentru un hipercub n dimensional
- set $direcții \ = \{0, \dots n-1\}$
- condiția de consistență: **două noduri $(b_0, \dots, b_{n-1})$, $(c_0, \dots, c_{n-1})$ legate prin muchie cu eticheta i diferă doar în bitul i**

Notație transmitere mesaje

1. transmitere prin adresare directă

send ch[q] (mesaj)

- unde **q** este identitatea unică, globală a canalului către receptor

2. transmitere prin adresare indirectă

send ch[direcție] (mesaj)

- unde **direcție** este locală procesului care execută operația send
- identifică unul din canalele pe care procesul poate transmite

Algoritm inel

Algoritmul inel

- fiecare proces are un vecin dedicat, Urm
- transmiterea folosește **adresarea prin direcție** (Urm, Prec)
- toate canalele selectate prin Urm formează un ciclu Hamiltonian
- algoritmul este **centralizat**:
  - inițiatorul trimite un token (jeton) care este pasat de fiecare proces de-a lungul ciclului până ajunge înapoi la inițiator;
  - inițiatorul ia apoi decizia

Algoritmul inel

```
chan token[1..n](tok_type tok);
```

```
/* inițiator */
process P[I]{
        tok_type tok;
        send  token[Urm](tok);
        receive token[I](tok);
        decide
}
```

```
/* neinițiatori */
process P[k=1 to n, k<>I] {
        tok_type tok;
        receive token[k](tok);
        send token[Urm](tok)
}
```

Număr de mesaje = n, Timp = n

Algoritm arbore

Algoritmul arbore

- Se aplică:
  - unei topologii arbore
  - unei topologii arbitrare în care se cunoaște un arbore de acoperire
- Fiecare nod cunoaște identitatea proprie și identitățile vecinilor
- Mulțimea tuturor identităților este **Ids**
- Pentru fiecare proces, se folosesc variabilele locale
  - Vecini – mulțimea identităților vecinilor (**q** = identitatea procesului q)
  - rec[q] – true dacă procesul a primit un mesaj de la vecinul q
- Inițiatorii sunt toate nodurile frunză
- **Algoritm:**
  - fiecare proces trimite exact un mesaj
  - când un proces a primit un mesaj pe fiecare canal incident mai puțin unul (condiție îndeplinită inițial de frunze) el trimite un mesaj pe canalul rămas
  - când un proces a primit câte un mesaj pe toate canalele sale atunci **decide**

Algoritmul arbore (2)

```
chan ch[1:n](int id, tok_type tok);
   /* fiecare proces are un canal propriu */
process Proc[p = 1 to n]{
   bool Vecini[1:n] = vecinii_lui_p;
   bool rec[1:n] = ([|n|] * false);
   int  r = numar_vecini_p;
   tok_type tok; int id, q0;
   while (r > 1){ receive ch[p](id, tok);
        rec[id] = true; r = r - 1 }
   /* de la un singur vecin, q0, nu s-a primit mesaj */
   afla q0 : Vecini[q0] and NOT rec[q0];
   send ch[q0](p, tok);
   receive ch[p](q0, tok); rec[q0] = true;
   decide
   /* informeaza celelalte procese despre decizie */
   /* for [q = 1 to n st Vecini[q] and q<>q0] send ch[q](p, tok); */
}
```

Număr de mesaje = N (egal cu nr. procese), Timp = O(D)

Algoritmul arbore (3) exemplu de execuție

(b) (a) și apoi le fiu vecin transmit iar după recepție, decid transmit

Algoritmul arbore (4)

Teoremă: Algoritmul "arbore" este un algoritm undă.

1. calcul finit
   - algoritmul foloseşte cel mult N mesaje → algoritmul atinge o configuraţie terminală γ după un număr finit de paşi.
2. în γ, cel puţin un proces a executat un eveniment "*decide*"
3. "*decide*" este precedat de un eveniment în fiecare proces

Subseturile T_{pq} *şi* T_{qp}

Algoritmul arbore (5)

Descompunera lui T_{pq}

Descompunera lui T

Algoritmul arbore

(2) în γ, cel puţin un proces a executat un eveniment "*decide*“

- demo prin reducere la absurd
- N noduri ➔ N-1 legături bidirecţionale între vecini ➔ 2N-2 posibile recepţii de mesaje
- K procese au trimis un mesaj în γ
- mesajele au fost receptionate
- ➔ F = 2N–2–K receptii neconsumate
- Pp. că nici un proces nu a executat "*decide*" în γ
  - fiecare proces este “blocat” la una din operatiile **receive**
- N-K procese nu au trimis mesaj (sunt in bucla while r>1 ...)

  ⇒ fiecare din ele mai are de primit cel puţin 2 mesaje
- K procese au trimis dar nici unul nu a decis

  => fiecare mai are de primit un mesaj

  => receptii neconsumate F >= 2(N-K)+K

Impreuna cu F = 2N–2–K

➔ 2N–2–K >= 2(N-K)+K

-2 >= 0

➔ cel puţin un proces a executat "*decide*" în γ

```
while (r>1){receive ch[p](id,tok);
                    rec[id] = true; r = r-1;}
/* de la un singur vecin, q0, nu s-a primit mesaj */
afla q0: Vecini[q0] and NOT rec[q0];
send ch[q0](p, tok);
receive ch[p](q0, tok); rec[q0] = true;
decide;
```

Algoritmul arbore

- (3) "decide" este precedat de un eveniment in fiecare proces
- Fie: f_{pq} **transmiterea** unui mesaj de la p la q
  - g_{pq} **receptia** la q a unui mesaj transmis de p
  - Cs submulţimea evenimentelor **e** produse în procesul **s**
  - ≤ relaţia de precedenţă
- Se dem. prin inductie peste evenimentele de recepţie:
- **pentru orice s din Tpq există e ∈ Cs : e ≤ g_{pq}**
- Presupunem că propozitia este adevărată pentru toate evenimentele *receive* care preced g_{pq} şi demonstrăm că este adevărată şi pentru g_{pq}.
- cf. algoritm, p trimite mesaj lui q dupa ce primeste mesaj de la toti ceilalti vecini r
- In Trp (similar in celilalti sub-arbori din Tpq):
- **e ≤ g_{rp} (ipoteza inductie)**
- **g_{rp} ≤ f_{pq}**
- **f_{pq} ≤ g_{pq}**
- **➔ e ≤ g_{pq}**

Algoritm ecou

Algoritmul ecou

- se aplică unor **topologii arbitrare**
- este **centralizat**; există un singur inițiator, I
- propus de Chang; prezentam versiunea mai eficientă Segall
- bazat pe inundarea rețelei cu mesaje **tok**
  - se stabilește un arbore de acoperire
  - mesaje **tok** sunt transmise înapoi spre rădăcină prin canalele arborelui de acoperire

Algoritmul ecou

```
chan ch[1:n](int id, tik_type tok);
const int I = id_initiator;

process Proc(I){
   bool Vecini[1:n] = vecinii_lui_I;
   int r = numar_vecini_I, id;
   tok_type tok;

   for [i=1 to n st Vecini[i]] send ch[i](I, tok);
   while (r > 0) {
       receive ch[I](id, tok);
       r = r - 1;
   }
   decide;
}
```

Algoritmul ecou (2)

```
process Proc[p=1 to n, p <> I]{
   bool Vecini[1:n] = vecinii_lui_p;
   int r = numar_vecini_p, id, parinte;
   tok_type tok;

   receive ch[p](parinte, tok); r = r - 1;
   for [q=1 to n st Vecini[q] and q<> parinte)
       send ch[q](p, tok);
   while (r > 0) {
       receive ch[p](id, tok); r = r - 1; }
   send ch[parinte](p, tok)
}
```

$$Mesaje = 2|E|\,, Timp = O(N)$$

Algoritmul ecou (3)
exemplu de execuție

	Nodul 1	Nodul 2	Nodul 3	Nodul 4
Vecini =	{2}	{1, 3, 4}	{2, 4}	{2, 3}
părinte =		1	2	2
r =	0	0	0	0

Algoritmul fazelor
(pulsatiilor, heartbeat, gossiping..)

Algoritmul fazelor

- algoritm **descentralizat**
- **topologii arbitrare**
- **canale unidirecționale**
- vecinii sunt: **in**-vecini și **out**-vecini
- procesele cunosc diametrul grafului D (sau o valoare $D' > D$)
- fiecare proces trimite exact D mesaje fiecărui out-vecin
- mesajul $i + 1$ este trimis fiecărui out-vecin numai după ce i mesaje au fost primite de la fiecare in-vecin

Algoritmul fazelor

```
chan ch[1:n](int id, tok_type tok);
const int D = diametrul_retelei;

process Proc[p = 1 to n] {
   bool in[1:n] = in-vecinii_lui_p;
   bool out[1:n] = out-vecinii_lui_p;
   int rec[1:n] = ([n] 0);
   /* rec[q] = număr mesaje primite de la q */
   int sent = 0;
   /* număr de mesaje transmise fiecărui out-vecin */
   tok_type tok;
   int id;
```

Algoritmul fazelor (2)

```
    if (p este inițiator) {
        for [q = 1 to n st out[q]] send ch[q](p, tok);
        sent = sent + 1
    }

    /* min(rec) este min(rec[q], ∀q ∈ in) */
    while (min(rec) < D) {
        receive ch[p](id, tok); rec[id] = rec[id] + 1;
        if (min(rec) >= sent and sent < D) {
            for [q=1 to n st out[q]] send ch[q](p, tok);
            sent = sent + 1;
        }
    }
    decide;
}
```

Mesaje = 2D|E|, unde E este mulț. canalelor nedirijate (2 canale dirijate)
Timp = 2D

Algoritmul fazelor (3)
exemplu de execuție

D (diametrul rețelei) = 3

	Nodul 1	Nodul 2	Nodul 3	Nodul 4
rec[4] =	{2:3}	{1:3, 4:3}	{2:3}	{3:3}
sent =	3	3	3	3

Algoritmul fazelor (4)

Teoremă: Algoritmul "fazelor" este un algoritm undă

1. calcul finit

   fiecare proces trimite cel mult D mesaje prin fiecare canal ⇒ algoritmul atinge o configurație terminală γ după un număr finit de pași

2. în γ, fiecare proces a executat un eveniment "decide"

   presupunem cel puțin un inițiator în C (pot fi mai mulți)

   2.1. fiecare proces a trimis cel puțin un mesaj

   2.2. fiecare proces a decis

3. "decide" este precedat de un eveniment în fiecare proces

- **T. Algoritmul "fazelor" este un algoritm unda**
- (1) **calcul finit**

  fiecare proces trimite cel mult D mesaje fiecarui out-vecin

  → algoritmul atinge o configuratie terminala **γ** intr-un calcul C cu numar finit de pasi
- (2) **in γ, fiecare proces a inregistrat un eveniment "decide"**

  presupunem cel putin un initiator in calculul C (pot fi mai multi)

  (2.1) fiecare proces a trimis cel putin un mesaj

```
if (p este initiator){
   for [q = 1 to n st out[q]]{
     send ch[q](p, tok); sent = sent+1;}

while (min(rec) < D){
  receive ch[p](id, tok); rec[id] = rec[id]+1;
  if (min (rec) >= sent and sent < D){
     for (q=1 to N st out[q]]
      send ch[q](p, tok);
     sent = sent+1;}
  ...
```

daca p ne-initiator: **sent = 0** → cand p primeste primul mesaj, transmite unul tuturor out-vecinilor

(2.2) In configuratia terminala γ fiecare proces a decis

```
while (min(rec) < D){
  receive ch[p](id, tok); rec[id] = rec[id]+1;
  if (min (rec) >= sent and sent < D){
     for (q=1 to N st out[q]]
      send ch[q](p, tok);
     sent = sent+1;}
} decide;
```

Demonstram ca orice proces iese din ciclu, adica min(rec) = D , deci nu exista in-vecin care sa-i fi trimis mai putin de D mesaje

Fie p procesul cu cel mai mic $sent_p$ in γ ➔ ∀q $sent_q >= sent_p$ (1)

In particular, se aplica pentru toti in-vecinii q ai lui p

In γ, toate mes. trimise de q sunt recept. de p ➔ $sent_q = rec_p[q]$ (2)

(1)&(2) → $min_q (rec_p[q]) >= sent_p$ conditia din **if** este semi-indeplinita

➔ $sent_p$ =D; altfel p ar fi trimis mesaje suplimentare la ultima receptie

$sent_q$ =D ∀ q ➔ $rec_p[q]$ =D pentru orice canal qp

min(rec) = D ➔ fiecare proces a iesit din ciclu si a decis

- **(3) "decide" este precedat de un eveniment in fiecare proces**

  Ne referim la decizia din procesul **p**

  D este diametrul → pentru orice **q** exista o cale la **p** cu lungime <= D

  q = p(0), p(1), …, p(k) = **p**, cu lungimea k<=D

  Intr-o faza, transmisia precede receptia aceluiasi mesaj

  $f_{p(i)p(i+1)}^{(i+1)} \leq g_{p(i)p(i+1)}^{(i+1)}$ pentru 0<= i <k

  Din algoritm, receptia unui mesaj precede transmiterea lui in faza urmatoare

  $g_{p(i)p(i+1)}^{(i+1)} \leq f_{p(i+1)p(i+2)}^{(i+2)}$ pentru 0<= i <k-1

  Aplicand succesiv relatiile, rezulta $f_{p(0)p(1)}^{(1)} \leq g_{p(k-1)p(k)}^{(k)}$

  In plus, din algoritm, $g_{p(k-1)p(k)}^{(k)} \leq d_p$

  ➔ decizia lui p este precedata de un eveniment in orice q: $f_{qp(1)}^{(1)} \leq d_p$

Algoritmul fazelor pentru clici

```
chan ch[1:n](int id, tok_type tok);
process Proc[p = 1 to N] {
   bool Vecini[1:n] = vecinii_lui_p;
   int n_vecini = numar-vecini_p;
   int rec = 0, sent = 0, id;
   tok_type tok;
   if (p este initiator) {
        for [q=1 to N st Vecini[q]] send ch[q](p, tok);
        sent = sent+1;
   }
   while (rec < n_vecini) {
        receive ch[p](id, tok); rec = rec + 1;
        if (sent == 0) {
                for [q=1 to N st Vecini[q]] send ch[q](p, tok);
                sent = sent + 1;
   }    }
   decide;
}
```

$$Mesaje = N(N-1), Timp = 2$$

Algoritmul fazelor pentru clici (2) exemplu de execuție

	Nodul 1	Nodul 2	Nodul 3	Nodul 4
rec =	3	3	3	3
sent =	1	1	1	1

Algoritmul lui Finn

Algortimul lui Finn

- nu cere cunoașterea diametrului
- se bazează pe cunoașterea identificatorilor proceselor
- în mesaje se transmit seturi de identificatori de procese
- proc **p** păstreaza două mulțimi de ids

  Inc – mulțimea proceselor **q** pentru care un eveniment în **q** precede cel mai recent eveniment în **p**

  NInc – mulțimea proceselor **q** pentru care fiecare vecin **r** are un eveniment care precede cel mai recent eveniment în **p**

- **Algoritmul:**

  - inițial: $Inc = \{p\}, \quad NInc = \emptyset$
  - p trimite mesaje cu Inc și $NInc$ de fiecare dată când Inc sau $NInc$ crește
  - când p primește mesaje cu Inc și $\mathrm{N}Inc$, actualizează Inc și $\mathrm{N}Inc$ (reuniune)
  - când p a primit un mesaj de la toți in-vecini, p este inserat în $\mathrm{N}Inc$
  - când Inc devine egal cu $\mathrm{N}Inc$, p decide

Algortimul lui Finn (2)

```
typedef SOP {multime identificatori};
chan ch[1:N](int id, SOP Inc, SOP NInc);

process Proc[p = 1 to n]{
   SOP Inc = {p};
   SOP Ninc = Ø;
   /* de la cine a recepționat  */
   bool rec[1:n] = ([n] false);
   bool out[1:n] = out-vecinii_lui_p;
   bool in[1:n] = in-vecinii_lui_p;
   SOP rInc, rNInc;
   int id;

   if (p este initiator)
       for [q = 1 to n st out[q]]
          send ch[q](p, Inc, NInc);
```

Algortimul lui Finn (2)

```
  while (Inc <> Ninc) {
      receive ch[p](id, rInc, rNInc);

      Inc  = Inc  U rInc;
      NInc = NInc U rNInc;

      rec[id] = true;
      if (rec[id, for ∀id=1..n and in[id])
            NInc = NInc U {p};
      if (Inc sau NInc modificat)
            for [q=1 to n st out[q]]
                  send ch[q](p, Inc, NInc);
  }
  decide;
}
```

$$Mesaje \leq 2N|E|, \qquad Timp = O(D)$$

Algoritmul lui Finn (3)
exemplu de execuție

	Nodul 1	Nodul 2	Nodul 3	Nodul 4
Inc =	{1, 2, 3, 4}	{1, 2, 3, 4}	{1, 2, 3, 4}	{1, 2, 3, 4}
Ninc =	{1, 2, 3, 4}	{1, 2, 3, 4}	{1, 2, 3, 4}	{1, 2, 3, 4}
rec =	{2:T}	{1:T, 4:T}	{2:T}	{3:T}

Algoritmul lui Finn (4)

Teoremă: Algoritmul lui Finn este un algoritm undă

1. calcul finit

2. în γ, fiecare proces a executat un eveniment "decide“

   2.1. în γ fiecare proces a trimis cel puțin un mesaj pe fiecare canal (dem. similară alg. fazelor)

   2.2. în γ fiecare proces a decis

      2.2.1. $\forall p,\ Inc_p\ conține\ toate\ procesele\ (\text{în}\ \gamma)$

      2.2.2. $\forall p\ și\ q,\ NInc_p = NInc_q$

3. "decide" este precedat de un eveniment în fiecare proces

T. Algoritmul lui Finn este un algoritm unda

- (1) **calcul finit**
- Se trimite un mesaj la fiecare modificare Inc sau Ninc
- Dimensiunea celor doua seturi luate impreuna este initial 1 si ajunge la 2N
  → numarul de mesaje transmise este limitat superior la 4N|E|

- (2) **In γ, fiecare proces a executat un eveniment "decide"**
- 2.1. in **γ** fiecare proces a trimis cel putin un mesaj pe fiecare canal:

  p trimite tuturor vecinilor q → se schimba Inc_q → q transmite vecinilor sai; in plus graful este tare conex etc.
- 2.2. in **γ** fiecare proces a decis

  2.2.1. ∀ p, Inc_p contine toate procesele (in γ)

  daca exista arc pq, la modificare Inc_p, p il trimite lui q; q il include in Inc_q

  graf tare conex → pentru orice p si q: Inc_q include Inc_p

  impreuna cu Inc_p include Inc_q → $Inc_p = Inc_q$ pentru **∀** p si q

  → Inc_p contine identitatile tuturor proceselor

2.2.2. ∀ p si q, $NInc_p = NInc_q$

Fiecare proces a trimis un mesaj pe fiecare canal

→ fiecare proces a receptionat de la toti vecinii (∀ id € in: rec[id])

→ (cf alg) p este adaugat la NInc

$NInc_p$ contine toate procesele – demo similara cu Inc_p

→ $Inc_p = NInc_p$ in γ

→ fiecare proces a decis in γ

- (3) In procesul p, "decide" este precedat de un eveniment in fiecare proces

Fie q un in-vecin al lui p

Fie evenimentul intern lui q, **a_q: NInc := NInc U {q}**

conform algoritmului, a_p se executa daca pentru fiecare in-vecin r al lui q un eveniment din r precede a_q

Procesul p decide doar cand $Inc_p = NInc_p$ = setul tuturor proceselor

→ cand procesul p ia decizia:

- p € Inc
- in-vecinul q € Inc si q € NInc
- deci decizia lui p este precedata de un eveniment din q si un eveniment din fiecare din in-vecinii lui q
- relatia este valabila pentru oricare doua procese din care unul este in-vecinul celuilalt
- retea conexa → decizia lui p este precedata de un eveniment din fiecare din celelalte procese

Sumar

- Sisteme de tranziții
- Caracteristici ale algoritmilor undă
- Algoritmul inel
- Algoritmul arbore
- Algoritmul ecou
- Algoritmul fazelor
- Algoritmul lui Finn

Algoritmi Paraleli și Distribuiți
Alegerea unui lider

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Alegerea unui lider

- Gerard LeLann a propus problema alegerii unui lider într-un paper celebru din 1977
- Multe sisteme distribuite sunt bazate pe paradigma client-server:
  - Un server/coordonator (lider)
  - Mai mulți clienți
- Ce se întâmplă dacă liderul se defectează?
  - Soluție: se alege un nou lider
- Problema generică: dintr-o colecție de procese, se alege unul singur care urmează să joace un rol special *(lider)*:
  - Execută operații de inițializare
  - Controlează alte procese

INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR
© IFIP, NORTH-HOLLAND PUBLISHING COMPANY (1977)

DISTRIBUTED SYSTEMS—TOWARDS A FORMAL APPROACH

GÉRARD LE LANN
IRISA—Université de Rennes—BP 25 A
35 031 Rennes Cedex, France

Le Lann, Gérard. "Distributed Systems-Towards a Formal Approach." *IFIP congress*. Vol. 7. 1977.

Packet-switching computer communication networks are examples of distributed systems. With the large scale emergence of mini and micro-computers, it is now possible to design special or general purpose distributed systems. However, as new problems have to be solved, new techniques and algorithms must be devised to operate such distributed systems in a satisfactory manner. In this paper, basic characteristics of distributed systems are analysed and fundamental principles and definitions are given. It is shown that distributed systems are not just simple extensions of monolithic systems. Distributed control techniques used in some planned or existing systems are presented. Finally, a formal approach to these problems is illustrated by the study of a mutual exclusion scheme intended for a distributed environment.

1. INTRODUCTION

Computer communication networks using packet-switching technology provide for the interconnection of data-processing equipments of any kind. Such systems, sometimes simply referred to as computer networks, may be viewed as multi-macroprocessors whenever the goals of resource-sharing are achieved. With the large-scale emergence of mini and microcomputers, it is now possible to envision building general or special purpose multimini and multimicrocomputers to be operated in a non-centralized manner. The need for automatic resource-sharing arises here as in a similar way it does for multimacroprocessor systems.

Two kinds of resources must be considered :
- system resources, multi-accessed by users and for which multiplexing is required (hidden sharing)
- user resources, which users agree to share according to some protocol of their own (explicit sharing).

This paper discusses the problems of system resource-sharing in a distributed environment. An example of a user-sharing problem is distributed data-base sharing.

2. DISTRIBUTED SYSTEMS-ELEMENTS FOR A FORMAL APPROACH

Experimental and public packet-switching computer

cesses called logical entities.

Let $F = \{f_i, i \varepsilon I\}$ be the set of the operating functions and $E_i = \{e_j^i, j \varepsilon J(i)\}$ be the set of entities participating in fuction f_i. At any instant t, it is possible to define $s_t(e_j^i)$ as the instantaneous state of entity e_j^i. It is therefore theoretically possible to define the global state of E_i at instant t as the vector $S_t(E_i) = \{..., s_t(e_j^i), ...$ for all $j \varepsilon J(i)\}$.

A system will be said to be f_i-centralized if there exists $k \varepsilon J(i)$ such that $S_t(E_i)$ is known to e_k^i.

An example is a system in which $\dim(E_i) = 1$; another example is a system in which entities run the operating function f_i by using a common "system table".

A system will be said to be totally centralized if it is f_i-centralized for any $i \varepsilon I$.

In contrast, a system will be said to be f_i-distributed if there does not exist $k \varepsilon J(i)$ such that $S_t(E_i)$ is known to e_k^i.

A system will be said to be totally distributed if it is f_i-distributed for any $i \varepsilon I$.

Typical cases of distributed systems are systems in

Alegerea unui lider

- Desemnarea statică a liderului ne-aplicabilă
  - nu se cunoaște compoziția exactă a grupului de procese
- Fiecare proces își cunoaște propria identitate și vecinii.
- Identitățile proceselor aparțin unei mulțimi total ordonate
- Alegerea liderului = determinarea procesului care are identitatea cea mai mică *(sau cea mai mare)*.
- Alegerea se face prin algoritmi descentralizați, cu participarea tuturor proceselor din colecție.

Alegerea unui lider

- Problemă:
  - Atunci când un proces se defectează, mai multe procese pot depista problema simultan
- Nu putem presupune că un singur proces detectează defectul
  - Algoritmul de alegere trebuie să conducă în final la situația în care toate procesele decid asupra aceluiași proces ca nou lider

Alegerea unui lider cu algoritmi undă

Caracteristici:

- toate procesele au același algoritm local
- algoritmul este **descentralizat** *(poate fi inițiat de oricare subset de procese)*
- în fiecare calcul algoritmul atinge o configurație finală în care:
  - un proces este lider
  - toate celelalte au pierdut
- varianta mai slabă:
  - un proces este lider
  - celelalte procese nu știu că au pierdut

Alegerea unui lider cu algoritmi undă (2)

- Fiecare proces are o **stare** dintre:
  - leader
  - lost
  - sleep *(procesul nu a executat nici o acțiune)*
  - candidate *(a executat acțiuni dar nu e sigur dacă e leader sau lost)*

- Ipoteze:
  - Sistemul este asincron *(comunicație asincronă, fără timp global, timpi arbitrari de transmisie)*
  - fiecare proces este identificat printr-un nume unic
  - Identitățile sunt extrase dintr-o **mulțime total ordonată**
  - aflarea liderului se transformă în **aflarea procesului cu cel mai mic identificator**

Algoritmi undă pentru alegerea liderului

- Algoritmii undă au cele trei proprietăți:
  - Terminare
    - Algoritmul se termină într-un număr finit de pași
  - Decizie
    - Cel puțin un nod execută un eveniment *decide*
  - Consistență
    - *Oricare* proces execută cel puțin un eveniment ce precede cauzal evenimentul *decide*

Proprietățile algoritmilor undă

Folosirea algoritmilor undă pentru alegerea liderului

- Ideea:
  - Fiecare proces are atașată o identitate în cadrul undei
    - Un nod decide atunci când un eveniment a precedat cauzal fiecare eveniment prin care procesele adaugă identitatea respectivă în vectorul de identități cunoscute
    - Procesul ce decide va cunoaște identitatea tuturor proceselor
    - Se alege procesul având identitatea cea mai mică ca nou lider
    - Se anunță *(broadcast)* identitatea liderului tuturor proceselor

Exemplu: alegere leader folosind Algoritmul Arbore

Alegerea unui lider cu algoritmi undă (3)

Alegerea liderului cu algoritm tree

- algoritmul impune ca cel puțin toate frunzele să fie inițiatori ⇒ adaugă o fază de *wakeup:*
  - inițiatorii trimit mesaje *wakeup* tuturor proceselor
  - fiecare proces folosește variabilele:
    - **ws** – boolean, asigură că fiecare proces transmite mesaje *wakeup* cel mult o dată
    - **wr** – int, contorizeaza mesajele de *wakeup* recepționate
- când un proces a primit *wakeup* prin fiecare canal, el începe algoritmul de alegere
- când un proces decide, el află identitatea liderului
- în funcție de ea procesul trece în starea leader sau lost

Algoritmul tree

```
chan ch[1:n](int id_transm, in min_id);
chan wakeup[1:n]();
process Proc[p = 1 to n]{
    bool ws = false;
    int wr = 0, r = număr_vecini_p;
    bool Vecini[1:n] = vecinii_lui_p;
    bool rec[1:n] = ([n] * false);
    int V = p;
    enum state{sleep, leader, lost};
    state stare = sleep;
    int id;
    /* aici începe faza wake-up */
    if (p este initiator) {
        ws = true;
        for [q = 1 to r st Vecini[q]]
            wakeup[q]();
    }
```

Algoritmul tree (2)

```
while (wr < număr_vecini_p){
    receive wakeup[p]();
    wr = wr + 1;
    if (NOT ws) {
        ws = true;
        for [q = 1 to n st Vecini[q]
            send wakeup[q]();
}   }

/* aici începe algoritmul tree */
while (r > 1){
    receive ch[p](q, id);
    rec[q] = true; r = r - 1;
    V = min(V, id);
}
```

Dacă este primul wakeup primit ⇒ trimit wakeup la toți vecinii

Aștept să primesc un răspuns de la N – 1 vecini și fac un minim între răspunsurile acestora.

N = numărul de vecini

Algoritmul tree (3)

```
/* de la un singur vecin, q0 nu s-a primit mesaj */
afla q0: Vecini[q0] and NOT rec[q0];
send ch[q0](p, V);
receive ch[p](q0, id);
V = min(V, id);
if (V == p)
    stare = leader
else
    stare = lost

/* informeaza celelalte procese despre decizie */
for [q = 1 to n st Vecini[q] and q<>q0]
    send ch[q](p,V);
```

Aștept rezultatul final de la ultimul nod a care nu am primit răspuns

Algoritmul tree (4)

Algoritmul tree (5)

- *Notații :*
  - **N** : Numărul de procese implicate
  - **D** : Diametru rețelei formate de procesele implicate

- *Numărul de mesaje* = O(N)
  - pe fiecare canal se trimit două mesaje *wakeup* și două *token*-uri ⇒ total = 4N - 4

- *Timp* = O(D)
  - în D pași după ce primul proces pornește algoritmul, fiecare proces a trimis mesajele *wakeup* ⇒ în D+1 pași fiecare proces a pornit unda.
  - Prima decizie se ia la D pași după pornire, ultima după alți D pași ⟹ totalul = 3D+1 pași

Alegere lider – cazul topologiilor inel

Alegerea liderului - topologie inel

Specificarea problemei:

- Se dă un aranjament circular de procese identificate prin numere distincte între ele.
- Dispunerea proceselor în inel este oarecare.
- Se cere desemnarea prin conses a procesului cu id **maxim**.
- Nu se cunoaște apriori numărul de procese.
- Nu există un control centralizat.

- Idee simplă:
  - Permitem fiecărui inițiator să trimită un token cu propria identitate pe inel
  - Nodurile nu mai au voie să inițieze după ce au primit un token

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Exemplu

- Nodurile *nu* mai au voie să inițieze un mesaj după ce au primit un token...

Algoritmul LeLann

Algoritmul LeLann

Atunci când un nod primește propriul id, a văzut deja toate celelalte id-uri

- Presupunând canale FIFO

  ⇒ Fiecare proces poate reține o listă cu identificatorii pe care i-a primit...

Algoritmul **LeLann:**

- Fiecare proces difuzează celorlalte un mesaj cu **id**-ul său
- Fiecare proces colectează numerele celorlalte procese
- Fiecare proces află maximul.
- Procesul al cărui număr este egal cu maximul devine lider.

Algoritmul LeLann

```
chan ch[1:n](token tok, int id_mare);
process Proc[p = 1 to n]{
   typedef SOP{multime identificatori};
   SOP List = {p};
   enum state (candidate, leader, lost);
   state stare = candidate;
   send ch[Urm](tok, p);
   receive ch[p](tok, q);
   while (q <> p) {
       List = List U {q};
       send ch[Urm](tok, q);
       receive ch[p](tok, q);
   }
   if (p == max(List)) stare = leader
   else state = lost
}
```

Număr de mesaje $\stackrel{?}{=} O(N^2)$

Algoritmul LeLann

Nod	ID-uri c	Status	
1	{5, 4, 3, 2, 1}	lost	5
2	{1, 5, 4, 3, 2}	lost	5
3	{2, 1, 5, 4, 3}	lost	5
4	{3, 2, 1, 5, 4}	lost	5
5	{4, 3, 2, 1, 5}	Leader	5

Algoritmul Lelann-Chang-Robert

Operating Systems

R. Stockton Gaines
Editor

An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes

Ernest Chang
University of Toronto

Rosemary Roberts
University of Waterloo

This note presents an improvement to LeLann's algorithm for finding the largest (or smallest) of a set of uniquely numbered processes arranged in a circle, in which no central controller exists and the number of processes is not known a priori. This decentralized algorithm uses a technique of selective message extinction in order to achieve an average number of message passes of order (n log n) rather than $O(n^2)$.

Key Words and Phrases: decentralized algorithms, distributed systems, operating systems

CR Categories: 4.32, 4.35, 5.25, 5.32

Introduction

Given a random circular arrangement of uniquely numbered processes where no a priori knowledge of the number of processes is known, and no central controller is assumed, we would like a method of designating by consensus a single unique process. The algorithm we propose works equally well for finding either the highest numbered or the lowest numbered process. Let us, without loss of generality, consider highest finding.

A situation in which this algorithm is important has been presented by LeLann [1]. In his example, a circle of controllers in which the control token is lost causes every controller to time out, and an election to find a new emitter for the control token is performed. LeLann's algorithm requires every controller to send a message bearing its number. Each controller thus collects, through the messages seen, the numbers of the other controllers in the circle. Every controller sorts its list, and the controller whose own number is the highest on its list is elected.

LeLann's algorithm, in a circle with n controllers, requires total messages passed proportional to n^2, written $O(n^2)$, where a message pass is a SEND of a message from a controller. This is clearly so, since each of the n controllers sends a message which is passed to all other nodes. Our algorithm requires, on the average, $O(n \log n)$ message passes.

The Algorithm

Each process is assumed to know its own number, and initially it generates a message with its own number, passing it to the left. A process receiving a message compares the number on the message with its own. If its own number is lower, the process passes the message (to its left). If its own number is higher, the process throws the message away, and if equal, it is the highest numbered process in the system.

Proposition: This algorithm detects one and only one highest number.

Argument: By the circular nature of the configuration and the consistent direction of messages, any message

Chang, E., & Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, *22*(5), 281-283.

Algoritmul Lelann-Chang-Roberts

- Mesajele sunt transmise în inel **în sensul acelor de ceasornic**.
  - Fiecare proces transmite procesului din dreapta un mesaj cu identificatorul său
  - Un proces care primește un mesaj **m** îl compară cu identificatorul propriu *(id)*:
    - **if** m **>** id -> transmite m în dreapta
    - **if** m **<** id -> elimină m
    - **if** m **=** id -> procesul curent devine lider

- ***Propoziție:***
  - Algoritmul detectează un singur cel mai mare număr

Algoritmul Lelann-Chang-Robert (2)

```
chan ch[1:n](token tok, int id_mare);

process Proc[p = 1 to n] {
    enum state(candidate, leader, lost);
    state stare = candidate;
    send ch[Urm](tok, p);
    while (state <> lider) {
        receive ch[p](tok, q);
        if (q == p) stare = leader;
        else if (q > p) {
            if (stare == candidate)
                stare = lost;
            send ch[Urm](tok, q);
        }
 }    }
```

Algoritmul Lelann-Chang-Roberts (3)

- Observații:
  - un proces **lost** ramâne în buclă pentru a pasa alte mesaje
  - doar procesul cu id maxim termină
  - pentru a debloca alte procese, liderul poate trimite un mesaj special

- *Algoritm modificat*:

  Când nu toate procesele sunt inițiatori:
  - Cel puțin un proces inițiază alegerea și se auto-marchează
  - Când un mesaj **m** ajunge la un proces nemarcat, acesta :
    - va genera un mesaj cu **id** propriu
    - se va marca
    - va continua cu prelucrarea mesajului **m**

Algoritmul Lelann-Chang-Roberts
Performanță

- Timp
  - Dacă toate procesele pornesc simultan, timpul este $\boldsymbol{O}(\boldsymbol{N})$, unde $\boldsymbol{N}$ este numărul de procese.
  - Dacă procesul cu numărul maxim pornește primul, timpul este $\boldsymbol{O}(\boldsymbol{N})$. Altfel, marcajul ia cel mult $\boldsymbol{N}-\mathbf{1}$ pași să ajungă la procesul cu id-ul maxim (și apoi $\boldsymbol{N}$ pași pentru propagare), deci timpul este $\boldsymbol{O}(\boldsymbol{N})$.

- Numărul de mesaje
  - *Cazul cel mai bun.*
    - Procesele sunt ordonate crescător în sensul acelor de ceasornic. $\mathbf{2}\boldsymbol{N}-\mathbf{1}$ mesaje.
  - *Cazul cel mai rău*.
    - Procesele sunt ordonate descrescător în sensul acelor de ceasornic**.** $\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{N}} \boldsymbol{i} = \frac{\boldsymbol{N}(\boldsymbol{N}+\mathbf{1})}{\mathbf{2}}$ mesaje.

Algoritmul Lelann-Chang-Roberts

Cazul cel mai rău

candidate

lost

leader

Algoritmul Lelann-Chang-Roberts

Cazul cel mai bun

candidate

lost

leader

Algoritmul Hirschberg-Sinclair

Technical Note
Operating Systems

R. Stockton Gaines
Editor

Decentralized Extrema-Finding in Circular Configurations of Processors

D.S. Hirschberg and J.B. Sinclair
Rice University

This note presents an efficient algorithm, requiring $O(n \log n)$ message passes, for finding the largest (or smallest) of a set of n uniquely numbered processors arranged in a circle, in which no central controller exists and the number of processors is not known a priori.

Key Words and Phrases: decentralized algorithms, distributed system, operating systems
CR Categories: 4.32, 4.35, 5.25, 5.32

Introduction

We are given n processors that are loosely coupled in a circular arrangement and work asynchronously. Each of the processors has an associated unique value (of which it alone is aware) and none of the processors has a priori knowledge of the number of processors in the circle. The problem is to designate by consensus a unique processor from the circle. The total number of data transmissions (messages passed) among the n processors pass messages in either or both directions, that these directions are distinguished, that processors can detect from which direction a received message originated, but that "left" may not mean the same to all processors. We propose an algorithm that requires $O(n \log n)$ messages in the worst case. The algorithm as given elects the processor with the highest value.

In the algorithm given below, a processor can initiate messages in both directions by a *sendboth* directive. A processor can pass a (possibly modified) message in a circular manner by a *sendpass* directive. A processor can send a responsive message back in the direction from which that processor received a message by a *sendecho* directive.

The Algorithm

```
To run for election:
    status ← "candidate"
    maxnum ← 1
    WHILE status = "candidate" DO
        sendboth ("from", myvalue, 0, maxnum)
        await both replies (but react to other messages)
 IF either reply is "no" THEN status ← "lost"
        maxnum ← 2*maxnum
    OD
On receiving message ("from", value, num, maxnum):
    IF value < myvalue THEN sendecho ("no", value)
    IF value > myvalue THEN DO
        status ← "lost"
        num ← num + 1
        IF num < maxnum THEN sendpass ("from", value, num,
        maxnum)
            ELSE sendecho ("ok", value)
    OD
    IF value = myvalue THEN status ← "won"
On receiving message ("no", value) or ("ok", value)
    IF value ≠ myvalue THEN sendpass the message
        ELSE this is a reply the processor was awaiting
```

The processors initiate messages that are passed in both directions along paths of predetermined lengths (which are successive powers of 2). Processors on the path read the message. If a processor determines, from reading the message, that it cannot win the election, then

Hirschberg, D. S., & Sinclair, J. B. (1980). Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, *23*(11), 627-628.

Algoritmul Hirschberg-Sinclair

- Complexitate $\boldsymbol{O(N\ logN)}$ în cel mai defavorabil caz.
- Lucrează pe un **inel bidirecțional**.
- Procesele pot detecta din ce direcție vine un mesaj și pot trimite un răspuns în acea direcție
- Operații de comunicare folosite de procese:
  - Un proces poate iniția mesaje în ambele direcții prin *sendboth*.
  - Un proces poate pasa un mesaj *(eventual modificat)* prin *sendpass*.
  - Un proces poate trimite un răspuns în direcția din care a primit un mesaj prin *sendecho*.

- Algoritmul opereaza in faze.
- In **faza k**, un proces initiaza mesaje care sunt transmise in ambele directii pe cai de lungime 2^k (1, 2, 4, 8 …)

- Procesele de pe o cale analizeaza mesajul
  - Daca un proces decide ca nu poate castiga alegerea el va pasa mesajul si nu va mai initia mesaje proprii
  - Daca procesul decide ca sursa nu poate castiga alegerea transmite un ecou negativ sursei
  - Ultimul proces din calea 2^k transmite un ecou pozitiv sursei (daca nu poate castiga alegerea)
- Un proces care primeste propriul mesaj castiga
  - in acest caz, calea “acopera” toate procesele
- El va trimite apoi un mesaj pentru a anunta celelalte procese ca a castigat.

Algoritmul Hirschberg-Sinclair (2)

Execuția pentru alegere:

```
stare = "candidate"
maxnum = 1
while (stare == "candidate") {
    sendboth ("from", myvalue, 0, maxnum);
    await both replies (but react to other
                        messages);
    if (either reply is "no")
        stare = "lost";
    maxnum = 2 * maxnum;
}
```

Algoritmul Hirschberg-Sinclair (3)

La recepție mesaj ("from",value,num,maxnum):

```
 if (value < myvalue)
    sendecho ("no", value)
else if (value > myvalue) {
        stare = "lost";
        num = num + 1;
        if (num < maxnum)
           sendpass ("from",value, num, maxnum)
        else
           sendecho ("ok", value)
    }
 else
     stare = "won";
```

Algoritmul Hirschberg-Sinclair (4)

```
La recepție mesaj ("no",value) sau ("ok",value):

  if (value <> myvalue) sendpass the message
  else this is a reply the processor was
  awaiting
```

- Netratat în algoritm – acțiunile unui proces neinițiator :
  - la primirea unui mesaj devine conștient de alegerea în curs
  - poate deveni *candidate* dacă identitatea are o valoare mai mare decât sursa mesajului.

Algoritmul Hirschberg-Sinclair (5)

Complexitate

Numar de mesaje

Un proces initiaza mesaje pe cai de lungime 2^i numai daca el a primit OK in faza pentru distanta 2^{i-1} (in oricare directie).

In orice grup de $2^{i-1} + 1$ procese consecutive, cel mult unul poate initia mesaje pe cai de lungime 2^i ➔

- toate cele n procese initiaza cai de lungime 1
- cel mult sup[n/2] procese vor initia cai de lungime 2,
- cel mult sup[n/3] de lungime 4,
- cel mult sup[n/5] de lungime 8 etc.

Pentru fiecare cale de lungime 2^i un proces declanseaza cel mult 4 mesaje (2 dus, 2 intors)

Vor fi $4*2^i$ transmiteri declansate de un proces

Complexitate

Numarul total de mesaje este cel mult

$$4*(l*n + 2*[n/2] + 4*[n/3] + 8*[n/5] + \ldots + 2^i*[n/(2^{i-1} + 1)] + \ldots).$$

- Fiecare termen (= lung cale * nr procese) mai mic de 2n
- Sunt cel mult 1+log n termeni
  - in log n faze, calea “acopera” n procese

→ numar mesaje O(n log n)

Complexitate in timp

Fiecare mesaj este transmis intr-o unitate de timp

Un proces trimite in paralel in cele doua sensuri

Mesaje de la procese diferite se transmit in paralel

$$Timp = 2*(1 + 2 + 4 + 8 + \ldots + 2^i + \ldots + n)$$

Daca n este putere a lui 2 → $Timp = 2*(2^{\log n+1} -1)/(2-1) = 4*n-2$

In general Timp = O(n)

Sumar

- Alegerea unui lider cu algoritmi undă
- Algoritmul tree
- Alegerea liderului – topologie inel
- Algoritmul LeLann
- Algoritmul LeLann-Chang-Robert
- Algoritmul Hirschberg-Sinclair

Algoritmi Paraleli și Distribuiți
Stabilirea Topologiei

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Difuzarea mesajelor folosind sondaje

```
typedef tip_arb int[1:N,1:N];
chan sondaj[1:N](tip_arb,tip_mesaj);
const int inițiator = indexul nodului inițiator;

process Nod[p=1 to N]{
   tip_arb arb, T;
   tip_mesaj m;
   bool top[1:N,1:N];

   if (inițiator == p) {
      calculeaza T pe baza top;
      m = mesaj_de_transmis
   } else
      receive sondaj[p](arb,m)

   for [q = 1 to N st q este fiul lui p in arb]
      send sondaj[q](arb,m);
}
```

top este o matrice cu topologia arborelui

Numărul de mesaje = $N - 1$

Difuzarea mesajelor folosind sondaje

Inițiator

Matrice de adiacență

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		T	T												
2	T			T	T										
3	T					T	T								
4		T						T	T						
5		T								T	T				
6			T									T	T		
7			T											T	T
8				T											
9				T											
10					T										
11					T										
12						T									
13						T									
14							T								
15							T								

Difuzarea mesajelor prin inundare

```
chan sondaj[1:n](tip_mesaj);
const int inițiator = indexul nodului inițiator;

process Nod[p=1 to n]{
   bool leg[1:n] = vecinii_lui_p;
   int num = numărul_vecinilor, răspunsuri = num;
   tip_mesaj m;

   if (p <> inițiator) {
      receive sondaj[p](m);
      răspunsuri = răspunsuri - 1;
   else
      m = mesaj_de_transmis;

   for [q = 1 to n st leg[q]]
      send sondaj[q](m)

   for [q = 1 to răspunsuri]
      receive sondaj[p](m)
}
```

Numărul de mesaje = $2m$

(m = numărul legăturilor)

Difuzarea mesajelor prin inundare

Stabilirea topologiei unei reţele de procese

Definiţii

- Colecție de procese **Nod(p:1..N)**
- Vecinii nodului p **leg[1:N]:**
  - $leg[q] = TRUE$ dacă q este vecin cu p,
  - altfel $leg[q] = FALSE$.
- Relație simetrică:
  - pentru p, $leg[q] = TRUE \leftrightarrow$ pentru q, $leg[p] = TRUE$

Problema

- Topologia reprezentată prin matricea de adiacenţe:
  - $top[i,j] = TRUE$, dacă i este vecin cu j
  - $top[i,j] = FALSE$, în caz contrar.
- Calculată pe baza cunoştinţelor locale
  - pentru orice p,q: $1 \leq p, q \leq N : top[p,q] = leg_p[q]$

Algoritmul pulsatiilor

Algoritmul pulsaţiilor

- Fiecare proces calculează singur topologia, folosind informaţiile provenite de la vecini
- Fiecare nod transmite vecinilor matricea sa de adiacenţă, top şi primeşte matricea de adiacenţă a fiecărui vecin
  - după un rund complet, fiecare nod va dispune de informaţii de la nodurile vecine (figura stg. pentru nodul g)
  - După două runde complete, orice nod va avea informaţii de la noduri aflate la distanţa 2 (figura dr.), etc.
  - Dupa r runde sunt completate liniile din top corespunzătoare nodurilor q aflate la o distanţă <= r

Algoritmul pulsaţiilor

Următorul predicat este satisfăcut pentru fiecare proces p:

RUND: oricare ar fi q cu 1<=q<=N: dist(p,q) <=r → top[q,*] este completat

dist(p,q) este distanţa de la p la q, adică lungimea căii celei mai scurte între cele două noduri.

Algoritmul pulsaţiilor

```
typedef tip_top bool[1:N,1:N];
typedef tip_leg bool[1:N];
chan topologie[1:N](tip_top);

process Nod[p=1 to n]{ tip_leg leg = vecinii_lui_p;
  int r = 0;
  tip_top top = ([N*N]FALSE), top_nou;
  top[p,1:N] = leg;
  /* actualizează liniile vecinilor */
  while (r < D) {
    /* transmite topologia curentă tuturor vecinilor */
    for [q = 1 to N st leg[q]]
      send topologie[q](top)
    for [q = 1 to n st leg[q]]{
      receive topologie[p](top_nou);
      top = top or top_nou;
    }
    r = r + 1;
  }
}
```

D = diametrul rețelei

Algoritmul pulsaţiilor (2)

- Neajunsuri:
  - În general, diametrul rețelei nu este cunoscut
  - Traficul inutil
- Rețea conexă
  - un nod cunoaşte întreaga topologie dacă fiecare linie ***top*** are cel puţin un element TRUE ⇒ mai execută un rund și poate termina
  - evitare blocare ⇒ un proces comunică doar cu vecinii care nu au terminat

Algoritmul pulsaţiilor (3)

```
chan topo[1:N](tip_top);
process Nod[p=1 to N] {
    tip_leg leg = vecinii_lui_p;
    tip_leg activ = leg;                    /* vecinii
   activi */
    tip_top top = ([N * N]FALSE);
    bool gata = FALSE;
    int transm;
    bool qgata;
    tip_top top_nou;

    /* actualizează vecinii lui p */
    top[p, 1:N] = leg;
```

Algoritmul pulsaţiilor (4)

```
 while (NOT gata) {
      /* transmite topologia curentă tuturor
vecinilor */
      for [q = 1 to N st leg[q]]
         send topologie[q](p, FALSE, top)
      /* recepționează topologiile de la vecini */
      for [q = 1 to N st leg[q]] {
         receive topologie[p](transm, qgata,
top_nou);
         top = top OR top_nou;
         if (qgata) activ[transm] = FALSE;
      }
      if (toate liniile din top au un element TRUE)
         gata = TRUE;
 }

 /* transmite top vecinilor activi
    în ultimul rund */
 for [q = 1 to N st activ[q]]
     send topologie[q](p,TRUE,top)
```

se executa D+1 runde

numar mesaje nu depaseste:

Numărul de mesaje = $2m(D+1)N$

m = numărul legăturilor

N = numărul de noduri

D = diametrul rețelei

Algoritmul pulsaţiilor (5)

Nodurile 3 și 4 au cel puțin un element ***true*** pe fiecare linie

Sau logic între tabela de adiacență proprie și cele primite de la vecini

$topologie[1..N]$

1	1	2	3	4	5
1			T		
2			T		
3	T	T		T	
4			T		T
5				T	

2	1	2	3	4	5
1			T		
2			T		
3	T	T		T	
4			T		T
5				T	

3	1	2	3	4	5
1			T		
2			T		
3	T	T		T	
4			T		T
5				T	

4	1	2	3	4	5
1			T		
2			T		
3	T	T		T	
4			T		T
5				T	

5	1	2	3	4	5
1			T		
2			T		
3	T	T		T	
4			T		T
5				T	

1	1	2	3	4	5
			F		

2	1	2	3	4	5
			F		

3	1	2	3	4	5
	T	T		F	

4	1	2	3	4	5
			F		T

5	1	2	3	4	5
				F	

$activ[1..N]$

Algoritmi cu mesaje de sondaj cu ecou

Algoritmi cu mesaje de sondaj cu ecou

- Ideea :
  - un **Iniţiator** colectează informaţii despre topologia locală a tuturor celorlalte noduri, determină topologia întregii reţele şi o difuzează nodurilor.
- **Cazul topologiilor arbore ⇒ sursa (initiatorul) este rădacina arborelui**
  - Două faze :
    - mesajele de sondaj se propagă de la inițiator spre frunze
    - mesajele de ecou *(care conțin topologii parțiale)* se propagă de la frunze spre inițiator.

Algoritmi cu mesaje de sondaj cu ecou (2)

```
const int inițiator = indexul nodului
inițiator;

typedef tip_leg bool[1:N];
typedef tip_top bool[1:N, 1:N];

chan sondaj[1:N](int transm);
chan ecou[1:M]  (tip_top top);
chan ecou_final (tip_top top);

process Nod[p=1 to N]{
    tip_leg leg = vecinii_lui_p;
    tip_top top = ([N * N] FALSE);
    int părinte = N + 1;
    tip_top top_nou;

    top[p, 1:n] = leg;
```

Algoritmi cu mesaje de sondaj cu ecou (3)

```
if (p <> inițiator)
   receive sondaj[p](părinte);

/* transmite sondaje celorlalte noduri vecine, copiii
lui p */
for [q = 1 to N st (leg[q] AND (q <> părinte))]
   send sondaj[q](p);


/* primește ecourile și fă reuniunea lor */
for [q = 1 to N st (leg[q] AND (q <> părinte))] {
   receive ecou[p](top_nou);
   top = top OR top_nou;
}


if (p <> inițiator)
   send ecou[părinte](top);
}
```

Algoritmi cu mesaje de sondaj cu ecou (4)

- **Cazul general**
  - după ce primeşte un prim mesaj de sondaj, un nod îl propagă tuturor celorlalţi vecini
  - nodul aşteaptă ecouri de la vecini
  - **există cicluri** $\Longrightarrow$ nodul poate primi și sondaje
    - topologia locală va fi transmisă doar ca ecou la primul sondaj
    - pentru restul sondajelor se transmite o topologie nulă
    - sondajele și ecourile trebuie recepţionate pe acelaşi canal

Algoritmi cu mesaje de sondaj cu ecou (5)

```
/* index inițiator */
const inițiator = indexul nodului inițiator;

enum fel{sondă, ecou};
typedef tip_leg bool[1:N];
typedef tip_top bool[1:N, 1:N];

chan sondă-ecou[1:N](fel tip_mesaj, int transm, tip_top
top);
chan ecou_final(tip_top top);

process Nod[p=1 to N]{
   tip_leg leg = vecinii_lui_p;
   tip_top top = ([N * N] FALSE);
   int părinte = n + 1;
   tip_top top_nou;
```

Algoritmi cu mesaje de sondaj cu ecou (6)

```
/* actualizează vecinii lui p */
top[p, 1:N] = leg;

fel k; int transm;
int nr_ecouri = 0;

if (p <> inițiator)
   receive sondă-ecou[p](k, părinte, top_nou);

/* transmite sondaje celorlalte noduri vecine, copiii lui p
*/
for [q = 1 to N st (leg[q] AND (q <> părinte))] {
   send sondă-ecou[q](sondă, p, O);  /* topologie nulă */
   nr_ecouri = nr_ecouri + 1;
}
```

Algoritmi cu mesaje de sondaj cu ecou (7)

```
   /* primește ecourile și fă reuniunea lor, sau sonde și
ignoră */
   while (nr_ecouri > 0) {
      receive sondă-ecou[p](k, transm, top_nou);

      if (k == sondă)
         send sondă-ecou[transm](ecou, p, 0);
      else if (k == ecou) {
         top = top OR top_nou;
         nr_ecouri = nr_ecouri - 1;
      }
   }

   if (p <> inițiator)
      send sondă-ecou[părinte](ecou, p, top);
```

Complexitate

Numar mesaje

Fiecare legătură (aparţinând arborelui de acoperire) corespunzătoare primei sonde poartă două mesaje (sonda şi ecoul)

Celelalte, câte patru (două sonde şi două ecouri)

Transmiterea topologiei de la iniţiator la noduri necesită alte N mesaje.

Algoritmi cu mesaje de sondaj cu ecou (8)

Inițiator

S Sondă

O Ecou vid

E Ecou

Sumar

- Difuzarea mesajelor prin inundare
- Difuzarea mesajelor folosind sondaje
- Problematica stabilirii topologiei
- Algoritmul pulsațiilor
- Algoritmi cu mesaje de sondaj cu ecou

Algoritmi Paraleli și Distribuiți
Terminarea programelor distribuite

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Terminarea programelor distribuite

- **Problema:** detecția terminării
- Algoritmi paraleli:
  - toate procesele se termină
    - cicluri infinite?
  - procese terminate sau blocate și nu există I/E în curs
  - simplu pentru un singur procesor - coada *ready* goală

- Algoritmi distribuiți:
  - fiecare proces să fie liber
    - a terminat execuția *sau*
    - se află în aşteptarea unui mesaj (**receive**)
  - pe canale să nu existe mesaje în tranzit

Procese organizate în inel

Procese organizate în inel

P(1) initiaza terminarea

termina?

inainte de a primi jeton, P(1) poate fi reactivat de un mesaj de la P(6)!

Procese organizate în inel (tehnica jetoanelor) (1)

- Inițiatorul terminării este procesul P[1]
- acțiunile lui P[1] cînd devine liber prima dată:

```
culoare[1] = albastru;
jeton = 0;
send ch[2](jeton);
```

- acțiunile lui P[i=1 to N] la recepția unui mesaj obişnuit:

```
culoare[i] = roșu;
```

Tehnica jetoanelor (2)

- acțiunile lui P[i=2 to N] la primirea unui jeton (și după ce devine liber):

```
culoare[i] = albastru;
jeton = jeton +1;    /* nesemnificativ */
send ch[(i + 1) mod n](jeton);
```

- acțiunile lui P(1) la recepția jetonului:

```
if (culoare[1] == albastru) {
  anunță terminarea; stop
}
culoare[1] = albastru;
jeton = 0;
send ch[2](jeton);
```

Tehnica jetoanelor (3)

Jeton
1

STOP

Terminarea în cazul general (tehnica jetoanelor)(1)

- topologie generală de **graf**
- **c** un ciclu care include toate arcele grafului; fie **nc** lungimea lui

Terminarea în cazul general (tehnica jetoanelor)(2)

- un jeton care se transmite după mesajele de date
- are inițial valoarea 0
- **acțiunile lui P[i=1 to N] la primirea unui mesaj uzual:**

```
culoare[i] = roșu;
```

- **acțiunile lui P[i=1 to N] la recepția jetonului:**

```
if (jeton == nc) { anunță terminarea; stop }
if (culoare[i] == roșu) { culoare[i] = albastru;
                          jeton = 0; }
else if (culoare[i] == albastru) jeton = jeton +1;

setează j la canalul următor din ciclul c;
send ch[j](jeton);
```

Terminarea în cazul general (tehnica jetoanelor)(3)

nc = 5

Jeton
4

STOP

Terminare (topologii arbore)

- Tehnica confirmarii mesajelor -

Terminarea in topologii arbore

- Când un proces frunză se termină, el anunţă părintele său.
- Când un nod oarecare al arborelui se termină, el aşteaptă notificarea terminării de la toţi fiii săi şi apoi îşi anunţă părintele.
- Notificările se propagă astfel pînă la procesul sursa, aflat în rădăcina arborelui.

Confirmarea mesajelor (1)

- Funcția de determinare a stării canalelor se poate împărți între procesele programului: fiecare proces verifică canalele pe care a trimis mesaje de date altor procese (canalele sale de ieşire).

- Pe un canal
  - se trimit **mesaje de date** și se primesc **semnale de confirmare** *sau*
  - se primesc **mesaje de date** și se trimit **semnale de confirmare**

- **Confirmarea mesajelor**: dacă pentru fiecare mesaj de date transmis pe un canal procesul primeşte de la receptor un semnal de confirmare, atunci canalul respectiv este gol.

- Presupune existența unui proces **sursă**:
  - care nu are legături de intrare *şi*
  - poate accesa orice alt proces pe o cale oarecare din graf

Confirmarea mesajelor (2)

- Exemplu:
  - Procesul sursă: 1
  - Arbore de acoperire

Algoritmul Dijkstra-Scholten (1)

Volume 11, number 1 INFORMATION PROCESSING LETTERS 29 August 1980

TERMINATION DETECTION FOR DIFFUSING COMPUTATIONS *

Edsger W. DIJKSTRA
Burroughs, 5671 AL Nuenen, The Netherlands

C.S. SCHOLTEN
Philips Research Laboratories, 5656 AA Eindhoven, The Netherlands

Received 2 June 1980

Concurrency, termination detection, distributed control, separation of concerns, message-based systems, correctness proving, diffusing computations, networks, cornets, activation tree, nondeterminacy

The following seems to capture the quintessence of a situation that is not unusual in distributed processing. Consider a finite, directed graph. If the graph contains an edge from node A to node B, we call B 'a successor of A', and A 'a predecessor of B'. We assume the existence of a node without incoming edges; this node will be called 'the environment' (because it acts as such with respect to the rest of the graph). The other nodes will be called 'the internal nodes'.

For each node its initial state will be called 'the neutral state'. A so-called 'diffusing computation' is started when the environment sends – of its own accord, so to speak – a message to one or more of its successors; it is supposed to do this just once. After reception of its first message, an internal node is free to send messages to its successors. It is this feature

proper – such that, when the diffusing computation proper has thus terminated, the fact of this completion will eventually be signalled back to the environment. Besides a node's ability to receive messages from its predecessors and to send messages to its successors, we assume each node also to be able to receive 'signals' from its successors and to send 'signals' to its predecessors; in other words, each edge is assumed to be able to accommodate two-way traffic, but only messages of the computation proper in the one direction and signals in the opposite direction. We shall impose that in the total computation – i.e. from the moment that the environment sent its messages to the rest of the graph until it has received the completion signal – each edge will have carried as many messages in the one direction as it has carried signals in the op-

Dijkstra, Edsger W., and Carel S. Scholten. "Termination detection for diffusing computations." *Information Processing Letters* 11.1 (1980): 1-4.

Algoritmul Dijkstra-Scholten (1)

- Bazat pe transmiterea unor semnale de confirmare
- Cazul unei rețele cu topologie **arborescentă**:
  - Când un proces *frunză* se termină, el îl anunță pe *părintele* său.
  - Când un *nod* oarecare al arborelui termină, el aşteaptă notificarea terminării de la toți *fiii* săi și apoi își anunță *părintele*.
  - Notificările se propagă până la procesul sursă.
- Cazul unui **graf dirijat aciclic** de procese
  - Deficitul unei legături = diferența dintre numărul de mesaje de date transmise şi numărul de semnale de confirmare primite pe acea legătură
  - Când un nod doreşte să termine:
    - aşteaptă primirea unor semnale pe legăturile de ieşire a datelor, semnale care reduc deficitele la zero (pe acele legături).
    - trimite pe fiecare legătură de intrare a datelor un număr de semnale egal cu deficitul (legăturii respective).

Algoritmul Dijkstra-Scholten (2)

- **Cazul general**: cicluri în graful proceselor
- Rezolvare: generare **arbore de acoperire**:
  - fiecare legătură generată la prima recepție (adică prima legătură de intrare, numită **prim** – de la părinte)
- Pentru terminare:
  - trimitere semnale pentru toate legăturile de intrare, mai puțin **prim**
  - recepție semnale pentru toate legăturile de ieşire
  - trimitere semnale spre **prim**.
- Schema de semnalare este separată și suplimentară față de prelucrarea propriu-zisă

Algoritmul Dijkstra-Scholten

Algoritmul Dijkstra-Scholten (3)

LORDUL

SERVITORUL

Algoritmul Dijkstra-Scholten (4)

- Comunică doar cu alți lorzi și cu propriul servitor
- Comunică cu lorzii prin scrisori (prin poștă) și cu servitorul prin semnale de comunicare (la telefon)
- Își anunță servitorul când trimite/primește o scrisoare (spunându-i și destinația/sursa)
- După ce se plictisește întreabă servitorul dacă a terminat treaba ca să poată merge la vânătoare

- Comunică doar cu alți servitori și cu propriul lord
- Comunică doar prin semnale (la telefon)
- Trimite semnale de confirmare pentru scrisorile primite și așteaptă semnale de confirmare pentru scrisorile trimise
  - Scrisoarea primită de la lordul$_i$ se confirmă servitorului$_i$
- Răspunde afirmativ doar dacă a primit și trimis toate confirmările pentru mesajele trimise și primite, altfel îl roagă pe lord să aștepte

Algoritmul Dijkstra-Scholten (5)

În continuare

- Lord = Prelucrare *(prelucrează mesaje de date)*
- Servitor = Nod *(gestionează semnalele de confirmare cu alte noduri și semnalele de comunicare cu Prelucrare)*
- Canalul poștal = canalul chm
  - chm_i este „adresa” lui $Prelucrare_i$
  - $Prelucrare_i$ primește mesaje pe chm_i și îi trimite lui $Prelucrare_j$ mesaje pe chm_j
- Canalul telefon pentru Nod = canalul chs
  - chs_i este „numărul de telefon” al lui Nod_i
  - Nod_i primește semnale pe chs_i (de la alte noduri sau de la $Prelucrare_i$) și trimite semnale de confirmare lui Nod_j pe chs_j
- Canalul telefon pentru Prelucrare = canalul stop
  - $stop_i$ este „numărul de telefon” al lui $Prelucrare_i$
  - $Prelucrare_i$ primește semnale de comunicare (da/nu) pe $stop_i$ de la Nod_i

Algoritmul Dijkstra-Scholten (6)

Algoritmul Dijkstra-Scholten (7)

```
enum fel{semnal, transmitere, receptie, terminare};
typedef struct {bool există;
                int deficit} arc;
chan chs[1:N](fel op, int id);
chan chm[1:N](int id, tip_date date);
chan stop[1:N](bool);

process Prelucrare[i=1 to N]{
   tip_date data; int id; bool term;
   inițializări;
   term = false;
   while (NOT term) {
      receive chm[i](id, data);                    /* primește mesaj */
      send chs[i](recepție, id);                   /* semnalează nod - recepție mesaj */
      realizează prelucrare
      ....
      send chm[j](i, data);                        /* trimite mesaj */
      send chs[i](transmitere, j);                 /* semnalează nod - transmitere mesaj */
      ...
      if (decizie de terminare) term = true;
   }
   term =false;
   while (NOT term) {
      send chs[i](terminare, i);                   /* anunță nod că a terminat */
      receive stop[i](term);                       /* așteaptă să termine nodul */
   }
   stop;
```

Algoritmul Dijkstra-Scholten (8)

```
process Nod[i=1 to N] {
   arc in[1:N]; /* arcele de intrare */
   int nsemnale = 0;      /* număr semnale așteptate pentru mesajele trimise */
   int prim = 0;
   var id: int; fel k;
   while (true) {
      receive chs[i](k, id);
      if (k == recepție) { if (prim == 0) prim = id fi;
                           in[id].deficit = in[id].deficit + 1; }
      else if (k == transmitere) nsemnale = nsemnale+1;
      else if (k == semnal) nsemnale = nsemnale-1;
      else if (k == terminare){
      for [j = 1 to N st (in[j].există AND (j <> prim))] /* se confirmă mesajele primite */
         while (in[j].deficit > 0) { send chs[j](semnal, i);
                                     in[j].deficit = in[i].deficit - 1;
         }
      }
      if (nsemnale <> 0) send stop[i](false);            /* mai sunt semnale de primit */
      else if (nsemnale == 0) {
         if ((i <> sursa) AND (prim <> 0)) {               /* se confirmă lui prim */
            while (in[prim].deficit > 0) {
               send chs[prim](semnal, i);
               in[prim].deficit = in[prim].deficit - 1;
            }
         }
         send stop[i](true);
      }}
   }
```

semnal de la Prelucrare: a primit un mesaj

semnal de la Prelucrare: a trimis un mesaj

semnal de confirmare de la alt nod

Prelucrare dorește terminarea

Detecția terminării folosind marcaje

DETECTING TERMINATION OF DISTRIBUTED COMPUTATIONS USING MARKERS

Jayadev Misra
Department of Computer Sciences, University of Texas, Austin, 78712

Abstract

A problem of considerable importance in designing computations by process networks, is detection of termination. We propose a very simple algorithm for termination detection in an arbitrary network using a single marker We show an application of this scheme in solving the problem of token loss detection and token regeneration in a token ring.

Introduction

We study the problem of detecting termination of computation in a network of processes. If every process in a network is *idle*, i.e. waiting for messages in order to carry out further computations, and there are no messages in transit, i.e. all messages that have been sent have been received, then no process will carry out any further computation. It is often important to detect such a situation. Multiphase algorithms [15] in which a phase is to be initiated only upon completion of the previous phase, requires termination detection of a phase. Francez, Rodeh and Sintzoff [11] suggest that it may be easier to devise a distributed algorithm in two steps: (1) design an algorithm that maintains the desired safety properties and eventually guarantees a terminating *global* state; (2) superimpose a termination detection algorithm on the basic algorithm. Deadlock detection [8] which is related to termination detection is of fundamental importance in distributed data bases. Detection of token loss in a token ring can be shown to be a termination detection problem.

We suggest an algorithm for termination detection in an arbitrary network of processes. The algorithm uses a single marker which repeatedly traverses the edges of the network until it detects termination. We make no assumption about the network structure, process behavior or message delays. Our only assumption is that a process receives (only and all) messages from another process in the order sent by the sender. Our solution is symmetric among the processes, i.e. process id's are not used in the solution. As an application of the scheme in this paper, we give a simple and efficient algorithm for detecting token loss and regenerating the token in a token ring network. The marker algorithm has also been applied [14] in *avoiding* deadlocks in distributed simulations.

There has been a considerable amount of work in termination detection. For a specific class of computations, called *diffusing computations*, Dijkstra and Scholten [2] proposed a very elegant algorithm. Their approach has been extended and applied to a variety of problems [4,5,6,7]. The major drawbacks of their approach are: (1) termination detection algorithm must be initiated whenever the diffusing computation starts and (2) the number of messages used for termination detection is equal to the number of messages in the diffusing computation itself. Our solution does not suffer from these drawbacks.

A series of papers has been published by Francez and co-workers [9,11,12] leading to a marker type algorithm. The algorithm proposed in this paper refines, simplifies and removes some of the restrictions of their approach. Independently Gouda [1] and Dijkstra [3] have proposed similar schemes, again with certain restrictions. All of these schemes use the marker to determine if a process has remained "continuously idle" over an interval of time; the idea of continuous idleness also appears in [8] for deadlock detection.

Recently Chandy and Lamport [10] have proposed a very elegant and general scheme for detecting *stable properties* of a network; a property (proposition) P is stable if it remains true once it becomes true. Clearly properties such as "network computation has terminated," and "a subset of processes are deadlocked" are all stable properties. Even though we treat the narrower problem of termination detection, our solution is simpler and more efficient for this specific problem.

Problem Description

*This work was supported by a grant from IBM.

Misra, Jayadev. "Detecting termination of distributed computations using markers." *Proceedings of the second annual ACM symposium on Principles of distributed computing*. ACM, 1983.

Detecția terminării folosind marcaje (1)

- Procesele transmit mesaje de marcaj la sfârşitului comunicației.
- Procesele primesc semnale de confirmare doar pentru marcaje.
- Pentru terminare:
  - aşteaptă marcaje pe legăturile de intrare
    - *(să termine cei de la care a primit mesaje)*
  - transmite marcaje pe legăturile de ieşire
    - *(anunță propria terminare celor cărora le-a trimis mesaje)*
  - transmite semnale pe legăturile de intrare (mai puțin **prim**)
    - *(confirmă marcajele primite)*
  - aşteaptă semnale pe legăturile ieşire
    - *(așteaptă confirmări pentru marcajele trimise)*
  - transmite semnal pentru **prim**

Detecția terminării folosind marcaje (2)

- Pentru fiecare arc se țin două atribute: ***activ*** și ***marcaj.***

- Pentru fiecare **arc de intrare**, se înregistrează dacă:
  - s-a primit un mesaj (se pune ***activ*** pe true);
  - s-a primit un marcaj (se pune ***marcaj*** pe true);
  - s-a transmis un semnal de confirmare (se pun ambele pe false).

- Pentru fiecare **arc de ieşire**, se înregistrează dacă:
  - s-a transmis un mesaj (se pune ***activ*** pe true);
  - s-a transmis un marcaj (se pune ***marcaj*** pe true);
  - s-a primit un semnal de confirmare (se pun ambele pe false).

Detecția terminării folosind marcaje (3)

```
const marcaj = mesaj-special-de-marcaj-sfârșit-date;
typedef enum {semnal, transmitere, recepție,
                recepție-marcaj, terminare} fel;
typedef struct {bool există;
                bool activ;        /*inițial false*/
                bool marcaj} arc;     /*inițial false*/
chan chs[1:N](fel op, int id);
chan chm[1:N](id:int, tip_date date);
chan stop[1:N](bool);
process Nod[i=1 to N]{
  bool term = false;
  arc in[1:N], out[1:N];
  int nsemnale = 0;/*număr semnale așteptate pe out*/
  int prim = 0;
  int nmarcaje;     /*număr marcaje așteptate pe in*/
  int id; fel k;
```

Detecția terminării folosind marcaje (4)

```
while (true) {
    receive chs[i](k, id);
    if (k == recepție) {
      if (prim == 0) prim = id;
      if (NOT in[id].activ) { in[id].activ = true;
                              nmarcaje := nmarcaje + 1;
      }
    else if (k == recepție-marcaj) {
       in[id].marcaj = true; nmarcaje = nmarcaje - 1;
       if (id == prim) term = true; }
    else if (k == transmitere) {
      if (NOT out[id].activ) { out[id].activ = true;
                               nsemnale = nsemnale + 1; }
    else if (k == semnal) {
      out[id].activ = false; out[id].marcaj = false;
      nsemnale = nsemnale - 1;
    }
```

semnal de la Prelucrare: a primit un mesaj

semnal de la Prelucrare: a primit un marcaj

semnal de la Prelucrare: a trimis un mesaj

semnal de confirmare de la alt nod

Detecția terminării folosind marcaje (5)

```
else if (k == terminare) {
      if (NOT term) send stop[i](false);
      else if (term) {
         for [j = 1 to N st (j <> prim AND out[j].activ AND
                                 NOT out[j].marcaj)] {
          send chm[j](i, marcaj); out[j].marcaj = true;
         }
         if (nmarcaje <> 0) send stop[i](false);
         else if (nmarcaje == 0) {
             for [j = 1 to n st (in[j].există AND (j<>prim)
                                          AND in[j].activ)] {
                send chs[j](semnal, i);
                in[j].activ = false; in[j].marcaj = false;
             }
             if (nsemnale <> 0) send stop[i](false);
             else if (nsemnale == 0) {
                  if ((i <> sursa) AND (prim <> 0)) {
                     send chs[prim](semnal, i);
                     in[prim].activ = false;
                     in[prim].marcaj = false;
                  }
                  send stop[i](true);
    } } } }
```

Detecția terminării folosind marcaje (6)

```
process Prelucrare[i=1 to N] {
  tip_date data; int id; bool term;
  initializari;
  term = false;
  while (NOT term) {
     receive chm[i](id, data);
     if (data==marcaj) send chs[i](receptie-marcaj,id);
     else if (data<>marcaj) {
         send chs[i](receptie, id);
         realizează prelucrare
         ....
         send chm[j](i, data);
         send chs[i](transmitere, j);
         ...
         if (decizie de terminare) term = true;
     }
  }
  term =false;
  while (NOT term) {
     send chs[i](terminare, i);
     receive stop[i](term);
  }
  stop;
```

Algoritmul lui Huang

Conferences > [1989] Proceedings. The 9th I...

Detecting termination of distributed computations by external agents

Publisher: IEEE

1 Author(s) S.-T. Huang **View All Authors**

34 Paper Citations

408 Full Text Views

- **Abstract**
- Authors
- References
- Citations
- Keywords
- Metrics

Abstract:
An algorithm is presented that defects for termination of distributed computations by an auxiliary controlling agent. The algorithm assigns a weight W, 0>

Published in: [1989] Proceedings. The 9th International Conference on Distributed Computing Systems

Date of Conference: 5-9 June 1989

Date Added to IEEE *Xplore*: 06 August 2002

Print ISBN: 0-8186-1953-8

INSPEC Accession Number: 3472185

DOI: 10.1109/ICDCS.1989.37933

Publisher: IEEE

Conference Location: Newport Beach, CA, USA, USA

First Page of the Article

Hide First Page Preview

Huang, S-T. "Detecting termination of distributed computations by external agents." *[1989] Proceedings. The 9th International Conference on Distributed Computing Systems*. IEEE, 1989.

Algoritmul lui Huang (1)

- Modelul se bazează pe următoarele:
  - procesele pot fi active sau libere (idle)
  - doar procesele active transmit mesaje
  - procesele libere pot deveni active la recepția unui *mesaj de calcul*
  - procesele active pot deveni libere în orice moment
  - **terminarea**: toate procesele sunt libere şi nu există mesaje de date în tranzit
  - topologia este un graf conex

Algoritmul lui Huang (2)

- **Ideea:**
  - Un *agent de control* are inițial ponderea 1.
  - Toate celelalte procese sunt libere și au inițial ponderea 0.
  - Calculul porneşte atunci când *agentul de control* trimite un mesaj de date unui proces.
  - Un proces liber devine activ la recepția unui mesaj de date.
- **Notații:**
  - B(DW)
    - Mesaj de date cu ponderea DW
    - Trimis doar de *agentul de control* sau de un proces activ
  - C(DW)
    - Mesaj de control cu ponderea DW
    - Trimis de un proces activ *agentului de control* când devine liber
  - W
    - Ponderea curentă a unui proces

Algoritmul lui Huang (3)

1. **Send B(DW)**:
   - Găseşte W1, W2 astfel încât W1 > 0, W2 > 0, W1 + W2 = W
   - Pune W = W1 şi transmite B(DW = W2)
2. **Receive B(DW)**:
   - W = W + DW
   - **if** liber -> devine activ **fi**
3. **Send C(DW)**:
   - send C(DW = W) *agentului de control*
   - devine liber
4. **Receive C(DW) (doar *agentul de control*)**:
   - W = W + DW
   - **if** W = 1 -> declară “terminarea” **fi**

Algoritmul lui Huang (4)

Sursă	Destinație	Mesaj
A	3	B(DW:=0.5)

TIMP
0

Algoritmul lui Huang (4)

Sursă	Destinație	Mesaj
A	2	B(DW:=0.25)
3	4	B(DW:=0.25)

TIMP
1

Algoritmul lui Huang (4)

Sursă	Destinație	Mesaj
4	1	B(DW:=0.125)
3	A	C(DW:=0.25)
2	5	B(DW:=0.125)

TIMP
2

Algoritmul lui Huang (4)

Sursă	Destinație	Mesaj
4	1	B(DW:=0.0625)
2	A	C(DW:=0.125)
1	3	B(DW:=0.0625)
5	1	B(DW:=0.0625)

TIMP
3

Algoritmul lui Huang (4)

Sursă	Destinație	Mesaj
1	A	C(DW:=0.1875)
3	A	C(DW:=0.0625)
4	A	C(DW:=0.0625)
5	A	C(DW:=0.0625)

TIMP
4

Algoritmul lui Huang (4)

Sursă	Destinație	Mesaj

TIMP
5

Sumar

- Problematica terminării programelor distribuite
- Cazul proceselor organizate în inel
- Tehnica jetoanelor pentru cazul general
- Confirmarea mesajelor
- Algoritmul Dijkstra-Scholten
- Detecția terminării folosind marcaje
- Algoritmul lui Huang

Algoritmi Paraleli și Distribuiți
Algoritmi pentru sisteme tolerante la defecte

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Fiabilitate la nivelul proceselor

- De obicei mascam defectarea proceselor prin replicare
- Organizam procesele in grupuri, un mesaj trimis grupului fiind livrat tuturor membrilor acelui grup
- Daca un membru se defecteaza, un altul ii ia locul

Comunicatie: diferenta intre Egalitate (a) vs. Ierarhie (b)

La cursul practic... failure models

- Pana in acest punct, modelele presupuse pentru sisteme distribuite imperfecte au constat din:

1. **Fail-recover faults**
   - Un nod se defecteaza si nu mai raspunde la mesaje
   - Alte noduri pot detecta defectul prin timeout-uri: “if 127.0.0.10 doesn’t respond in 100ms, it’s down”
   - … dar nodurile respective pot reveni in operatie
2. **Delayed / lost messages**
   - Mesajele intre noduri pot fi arbitrar intarziate sau pierdute, dar **nu** corupte

- Ne putem folosi imaginatia. Ce facem daca…
  - reteaua incepe sa corupa mesaje?
  - un atacator intercepteaza si modifica date in trafic?
  - unele noduri sunt… malitioase?

Toleranta la defecte bizantine

- **Defectul bizantin:** Orice fel de defect la care ne putem gandi, precum…
  - coruperea de mesaje
  - noduri ce se comporta diferit functie de **context** (ex., comunica unor noduri informatii diferite)
  - nodurile **conspira** intre ele pentru a cauza distrugeri

The Saddest Moment

JAMES MICKENS

James Mickens is a researcher in the Distributed Systems group at Microsoft's Redmond lab. His current research focuses on Web applications, with an emphasis on the design of JavaScript frameworks that allow developers to diagnose and fix bugs in widely deployed web applications. James also works on fast, scalable storage systems for datacenters. James received his PhD in computer science from the University of Michigan, and a bachelor's degree in computer science from Georgia Tech.
mickens@microsoft.com

Whenever I go to a conference and I discover that there will be a presentation about Byzantine fault tolerance, I always feel an immediate, unshakable sense of sadness, kind of like when you realize that bad things can happen to good people, or that Keanu Reeves will almost certainly make more money than you over arbitrary time scales. Watching a presentation on Byzantine fault tolerance is similar to watching a foreign film from a depressing nation that used to be controlled by the Soviets—the only difference is that computers and networks are constantly failing instead of young Kapruskin being unable to reunite with the girl he fell in love with while he was working in a coal mine beneath an orphanage that was atop a prison that was inside the abstract concept of World War II. "How can you make a reliable computer service?" the presenter will ask in an innocent voice before continuing, "It may be difficult if you can't trust anything and the entire concept of happiness is a lie designed by unseen overlords of endless deceptive power." The presenter never explicitly says that last part, but everybody understands what's happening. Making distributed systems reliable is inherently impossible; we cling to Byzantine fault tolerance like Charlton Heston clings to his guns, hoping that a series of complex software protocols will somehow protect us from the oncoming storm of furious apes who have somehow learned how to wear pants and maliciously tamper with our network packets.

Figure 1: Typical Figure 2 from Byzantine fault paper: Our network protocol

How can you make a reliable computer service? […] It may be difficult if you can't trust anything and the entire concept of happiness is a lie designed by unseen overlords of endless deceptive power.

The Saddest Moment, James Mickens

Agreement algorithms

- In sistemele distribuite, avem diverse cazuri cand procesele au nevoie sa ajunga la un acord :
  - Leader election
  - Commit
  - Synchronize

- **Distributed Agreement**: *toate procesele non-faulty ajung la consens intr-un numar finit de pasi*
- Procese perfecte, canale faulty: problema celor doua armate
- Procese faulty, canale perfecte: problema generalilor Bizantini

Distributed consensus

- Un numar de noduri ajung la nu **acord** despre o **valoare**
- Orice nod se poate **defecta** sau **recupera/reveni** in orice moment de timp

How does RAFT work?

RAFT IS A CONSENSUS ALGORITHM

a process for getting multiple machines to "agree"!

- RAFT doesn't stand for anything. It's a bunch of logs tied together.

used in many projects in the real world - ex. etcd

WHY? Any part of a distributed system can fail! Consensus across several nodes gives us backups of data.

WHAT ABOUT NETWORK PARTITIONS?!

In a partition, if the leader falls on the smaller side, a new leader is elected within the majority side. When the partition ends, messages received by the minority side are discarded, and those nodes converge their state to match the majority's.

WHEN A NEW WRITE HAPPENS:

Any node can accept writes!

Message forwarded to leader node

Message is queued...

COPY me!

ACK!

ACK

...while followers copy it to their logs

When a quorum have copied the message, leader "commits" the new data & will now return it on read

FISH :

Followers also commit so they return new data!

@deniseyu21 with @mt165

Algoritmul *raft* pentru consens

- Raft este un algoritm dovedit a functiona corect, si care asigura *strong consistency*

- Exista si alti algoritmi pentru stabilirea consensului, cel mai renumit fiind *paxos*

Replicare master – slave

- Unul dintre noduri este ales *master* (sau *leader*), ceilalti sunt *followers*

Replicare master – slave

- Furnizeaza o **ordonare globala** scrierilor provenind din partea clientilor – putem reconcilia scrieri concurente
- Clientii ce contacteaza noduri follower sunt **redirectionati**

Defectarea liderului

- Liderul trimite periodic mesaje heartbeat (alive) tuturor nodurilor follower
- Daca un nod follower nu mai primeste mesaje heartbeat, presupune ca leaderul s-a defectat si initiaza o **alegere**
- O alegere este castigata daca un nod primeste voturi pozitive din partea unui cluster **majoritar** de noduri

Mesaje Heartbeat

Alegere lider

O alegere finalizata cu succes: 2 din 3 noduri sunt de acord cu noul lider

Aducerea nodurilor outdated la zi

- Unul din follower iese offline pentru 10 minute – cum il re-aducem intr-o stare consistenta?
- Inregistram *toate scrierile* intr-un “indexed log", pe care il replicam
  - Contine momentul cand o inregistrare a fost scrisa

Index	Term	Contents
0	1	SET food pizza
1	1	SET language c++
2	1	SET food pickles
3	5	SET answer_to_life 42
…		….

Replicarea Logului

Aplicarea intrarilor din log

- Intrarile din Log de obicei conduc la *modificari* ale unei baze de date (cunoscuta si ca *state machine*)
- La un moment dat, aceste modificari trebuie *aplicate* in state machine

Replicarea State Machine

- Nu e sigur sa aplicam o intrare imediat, din moment ce exista situatii (rare) cand un alt lider (subsecvent) poate cere **roll back**
- Intrarile din Log sunt considerate a fi **comise** doar cand o majoritate de noduri le-au aplicat
- Odata ce o intrare a fost comisa, se garanteaza ca nu va mai fi retrasa si poate fi *aplicata* sigur in state machine
- Doar nodurile care au *toate intrarile comise* pot participa ca posibili lideri in alegeri

... Ne reintoarcem la defecte...

- Raft nu este considerat a fi un protocol “byzantine fault tolerant” – diverse oportunitati pentru un nod malitios sa cauzeze probleme
- Un lider byzantine poate modifica cererile clientilor

Probleme cu raft…

- Un nod bizantin poate f. usor sa pretinda ca este lider, si toate celelalte noduri il vor crede
- Nu trebuie sa **dovedeasca** ca a castigat alegerile!

Probleme cu raft…

... sau in mod continuu sa declanseze alegeri, ne-permitand aparitia vreunui alt lider– clusterizarea devine *useless* (my way or the highway)

Diverse cause ale defectelor bizantine

- Cauze benigne
  - **Coruperea** unor pachete pe retea, a datelor pe disc
  - Un **memory bitflip** complet aleatory cauzeaza un nod sa se comporte imprevizibil

```
bool isLeader;   // gets hit by cosmic ray
```

  - Un bug rar in cod (as if) cauzeaza un nod sa nu respecte protocolul

- Cauze malitioas
  - Un atacator modifica *traficul inter-node*
  - Un atacator *preia abuziv* un nod

Posibile contra-masuri

- **Semnaturi criptografice, checksums**: pot ajuta la **autentificarea** si verificarea **integritatii** mesajelor si datelor

- **ECC (Error-Correcting Code) memory:** poate detecta si uneori repara eventuale erori la nivelul memoriei

- E extrem de dificil sa asiguram protectia impotriva *tuturor* tipurilor de defecte… unele **protocoale** exista
  - dar toate au limitarile lor

Reality check

- Este necesar un asemenea nivel de paranoia?
- Cand rulezi 50.000 de servere in productie, ce se intampla cand unul “goes byzantine”?
- Ar putea acesta sa distruga functionarea sistemului ca ansamblu? poate…
- O idee buna sa asiguram protectia macar impotriva *unor defecte bizantine*

Rezolvarea prin replicare...

- Replicam un proces si replicile la nivelul grupului intr-un singur grup
- Cate replici vom crea?
- Tipuri de defecte ale proceselor
  - ***crash***: procesul devine nefuncțional
  - ***byzantine***: procesul trimite mesaje cu un conținut arbitrar
- Un sistem se numeste *k fault-tolerant* daca poate supravietui si functiona chiar si cu k procese defecte
  - Pentru defectele de tip crash (a faulty process halts, but is working correctly until it halts)
    - k+1 replici
  - Pentru defectele Byzantine (a faulty process may produce arbitrary responses at arbitrary times)
    - 2k+1 replici

Problema celor doua armate

Paradoxul celor doi generali

- Doua armate au inconjurat un oras
- Generalii acestora trebuie sa decida **impreuna** daca sa atace sau sa mearga in retragere
- Ei comunica prin mesageri, care insa pot fi **interceptati**
- Ambii generali trebuie sa ajunga la **aceeasi** decizie

Destul de usor, nu?

Probabil ca nu…

Paradoxul celor doi generali

- Nu exista un protocol care sa garanteze ca ambii generali (noduri) sunt 100% siguri asupra deciziei pe care celalalt o va lua
  - Exista demonstratii stiintifice
- Dupa **500 confirmari**, ambii pot fi destul de siguri ca amandoi vor ataca (sau, vor lua aceeasi decizie)
- Dar “destul de siguri” nu **garanteaza 100% siguranta**

Varianta simplificata a demonstratiei:

- Presupunem ca exista un protocol care asigura ca dupa schimbul a **N mesaje**, se garanteaza siguranta in luarea deciziei
- Al N-lea mesaj ar putea fi pierdut… deci, doar primele **N-1 mesaje** *inseamna ca au fost suficiente* pentru a garanta siguranta in luarea deciziei
- **Deci,** trebuie sa existe un protocol care schimba doar **N-1 mesaje**
- Concluzia: exista un protocol care schimba in final **0 mesaje**

- Exista protocoale ce mitigheaza nesiguranta data de trimiterea mesajelor, folosind reincercari, ACK-uri si timeouturi
  - TCP …

Problema generarilor bizantini

Problema generalilor Bizantini

- Generalizare a problemei celor doi generali
- N generali (noduri) trebuie sa decida asupra unei decizii de atac sau retragere pe baza *votului majoritatii*
- Unii generali pot fi in secret tradatori, si vor incerca sa *manipeze* votul…
- **Scop:** atingerea consensului intre noduri oneste
  - #1 wants to **attack**
  - #3 wants to **retreat**

  - #1 receives **retreat** votes from #2 and #3
  - #3 receives **attack** votes from #1 and #2
  - Result: #3 attacks *alone*, #1 retreats

Problema generalilor bizantini

- Fie v(i) – informația comunicată de către Generalul cu numărul *i*
- Fiecare General folosește o metodă pentru a combina valorile v(1), v(2), ..., v(n) într-un *plan de acțiune*

- Dacă decizia care poate fi luată este *atac* sau *retragere*, v(i) este opțiunea Generalului *i* dintre cele două variante, iar decizia finală pentru fiecare General se bazează pe votul majoritar (dintre cele *n* valori)

- Generalii își comunică unii altora valorile v(i)
- Generalii trădători pot trimite valori diferite celorlalți Generali
- Fiecare General ia decizia finală bazată pe aceeași mulțime v(1), v(2), ..., v(n)

Problema generalilor bizantini

- Restrângem abordarea problemei la modul în care *un* General își trimite valoarea celorlalți Generali
- Termenii problemei se schimbă: un General comandant trimite ordinul Locotenenților, obținându-se astfel problema:

- ***Problema Generalilor Bizantini***:
  - Un General comandant trebuie să trimită un ordin celor n – 1 Locotenenți astfel încât:
  - IC_1: Toți Locotenenții loiali se supun aceluiași ordin
  - IC_2: Dacă Generalul comandant este loial, atunci fiecare Locotenent loial se supune ordinului Generalului comandant

- IC_1, IC_2 – *interactive consistency conditions*
- Se observă că, pentru un Comandat loial, IC_1 rezultă din IC_2; totuși, Comandantul nu este în mod necesar loial

Problema generalilor bizantini

- Evenimentele sunt următoarele:
  - Comandantul trimite ordinul tuturor Locotenenților
  - Un Locotenent trimite celorlalți Locotenenți mesajul primit de la Comandant
  - După primirea mesajului de la Comandant și a copiilor de la ceilalți Locotenenți, un locotenent decide *prin vot majoritar* ce decizie va lua

Problema generalilor bizantini
Exemplul 1

- $Locotenent_3$ este trădător
- $X \in \{atac, retragere\}$
- Oricare ar fi mesajul transmis de trădător, cei doi Locotenenți loiali vor lua aceeași decizie (*atac*)

Problema generalilor bizantini
Exemplul 2

- $Comandantul$ este trădător
- $X \in \{atac, retragere\}$
- Oricare ar fi mesajul transmis de trădător, toți Locotenenții vor lua aceeași decizie X

Problema generalilor bizantini

- Dificultatea problemei constă în faptul că:
  - Dacă Generalii (Locotenenții) pot trimite doar mesaje orale, atunci nu există soluție decât pentru cazul în care *2/3 din Generali sunt loiali*
- Un mesaj *oral* este aflat complet sub controlul emițătorului, deci un trădător poate trimite orice mesaj
- Mesajul trimis este *atac* / *retragere*
- Următoarele exemple: nu există soluție pentru 3 Generali, din care un trădător

Problema generalilor bizantini
Exemplul 3

- $Locotenent_2$ este trădător

Problema generalilor bizantini
Exemplul 4

- $Comandantul$ este trădător

Bitcoin / blockchain consensus

Ce este bitcoin?

- O valuta ce nu este influentata de **nicio** autoritate centrala, bazata pe criptografie si nu pe increderea intre oameni

- O retea distribuita de noduri ce proceseaza tranzactii, *puternic rezilienta* la atacuri bizantine
  - **Oricine** poate rula propriul nod bitcoin
  - … ceea ce inseamna ca **nimeni** nu e considerat de incredere

Scurt istoric

- **Noiembrie 2008:** “Satoshi Nakamoto” publica lucrarea despre un “peer-to-peer electronic cash system”
- **Ianuarie 2009:** Reteaua bitcoin apare online, cu “Satoshi” fiind cel care mineaza primul block

Adrese Bitcoin

- Oricine poate genera o adresa pornind de la o pereche public-private key

Public: 1DwAdnHZ3w2Ecww6SPZZaGNMXVuWbdGZNY

Secret: L3H8Yz7YPshSHw5yjTH72cYy12UnUerFnzd8wjEDyhsYXSbhWU5q

- Oricine are o adresa publica poate **trimite** bitcoins la acea adresa
- Pentru a-i **cheltui**, trebuie sa stiti partea secreta a cheii

O tranzactie bitcoin (super simplificata)

- Intrari: iesirile unor tranzactii *anterioare*
  - Iesirea #2 tranzactiei *3b96bb7e197e...*
  - Iesirea #1 tranzactiei *e1afd89295b68…*
  - Iesirea #6 tranzactiei *e79fc1dad370e…*

- Iesiri
  - 0.1 BTC catre adresa **1HmxmBAX413yGY2LDoEN8FHBok61aT4w2d**
  - 0.2 BTC catre adresa **1Hozk3UFZDsGGd7PENAVgy3ouFHzvsCFJ7**

- Semnaturi ce dovedesc **ownership-ul** adreselor de intrare

E suficient? Nu!

- Vanzatorul trebuie sa poata verifica ca tranzactia este **valida**
- **Sursele de intrare** pot sa **nu existe**, sau cumparatorul poate ca nu are **dreptul de a le cheltui** (invalid signature)

Blockchain-ul bitcoin (super simplificat)

- Un log replicat al *tuturor* tranzactiilor bitcoin de la inceputul timpului
- Fiecare block contine o lista a tranzactiilor semnate, dar si hash-ul *SHA256* al unui bloc *anterior*

Block #0 → Block #1 → …

SHA256:
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Este de ajuns? Inca nu

- Noua tranzactie trebuie sa ajunga in **blockchain-ul** tuturor nodurilor bitcoin
- **Toate nodurile** din retea trebuie sa fie constiente de tranzactie, altfel vanzatorul **nu poate folosi** monedele

Transaction broadcasting

"I declare I want to send 0.1 BTC from *12vEsiYhQ3iUJK2Mpt5eLq2cgxewQLw4tr* to *16yZiBEmG3AKSUkHm3oHo3PUMPUWUe72tv*, and here's a signature to prove I own these bitcoins. Please add my transaction to your blockchains"

Este de ajuns? Poate…

- Nu este sufficient ca **un singur nod** confirma tranzactia
- Vanzatorul trebuie sa fie sigur ca **intreaga retea** o confirma
  - daca nu, nu va mai putea cheltui monedele in viitor

Reaching blockchain consensus

- Nodurile Bitcoin vor primi tranzactia intr-o ordine posibil **diferita**
- **Este critic** ca nodurile sa ajunga la un *acord* asupra ordinii
  - Si nu, eventual consistency nu mai functioneaza aici
- **Double-spending**: Doua tranzactii ce consuma aceiasi sursa de intrare (cineva vinde pe aceeasi bani de doua ori) – doar *prima* tranzactie poate fi valida
- Toate nodurile trebuie sa ajunga la consens asupra *tranzactiei care e prima*, respectiv care trebuie eventual invalidata

Blockchain forks

- Prin definitie, “adevaratul” blockchain este cel mai lung
- Un nod onest notificat de un **blockchain mai lung** trebuie sa comute pe folosirea acestuia **imediat**

- Tranzactiile din blocurile ramase “orfane” sunt retrase/anulate

Blockchain forks (2)

- “Longest blockchain wins”: toate nodurile bitcoin converg rapid catre un blockchain comun

- Cu cat sunt mai multe blocuri adaugate pe chain, cu *atat mai improbabil* e ca tranzactiile noastre sa fie retrase

- In plus, extra blocuri servesc unor *confirmari suplimentare* prin care nodurile bitcoin confirma tranzactia noastra
  - Aceste blocuri confirma **validitatea** tranzactiei
  - si confirma ca tranzactia face parte din **cel mai lung** blockchain cunoscut de respectivele noduri

E suficient? Aproape…

- Vanzatorul poate confirma criptografic ca sunt alte 15 blocuri valide adaugate dupa cel cu tranzactia curenta
  - aproape sigur ca orice fork de aici incolo va contine tranzactia… **sau nu?**

Un atac bizantin

Ceva lipseste… urmatorul atac ar putea avea success:

1. Publish a transaction, which is added to block *#190*
2. Wait for *20* confirmations
3. Seller is reasonably sure the network reached consensus on this transaction
4. Seller gives you bread
5. Publish a **new blockchain**, forking from node *#189* with *30* empty blocks after it
6. Network switches to your blockchain, since it's **longer**
7. You get to keep **both** the bread and the coins

Proof of work

- Solutia la scenariul anterior: *sa facem costisitoare* operatie de adaugare a unor noi blocuri

- Orice hash SHA256 al unui bloc trebuie sa inceapa cu un anuimit numar de zero-uri, altfel nodurile oneste il vor *rejecta*

- Singura metoda pentru a produce un bloc consta in **forta bruta**
  - Formatul de bloc contine un contor exact pentru acest scop, ce influenteaza rezultatul aplicarii SHA256
  - Acesta trebuie incrementat pana cand reusim sa generam un SHA256 valid
  - In alte variante de monede virtuale exista si alte versiuni ale Proof-of-Work

Atacuri 51%

- Nodurile Bitcoin constant incearca adaugarea de noi blocuri – exista **recompense** pentru cele care reusesc

- Un atacator trebuie sa le poata **intrece** pentru a produce blocuri mai rapid decat **restul retelei combinat**

- Acest lucru este posibil doar de catre atacatori ce detin peste *50%* din totalul **puterii computationale de generare a SHA256** a intregii retele
  - Un astfel de atac devine extrem de scump

“SHA256 computation arms race”

- Cele mai bune **CPU-uri**: *10 – 100* milioane de hash-uri per sec
- Cele mai bune **GPU-uri**: *100 – 1000* milioane de hash-uri per sec

Hardware specializat pe minare de bitcoin: **FPGA**, **ASIC**

- Un **ASIC** lansat in 2012: *60,000* M hashes / s
- Un **ASIC** astazi (2019): *14,000,000* M hashes / s

Antminer S9

Examen

Condiții examen (primavara)

- Timp de lucru: 1 ora si 20 minute
- *8 intrebari, fiecare întrebare valorează 0.5 puncte*
- *pot fi date întrebări din oricare din cursurile teoretice sau practice predate la aceasta materie.*
- Fără documentație (*closed book*)

Exemple de subiecte

1. Definiți conceptul de “Split binary semaphore”. Explicați cum se realizează tehnica de pasare a ștafetei.
2. Reprezentați grafic modelul Replicated Workers. Care este condiția de terminare?
3. Corectați pseudocodul de mai jos *(parallel Scan)* pentru a funcționa întotdeauna corect in cazul a n procese (thread-uri) executate in paralel pe o arhitectura de tipul *MIMD*.

```
int a[1:n];
process parallel_Scan[k=1 to n] {
    for (j = 1; j < sup(log2 n); j++)
        if (k - 2^(j-1) >= 1)
         a[k] = a[k-2^(j-1)] ⊕ a[k];
  }
```

⊕ = operație ce se aplica pe elementele date ca input

4. Definiți timpul total de execuție in *modelul Foster* (distribuit) si cum se calculează acesta (T = ?) Explicați pe scurt fiecare din cele 3 subcomponente ale acestuia.
5. Considerați o implementare de semafor distribuit cu doua tipuri de procese: procesele ce apelează operațiile semaforului (P si V) - **Utiliz(i)**, si cele care implementează logica semaforului distribuit - **Ajutor(i).** Scrieți pseudocodul pentru procesele *Ajutor* in cazul recepționarii unui mesaj:

```
while (true) {
        receive opsem[i](transm, k, ts);   // transm = ID sender,  k = felul mesajului,
                                           // ts = timestamp-ul de trimitere
        // TODO
```

Condiții examen (toamna)

- Timp de lucru: 2 ore
- Fără documentație (*closed book*)

Un model de subiect de examen la APD

Subiectul 1. (2 puncte) Ceasuri logice vectoriale. Tratați:

1.1 Conceptul general (0.7 p)

Motivatia folosirii ceasurilor logice: de ce este nevoie de ele (ce aduce nou față de alte metode).

Principiul ceasurilor logice vectoriale.

1.2 Pentru procesele din figură, precizați vectorii de timp asociați evenimentelor specificate. Axele orizontale reprezintă timpul. (0.3 p)

1.3 Aplicație - ordonarea cauzală *multicast*. (1 p)

Specificarea problemei.

Soluția – cu descrierea acțiunilor la trimiterea, recepția și livrarea mesajelor.

Un model de subiect de examen la APD

Subiectul 2. (1 punct)

Propuneti un algoritm de *tip heartbeat* pentru calculul sumei a n^2 valori de tip întreg. Folosiți n^2 procese dispuse într-o grilă neperiodică.
Deci, fiecare proces poate comunica cu vecinii de la *est*, *vest*, *nord* și *sud*.
Fiecare proces deține, inițial, o singură valoare *v*.
În final, fiecare proces trebuie să dețină valoarea sumei.

Calculați complexitatea soluției oferite.

Un model de subiect de examen la APD

Subiectul 3. (1 punct) Tratați unul din subiectele urmatoare, la alegere:

3.1. Cautarea paralelă

Prezentarea problemei (0.2 p)

Descrierea algoritmului (0.4 p)

Analiza complexității (0.4 p)

3.2. Problema cititorilor și scriitorilor

Prezentarea problemei (0.2 p)

Descrierea algoritmului (0.4 p)

Politici cu prioritate asupra cititorilor și asupra scriitorilor (0.4 p)

Detalii finale

- Punctajele vor fi afișate pe site
- Nu aveți voie să veniți cu alta grupă decât cu aprobarea titularului de curs (pentru probleme speciale, vă rog să mă contactați)
- Pentru nelămuriri, documentație, explicații suplimentare:
  - open office: *PRECIS 605*
  - ciprian.dobre@cs.pub.ro
- …

Baftă în sesiune ☺ !

Si nu uitati ca acceptam studenti la practica de vara!
(nu e rau sa porniti lucrarea de licenta inca din vara)

Distributed Commit

Distributed Commit

- **Scop**: fie **toti** membrii unui grup decid asupra efectuarii unei operatii, fie **niciunul** nu efectueaza operatia
- **Atomic transaction**: o tranzactie ce se efectueaza cu totul sau deloc
- Defecte:
  - Cele de tip Crash ce pot fi recuperate
  - Defecte de comunicatie detectabile prin timeouturi
- Observatii:
  - Commit-ul necesita un set de procese sa ajunga la un acord…
  - …similar problemei generalilor Bizantini …
  - … dar solutia este mult mai simpla deoarece avem ipoteze/constrangeri mai puternice

O tranzactie bancara distribuita

One-phase Commit

- Protocolul one-phase commit
  - Un site este desemnat ca si coordinator
  - Coordinatorul instiinteaza toate celelalte procese daca efectueaza sau nu local o opearatie
  - Schema aceasta insa nu este fault tolerant

Transaction Processing (1)

Transaction Processing (2)

Two Phase Commit (2PC)

Two-Phase Commit

a) Masina de stari finite pentru coordonator, in 2PC.

b) Masina de stari finite pentru un participant.

Two-Phase Commit

actions by coordinator:

```
write START _2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast  GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator
votes COMMIT{
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT  to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Two-Phase Commit

actions by participant:

```
write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send  VOTE ABORT to coordinator;
}
```

Two-Phase Commit

actions for handling decision requests: /* executed by separate thread */

```
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip;  /* participant remains blocked */
```

Steps taken by participant process for handling incoming decision requests.

Algoritmi Paraleli și Distribuiți
Pthread + Primitive sincronizare

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

POSIX threads

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{

}

pthread_join(thread, NULL);
```

Compilare pthread

gcc –o executabil cod.c –lpthread –lrt

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

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{

}

pthread_join(thread, NULL);
```

Acest element reprezintă thread-ul.
Poate fi considerat handle

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
}
```

Aici am putea să facem recomandări sistemului de operare. Gen să folosească anumite core-uri.

```
pthread_join(thread, NULL);
```

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
}

pthread_join(thread, NULL);
```

Când se crează thread-ul, acesta va începe la această funcție.

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
}

pthread_join(thread, NULL);
```

Așa trimitem date thread-ului

Pthread

```
pthread_t thread;
pthread_create(&thread, NULL, threadFunction, arg);

void * threadFunction(void* arg)
{
}

pthread_join(thread, NULL);
```

Astfel se pot extrage informații din thread

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
   a=a+1;
   return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
     pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
      pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int i, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
 }
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
  a=a+1;
  return NULL;
}
```

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
void* threadFunction(void *arg)
{
   a=a+1;
   return NULL;
 }
```

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
  }
  for(i = 0; i < P; i++) {
    pthread_join(tid[i], NULL);
  }
  return 0;
}
```

Pthread

```
int main(int argc, char **argv)
{
  int I, P=2;
  pthread_t tid[P];
  for(i = 0; i < P; i++) {
    pthread_create(&(tid[i]), NULL, threadFunction, NULL);
   }
   for(i = 0; i < P; i++) {
     pthread_join(tid[i], NULL);
   }
   return 0;
}
```

Mutex Pthread

ÎN MAIN

Înainte de a porni thread-urile

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Mutex Pthread

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Poate fi folosit să anunțe că acest mutex e împărțit mai multor procese

Mutex Pthread

pthread_mutex_lock(&**mutex**);

load(a, eax)

eax = eax + 2

write(a, eax)

pthread_mutex_unlock(&**mutex**);

Mutex Pthread

ÎN MAIN

După ce au terminat thread-urile

pthread_mutex_destroy(&**mutex**);

Semaphore

ÎN MAIN

Înainte de a porni thread-urile

```
sem_t semaphore;
int semaphore_value= 4;
sem_init(& semaphore, 0, semaphore_value);
```

Semaphore

```
sem_wait(&semaphore);
```

P() sau Proberen
 - Dijsktra

Semaphore

```
sem_wait(&semaphore);
```

Semaphore

```
sem_wait(&semaphore);
```

Semaphore

```
sem_wait(&semaphore);
```

Semaphore

Nu contează că doua thread-uri au ajuns simultan la semafor, acesta este protejat, la fel ca un mutex.

```
sem_wait(&semaphore);
```

Semaphore

```
sem_wait(&semaphore);
```

Semaphore

```
sem_wait(&semaphore);
```

Semaphore

```
sem_wait(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```

```
sem_post(&semaphore);
sem_post(&semaphore);
```

V() sau Verogen
- Dijsktra

Semaphore – Signaling

```
sem_wait(&semaphore);
```

```
sem_post(&semaphore);
```

```
sem_post(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```

-1

```
sem_post(&semaphore);
```

```
sem_post(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```

```
sem_post(&semaphore);
sem_post(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```

```
sem_post(&semaphore);
sem_post(&semaphore);
```

Semaphore – Signaling

```
sem_wait(&semaphore);
```

```
sem_post(&semaphore);
sem_post(&semaphore);
```

Semaphore

ÎN MAIN

După ce au terminat thread-urile

sem_destroy(& **semaphore**);

Barrier

Barrier

ÎN MAIN

Înainte de a porni thread-urile

```
pthread_barrier_t barrier;
int num_threads = 5;
pthread_barrier_init(&barrier, NULL, num_threads);
```

Barrier

pthread_barrier_wait(&barrier);

5

Barrier

pthread_barrier_wait(&**barrier**);

5

Pentru toate thread-urile codul acesta
Este executat înainte de codul acesta

Barrier

pthread_barrier_wait(&barrier);

5

Barrier

pthread_barrier_wait(&**barrier**);

5

Barrier

pthread_barrier_wait(&**barrier**);

5

Barrier

pthread_barrier_wait(&**barrier**);

3

Barrier

pthread_barrier_wait(&**barrier**);

2

Barrier

pthread_barrier_wait(&**barrier**);

1

Barrier

pthread_barrier_wait(&**barrier**);

1

Barrier

pthread_barrier_wait(&**barrier**);

1

Barrier

pthread_barrier_wait(&**barrier**);

1

Barrier

pthread_barrier_wait(&**barrier**);

0

Barrier

pthread_barrier_wait(&**barrier**);

0

Barrier

Cum știe o barieră când să se reseteze?

O soluție ar fi:
Reusable Barrier in
[The Little Book of Semaphores
By Allen B. Downey]

pthread_barrier_wait(&**barrier**);

5

Barrier

ÎN MAIN

După ce au terminat thread-urile

```
pthread_barrier_destroy(&barrier);
```

Unele probleme nu au soluție paralelă

Calculating the hash of a hash of a hash …

of a string.

Deep First Search

Huffman decoding

Outer loops of most simulations

P complete problems

Paralelizarea prin împărțirea muncii

Sunt o serie de probleme care sunt extrem de ușor paralelizabile.

Embarrassingly parallel

Probleme ”Embarrassingly parallel”

Multiplicare unui vector cu un scalar

9	6	9	4	2	7	6	5	6	1

* 3

27	18	27	12	6	21	18	15	18	3

Probleme ”Embarrassingly parallel”

Toate calculele pot fi efectuate în același timp

9	6	9	4	2	7	6	5	6	1

* 3

27	18	27	12	6	21	18	15	18	3

Probleme ”Embarrassingly parallel”

Câte elemente sunt?

9	6	9	4	2	7	6	5	6	1

Probleme ”Embarrassingly parallel”

Câte elemente sunt?

Probleme ”Embarrassingly parallel”

Câte elemente sunt? **N**

Probleme ”Embarrassingly parallel”

Dar câte elemente de procesare?

Probleme ”Embarrassingly parallel”

Dar câte elemente de procesare? **P**

Probleme ”Embarrassingly parallel”

Dar câte thread-uri?

Probleme ”Embarrassingly parallel”

Dar câte thread-uri? **P**

În majoritatea cazurilor obținem performanță maximă când numărul de thread-uri este egal cu numărul de elemente de procesare, sau core-uri.

Probleme ”Embarrassingly parallel”

Cum este P față de N?

Probleme ”Embarrassingly parallel”

P << N

Probleme ”Embarrassingly parallel”

Caz concret: P = 2

Cum împărțim?

Probleme ”Embarrassingly parallel”

Caz concret: P = 2
Cum împărțim?

9	6	9	4	2	7	6	5	6	…	1

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2
Cum împărțim?

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2
Cum împărțim?

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2
Cum împărțim? Putem și random

9	6	9	4	2	7	6	5	6	…	1

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2

Cum împărțim?

9	6	9	4	2	7	6	5	6	…	1

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2

Cum împărțim?

9	6	9	4	2	7	6	5	6	…	1

Este utilă?

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2

Cum împărțim?

9	6	9	4	2	7	6	5	6	…	1

Ce ne dorim?

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Caz concret: P = 2

Cum împărțim?

9	6	9	4	2	7	6	5	6	…	1

Ce ne dorim?

Thread 1 Thread 2

Aproximativ același număr elemente

Probleme ”Embarrassingly parallel”

Aproximativ N/P elemente

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Aproximativ N/P elemente

Dacă N nu se divide perfect la P?

Thread 1

Thread 2

Probleme ”Embarrassingly parallel”

Aproximativ N/P elemente

Dacă N nu se divide perfect la P?

Probleme ”Embarrassingly parallel”

floor(N/P) elemente **floor(15/2) = 7**

Probleme ”Embarrassingly parallel”

ceil(N/P) elemente **ceil(15/2) = 8**

Probleme ”Embarrassingly parallel”

A = floor(N/P)

Probleme ”Embarrassingly parallel”

A = ceil(N/P)

Probleme ”Embarrassingly parallel”

Formule elegante:

Tid este identificator de thread, are valori de la 0 la P

start = Tid * ceil(N/P)

end = min(N, (Tid+1) * ceil(N/P))

Măsurarea timpului de execuție

time ./executabil p a r a m e t r i

Măsurare timp – Linia de comandă

time *sleep 5*

real 0m5.001s
user 0m0.000s
sys 0m0.001s

time sleep 5

sleep 5 0.00s user 0.00s system 0% cpu 5.002 total

/usr/bin/time sleep 5

0.00user 0.00system 0:05.00elapsed 0%CPU (0avgtext+0avgdata 2076maxresident)k
0inputs+0outputs (0major+73minor)pagefaults 0swaps

Măsurare timp – Linia de comandă

***time** sleep 5*

real 0m5.001s
user 0m0.000s
sys 0m0.001s

Wall clock time – Timpul trecut de la pornirea programului – Pe acesta îl folosim

time sleep 5

sleep 5 0.00s user 0.00s system 0% cpu 5.002 total

/usr/bin/time sleep 5

0.00user 0.00system 0:05.00elapsed 0%CPU (0avgtext+0avgdata 2076maxresident)k
0inputs+0outputs (0major+73minor)pagefaults 0swaps

Măsurare timp – Linia de comandă

time sleep 2

real 0m2.0**21**s
user 0m0.000s
sys 0m0.000s

time sleep 2

real 0m2.0**18**s
user 0m0.000s
sys 0m0.016s

time sleep 2

real 0m2.0**16**s
user 0m0.000s
sys 0m0.000s

time sleep 2

real 0m2.0**15**s
user 0m0.000s
sys 0m0.000s

Timpii măsurați nu sunt exacți.

- Dacă avem timpi foarte mici nu sunt relevanți. De preferat timpi peste 1 secundă.
- Pentru a măsura corect trebuie să facem medie a timpilor după mai multe rulări.

Măsurare timp – Linia de comandă

***time** sleep 5*

real 0m5.001s
user 0m0.000s
sys 0m0.001s

Suma timpului petrecut în user space pe fiecare core.

time sleep 5

sleep 5 0.00s user 0.00s system 0% cpu 5.002 total

/usr/bin/time sleep 5

0.00user 0.00system 0:05.00elapsed 0%CPU (0avgtext+0avgdata 2076maxresident)k
0inputs+0outputs (0major+73minor)pagefaults 0swaps

Măsurare timp – Linia de comandă

Suma timpului petrecut în user space pe fiecare core.

```
time timeout 5 ./useAllCPU 12

real    0m4.075s
user    0m47.797s
sys     0m0.031s
```

Măsurare timp – Linia de comandă

***time** sleep 5*

real 0m5.001s
user 0m0.000s
sys 0m0.001s

Suma timpului petrecut în kernel pe fiecare core.

time sleep 5

sleep 5 0.00s user 0.00s system 0% cpu 5.002 total

/usr/bin/time sleep 5

0.00user 0.00system 0:05.00elapsed 0%CPU (0avgtext+0avgdata 2076maxresident)k
0inputs+0outputs (0major+73minor)pagefaults 0swaps

Măsurare timp – Linia de comandă

Orice I/O este făcut de Kernel

```
time dd if=/dev/zero of=file.txt count=1024 bs=1048576
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 9.4847 s, 113 MB/s

real    0m9.490s
user    0m0.000s
sys     0m0.992s
```

Măsurare timp – Din program

```
clock_t t;
t = clock();
WORK();
t = clock() - t;
double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
```

Măsurare timp – Din program

```
clock_t t;
t = clock();
WORK();
t = clock() - t;
double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
```

DO NOT USE

Măsurare timp – Din program

```
clock_t t;
t = clock();
WORK();
t = clock() - t;
double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
```

DO NOT USE

Din **man**: the value returned by **clock**() also includes the times of any children.

Măsurare timp – Din program

```
struct timespec start, finish;
double elapsed;
clock_gettime(CLOCK_MONOTONIC, &start);
WORK();
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed = (finish.tv_sec - start.tv_sec);
elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
```

Măsurare timp cu sau fără I/O?

Măsuri

- T - Timpul total necesar execuției programului paralel
- P - Numărul de procesoare utilizate
- G – Timp execuție cel mai rapid algoritm secvențial
- S – Speedup
  - $S = \frac{G}{T}$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1			

$$S = \frac{6.14}{6.14}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8		

$$S = \frac{6.14}{7.76}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 0.78	

$$S = \frac{6.14}{7.82}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 0.78	S = 0.78

$$S = \frac{6.14}{7.87}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 0.78	S = 0.78

WRONG

$$S = \frac{6.14}{7.87}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1			

$$S = \frac{6.15}{6.15}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8		

$$S = \frac{6.15}{7.77}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 1.55	

$$S = \frac{6.15}{3.95}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 1.55	S = 3.05

$$S = \frac{6.15}{2.01}$$

Timpi adunare a doi vectori C = A + B

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 1.55	S = 3.05

De ce nu **S = P** ?

Timpi adunare a doi vectori C = A + B

Nu se ține cont de timpul de citire/scriere RAM.

Sequential	Pthread (1 Thread)	Pthread(2 Thread)	Pthread(4 Thread)
real 0m6.151s user 0m6.141s sys 0m0.000s	real 0m7.777s user 0m7.766s sys 0m0.000s	real 0m3.954s user 0m7.828s sys 0m0.000s	real 0m2.011s user 0m7.875s sys 0m0.031s
S = 1	S = 0.8	S = 1.55	S = 3.05

De ce nu **S = P** ? S = O(P) este **ideal**.

Consistența?

- Demonstrație formală

- Stress test
  - mereu comparați cu rezultatul algoritmului secvențial

Workflow - Testarea programelor

Sanity check

- Test mici, rapide pentru a salva timp dacă sunt probleme majore

Stress test consistency

- Singura metodă ”acceptabilă” de a confirma că programul nu are bug-uri ce apar rar

Measure time

- Pentru a determina că programul e scalabil și întradevăr implementat în paralel

Algoritmi Paraleli și Distribuiți Java

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

APD

Programarea concurentă în Java

Java multithreading

Stările posibile ale unui thread

- Creat: obiectul a fost creat cu operația `new();` se poate apela metoda `start()`
- Gata de execuție: a fost apelată metoda `start()`, firul poate fi executat
- Suspendat: a fost apelat `sleep()` sau `wait()`
- Terminat: metoda `run()` a fost terminată

Java multithreading

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

start method

Thread-urile sunt pornite prin apelul metodei **start**() dintr-un obiect Thread

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

start method

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```
void start() {
	Crează stack-ul pentru thread
	Informează JVM și OS
	Thread-ul execută metoda run();
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

join method

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Thread-ul main așteaptă terminarea celorlate thread-uri prin apeluri **join**().

Trebuie să fie într-un bloc try/catch în caz că thread-ul se termină din cauza unei întreruperi.

join method

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

```
void join() {
	aș teaptă ca un thread să termine execuția
}
```

thread execution

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}
		// Main thread continues execution
		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

implements instead of extends

```java
public class MyThread extends Thread {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

Putem să folosim și implements pentru a crea obiecte ce au punctul de start al unui thread.

De ce? Pentru că în Java un obiect poate moșteni un singur obiect și există cazuri când este necesară extinderea unui alt obiect.

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		MyThread threads[] = new MyThread[N];

		for (int i = 0; i < N; i++) {
			threads[i] = new MyThread();
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

implements instead of extends

```java
public class MyRunnable implements Runnable {
	public void run() {
		System.out.println("Hello world!");
	}
}
```

```java
public class Main {
	public static void main(String[] args) {
		int N = 4;
		Thread threads[] = new Thread[N];

		for (int i = 0; i < N; i++) {
			Runnable aux = new MyRunnable();
			threads[i] = new Thread(aux);
		}
		for (int i = 0; i < N; i++) {
			threads[i].start();
		}

		System.out.println("Hello from main thread");

		for (int i = 0; i < N; i++) {
			try {
				threads[i].join();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
}
```

Java Concurrency

Sincronizarea firelor de execuție

- **Două situații:**
  - Concurență
  - Cooperare
- **Sincronizarea**
  - asigură ***excluderea mutuală*** – un singur fir poate executa la un moment dat o metodă (secvență de cod) sincronizată: **secțiune critică**
  - Folosește mecanismul de **zăvor** :
    - Fiecare obiect are asociat câte un **zăvor**
    - **synchronized(o)** asigură intrarea în secțiunea critică
    - se poate asocia unei **metode** sau unei **secvențe** de cod
    - pe parcursul execuției secvențelor de cod sincronizate, zăvorul este „închis”

Concurrency problems

```
i = 0;
```

thread 1

```
i = i + 2;
```

thread 2

```
i = i + 2;
```

```
i is 2 or 4
```

Concurrency problems

```
i = 0;
```

thread 1

```
mov eax i;
add eax 2;
mov i eax;
```

thread 2

```
mov eax i;
add eax 2;
mov i eax;
```

```
i is 2 or 4
```

Concurrency problems

```
i = 0;
```

thread 1

```
mov eax i;

add eax 2;

mov i eax;
```

thread 2

```
mov eax i;

add eax 2;
mov i eax;
```

i is **2** or 4

Concurrency problems

```
i = 0;
```

thread 1

```
mov eax i;
add eax 2;
mov i eax;
```

thread 2

```
mov eax i;
add eax 2;
mov i eax;
```

i is 2 or **4**

Solution: synchronized

```
i = 0;
```

thread 1

```
synchronized (object) {
        i = i + 2;
}
```

thread 2

```
synchronized (object) {
        i = i + 2;
}
```

```
i is 4
```

synchronized with object

```
synchronized (object) {
    i = i + 2;
}
```

Echivalent cu:

```
lock (object)
    i = i + 2;
unlock (object)
```

object can be **any** object
object can be **this**
object can be **.class**

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
    if(instance == null){
        synchronized (ThreadSafeSingleton.class) {
            if(instance == null){
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

synchronized method

Secțiune critică între toate thread-urile care folosesc același **obiect**

```
increment() {
    synchronized (this) {
        i = i + 2;
    }
}
```

```
synchronized increment() {
        i = i + 2;
}
```

Secțiune critică între toate thread-urile care folosesc **clasa *Name***

```
increment() {
    synchronized (Name.class) {
        i = i + 2;
    }
}
```

```
static synchronized increment() {
        i = i + 2;
}
```

```java
class Table {

  void printTable(int n) {// method not synchronized
    for (int i = 1; i <= 5; i++) {
      System.out.println(n * i);
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

src > MyThread1.java > MyThread1

```java
class MyThread1 extends Thread {
  Table t;

  MyThread1(Table t) {
    this.t = t;
  }

  public void run() {
    t.printTable(5);
  }

}
```

src > MyThread2.java > MyThread2

```java
class MyThread2 extends Thread {
  Table t;

  MyThread2(Table t) {
    this.t = t;
  }

  public void run() {
    t.printTable(100);
  }
}
```

src > TestSynchronization1.java > TestSynchronization1

```java
class TestSynchronization1 {
  Run | Debug
  public static void main(String args[]) {
    Table obj = new Table();// only one object
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `
For more details, please visit https://support.
bash-3.2$  /Library/Java/JavaVirtualMachines/ad
85a2b4998925731581b9dea5f/redhat.java/jdt_ws/Co
5
100
10
200
15
300
20
400
25
500
bash-3.2$
```

```java
class Table {

  synchronized void printTable(int n) {// r
    for (int i = 1; i <= 5; i++) {
      System.out.println(n * i);
      try {
        Thread.sleep(400);
      } catch (Exception e) {
        System.out.println(e);
      }
    }
  }
}
```

src > MyThread1.java > MyThread1

```java
class MyThread1 extends Thread {
  Table t;

  MyThread1(Table t) {
    this.t = t;
  }

  public void run() {
    t.printTable(5);
  }

}
```

src > MyThread2.java > MyThread2

```java
class MyThread2 extends Thread {
  Table t;

  MyThread2(Table t) {
    this.t = t;
  }

  public void run() {
    t.printTable(100);
  }
}
```

src > TestSynchronization1.java > TestSynchronization1

```java
class TestSynchronization1 {
  Run | Debug
  public static void main(String args[]) {
    Table obj = new Table();// only one object
    MyThread1 t1 = new MyThread1(obj);
    MyThread2 t2 = new MyThread2(obj);
    t1.start();
    t2.start();
  }
}
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
20
400
25
500
f/redhat.java/jdt_ws/Concurrency_5e4940f0/bin" TestSynch
5
10
15
20
25
100
200
300
400
500
bash-3.2$
```

Metoda `wait()`

- Permite manevrarea zăvorului asociat cu un obiect
- La apelul metodei `wait()` pentru un obiect ***m*** de către un fir de execuție ***t***:
  - se deblochează zăvorul asociat cu ***m*** și ***t*** este adăugat la un set de thread-uri blocate, *wait set*-ul lui ***m***
  - dacă ***t*** nu deține zăvorul pentru ***m***: `IllegalMonitorStateException`
  - ***t*** va continua execuția doar când va fi scos din *wait set*-ul lui ***m***, prin:
    - o operație **notify() / notifyAll()**
    - expirarea timpului de așteptare
    - o acțiune dependentă de implementare

Notificări

- Metode: `notify()`, `notifyAll()`
- La o notificare apelată din thread-ul ***t*** pentru obiectul ***m***:
  - `notify()`: un thread ***u*** din *wait set*-ul lui ***m*** este scos și repus în execuție
  - `notifyAll()`: toate thread-urile sunt scoase din *wait set*-ul lui ***m*** – dar ***numai unul*** va putea obține zăvorul
  - daca ***t*** nu deține zăvorul pentru ***m***: `IllegalMonitorStateException`

Java – wait()/notify()/notifyAll()

```
this.wait() {
        unlock()
        put_thread_in_wait_state()  //nu mai este pe CPU
        lock()
}
```

Java – wait()/notify()/notifyAll()

Wait funcționează doar dacă ai deja un lock (apelul se face într-o secțiune synchronized)

```
object.wait() {
        unlock()
        put_thread_in_wait_state() //nu mai este pe CPU
        lock()
}
```

Java – wait()/notify()/notifyAll()

```
this.
object.notify() {
        thread = pick_random_thread_waiting_on_object()
        move_from_wait_to_running(thread)
}
```

Java – wait()/notify()/notifyAll()

this.notifyAll()
object.notifyAll()

Pune toate thread-urile care fac wait pe object în starea de running.
Un singur thread va putea lua lock()

Producator-consumator - pseudocod

```
int buf, p=0, c=0;
co process Producer{
        int a[1:n];
    while (p<n){<await p==c>;
          buf = a[p+1];
          p = p+1;}
    }
|| process Consumer{
              int b[1:n];
     while (c<n){<await p>c>;
          b[c+1] = buf;
          c = c+1;}
     }
oc
```

- buf, p, c sunt partajate de cele doua procese
- accesul este exclusiv
- accesul este direct (nu este intermediat de un alt proces)

```java
class ZonaTampon {
  private int buf; // val. curenta din tampon
  private boolean disponibil = false; // daca o valoare este disponibila in buf

  public synchronized int Consuma() {
    if (!disponibil) { // val. indisponibila
      try {
        wait(); // suspenda executia si deblocheaza zavor
      } catch (InterruptedException e) {
      }
    }
    disponibil = false;
    notify(); // pune in executie un thread in asteptare
    return buf;
  }

  public synchronized void Produ(int valoare) {
    if (disponibil) {// o valoare in tampon
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    buf = valoare;
    disponibil = true;
    notify();
  }
}
```

Exemplu – Producator

```java
public class Producator extends Thread {
  private ZonaTampon Tampon;
  private int numar; // ID-ul producatorului

  public Producator(ZonaTampon z, int numar) {
    Tampon = z;
    this.numar = numar;
  }

  public void run() {
    for (int i = 0; i < 10; i++) {
      Tampon.Produ(i);
      System.out.println("Producator " + this.numar + " a transmis: " + i);
      try {
        sleep((int)Math.random() * 100);
      } catch (InterruptedException e) {}
    }
  }
}
```

Exemplu - Consumator

```java
public class Consumator extends Thread {
  private ZonaTampon Tampon;
  private int numar; // ID-ul consumatorului

  public Consumator(ZonaTampon z, int numar) {
    Tampon = z;
    this.numar = numar;
  }

  public void run() {
    int valoare = 0;
    for (int i = 0; i < 10; i++) {
      valoare = Tampon.Consuma();
      System.out.println("Consumator" + this.numar + " a preluat " + valoare);
    }
  }
}
```

Exemplu - Test

```java
public class App {
    Run | Debug
    public static void main(String[] args) {
        ZonaTampon z = new ZonaTampon();
        Producator p1 = new Producator(z, 1);
        Consumator c1 = new Consumator(z, 1);
                                //p1 si c1 folosesc aceeasi ZonaTampon
        p1.start();
        c1.start();
    }

}
```

```
Producator 1 a transmis: 0
Consumator1 a preluat 0
Producator 1 a transmis: 1
Consumator1 a preluat 1
Consumator1 a preluat 2
Producator 1 a transmis: 2
Producator 1 a transmis: 3
Consumator1 a preluat 3
Consumator1 a preluat 4
Producator 1 a transmis: 4
Producator 1 a transmis: 5
Consumator1 a preluat 5
Consumator1 a preluat 6
Producator 1 a transmis: 6
```

Clase utile pentru sincronizare în JDK 1.5

- `Semaphore`
- `Mutex`
- `CyclicBarrier`
  - barieră reutilizabilă
  - are ca argument un contor care arată numărul de fire din grup
- `CountDownLatch`
  - similar cu bariera, are un contor, dar decrementarea contorului este separată de așteptarea ajungerii la zero
  - decrementarea semnifică terminarea unor operații
- `Exchanger`
  - *rendez-vous* cu schimb de valori în ambele sensuri între threaduri

Facilități pentru sincronizare de nivel scăzut

- `Lock`
  - generalizare *lock monitor* cu așteptări contorizate, întreruptibile, teste etc.
- `ReentrantLock`
- `Conditions`
  - permit mai multe condiții per lock
- `ReadWriteLock`
  - exploatarea faptului că, la un moment dat, un singur fir modifică datele comune și ceilalți doar citesc
- Variabile atomice:
  - `AtomicInteger`
  - `AtomicLong`
  - `AtomicReference`
  - permit execuția atomică *read-modify-write*

volatile

```
volatile int i;
```

Fiecare scrierea asupra lui i trebuie să ajungă la toate thread-urile

Toate operațiile cu i pot fi mai lente

```
int i;
```

i poate fi ținut în cache (cache posibil ar fi regiștrii CPU)

Toate operațiile cu i pot fi mai rapide

Semaphore

Declararea unui semafor

Permits
(poate fi negativ)

```
Semaphore semaphore = new Semaphore(4);
```

Luarea unui semafor **P()**
(Decremenetează permits)
(Așteaptă dacă permits e 0)

```
try {
        semaphore.acquire();
} catch (InterruptedException e) {
        e.printStackTrace();
}
```

Se eliberează semaforul **V()**
(Crește numărul de permits)

```
semaphore.release();
```

Un semafor inițializat cu 1 este echivalentul unui mutex/lock

Cyclic Barrier

Declararea barierei

Numărul de thread-uri care trebuie să aștepte la barieră ca aceasta să se deschidă

```
CyclicBarrier barrier = new CyclicBarrier(5);
```

Așteaptă până numărul necesar de thread-uri ajung la acest apel.

```
try {
        barrier.await();
} catch (InterruptedException | BrokenBarrierException
e) {
        e.printStackTrace();
}
```

Cyclic Barrier

- O barieră garantează că toate thread-urile execută tot codul de dinainte de barieră înainte de a executa orice linie de cod de după barieră

- Barierele clasice nu puteau fi folosite înăuntrul unei iterații
    - După ce toate thread-urile ajung la barieră unul o putea lua înainte și ajunge iar la barieră înainte să se închidă și ar mai trece o dată de ea.

- O barieră **CyclicBarrier** sau **Reentrantă** poate fi refolosită (Este sigură pentru a fi folosită în for/while)

Executor Service – replicated workers

Executor Service

```
ExecutorService tpe =
Executors.newFixedThreadPool(4);
```

Executor Service

```java
public class MyRunnable implements Runnable {
	int a;
	ExecutorService tpe;

	public MyRunnable(ExecutorService tpe, int a) {
		this.a = a;
		this.tpe = tpe;
	}

	@Override
	public void run() {
		if (a > 10) {
			tpe.shutdown();
			return;
		}
		System.out.println(a);
		tpe.submit(new MyRunnable(tpe, a + 3));
	}
}
```

Executor Service

```
tpe.submit(new MyRunnable(tpe, 0));
tpe.submit(new MyRunnable(tpe, 1));
tpe.submit(new MyRunnable(tpe, 2));
……
```

Executor Service

Executor Service

Task-urile pot genera alte task-uri

```
tpe.submit(new MyRunnable(tpe, a + 3));
```

Executor Service

Executor Service

Executor Service

Executor Service

Executor Service

Când oprim thread-urile?

- Depinde de problemă.
- Uneori ne ajunge o singură soluție.
- Alte probleme pot necesita toate soluțiile.

```
tpe.shutdown();
```

```java
public class MyRunnable implements Runnable {
  private final long countUntil;

  MyRunnable(long countUntil) {
      this.countUntil = countUntil;
  }

  @Override
  public void run() {
      long sum = 0;
      for (long i = 1; i < countUntil; i++) {
          sum += i;
      }
      System.out.println(sum);
  }
}
```

```java
public class App {
    private static final int NTHREDS = 10;

    Run | Debug
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
        for (int i = 0; i < 500; i++) {
            Runnable worker = new MyRunnable(10000000L + i);
            executor.execute(worker);
        }
        // This will make the executor accept no new threads
        // and finish all existing threads in the queue
        executor.shutdown();
        // Wait until all threads are finish
        try{
            executor.awaitTermination(60, TimeUnit.SECONDS);
        } catch(InterruptedException ie) {
            // (Re-)Cancel if current thread also interrupted
            executor.shutdownNow();
        }
    }
}
```

CompletableFuture – async tasks

```
CompletableFuture.supplyAsync(this::doSomething);
```

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;

public class CompletableFutureCallback {
    public static void main(String[] args) {
        long started = System.currentTimeMillis();

        CompletableFuture<String>  data = createCompletableFuture()
                .thenApply((Integer count) -> {
                    int transformedValue = count * 10;
                    return transformedValue;
                }).thenApply(transformed -> "Finally creates a string: " + transformed);

            try {
                System.out.println(data.get());
            } catch (InterruptedException | ExecutionException e) {

            }
    }

    public static CompletableFuture<Integer> createCompletableFuture() {
        CompletableFuture<Integer>  result = CompletableFuture.supplyAsync(() -> {
            try {
                // simulate long running task
                Thread.sleep(5000);
            } catch (InterruptedException e) { }
            return 20;
        });
        return result;
    }

}
```

```java
public class CompletableFutureSimpleSnippet {
    public static void main(String[] args) {
        long started = System.currentTimeMillis();

        // configure CompletableFuture
        CompletableFuture<Integer> futureCount = createCompletableFuture();

            // continue to do other work
            System.out.println("Took " + (started - System.currentTimeMillis()) + " milliseconds"
);

            // now its time to get the result
            try {
              int count = futureCount.get();
                System.out.println("CompletableFuture took " + (started -
System.currentTimeMillis()) + " milliseconds" );

               System.out.println("Result " + count);
             } catch (InterruptedException | ExecutionException ex) {
                // Exceptions from the future should be handled here
            }
    }

    private static CompletableFuture<Integer> createCompletableFuture() {
        CompletableFuture<Integer> futureCount = CompletableFuture.supplyAsync(
                () -> {
                    try {
                        // simulate long running task
                        Thread.sleep(5000);
                    } catch (InterruptedException e) { }
                    return 20;
                });
        return futureCount;
    }

}
```

Structuri de date

HashMap

Keys	Values
A	11111
DD	2222
C	33333
S	4444

Hash(Keys)	Values
1	2222
2	11111
3	33333
4	
5	4444

Search **O(1)**
Insertion **O(1)**
Deletion **O(1)**

ConcurrentHashMap

- Folosește lock-uri în interiorul tuturor metodelor: **get**(), **put**()
- Asta înseamnă că se poate folosi structura fără cod special pentru sincronizare.
- Are și metode speciale gen **putIfAbsent**()

- Există mai multe astfel de structuri (idem pentru HashTable)

ConcurrentHashMap – privire sub capotă

- Dacă folosim un singur lock pentru accesarea HashMap: 2 thread-uri nu îl vor putea modifica simultan.
- Dacă folosim un lock pentru fiecare element din HashMap: Vom folosi multă memorie și avem nevoie de mecanisme speciale pentru inserție sau ștergere.
- Soluția e folosirea unui număr limitat de lock-uri (de obicei egal cu numărul de thread-uri).

Hash(Keys)	Values	
1	2222	Lock A
2	11111	
3	33333	Lock B
4		
5	4444	Lock C

```java
class HashMapDemo extends Thread {
    static HashMap<Integer,String> l=new HashMap<Integer,String>();

    public void run() {
        try {
            Thread.sleep(1000);
            l.put(103,"D");
        } catch(InterruptedException e) {
            System.out.println("Child Thread going to add element");
        }
    }

    Run | Debug
    public static void main(String[] args) throws InterruptedException {
        l.put(100,"A");
        l.put(101,"B");
        l.put(102,"C");
        HashMapDemo t=new HashMapDemo();
        t.start();

        for (Object o : l.entrySet())  {
            Object s=o;
            System.out.println(s);
            Thread.sleep(1000);
        }
        System.out.println(l);
    }
}
```

```
100=A
101=B
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.HashMap$HashIterator.nextNode(HashMap.java:1493)
        at java.base/java.util.HashMap$EntryIterator.next(HashMap.java:1526)
        at java.base/java.util.HashMap$EntryIterator.next(HashMap.java:1524)
        at HashMapDemo.main(HashMapDemo.java:23)
```

```java
public class App extends Thread {

    static ConcurrentHashMap<Integer, String> l = new ConcurrentHashMap<Integer, String>();

    public void run() {
        l.put(103, "D");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            System.out.println("Child Thread going to add element");
        }
    }

    Run | Debug
    public static void main(String[] args) throws Exception {
        l.put(100, "A");
        l.put(101, "B");
        l.put(102, "C");
        HashMapDemo t = new HashMapDemo();
        t.start();

        for (Object o : l.entrySet()) {
            Object s = o;
            System.out.println(s);
            Thread.sleep(1000);
        }
        System.out.println(l);
    }
}
```

```
a/jdt_ws/HashMapDemo_eca003ea/bin" App 2fa1c97cda8748c6465d77ce70e076/redhat
100=A
101=B
102=C
{100=A, 101=B, 102=C}
```

Java – Atomic*

- **AtomicInteger**
- **AtomicLong**
- **AtomicBoolean**
- **AtomicFloat**
- **AtomicReference**

- Problema: 32bit/64bit systems (+ alte arhitecturi)
  - Copierea elementelor de 64 de biți pe procesor de 32 de biți ia mai multe operații iar thread-ul poate fi scos de pe CPU în timpul unei scrieri
  - Un alt thread poate citi doar o parte din variabilă

- **Atomic*** asigură că toate operațiile cu aceea variabilă sunt atomice (nu se pot face citiri în mijloc)
- Oferă și oprații mai complexe **getAndSet/getAndIncrement** implementate atomic

```java
import java.util.concurrent.atomic.AtomicInteger;

public class Counter {
    private AtomicInteger value = new AtomicInteger();
    public int getValue(){
        return value.get();
    }
    public int increment(){
        return value.incrementAndGet();
    }

    // Alternative implementation as increment but just make the
    // implementation explicit
    public int incrementLongVersion(){
        int oldValue = value.get();
        while (!value.compareAndSet(oldValue, oldValue+1)){
             oldValue = value.get();
        }
        return oldValue+1;
    }

}
```

```java
public final int incrementAndGet() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return next;
        }
    }
```

Algoritmi Paraleli și Distribuiți
Mecanisme de sincronizare

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Race Conditions

Process a:

```
while(i < 10)
    i = i +1;
print "A won!";
```

Process b:

```
while(i > -10)
    i = i - 1;
print "B won!";
```

– Cine câștigă?
– Există un câștigător?

Problema secțiunilor critice

- Problema: Dezvoltarea unui protocol pentru cooperare între procese, a.î. execuția să aibă loc mutual exclusiv (secțiuni critice)
  - Cum să facem ca mai multe instrucțiuni să pară ca una singură?

Shared vars:

Initialization:

Process:

. . .

. . .

Entry Section

Critical Section

Exit Section

Soluția constă în introducerea acestor mecanisme

Cerințe ale soluției

Discuție asupra folosirii semafoarelor pentru soluționare...

- **Mutual Exclusion**
  - Doar un process poate fi la un moment dat în secțiunea critică
- **Progress**
  - Decizia cine intră în regiune nu poate fi amânată indefinit
    - No deadlock
- **Bounded Waiting**
  - Limită asupra #times alte procese pot intra în regiune, în timp ce aștept
    - No livelock
- La care se adaugă eficiență, fairness …

Reminder: Dekker's Algorithm

- Presupune două thread-uri, numerotate cu 0 și 1

```
CSEnter(int i)
 {
   inside[i] = true;
   while (inside[j])
   {
     if (turn == j)
     {
        inside[i] = false;
        while(turn == j) continue;
        inside[i] = true;
      }
   }}
```

```
CSExit(int i)
{
  turn = j;
  inside[i] = false;
  }
```

Peterson's Algorithm (1981)

```
CSEnter(int i)
{
      inside[i] = true;
      turn = j;
      while (inside[j] && turn == j)
            continue;
}
```

```
CSExit(int i)
{
  inside[i] = false;
  }
```

- Mai simplu e mai bine!!

O analiză a algoritmului lui Peterson:

- Safety (prin contradicție):
  - Presupuneți că două procese (Ana și Ion) sunt în regiunea critică (ambele au flag-ul *inside* setat). Deoarece doar unul, să spunem Ana, poate avea *turn*, celălalt (Ion) **trebuie** să fi ajuns la testul *while()* înainte ca Ana să seteze flag-ul *inside.*
  - Totuși, după setarea flag-ului *inside*, Ana a cedat *turn* către Ion. Ion deja a modificat *turn* și **nu poate** să îl modifice din nou, ceea ce contravine presupunerii inițiale.

Liveness & Bounded waiting => date de variabila *turn*.

Putem generaliza pentru mai multe thread-uri?

- Abordarea anterioară nu funcționează:

```
CSEnter(int i)
{
    inside[i] = true;
    for (j = 0; j < N; j++)
        while (inside[j] && turn == j)
            continue;
}
```

```
CSExit(int i)
{
    inside[i] = false;
}
```

- Problema: Al cui e rândul?

Conceptul de Bakery

- Introdus de Leslie Lamport
- Gândiți-vă la un magazin popular cu un vânzător (ex. cazul unui Apple Store și lansarea următorului iPhone)
  - Oamenii iau un bilet de la un automat
  - Dacă nimeni nu așteaptă, biletul nu mai contează
  - Dacă mai mulți clienți așteaptă, ordinea biletelor determină ordinea în care vor fi serviți

Algoritmul Bakery : “Runda 1”

- int ticket[n];
- int next_ticket;

```
CSEnter(int i)
{
    ticket[i] = ++next_ticket;
    for (j = 0; j < N; j++)
        while(ticket[j] && ticket[j] < ticket[i])
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Oops… accesul la *next_ticket* este o problemă!

Algoritmul Bakery: “Runda 2”

- int ticket[n];

Adăugăm 1 la maxim!

```
CSEnter(int i)
{
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    for (j = 0; j < N; j++)
        while (ticket[j] && ticket[j] < ticket[i])
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Idee: adăugăm 1 la maxim.
- Oops… două procese pot prelua o aceeași valoare!

Algoritmul Bakery: “Runda 3”

Dacă i și j aleg un același bilet, stabilim o regulă tip “break tie”:

(ticket[j] < ticket[i]) || (ticket[j]==ticket[i] && j<i)

Notație: (B,j) < (A,i) pentru a simplifica scrierea codului:

(B<A || (B==A && j<i)), e.g.:

(ticket[j],j) < (ticket[i],i)

Algoritmul Bakery: “Runda 4”

- int ticket[N];
- boolean picking[N] = false;

```
CSEnter(int i)
{
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    for (j = 0; j < N; j++)
        while(ticket[j] && (ticket[j], j) < (ticket[i],i))
            continue;
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

- Oops… aș putea să mă uit la j atunci când j încă nu deține biletul, și totuși j ar putea avea un id mai mic ca al meu (i)!

Algoritmul Bakery: Aproape gata

- int ticket[N];
- boolean choosing[N] = false;

```
CSEnter(int i)
{
    choosing[i] = true;
    ticket[i] = max(ticket[0], … ticket[N-1])+1;
    choosing[i] = false;
    for (j = 0; j < N; j++) {
        while(choosing[j]) continue;
        while(ticket[j] && (ticket[j], j) < (ticket[i],i))
            continue;
    }
}
```

```
CSExit(int i)
{
    ticket[i] = 0;
}
```

Algoritmul Bakery: Probleme?

- Dacă nu știm câte thread-uri rulează?
  - Algoritmul depinde de valoarea N
  - Cumva avem nevoie de un mecanism de ajustare a N atunci când un thread e creat sau termină execuția
- De asemenea, tehnic, un bilet poate da overflow!
  - Soluția: Modificarea codului ca, atunci când un bilet ajunge la o valoare prea mare, să îl resetăm la 0

Algoritmul Bakery: Final

- int ticket[N]; /* Important: Disable thread scheduling when changing N */
- boolean choosing[N] = false;

```
CSEnter(int i)
{
        do {
            ticket[i] = 0;
            choosing[i] = true;
            ticket[i] = max(ticket[0], … ticket[N-1])+1;
            choosing[i] = false;
        } while(ticket[i] >= MAXIMUM);
        for (j = 0; j < N; j++) {
                while(choosing[J]) continue;
                while(ticket[J] && (ticket[J],J) < (ticket[i],i))
                        continue;
        }
}
```

```
CSExit(int i)
{
        ticket[i] = 0;
}
```

Ne reîntoarcem la semafoare

- Unele sisteme (acolo unde busy-waiting nu e o problemă, gen există suficiente core-uri) implementează soluții bazate pe Algoritmul Bakery
- Alteori (cel mai des însă) avem nevoie de *semafor*
  - Soluție cu *blocare* în loc de *busy-waiting* de la SO

Blocare vs. non-blocare

```
acquire() {
    while (value <= 0)
        ; // no-op
    value--;
}

release() {
    value++;
}
```

```
acquire() {
    value--;
    if (value < 0) {
        add this process to list
        block();
    }
}

release() {
    value++;
    if (value <= 0) {
        remove a process P from list
        wakeup(P);
    }
}
```

Un pic de recap... ce e greșit?

Shared: Semaphores mutex, empty, full;

Init: mutex = 1; /* for mutual exclusion*/
empty = N; /* number empty bufs */
full = 0; /* number full bufs */

Producer

```
do {
    . . .
    // produce an item in nextp
    . . .
    P(mutex);
    P(empty);
    . . .
    // add nextp to buf
    . . .
    V(mutex);
    V(full);
} while (true);
```

Ce se întâmplă dacă buffer = full?

Oops! Ordinea în care realizați operații poate avea un impact semnificativ asupra corectitudinii

Consumer

```
do {
    P(full);
    P(mutex);
    . . .
    // remove item to nextc
    . . .
    V(mutex);
    V(empty);
    . . .
    // consume item in nextc
    . . .
} while (true);
```

Exercițiu

- Cum ati implementa o *barieră* folosind semafoare?
- Ex. (suma elementelor unui vector):

```
int a[1:n];
process suma[k=1 to n]{
      for [j = 1 to sup(log2 n)]{
            if (k mod 2^j = 0){
                  a[k] = a[k-2^(j-1)] + a[k];
            }
            barrier
      }
}
```

Posibila solutie… pentru bariera

```
sem b = 0; e = 1;
int nb = 0;
process proc[k=1 to n]{
    # enter barrier
    P(e);
    nb=nb+1;
    if (nb = n){ V(e); V(b);}
    else if (nb > 1){ V(e); P(b); V(b);}
    else if (nb = 1) { V(e); P(b);}
    P(e);
    nb=nb-1;
    V(e);
    # exit barrier
}
```

```
# if last process
to enter barrier
```

```
# if first process
to enter barrier
```

Un pic de practice: Cigarette smokers problem

- Un agent și trei fumători
- Fumătorii:
  - Așteaptă ingrediente (tutun, hârtie, chibrit)
  - Confecționează țigară
  - Fumează
- Agentul deține toate 3 ingredientele
- Un fumător are tutun, un altul hârtie, al 3-lea chibrituri)
- Agentul selectează două ingrediente (random) pe care le dă fumătorilor
  - Doar fumătorul ce are nevoie de exact acele 2 ingrediente trebuie să le preia
  - Agentul nu poate semnaliza exact acelui fumător pentru că nu știe care fumător e care, respectiv ingredientele sunt random extrase

Cigarette smokers problem

```
sem tobacco = 0;
sem paper = 0;
sem match = 0;
Sem agent = 1;
process Agent{
    while (true){
      if (draw1){ P(agent); V(tobacco); V(paper); }
      else if (draw2){ P(agent); V(paper); V(match); }
      else if (draw3){ P(agent); V(tobacco); V(match);}
    }
}
process Smoker1{
    P(tobacco); P(paper); V(agent);
}
process Smoker2{
    P(paper); P(match); V(agent);
}
process Smoker3{
    P(tobacco); P(match); V(agent);
}
```

Funcționează ???

Cigarette smokers problem - deadlock

OK!

P(match)

P(tobacco)

DEADLOCK!

P(paper)

P(match)

P(tobacco) P(paper)

Practice

- Gandiți-vă la o soluție pentru evitarea deadlock-ului…

Cigarette smokers problem

```
sem tobacco = 0;
sem paper = 0;
sem match = 0;
sem agent = 1;

bool isTobacco = false;
bool isPaper = false;
bool isMatch = false;

sem tobaccoSem = 0;
sem paperSem = 0;
sem matchSem = 0;

process Agent{
    while (true) {
      if (draw1) { P(agent); V(tobacco); V(paper);}
      else if (draw2) { P(agent); V(paper); V(match);}
      else if (draw3) { P(agent); V(tobacco); V(match);}
    }
}
```

Cigarette smokers problem

```
process PusherA{
    P(tobacco);
    P(e);
    if (isPaper) { isPaper =  false; V(matchSem);}
    else if (isMatch) { isMatch = false; V(paperSem);}
    else if (isPaper == isMatch == false) isTobacco = true;
    V(e);
}

proces PusherB{
    P(match);
    P(e);
    if (isPaper) { isPaper =  false; V(tobaccoSem); }
    else if (isTobacco) { isTobacco = false; V(paperSem);}
    else if (isPaper == isTobacco == false) isMatch = true;
    V(e);
}

process PusherC{
    P(paper);
    P(e);
    if (isTobacco) { isTobacco =  false; V(matchSem);}
    else if (isMatch) { isMatch = false; V(tobaccoSem);}
    else if (isPaper == isMatch == false) isPaper = true;
    V(e);
}
```

Cigarette smokers problem

```
process SmokerWithTobacco{
    P(tobaccoSem);
    # makeCigarette
    V(agent);
    # smoke
}
process SmokerWithPaper{
    P(paperSem);
    # makeCigarette
    V(agent);
    # smoke
}
process SmokerWithMatch{
    P(matchSem);
    # makeCigarette
    V(agent);
    # smoke
}
```

O altă problemă - East-West Bridge Problem

- Considerați un pod, prea îngust pentru a permite mașinile să meargă în ambele direcții.
  - Mașinile trebuie să alterneze pt. trecerea podului.
  - Podul nu este, de asemenea, suficient de puternic pentru a ține mai mult de **trei** mașini simultan.
- Găsiți o soluție la această problemă care să evite **starvation**.
  - Mașinile care doresc să treacă ar trebui să ajungă în cele din urmă.
- Încercați să maximizați capacitatea podului (adică trei mașini ar trebui să meargă la un moment dat).
  - Dacă o mașină părăsește podul care se îndreaptă spre est și nu există mașini spre vest, atunci următoarea mașină spre est trebuie lăsată să treacă.

alamy stock photo
DYC3N7
www.alamy.com

Soluția problemei

- Soluția: multiple readers writers problem.
- Intuitiv, dacă esti un reader ce intră în cameră, și vezi writers în cameră, iei loc (dacă esti o masină din est, și mașinile din vest traversează, aștepți).
- Dacă ești un reader, și sunt mai puțin de trei readers, și nu există writers în așteptare, intri în cameră.
- Dacă ești un reader, și sunt exact trei readers, aștepți.
- Dacă ești un reader ce părăsește camera, dacă nu mai sunt writers dar sunt readers în așteptare, permiți unui reader să intre.
- Dacă sunt writers în așteptare, și ești ultimul reader, permiți următorilor trei writers să intre.
- Principala diferență față de readers-writers e că vrem să permitem la **exact** trei readers sau writers să intre simultan în cameră.

Invarianți

- Safety conditions: Readers și writers nu pot fi activi în același timp:
  - nr_active <= 3
  - nw_active <= 3
  - nr_active > 0 --> nw_active == 0
  - nw_active > 0 --> nr_active == 0
- Alte condiții:
  - nr_active > 0 & nr_waiting > 0 --> nw_waiting > 0
  - nw_active > 0 & nw_waiting > 0 --> nr_waiting > 0

Soluție pentru readers:

```
P( lock ); // lock is a mutex and is initializes to 1.
if (( nw_active + nw_waiting == 0 ) && ( nr_active < 3 )) {
    nr_active++; // notify we are active
    V( r_sem ); // allow ourself to get through
}  else
    nr_waiting++; // we are waiting
V( lock );
P( r_sem ); // readers will wait here, if they must wait.

READING...

P( lock );
nr_active--;
if (( nr_active == 0 ) && ( nw_waiting > 0 ))  { // if we are the last reader
   count = 0; // makes sure we only at most 3 writers in
   while (( nw_waiting > 0 ) && ( count < 3 )) {
        V( w_sem ); // wake a writer;
        w_active++; // one more active writer
        w_waiting--; // one less waiting writer.
        count++;
   }
}  else if (( nw_waiting == 0 ) && ( nr_waiting > 0 )) { // allow another waiting reader to go in, if no waiting
writers
         V( r_sem );
         w_active++; // one more active reader
         w_waiting--; // one less waiting reader.
}
V( lock );
```

Initializați semafoarele

Să scriem soluția pentru writers...

Ordinea semafoarelor

- În momentul în care apelați V(), oricare thread blocat pe P() va fi trezit
  - Dar dacă doriți păstrarea ordinii FIFO?

Example: P and V

```
void v(struct semaphore *s) {
    struct proc_desc *p = 0;

    save = intr_enable( ALL_DISABLE );
    while ( TestAndSet( &s->sem_lock ) );
    s->sem_count++;
    if (p = get_from_queue( &s->sem_queue)) {
      p->runstate &= ~PROC_BLOCKED;
    }
    s->sem_lock = 0;
    intr_enable( save );
    if (p)
      reschedule(p);
}
```

```
void p(struct semaphore *s) {
    struct proc_desc *p = 0;
    for (;;) {
      save = intr_enable( ALL_DISABLE );
      while ( TestAndSet( &s->sem_lock ) );
      if (s->sem_count > 0) {
          s->sem_cont--;
          s->sem_lock = 0;
          intr_enable( save );
          return;
      }
      add_to_queue( &s->sem_queue, myproc );
      myproc->runstate |= PROC_BLOCKED;
      s->sem_lock = 0;
      intr_enable( save );
      yield();
    }
}
```

process A			**P**	**P**		**WAKE**
process B		**V**			**V**	
Semaphore						
lock	NO	YES	YES	YES	YES	YES
count	0	1	0		1	0
queue	0			**A**		
int disable	NO	YES	YES	YES	YES	YES

Semafor FIFO în Java

```
public class SemaphoreQueue
{
    private SemaphoreSlim semaphore;
    private ConcurrentQueue<TaskCompletionSource<bool>> queue =
        new ConcurrentQueue<TaskCompletionSource<bool>>();
    public SemaphoreQueue(int initialCount)
    {
        semaphore = new SemaphoreSlim(initialCount);
    }
    public SemaphoreQueue(int initialCount, int maxCount)
    {
        semaphore = new SemaphoreSlim(initialCount, maxCount);
    }
    public void Wait()
    {
        WaitAsync().Wait();
    }
    public Task WaitAsync()
    {
        var tcs = new TaskCompletionSource<bool>();
        queue.Enqueue(tcs);
        semaphore.WaitAsync().ContinueWith(t =>
        {
            TaskCompletionSource<bool> popped;
            if (queue.TryDequeue(out popped))
                popped.SetResult(true);
        });
        return tcs.Task;
    }
    public void Release()
    {
        semaphore.Release();
    }
}
```

Algoritmi Paraleli și Distribuiți
MPI

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

MPI - *Message Passing Interface*

- **MPI** este un **standard** pentru comunicarea prin mesaje elaborat de *MPI Forum*.
- Sisteme anterioare:
  - Intel (NX/2) Express
  - nCUBE (Vertex)
  - PARMACS
  - Zipcode
  - Chimp
  - PVM
  - Chameleon
  - PICL
- MPI are la bază modelul proceselor comunicante prin mesaje.

MPI - *Message Passing Interface*

- MPI este o (*specificație* pentru o) **bibliotecă**, nu un limbaj.
- MPI specifică reguli de apel pentru:
  - C / C++
  - FORTRAN 77 / FORTRAN 90
- Istoric:
  - **MPI 1** (1993 – SC93) - orientat pe comunicarea punct la punct
  - **MPI 2** (1997) include:
    - Procese dinamice
    - Comunicarea *one-sided*
    - Operații de I/E paralele

Obiective MPI

- Standardizare
  - Singura bibliotecă de transmitere de mesaje ce poate fi considerată standard
- Portabilitate
  - Codul nu trebuie modificat la portarea aplicațiilor
- Performanță
  - Implementarile producătorilor pot exploata carcateristici hardware native
- Funcționalitate
  - Peste 115 rutine doar în **MPI 1**
- Disponibilitate
  - Implementări atât publice, cât și de la producători

Elemente noi introduse de MPI

- **Tampon de comunicație** - definit de tripleta:
  - *(adresă, contor, tip_de_date)*
- **Context**
  - Fiecare comunicare de mesaj se derulează într-un *context*.
  - Mesajele sunt întotdeauna primite în contextul în care au fost transmise.
  - Mesajele transmise în contexte diferite nu interferă.
  - Mesajele sunt caracterizate de**:**
    - **context**
    - **tag**
- **Grup de procese**
  - Contextul este partajat de un *grup de procese*.
  - Fiecare proces are un rang (număr întreg).
  - Deci, fiecare proces e caracterizat de:
    - **grup**
    - **rang în grup**

Elemente noi introduse de MPI

Există un comunicator predefinit: ***MPI_COMM_WORLD***

Structura generală a unui program MPI

```
1   #include <mpi.h>
2
3   .
4   .
5   .
6   /* Initialize MPI Environment */
7   int MPI_Init(int *argc, char ***argv);
8   int MPI_Initialized(int *flag);
9   .
10  .
11  .
12  /* Do work and make message passing calls */
13  int MPI_Comm_rank(MPI_Comm comm, int *rank);
14  int MPI_Comm_size(MPI_Comm comm, int *size);
15  .
16  .
17  .
18  /* Terminate MPI Environment */
19  int MPI_Finalize();
```

Comunicarea punct la punct – operații de bază

- Forma clasică:
  - send (adresă, lungime, destinație, tag)
  - recv (adresă, lungime_maximă, sursă, tag, lungime_efectivă)
- În MPI:
  - MPI_SEND (buf, count, datatype, dest, tag, comm)
  - MPI_RECV (buf, count, datatype, source, tag, comm, status)
- Limbajul C:

```
int MPI_Send (void *buf, int count,
              MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
int MPI_Recv (void *buf, int count,
               MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Status *status)
```

Exemplu

```
1   #include "mpi.h"
2
3   int main (int argc, char **argv)
4   {
5       char message[40];
6       int myrank;
7       MPI_Status status;
8
9       MPI_Init(&argc, &argv);
10      MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
11      if (myrank == 0) {
12          strcpy(message, Hello, there);
13          MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
14      } else {
15          MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
16          printf("\tReceived:%s \n", message);
17      }
18
19      MPI_Finalize();
20      return 0;
21  }
```

Moduri de comunicare

- Apelurile *send* și *receive* sunt **blocante**
- Moduri de comunicare:
  - cu buffer local
  - fără buffer local

Modurile de comunicaţie MPI

Tip	Condiția de terminare
Synchronous send	Finalizare atunci când operația de recepție a fost terminată cu succes.
Buffered send	Se termină întotdeauna (cu exceptia apariției unei erori), indiferent de operația de recepție.
Standard send	Trimiterea mesajului (fără a se cunoaște starea operatiei de recepție).
Ready send	Se termină întotdeauna (cu excepția apariției unei erori), însă folosirea e condiționată de apelul anterior la destinație a operației de recepție.

Probleme de transmitere în modul sincron

- Presupunem că:
  - avem 5 procese care execută o serie de *send* și *receive* + 1 proces care execută *receive*
  - inițial, cele 5 procese execută simultan *send* și un proces execută *receive*

Probleme de transmitere în modul sincron

- Soluție:
  - Ordonarea operațiilor *send* și *recv*

A doua secvență de operații are loc fără un timp de așteptare.

Operaţii *send-recv* combinate

```
MPI_Sendrecv

int MPI_Sendrecv(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int source, int recvtag,
    MPI_Comm comm,
    MPI_Status *status )


MPI_Sendrecv_replace

int MPI_Sendrecv_replace(
    void *buf, int count, MPI_Datatype datatype,
    int dest, int sendtag, int source, int recvtag,
    MPI_Comm comm,
    MPI_Status *status )
```

Transmitere prin tampon alocat explicit

```
1   #define BuffSize 100000
2   int size;
3   char *buff;
4
    buff = malloc(BuffSize); MPI_Buffer_attach(buff,
5   BuffSize);
6   /* Tamponul de BuffSize octeți poate fi folosit acum */
7
8   MPI_Bsend( ... );
9   MPI_Buffer_detach(&buff, &size);
10  /* Tamponul redus la zero */
11
12  MPI_Buffer_attach(buff, size);
13  /* Tamponul disponibil din nou */
14
15
16  int MPI_Buffer_attach(void *buffer, int size );
17  int MPI_Buffer_detach(void *bufferptr, int *size );
```

Transmitere sincronă și în mod pregătit

- Transmitere sincronă
  - MPI_Ssend
- Transmitere în modul pregătit
  - MPI_Rsend

Comunicația non-blocantă

- ```
  int MPI_Isend(void *buf, int count,
        MPI_Datatype datatype, int dest, int tag,
        MPI_Comm comm, MPI_Request *request)
  ```
- ```
  int MPI_Irecv(void *buf, int count,
        MPI_Datatype datatype, int source, int tag,
        MPI_Comm comm, MPI_Request *request)
  ```

- MPI_Test
- MPI_Testall
- MPI_Testany
- MPI_Testsome

- MPI_Wait
- MPI_Waitall
- MPI_Waitany
- MPI_Waitsome

Exemplu

```
1   int rank, size, i, index, a[10], flag, f[6];
2   MPI_Request request, r[6];
3   MPI_Status status, s[6];
4   if (rank == 0) {
5       for (i=0; i<10; i++)
6           a[i] = i+1;
7       MPI_Isend(a, 10, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
8       MPI_Test(&request, &flag, &status);
9       if (flag) {
10          printf("Procesul 0 ... operatia nu s-a terminat\n");
11      } else {
12          printf("Procesul 0 ... operatia s-a terminat\n");
13      }
14      sleep(10);
15      MPI_Wait(&request, &status);
16      printf("Procesul 0 ... operatia s-a terminat\n");
17  }
18  if (rank == 1) {
19      sleep(15);
20      a[0] = 255;
21      MPI_Irecv(a, 10, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
22      MPI_Test(&request, &flag, &status);
23      if (flag) {
24        printf("Procesul 1 ... am primit date");
25      } else {
26        MPI_Wait(&request, &status);
27      }
28  }
```

Evitarea întârzierii nelimitate (1)

```
1   typedef struct {
2       char data [MSIZE];
3       int dsize;
4   } Buffer;
5
6   Buffer buf[];
7   MPI_Request req[];
8   MPI_Status status[];
9   int index[];
10  ...
11  MPI_Comm_rank(comm, &rank);
12  MPI_Comm_size(comm, &size);
13
14  if(rank != size-1) {
15      /* client code */
16      buf = (Buffer *) malloc(sizeof(Buffer));
17      while(1) {
18          produce_request( buf->data, &buf->dsize);
19          MPI_Send(buf->data,buf->dsize,MPI_CHAR,size-1,tag,comm)
20      }
21  }
```

Evitarea întârzierii nelimitate (2)

```
22  else {
23
24      buf = malloc(sizeof(Buffer)*(size-1));
25      req = malloc(sizeof(MPI_Request)*(size-1));
26      status = malloc(sizeof(MPI_Status)*(size-1));
27      index = malloc(sizeof(int)*(size-1));
28
29      for(i=0; i < size-1; i++)
30          MPI_Irecv(buf[i].data,MSIZE,MPI_CHAR,i,tag,comm,&req[i]);
31      while (1){
32          MPI_Waitsome(size-1, req, &count, index, status);
33          for (i=0; i < count; i++){
34              j = index[i];
35              MPI_Get_count(&status[i], MPI_CHAR, &buf[j].dsize);
36              process_request(buf[j].data, buf[j].dsize);
37              MPI_Irecv(buf[j].data,MSIZE,MPI_CHAR,j,tag,comm,&req[j])
38          }
39     }
40  }
```

Cereri de comunicare persistentă

- De multe ori, o comunicație cu *aceeași listă de argumente* este repetată în cadrul unei bucle.
- În astfel de situații, este posibilă optimizarea comunicației prin prin *legarea* listei de argumente la o cerere de comunicare persistentă.
- Cererea de comunicare persistentă reduce timpul de comunicare dintre proces și *controller*-ul de comunicare, nu și timpul de comunicare scurs pe rețeaua externă.
- Nu este necesar ca mesajele trimise prin intermediul cererilor persistente să fie recepționate în mod persistent (sau viceversa).

Cereri de comunicare persistentă

- O cerere de comunicare persistentă este inactivă după ce a fost creată.
- O comunicare (operație *send* sau *receive*) care folosește o cerere persistentă trebuie inițiată (cu ajutorul funcției *MPI_Start*).
- Astfel, programatorul poate să:
  - creeze o cerere persistentă – *MPI_Send_init*, *MPI_recv_init*
  - declanșeze o operație – *MPI_Start*, *MPI_Startall*
  - aștepte terminarea unei operații – *MPI_Wait*
  - elimine înregistrarea cererii persistente – *MPI_Request_free*

Cereri de comunicare persistentă

- Sintaxă:
  - `int MPI_Start(MPI_Request *request)`
  - `int MPI_Wait(MPI_Request *request, MPI_Status *status)`
  - `int MPI_Request_free(MPI_Request *request )`

- Exemplu:

```
MPI_request *request;
MPI_Recv_init(buf, count, datatype,
              source, tag, comm, request);

/* MPI_Start are semantica MPI_Irecv */
MPI_Start(request);
MPI_Wait(request, status);
MPI_Request_free(request);
```

Tipuri de date predefinite în MPI

- **Tipuri predefinite**
  - toate tipurile tipurile de bază din C *(și din FORTRAN)* plus *MPI_BYTE* și *MPI_PACKED*.

Tip MPI	Tip echivalent în C
MPI_CHAR MPI_SHORT MPI_INT	signed char signed short int signed int
MPI_LONG MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT	signed long int unsigned char unsigned short int
MPI_LONG MPI_UNSIGNED_CHAR MPI_UNSIGNED_SHORT	unsigned int unsigned long int unsigned short int
MPI_DOUBLE MPI_LONG_DOUBLE	double long double

Tipuri de date derivate în MPI

- Motivație: realizarea unei modalități eficiente de a comunica tipuri *non-contigue* sau *diferite* într-un mesaj
- Formal, standardul MPI definește tipul de date general alcătuit din:
  - o secvență de tipuri de bază
  - o secvență de deplasări (deplasările au o valoare întreagă)
- Harta unui tip (*typemap*) este folosită pentru o exprimare facilă:
  - Harta tipului = secvență de perechi (tip, deplasare)
  - $\boldsymbol{Typemap} = \{(\boldsymbol{type_0}, \boldsymbol{displacement_0}), \ \dots, (\boldsymbol{type_{n-1}}, \boldsymbol{displacement_{n-1}})\}$
- Semnătura tipului (*typesig*) = secvența tipurilor
  - $\boldsymbol{Typesig} = \{\boldsymbol{type_0}, \boldsymbol{type_1}, \dots, \boldsymbol{type_{n-1}}\}$
- Titlul tipului (*handle*)
  - Exemplu: MPI_INT, cu harta $Typemap = \{(int, 0)\}$

Tipuri de date derivate în MPI

- Etapele necesare pentru a utiliza un tip derivat:
  - construirea tipului (utilizând o serie de constructori speciali)
  - alocarea sa (înregistrarea folosind *MPI_Type_commit*)
  - la final, tipul va fi dezalocat (*MPI_Type_free*)

- Exemple de constructori:
  - **MPI_Type_contiguous** (count, oldtype, *newtype)
  - **MPI_Type_vector** (count, blocklength, stride, oldtype, *newtype)
  - **MPI_Type_indexed** (count, blocklens[], offsets[], old_type, *newtype)
  - **MPI_Type_struct** (count, blocklens[], offsets[], old_types[], *newtype)

Tipuri de date derivate în MPI - constructori

- **MPI_Type_contiguous** (count, oldtype, *newtype)
  - Se produce un nou tip de date prin copierea *oldtype* de *count* ori

- Exemplu:

```
count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);

MPI_Send(&a[2][0], 1, row_type, dest, tag, comm);
```

Tipuri de date derivate în MPI - constructori

- **MPI_Type_vector** (count, blocklength, stride, oldtype, *newtype)
  - Se vor lua *count* blocuri de lungime *blocklength*
  - Un element din bloc este de tipul *oldtype*
  - Blocurile pot fi despărțite de zone care vor fi sărite (*strides*)

- Exemplu:

```
float mesh[10][20];
int dest, tag;
int count = 10;                /* numărul de blocuri */
int blocklen = 1;              /* o singură coloană */
int stride = 19;               /* salt de 20 de elemente */

MPI_Datatype newtype;
MPI_Type_vector(count,blocklen,stride,MPI_FLOAT,&newtype);
MPI_Type_commit(&newtype);
MPI_Send(&mesh[0][19], 1, newtype, dest, tag, MPI_COMM_WORLD);
```

Tipuri de date derivate în MPI - constructori

- **MPI_Type_struct** (count, blocklens, offsets, oldtypes, *newtypes)
  - Se vor lua *count* blocuri de lungime *blocklength*
  - Un element din bloc este de tipul *oldtype*
  - Fiecare bloc se găsește la deplasamentul corespunzător din *offsets*
- Exemplu: un apel **MPI_Type_struct** cu *count*=2, *blocklens[0]*=1, *old_types[0]*=MPI_INT, *blocklens[1]*=3 şi *old_types[1]*=MPI_DOUBLE

MPI_type_struct - exemplu

```
1   # define MAX_PART 1000
2   Struct Part_struct
3     { int class;        /* clasa particulei */
4       double d[6];      /* coordonatele particulei */
5       char b[7];        /* alte informații */
6     };
7   struct Part_struct particle[MAX_PART];

8   /* construcția tipului care descrie structura MPI a unei particule */
9   MPI_Datatype PartType;
10  MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR}; /* semnătura */
11  int blocklen [3] = { 1, 6, 7};
12  MPI_Aint disp[3];
13  int base;
```

MPI_type_struct - exemplu

```
14  /* se determină adresele absolute pentru a calcula ulterior deplasările
15   * MPI_Address este folosit pentru portabilitate
16   */
17  MPI_Address (particle, disp);
18  MPI_Address (particle[0].d, disp+1);
19  MPI_Address (particle[0].b, disp+2);
20
21  base=disp[0];
22  /* deplasările reprezintă adresele relative */
23  for (i=0; i<3; i++) disp [i] -= base;
24  MPI_Type_create_struct ( 3, blocklen, disp, type, &PartType);
25  MPI_Type_commit (&PartType);
26
27  /* se transmite întregul tablou */
28  MPI_Send (particle, MAX_PART, PartType, dest, tag, comm);
```

Topologii virtuale

- O topologie este un mecanism prin care se obține o modalitate facilă de a *numi* procesele dintr-un grup (comunicator).
- În cazul multor aplicații distribuite, simpla folosire a rangurilor nu reflectă modelul logic de comunicare între procese.
- Procesele pot fi văzute ca făcând parte dintr-un:
  - grid sau structură carteziană (două/trei/... dimensiuni)
  - graf (model generic)
- Trebuie făcută distincția între:
  - topologia virtuală
  - topologia hardware-ului
- Topologia virtuală poate fi concepută astfel încât să respecte modul în care procesele sunt repartizate procesoarelor (topologia hardware), pentru o sporire a performanței comunicării dintre procese.

Topologii virtuale

- `MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm comm_cart)`
- Funcția creează un nou comunicator în care topologia este una carteziană, conform parametrilor (procesele implicate sunt cele din *comm_old*):
  - *ndims* – numărul de dimensiuni din structura carteziană
  - *dims* – un vector de dimensiune *ndims* care specifică numărul de procese de pe fiecare dimensiune
  - *periods* – un vector cu valori de tip boolean care indică prezența/absența periodicității pe o anumită dimensiune
  - *reorder* – valoare booleană care permite (sau nu) MPI-ului să reordoneze procesele din noul comunicator în funcție de proximitățile fizice ale procesoarelor pe care rulează procesele (topologia hardware)

Exemple de grile carteziene 2D
Grilă neperiodică

Rang
(linie,coloană)

Exemple de grile carteziene 2D
Grilă periodică pe linii

Exemple de grile carteziene 2D
Grilă periodică pe coloane

Primitive pentru topologii carteziene

- Furnizare *dims*, *periods* și *coords* pentru procesul apelant:

```
int MPI_Cart_get(MPI_Comm comm, int maxdims,
                 int *dims, int *periods,
                 int *coords)
```

- Aflare *coords* pentru un proces cu rang cunoscut:

```
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int maxdims, int *coords)
```

- Pentru a determina sursa și destinația care vor fi folosite într-un apel ulterior *MPI_Sendrecv* (sursa și destinația pe aceeași direcție):

```
int MPI_Cart_shift(MPI_Comm comm, int direction,
                   int displ, int *source, int *dest)
```

Obs: sursa = rang proces apelant – deplasament

destinația = rang proces apelant + deplasament

Grilă periodică pe toate dimensiunile (tor) MPI_Cart_shift

Proces apelant: **5**
Direcție: 1 (linia comună)
Deplasament: 2

Sursa: **3**
Destinația: **4**

Exemplu (1)

```
1  int numtasks, rank, newrank, i, j;
2  float a[SIZE][SIZE] = {1.0, 2.0, 3.0, 4.0}, { 5.0, 6.0, 7.0, 8.0 },
3                        { 9.0, 10.0, 11.0, 12.0 }, { 13.0, 14.0, 15.0, 16.0} };
4  float t;
5  MPI_Status stat;
6  MPI_Request request;
7  MPI_Comm newcomm;
8  int ndims, reorder, dim_size[2], periods[2];
9  int coords[2], source, dest;
10 MPI_Init(&argc,&argv);
11 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
13 ndims = 2; /* numărul de dimensiuni */
14 dim_size[0] = 4; /* numărul de procese pe prima dimensiune */
15 dim_size[1] = 4; /* numărul de procese pe cea de-a doua dimensiune */
16 periods[0] = 1; /* dimensiune ciclică */
17 periods[1] = 0;
18 reorder = 1;
19 MPI_Cart_create(MPI_COMM_WORLD, ndims, dim_size, periods, reorder, &newcomm);
20 MPI_Comm_rank(newcomm, &newrank);
21 MPI_Cart_coords(newcomm, newrank, 2, coords);
22 printf("Rang=%d Pozitia=(%d,%d)\n", newrank, coords[0], coords[1]);
23 MPI_Barrier(MPI_COMM_WORLD);
24 MPI_Cart_shift(newcomm, 0, 1, &source, &dest);
25 printf("Procesul %d: trimit catre %d si primesc de la %d\n", newrank, dest, source);
```

Exemplu (2)

```
26  if (coords[0] % 2 == 0) {
27      MPI_Send(&a[coords[0]][coords[1]], 1, MPI_FLOAT, dest, 0, newcomm);
28      MPI_Recv(&t, 1, MPI_FLOAT, source, 0, newcomm, &stat);
29  } else {
30      MPI_Recv(&t, 1, MPI_FLOAT, source, 0, newcomm, &stat);
31      MPI_Send(&a[coords[0]][coords[1]], 1, MPI_FLOAT, dest, 0, newcomm);
32  }
33  /* se pune noul element pe poziția corespunzătoare */
34  a[coords[0]][coords[1]] = t;
35  MPI_Barrier(MPI_COMM_WORLD);
36  if (newrank == 0) {
37      for (i=0; i<4; i++) {
38          for (j=0; j<4; j++) {
39              coords[0] = i; coords[1] = j;
40              MPI_Cart_rank(newcomm, coords, &rank);
41              if (rank != 0) {
42                  MPI_Recv(&a[j][i], 1, MPI_FLOAT, rank, 0, newcomm, &stat);
43              }
44              printf("%.2f ", a[j][i]);
45          }
46          printf("\n");
47      }
48  } else {
49      MPI_Send(&a[coords[0]][coords[1]], 1, MPI_FLOAT, 0, 0, newcomm);
50  }
51  MPI_Finalize();
```

Comunicare colectivă

- Comunicarea colectivă se referă la acele funcții din MPI care implică **toate procesele** din comunicator.
- Caracteristici:
  - O operație colectivă presupune apelul funcției corespunzătoare de către toate procesele din comunicator, folosind argumente potrivite.
  - Dimensiunea datelor trimise trebuie să **coincidă** cu dimensiunea datelor primite.
  - Operațiile colective se pot executa doar în **modul blocant**.
- Operațiile de comunicare colectivă se împart în:
  - Operații de sincronizare
  - Operații de transfer de date
  - Calcule colective (de reducere)

Operații de sincronizare
Sincronizarea cu barieră

- `int MPI_Barrier(MPI_Comm comm)`

- MPI_Barrier sincronizează toate procesele din comunicator
- Un proces apelant se blochează până când toate celelalte procese din comunicator apelează, la rândul lor, această funcție

Operații de transfer de date

- Operații de transfer de date:
  - **Difuzarea** (*broadcast*)
    - propagarea unui mesaj de la *rădăcină* la restul proceselor
  - **Colectarea** (*gather*)
    - procesul *rădăcină* colectează date de la celelalte procese
  - **Distribuirea** (*scatter*)
    - inversa operației de colectare
  - Variații ale colectării și distribuirii (*allgather*, *alltoall*)
    - *allgather*
      - fiecare proces colectează datele de la celelalte procese
    - *alltoall*
      - fiecare proces realizează o distribuire a datelor, în funcție de rangul celorlalte procese

Operații de transfer de date

Difuzare

P_0	A			
P_1	A			
P_2	A			
P_3	A			

Distribuire

P_0	A	B	C	D
P_1	B			
P_2	C			
P_3	D			

Colectare

P_0	A	B	C	D
P_1	B			
P_2	C			
P_3	D			

Operații de transfer de date

Allgather

P_0	A	B	C	D
P_1	B	A	C	D
P_2	C	A	B	D
P_3	D	A	B	C

Alltoall

P_0	A_0	B_0	C_0	D_0
P_1	A_1	B_1	C_1	D_1
P_2	A_2	B_2	C_2	D_2
P_3	A_3	B_3	C_3	D_3

Calcule colective (operații de reducere)

- ```
  int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
                 MPI_Datatype datatype, MPI_Op op,
                 int root, MPI_Comm comm)
  ```

$\boldsymbol{P_0}$ *sendbuf*	A_0	A_1	A_2	A_3
$\boldsymbol{P_1}$ *sendbuf*	B_0	B_1	B_2	B_3
$\boldsymbol{P_2}$ *sendbuf*	C_0	C_1	C_2	C_3
$\boldsymbol{P_3}$ *sendbuf*	D_0	D_1	D_2	D_3

MPI_Op = MPI_SUM

Rezultat MPI_Reduce: $\boldsymbol{P_0}$ *recvbuf* va conține suma celor 4 vectori

$\boldsymbol{P_0}$	$A_0 + B_0 + C_0 + D_0$	$A_1 + B_1 + C_1 + D_1$	$A_2 + B_2 + C_2 + D_2$	$A_3 + B_3 + C_3 + D_3$
$\boldsymbol{P_1}$				
$\boldsymbol{P_2}$				
$\boldsymbol{P_3}$				
  ```

Calcule colective (operații de reducere)
Exemplu – aproximare valoare PI

- $\int_0^1 \frac{1}{1+x^2}\, dx = \frac{\pi}{4}$
- Suma Riemann: $\sum_{k=1}^{n} f(c_k)\,(x_k - x_{k-1})$

- Soluție secvențială:

```
for (i = 1; i <= n; i++) {
   x = h * ((double)(i) - 0.5);
   sum = sum + f(x);
}
pi = h * sum;
```

Calcule colective (operații de reducere) Exemplu – aproximare valoare PI

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
while (1) {
    if (myid == 0) {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) {
        break;
    } else {
        h   = 1.0 / (double) n;
        sum = 0.0;
        for (i = myid + 1; i <= n; i += numprocs) {
            x = h * ((double)i - 0.5);
            sum += (4.0 / (1.0 + x*x));
        }
        mypi = h * sum;
        MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0) {
                printf("pi is approximately %.16f\n", pi);
        }
    }
}
```

Sumar

- *Message Passing Interface*
  - Elemente introduse
  - Comunicarea punct – la – punct
  - Tipuri de date
  - Topologii de procese
  - Comunicare colectivă

Algoritmi Paraleli și Distribuiți
Toleranta la defecte.

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Obiectiv

- Asigurarea tolerantei la defecte
  - Solutii de detectie a erorilor in SD
  - Solutii de recuperare din defect in SD

"You know you have one when the crash of a computer you've never heard of stops you from getting any work done."

Leslie Lamport, 1993 – defectul in SD

Introducere

- Un sistem distribuit (SD) se defineste ca avand 6 caracteristici importante [Coulouris, et al, 1994]
  - Suportul pentru partajarea resurselor
  - Deschidere
  - Concurenta
  - Scalabilitate
  - Toleranta la defecte
  - Transparenta
- **Problema:** funcţionarea corectă în prezenţa unor defecţiuni ale sistemului
- Sistemele distribuite sunt compuse din multe componente hardware si software => probabilitate mai mare de aparitie a unor defecte in diverse parti

Eroare?

error

An error has occurred.

We apologize for this inconvenience.

RegOnline has been notified and will work to fix this problem

Please try the following:

- Click the back button and try again.
- Close your browser and try again.

Could you help us eliminate this error?...

Please click here to describe what you were doing when the error occurred.

Toleranta la defecte

- Problemele: greseli de functionare, erori, defecte

- **Toleranta la defecte**:
  - Evitarea greselilor in functionare chiar in prezenta defectelor .
  - Utillizatorii nu resimt defectarea unor componente.
- Doua abordari :
  - Redundanta Hardware
  - Recuperarea Software

Toleranta la defecte

- Aceeaşi ieşire pentru aceleaşi valori de intrare,
- Valori apropiate la ieşire, pentru valori de intrare apropiate

Tipuri de defecte în sistemele distribuite

- Crash: la un moment dat un procesor devine nefuncţional
  - În ultimul pas, înainte de a se defecta, se poate să fi trimis numai un subset din mesaje
- Byzantine: procesorul intră în stări arbitrare şi trimite mesaje conţinând conţinut arbitrar

Clasificarea defectelor si anomaliilor

Toleranta la defecte in sistemele distribuite

- Exemple:

COMPONENTA	DEFECT
Unitate de procesare (CPU)	Caderi sau defecte bizantine
Reteaua de comunicatie	Toate tipurile de defecte
Ceasurile	Temporizari, omisiuni, defecte bizantine
Unitate de stocare (disc)	Temporizari, omisiuni, caderi
Software	Software Toate tipurile de defecte

- Din perspectiva tolerantei la defecte, sistemele distribuite au un avantaj major: *se poate introduce usor o redundanta in sistem* deoarece imperfectiunile si posibilitatea unor defecte fizice nu pot fi ignorate.
- Probleme:
  - Defecte la nivelul nodurilor din sistemul distribuit.
  - Defecte la nivelul procesului de comunicatie.
  - Intarziere in procesul de transmisie al informatiei.

Toleranta la defecte in sistemele distribuite (2)

- Defecte la nivelul nodurilor din sistemul distribuit.
  - Defecte independente si diferite pe fiecare nod.
- Defecte la nivelul procesului de comunicatie.
  - *Defectiune hardware* a mediului de transmisie -> oprirea comunicatiei, partitionarea sistemului in mai multe sisteme.
  - *Intermittent failure*: mesajele pierdute, schimbate ordinea sau duplicate.
- Intarziere in procesul de transmisie al informatiei.
  - Nu este un defect, dar poate conduce la defecte.
  - Doua situatii frecvent intalnite:
    - *Jitter (variable delay)* -> timpul in care un mesaj ajunge la destinatie.
    - *Efectul relativistic* -> site-uri diferite pot primi aceleasi mesaje, dar in alta ordine.

Fazele tolerantei la defecte

- Proiectarea si implementarea unui sistem distribuit tolerant la defecte este dependenta de arhitectura sistemului si de functionalitatea sa. Prin urmare nu exista reguli generale pentru a adauga caracteristici de toleranta la defecte intr-un sistem, insa putem formula unele principii care trebuie sa urmeze patru faze:
  - Detectia erorilor;
  - Estimarea si izolarea zonelor afectate;
  - Recuperarea erorilor;
  - Tratarea defectelor.

Arhitectura generica de FT

Detectia erorilor

- Explozia de mesaje
- Scalabilitate
- Pierderi de mesaje
- Flexibilitate
- Dinamism
- Securitate

Detectoare

- Nivelul de FD este in general asigurat de detectoare locale dedicate

Problema celor două armate

- Puteți să vă gândiți la un algoritm?

Scheme de detectie

- Strategia Heartbeat – traditionala, (inca) cea mai intalnita
- Problema – scalabilitatea
  - Solutia 1) Structuri ierarhice de FD – Two-level Globus FD
  - Solutia 2) Protocoale de tip Gossip – cu o mare probabilitate, eventual toate procesele obtin aceleasi informatii
- Protocoale Adaptive
  - Se adapteaza dinamic la conditiile de trafic peste retea
  - E.g.: Un protocol ce adapteaza valoarea intervalului de timeout in functie de valoarea maxima a timpului de raspuns al mesajelor de heartbeat
- Detectoare de erori lenese
- Detectoare de erori progresive

Protocoale de detectie adaptive

- Asigura in aceeasi masura adaptarea la:
  - schimbarile conditiilor de retea (conditii diverse de trafic)
  - cerintele si schimbarile de context ale aplicatiilor
- La baza tot strategia *heartbeat,* insa intervalul de timeout este modificat dinamic conform conditiilor de retea.

Exemple FD adaptive (1)

- FETZER-FD
  - Intervalul de timeout este calculat ca fiind maximul dintre timpii de receptionare a mesajelor heartbeat.
  - In conditii reale:

1. in retele mari probabilitatea de pierdere a mesajelor este ridicata.
2. nu exista o limita superioara in ceea ce priveste intarzierile de mesaje.

Exemple FD adaptive (2)

- CHEN-FD
  - Se bazeaza pe analiza probabilistica a traficului de retea si a cerintelor aplicatiilor.
  - Estimarea timpului de primire a urmatorului mesaj heartbeat pe baza timpilor de receptionare a celor mai recente mesaje heartbeat.
  - Estimarea anterioara plus o margine de siguranta formeaza intervalul de timeout.

Dezavantaje

- Acuratetea este data de functia de estimare.
- Adaptarea la cerintele mai multor aplicatii ar necesita administrarea mai multor valori de timeout.
- O solutie alternativa -- aplicatiile sa-si seteze propria valoare de timeout
  - dar se invalideaza adaptarea la conditiile de retea.

Detectoare de erori lenese

- Ideea de baza este folosirea a cat mai multe mesaje transmise intre procese pentru a detecta erori. Se impune, astfel, ca fiecare mesaj primit sa fie confirmat.
- Cand procesele nu comunica, se folosesc mesaje de monitorizare.

Detectoare de erori progresive

- Se bazeaza pe asocierea unui proces cu o valoare progresiva, care creste monoton in timp daca procesul s-a defectat.
- Fiecare aplicatie are un prag finit stabilit conform propriilor cerinte.
- O aplicatie decide daca suspecteaza sau nu un proces comparand pragul propriu cu valoarea progresiva asociata procesului.

Detectoare de erori progresive

FD bazat pe algoritmi progresiv si cei de gossiping

- **Comunicaţie**

  Exploatează abordarea ierarhică, dar se bazează pe o topologie dinamică

- **Monitorizare**

  Foloseşte strategia heartbeat

- **Gossiping**

  Utilizează algoritmul de basic gossiping şi un algoritm optimizat de diseminare a informaţiei (Round-Robin)

- **Interpretare date de monitorizare**

  Are la bază o transformare olomorfă şi metoda de predicţie EMA combinată cu un algoritm de ajustare adaptivă a factorului de netezire

- **Api aplicaţie**

  Oferă acces la serviciile modulului de detecţie

Clusterizare

Interpretare date monitorizare (1)

Interpretare date monitorizare (2)

$$0 <= sl_{qp}(t) < 1 \mapsto sl_{qp}(t) = \left|\frac{t-1}{t+1}\right|,$$

$$t = \frac{t_{now}}{t_{pred}} \in (0, \infty),$$

t_{now} *timpul curent*,

t_{pred} *ultima valoare prezisa*

Gossiping (1)

Grup de *gossip*
Agent 0
Agent 1
Agent 2
Agent 3
Agent 4
Agent 5
Grup de *gossip*
Agent 0
Agent 1
Agent 2
Agent 3
Agent 4
Agent 5
Grup de *gossip*
Agent 0
Agent 1
Agent 2
Agent 3
Agent 4
Agent 5
Runda 1
Runda 2
Runda 3
Runda 4
Runda 5
Lista *gossip* locală
serviceID
suspicionLevel1
time1
+
Lista *gossip* primită
serviceID
suspicionLevel2
time2
Lista *gossip* actualizată
serviceID
(time1 < time2) ?
min(suspicionLevel1, suspicionLevel2) :
suspicionLevel1
time

Gossiping (2)

Cerere broadcast
informaţie gossip
A9
A1
A10
A4
A7
A3
A8
A2
A6
A5

Lista de obiecte monitorizate
Lista *gossip* difuzată
Obiect
monitorizat
+
serviceID
suspicionLevel
time
Actualizare nivel suspiciune pe
baza zvonurilor primite
Notificare aplicaţii
Verificare nivel suspiciune

Testare şi validare – Scenariul 1

Testare şi validare – Scenariul 2

Recuperarea din eroare

- Realizata prin “readucerea” aplicatiei la o stare corecta, de dinainte de aparitia defectului
- Gestiunea erorilor:
  - Rollback – stare salvata: checkpoint
  - Rollforward – urmatoarea stare fara eroare reprezinta noua stare
  - Compensare – mascarea erorii prin redundanta

Solutii de recuperare din eroare

- Abordarea tipica FT: sistemul asteapta aparitia unui defect si reactioneaza la detectarea acestuia pentru a-l corecta
- Daca defectele se propaga mai repede decat pot mecanismele de recuperare sa le trateze se poate ajunge la situatia cand nu se mai pot aplica tehnicile de recuperare
- Mecanismele de recuperare proactiva:
  - Cresterea rezistentei sistemului la posibilele propagari nedorite ale erorilor
  - Initierea recuperarii inainte sau concurent cu recunoasterea problemei din sistem
  - Adaptarea sistemului

FT pentru aplicatiile bazate pe schimburi de mesaje

- Re-executia intregii aplicatii
- Pierderea tuturor calculelor

Checkpoint

- Snapshot al unui proces
- Starea salvata pe medii de stocare persistente

Checkpoint

- p2 a trimis un mesaj ce nu a fost trimis de p1
- Este nevoie de salvea unei stari globale, consistente, a sistemului

Checkpointing Distribuit

Independent

- ☑ Simplitate
- ☑ Autonomie
- ☑ Scalabilitate
- ✖ Efectul de Domino

Coordonat

- ☑ Stari Consistente
- ☑ Performanta Buna
- ☑ Garbage Collection
- ✖ Scalabilitate

Indus de comunicatie

- ☑ Stari Consistente
- ☑ Autonomie
- ☑ Scalabilitate
- ✖ Performanta

Logarea Mesajelor

- Logarea datelor pe dispozitive persistente de stocare in timpul executiei fara erori
- Folosirea acestei informatii pentru recuperarea din eroare

- ***Orfan:*** procesul ce depinde de o stare ne-recuperabila a unui proces esuat

Procese orfan?

- Dupa ce procesul p trimite mesajul m, tot el trimite mesajul m' catre procesul q, care la randul lui il trimite mai departe.
- Daca mesajul m nu era logat, atunci defectarea lui p ar putea conduce la pierderea oricarei informatii despre m.
- La reinitializarea lui p, el poate ajunge intr-o stare in care a doua oara nu mai trimite mesajul m' lui q.
- In acest caz procesul q nu ar mai fi consistent cu procesul p, pentru ca q a trimis un mesaje m' ce nu a fost niciodata trimis de procesul p.
- q este un proces orfan.

Abordari Message Logging

Pesimista

- ☑ Nu exista procese orfan
- ☑ Recuperare usoara
- ✖ Performante scazute

Optimista

- ☑ Non-blocanta
- ✖ Procese orfan
- ✖ Recuperare complexa

Cauzala

- ☑ Non-blocanta
- ☑ Nu exista procese orfan
- ✖ Recupera Complexa

Abordarea pesimista

- Mesajele sunt logate sincron pe mediul de stocare.
- Overhead asupra comunicarii.

Abordare optimista

- Mesajele sunt logate asincron pe mediul de stocare.
- Protocolul de recuperare poate fi complex (risc de aparitie al proceselor orfane).

Recovery Oriented Computing

- Ideea: defectele hardware, software si erorile de operare sunt aspecte ce trebui tratate, nu prevenite.
- Axate pe MTTR:
  - Reducerea timpului de recuperare
  - Marirea disponibilitatii sistemului
- Repornirea sistemului dupa defect
  - Solutia naiva: reboot
  - Repornirea doar a unor parti ale sistemului: sisteme modulare, in care componentele pot fi repornite independent: serviciul din Unix /etc/rc
- Stateful recovery
  - Recuperarea din defect in cazul bazelor de date
  - Folosirea metodelor de checkpointing

Cauzele de Downtime: leader = operatorul
Efectul in 5 studii de caz a 6 tehnici ROC:

Public Switched Telephone Network

Average of Three Internet Sites

ROC technique	Mercury RR	FIG	Email with Undo	Pinpoint	ROC-1
Partitioning	√				√
Recovery experiments	√	√	√	√	√
Reversible systems			√		
Diagnosis aid				√	√
Independent mechanisms	√		√		√
Redundancy and fail fast					√

Dependable Service Engineering

- Arhitectura bazata pe folosirea de containere de servicii tolerante la defecte

  - Tehnici de toleranta la defecte suportate:
    - Rewind and retry
    - Resubmiterea catre implementari alternative
    - Executie pe mai multe versiuni cu votare

Voting Service Container
SOAP
Weather Service Wrapper
HTTP
BBC Weather Website
HTTP GET
Weather Service Wrapper
Met check Weather Website
SOAP
Web Browser Client
Weather Service Wrapper
Accuweather Website
Invokes
Weather Service Wrapper
Weather Channel Website
Weather Service Wrapper
Weather Online Website
Voting Visualisation

Dependable ≠ unaffordable

- Toleranta la defecte prin virtuzalizare
- Loosely Synchronized Redundant Virtual Machines (LSRVM)

Algoritmi Paraleli și Distribuiți
Teorema CAP.

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE

Teorema CAP

- Propusă de Prof. Eric Brewer la PODC (Principle of Distributed Computing) 2000
- Descrie *trade-off-urile* sistemelor distribuite
- Este imposibil pentru un serviciu web să furnizeze *trei garanții* în același timp:
  - **Consistency**
  - **Availability**
  - **Partition-tolerance**

Teorema CAP

- **C**onsistency (Consistență):
  - Toate nodurile ar trebui să vadă aceleași date în același timp
- **A**vailability (Disponibilitate):
  - Defectarea unui nod nu ar trebui să oprească operarea corectă a sistemului (doar) cu nodurile rămase
- **P**artition-tolerance (Toleranță la partiționare):
  - Sistemul continuă operarea cf. cu partiționarea rețelei
- Un sistem distribuit poate satisfice oricare două garanții, dar niciodată **toate trei împreună**

O mica precizare (diff. între Reliability și Availability)

- Diferența dintre Fiabilitate și Disponibilitate
  - Un sistem pica câte o milisecundă la fiecare oră
    - highly available (99.9%) , but highly unreliable
  - Un sistem care nu pica vreodată, dar are nevoie de 1 zi/lună pentru mentenanță
    - highly reliable, not highly-available

Teorema CAP

Teorema CAP

- Un exemplu:

Hotel Booking: are we double-booking the same room?

Teorema CAP

- Un exemplu:

Hotel Booking: are we double-booking the same room?

Teorema CAP

- Un exemplu:

Hotel Booking: are we double-booking the same room?

Teorema CAP: Demonstratie

- 2002: Demonstratia furnizata de Nancy Lynch si Seth Gilbert la MIT

Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." ACM SIGACT News 33.2 (2002): 51-59.

Teorema CAP: Demonstratie

- O demonstratie simpla folosind doua noduri:

Teorema CAP: Demonstratie

- O demonstratie simpla folosind doua noduri:

Teorema CAP: Demonstratie

- O demonstratie simpla folosind doua noduri:

Teorema CAP: Demonstratie

- O demonstratie simpla folosind doua noduri:

De ce e important?

- Viitorul bazelor de date consta in **distribuirea** lor (Big Data Trend, etc.)
- Teorema CAP descrie **trade-off-uri** implicate de sistemele distribuite

Scalabilitatea Bazelor de Date Relationale

- Bazele de Date Relaționale sunt construite pe principiul **ACID** (Atomicity, Consistency, Isolation, Durability)
- Implică că o bază de date relațională cu adevărat distribuită ar trebui să aibă toate proprietățile de **availability, consistency, partition tolerance**.
- Ceea ce e **imposibil** …

Ce oferă Teorema CAP

- Din următoarele trei garanții de performanță oferite:
  - Consistency
  - Availability
  - Partition tolerance
- Alegeți două
- Ceea ce înseamnă că sunt trei tipuri de sisteme distribuite:
  - CP
  - AP
  - CA

Vreo problemă?

O opinie greșită: 2 din 3

- Ce spuneți de CA?
- Poate un sistem distribuit (peste o rețea unreliable) să nu fie tolerantă la partiționare?
  - Toleranța la defecte?
  - Dacă tot timpul am nevoie de A, și A se defectează, pot pierde datele?
  - Cu ce diferă atunci sistemul de unul centralizat?

Ce spun alții...

- Coda Hale, Yammer software engineer:
  - “Of the CAP theorem’s Consistency, Availability, and Partition Tolerance, **Partition Tolerance is mandatory in distributed systems**. You cannot not choose it.”

http://codahale.com/you-cant-sacrifice-partition-tolerance/

Ce spun alții...

- Werner Vogels, Amazon CTO
  - “An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, **consistency and availability cannot be achieved at the same time**.”

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

Teorema CAP, 12 ani mai târziu

- Prof. Eric Brewer: părintele teoremei CAP
  - “Formularea “2 din 3” este greșită deoarece simplifică mult tensiunile dintre proprietăți...
  - **CAP interzice doar o mică parte din proiectarea distribuită**: disponibilitatea și consistența perfecte în prezența partițiilor, care sunt oricum rare.”

http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Consistență sau Disponibilitate

- Consistența și Disponibiltatea nu sunt decizii “binare”
- Sisteme AP relaxează consistența în favoarea disponibilității – dar nu sunt inconsiste prin definiție
- Sisteme CP sacrifice disponibilitatea pentru consistență – dar nu sunt indisponibile
- Aceasta sugerează că sisteme atât AP cât și CP, pot oferi un grad de consistență și disponibilitate, precum și toleranță la partiționare

AP: Best Effort Consistency

- Exemplu:
  - Web Caching
  - DNS
- Abordări ”de mijloc”:
  - Optimistic
  - Expiration/Time-to-live
  - Conflict resolution

CP: Best Effort Availability

- Exemple:
  - Majority protocols
  - Distributed Locking (Google Chubby Lock service)
- Abordări ”de mijloc”:
  - Pessimistic locking
  - Make minority partition unavailable

Tipuri de Consistență

- Strong Consistency
  - După ce actualizările au loc, **orice acces ulterior** va returna **aceeași** valoare actualizată
- Weak Consistency
  - **Nu există garanții** că accese successive vor returna valoarea actualizată
- **Eventual Consistency**
  - O formă specifică a weak consistency
  - Oferă garanții că dacă nu se fac **noi actualizări** asupra obiectelor **la un moment dat** (eventually) toate replicile vor returna valoarea actualizată (e.g., propagarea update-urilor către replici într-o manieră lazy)

Variații ale Eventual Consistency

- Causal consistency
  - Procesele aflate sub o relație cauzală vor vedea date consistente
- Read-your-write consistency
  - Un process ce va accesa datele anterior scrise de el nu va vedea vreodată o valoare neactualizată
- Session consistency
  - Pe durata unei sesiuni, sistemul garantează o consistență read-your-write
  - Garanții oferite între sesiuni ce nu se suprapun

Variații ale Eventual Consistency

- Monotonic read consistency
  - Dacă un process a văzut o valoare particulară a unei date, orice procese subsecvente nu vor returna valori mai vechi
- Monotonic write consistency
  - Sistemul garantează serializarea scrierilor de către *același* proces
- În realitate
  - Orice combinație de aceste proprietăți poate fi garantată
  - Monotonic reads și read-your-writes sunt cele mai dorite pentru sistemele distribuite

Eventual Consistency
- Un exemplu Facebook

- Bob are o știre interesantă pe care o partajează cu Alice pe wall-ul Facebook
- Bob o roagă pe Alice să verifice ce a postat
- Alice se loghează, verifică wall-ul Facebook și... :

Și nu regăsește nimic acolo!

Eventual Consistency
- Un exemplu Facebook

- Bob o roagă pe Alice să aștepte un pic și să verifice mai încolo
- Alice așteaptă un minut și se reloghează:

Și găsește postarea pe care Bob a partajat-o cu ea!

Eventual Consistency
- Un exemplu Facebook

- Motivul: este posibil ca Facebook să folosească modelul **eventual consistent**
- De ce alege Facebook modelul eventual consistency în locul unuia de tip strong consistency?
  - Facebook are peste 1 miliard de utilizatori activi
  - Nu este trivial să stocăm eficient și fiabil o cantitate imens de mare de date generată în orice moment de timp
  - Modelul de consistență eventuală oferă opțiunea **reducerii încărcării și crește disponibilitatea serviciilor oferite**

Eventual Consistency
- Un exemplu Dropbox

- Dropbox oferă consistență imediată prin intermediul sincronizării
- Totuși, ce se întâmplă când apare o partiționare peste rețea?

Eventual Consistency
- Un exemplu Dropbox

- Faceți un experiment:
  - Deschideți un fișier în folderul Dropbox
  - Deconectați rețeaua (e.g., WiFi, 4G)
  - Încercați să editați fișierul în folderul Dropbox: puteți?
  - Re-activați conexiunea la rețea: ce se întâmplă cu folderul Dropbox?

Eventual Consistency - Exemplu Dropbox

- Dropbox se bazează tot pe eventual consistency:
  - Consistența imediată este imposibil de realizat în cazul partiționării rețelei
  - Utilizatorii ar fi nemulțumiți dacă documentele lor Word devin neutilizabile de fiecare data când apasă Ctrl+S , datorită simplei latențe în actualizarea tuturor dispozitivelor peste WAN
  - Dropbox se orientează spre conceptul de **personal syncing**, nu pe colaborare, așa încât asta nu consistența eventuală în cazul lor nu e chiar o limitare reală.

Eventual Consistency - Un exemplu ATM

- În proiectarea unui automated teller machine (ATM):
  - Consistența puternică (strong) pare a fi o alegere naturală
  - Totuși, în practică, **A beats C**
  - O disponibilitate mărită înseamnă **încasări mai mari**
  - ATM vă permite retragerea de bani, *chiar atunci când mașina este partiționată de rețea*
  - Totuși, **se pune o limită** asupra cantității de bani pe care îi puteți retrage (e.g., $200)
  - Banca chiar vă poate suprataxa atunci când apare o retragere peste disponibil

Opțiuni în alegerea între C și A

- Un sistem de rezervare bilete de avion:
  - Când majoritatea locurilor sunt disponibile: este ok să ne bazăm pe unele date out-of-date, disponibilitatea fiind mai critică
  - Când avionul se apropie de limita de rezervare: avem nevoie de date actualizate, pentru a ne asigura că nu facem overbooking, consistența devenind mai critică

- Nu se oferă nici consistență puternică nici disponibilitate garantată, dar scopul e să creștem significant toleranța la eventualele probleme la nivelul rețelei

Eterogenitate: Segmentare între C și A

- Nu există o singură cerință uniformă
  - Unele aspecte necesită consistență strong
  - Altele necesită disponibilitate ridicată
- Segmentarea sistemelor între diverse componente
  - Fiecare furnizează diverse tipuri de garanții
- Per ansamblu sistemul nu garantează nici consistența și nici disponibiliate
  - Fiecare parte a serviciului obține exact ceea ce are nevoie
- O astfel de abordare permite partiționarea de-a lungul mai multor dimensiuni

Discuție

- Într-un sistem e-commerce (e.g., Amazon, e-Bay, etc), care sunt opțiunile de considerat când alegem între consistență și disponibilitate? Care ar fi strategia voastră?
- Hint -> Lucruri pe care le puteți lua în vizor:
  - Tipuri de date diferite (e.g., shopping cart, billing, product, etc.)
  - Tipuri diferite de operații (e.g., query, purchase, etc.)
  - Tipuri diferite de servicii (e.g., distributed lock, DNS, etc.)
  - Grupuri diferite de utilizatori (e.g., users in different geographic areas, etc.)

Exemple de scheme de partiționare

- Data Partitioning
- Operational Partitioning
- Functional Partitioning
- User Partitioning
- Hierarchical Partitioning

Exemple de Scheme de Partiționare

Data Partitioning

- Diverse date pot necesita diverse scheme de consistență și disponibilitate
- Exemplu:
  - Shopping cart: disponibilitate ridicată, timp mic de răspuns, pot suferi câteodată anomalii (clienți nefericiți în cel mai rău caz, dar vor pune lucrurile pe seama unei întâmplări)
  - Informațiile despre produse, în schimb, trebuie să fie disponibile, sunt acceptate doar mici variații în inventar
  - Checkout, billing, shipping records, toate sunt date ce trebuie ținute consistent

Exemple de scheme de partiționare

Operational Partitioning

- Fiecare operație poate necesita o abordare diferită între consistență și disponibilitate
- Exemple:
  - Reads: disponibilitate ridicată; e.g.., “query”
  - Writes: consistență ridicată, lock la writing; e.g., “purchase”

Exemple de scheme de partiționare

Functional Partitioning

- Sistemul constă dintr-o colecție de sub-servicii
- Diversele sub-servicii au nevoi diferite
- Exemplu: Un sistem distribuit complet
  - Distributed lock service (e.g., Chubby) :
    - Strong consistency
  - DNS service:
    - High availability

Exemple de scheme de partiționare

User Partitioning

- E nevoie să menținem date înrudite aproape între ele pentru garantarea performanței la procesare
- Exemplu: Craglist
  - E posibil să dorim divizarea serviciilor între câteva Data Centers,e.g., east coast / west coast
    - Utilizatorii obțin performanță ridicată (e.g., disponibilitate ridicată și consistență bună) dacă interoghează servere apropiate de ei
    - Performanță slabă dacă un utilizator din New Yor interoghează serviciul din San Francisco

Exemple de scheme de partiționare

Hierarchical Partitioning

- Servicii globale ce au “extensii” locale
- Diverse locații în ierarhie pot folosi diverse modele de consistență
- Exemplu:
  - Servere locale (mai bine conectate cu utilizatorul) oferă mai multă consistență și disponibilitate
  - Servere globale sunt mai partiționate și relaxează una dintre proprietăți

Ce se întâmplă când nu avem deloc partiționare?

- Balans între **Consistență** și **Latență**:
- cauzat de **posibilitatea de apariție a defectelor** în sisteme distribuite
  - Disponibilitate ridicată -> date replicate -> probleme de consistență
- Ideea de bază:
  - Disponibilitatea și latența sunt aproape **același lucru**: indisponibilitatea = latență f. mare
  - Atingerea unor diverse nivele de consistență/disponibilitate necesită timpi diferit

CAP -> PACELC

- O descriere mai completă a spațiului unor potențiale alegeri pentru un sistem distribuit:
  - Dacă există o **partiție (P)**, cum ajunge sistemul să aleagă între **availability și consistency (A și C)**; **altfel (else, E)**, când sistemul rulează în parametri normali în absența partiționării, cum ajunge sistemul să aleagă între **latency (L) și consistency (C)**?

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

PACELC

Partitioned

Normal

Exemple

- **Sisteme PA/EL:** Renunță la ambele C-uri pentru availability și lower latency
  - Dynamo, Cassandra, Riak
- **Sisteme PC/EC:** Refuză să renunțe la Consistență în detrimentul unui cost scump pentru disponibilitate și latență
  - BigTable, Hbase, VoltDB/H-Store
- **Sisteme PA/EC:** Renunță la Consistență atunci când apare o partiționare și păstrează Consistența doar în condiții normale de operare
  - MongoDB
- **Sisteme PC/EL:** Păstrează Consistența dacă apare o partiționare, dar renunță la consistență în cazul operării normale
  - Yahoo! PNUTS

Algoritmi Paraleli și Distribuiți
Algoritmi peer-to-peer

Prof. Ciprian Dobre
ciprian.dobre@cs.pub.ro

Peer-to-peer (P2P)

Ce este un sistem Peer-to-Peer (P2P)?

- O arhitectura de sistem distribuit in care:
  - **Nu exista un control centralizat**
  - Nodurile sunt aproape **simetrice** in functionalitate
- Numar mare de **noduri unreliable**
- **Capacitate mare de deservire a serviciilor**, prin paralismul adus de:
  - Multe discuri
  - Multe legaturi de retea
  - Multe CPI-uri
- **Absenta unui server centralizat** inseamna:
  - Mai putine sanse de overload
  - Deployment mai usor
  - Un defect nu poate strica intregul sistem
  - Sistemul per ansamblu e mai greu de atacat

Peer-to-peer (P2P)

Descentralizare!

Greu de coordonat cand nodurile intra si parasesc dinamic

Peer-to-peer (P2P)

Napster (1999)

Peer-to-peer (P2P)

Napster (1999)

Peer-to-peer (P2P)

Napster (1999)

Problema – server centralizat pentru indexarea datelor

This article is more than **19 years old**

Napster loses net music copyright case

A federal judge in San Francisco yesterday ordered Napster, the internet service that allows the trading of MP3 sound files by linking personal computers, to stop permitting the exchange of copyrighted music owned by major music labels.

The preliminary injunction, which could effectively shut down Napster's service, represents a major victory for the world's five largest record companies who are suing Napster for copyright infringement.

In granting the industry's request for a preliminary injunction, Chief Judg Marilyn Patel of the US district court in San Francisco ruled that the huge popular Napster service, with an estimated 20m users, was used primaril trade copyrighted material with the full knowledge of Napster.

"A majority of Napster users use the service to download and upload copyrighted music," she said.

The injunction is scheduled to take effect at midnight on Friday and will continue pending trial.

Advertisement

Spread your content, not your workflow.

Metallica sued Napster 15 years ago today

File sharing forever

By Nilay Patel | @reckless | Apr 13, 2015, 10:39am EDT

SHARE

April 13th, 2000. The day the music industry and the internet became best frenemies forever.

That's the day *Metallica v. Napster, Inc.* was filed in the United States District Court for the

Peer-to-peer (P2P)

Napster (1999)

Problema – server centralizat pentru indexarea datelor

Solutia - Distributed hash table

gnutella

KaZaA

limewire

Distributed Hash Tables (DHT)

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Distributed Hash Tables (DHT)

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Distributed Hash Tables (DHT)

Proprietati dorite:

- Descentralizare
- Load-balancing
- Scalabilitate
- Disponibilitate

Distributed Hash Tables

- *Distributed Hash Table:*

```
key = hash(data)
lookup(key) → IP addr
send-RPC(IP address, put, key, data)
send-RPC(IP address, get, key) → data
```

(Chord lookup service)

- **Partitionam datele** peste sisteme distribuite pe scara larga
  - Punem tupluri intr-un global database engine
  - si blocurile de date intr-un sistem global de fisiere
  - peste un P2P file-sharing system

*RPC = Remote Procedure Call

BitTorrent peste DHT

- BitTorrent poate folosi DHT in loc de (sau in conjuctie cu un) tracker

- Clientii BT folosesc DHT:
  - Key = **file content hash** (“infohash”)
  - Value = Adresa **IP a peer-ului** ce poate servi fisierul
    - Poate stoca mai multe valori (*i.e.* IP addresses) pentru o cheie

- Clientul apeleaza:
  - `get(infohash)` pentru a gasi alti clienti ce pot servi fisierul
  - `put(infohash, my-ipaddr)` pentru a se identifica ca detinator al fisierului

Chord: un DHT performant

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica* Robert Morris† David Liben-Nowell†
David Karger† M. Frans Kaashoek† Frank Dabek†
Hari Balakrishnan†

January 10, 2002

Abstract

A fundamental problem that confronts peer-to-peer applications is to efficiently locate the node that stores a particular data item. This paper presents *Chord*, a distributed lookup protocol that addresses this problem. Chord provides support for just one operation: given a key, it maps the key onto a node. Data location can be easily implemented on top of Chord by associating a key with each data item, and storing the key/data item pair at the node to which the key maps. Chord adapts efficiently as nodes join and leave the system, and can answer queries even if the system is continuously changing. Results from theoretical analysis and simulations show that Chord is scalable, with communication cost and the state maintained by each node scaling logarithmically with the number of Chord nodes.

1 Introduction

Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. A review of the features of recent peer-to-peer applications yields a long list: redundant storage, permanence, selection

*University of California, Berkeley. istoica@cs.berkeley.edu
†MIT Laboratory for Computer Science, {rtm, dln, karger, kaashoek, fdabek, hari}@lcs.mit.edu.

Authors in reverse alphabetical order.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, *31*(4), 149-160.

Algoritmul Chord pentru lookup

- **Interfata:** lookup(key) → IP address
- **Eficient:** O(log N) mesaje pentru operatia de lookup
  - N fiind numarul total de servere
- **Scalable:** O(log N) stari per nod
- **Robust:** poate supravietui unui numar mare de noduri defecte

Chord – DHT eficient

Cum asignam chei nodurilor peer?

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Chord

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Chord

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Chord

Hash table

k_0	v_0
k_1	v_1
k_2	v_2
k_3	v_3
k_4	v_4
k_5	v_5

Identificatori Chord

- **Key identifier** = SHA-1(key)
- **Node identifier** = SHA-1(IP address)
- SHA-1 asigura o distributie uniforma a ambelor
- ***Cum partitioneaza Chord datele?***
  - *i.e.*, map key IDs to node IDs

Chord (distributia cheilor)

Inel de identificatori peste spatiul hash 2^m

Chord (distributia cheilor)

Inel de identificatori peste spatiul hash 2^m

= node

= key

node id = hash(node)
key id = hash(key)

Consistent hashing [Karger '97]

Cheia este stocata la successor: nodul avand urmatorul cel mai mare ID

Chord: Successor pointers

Basic lookup

Simple lookup algorithm

```
Lookup(key-id)
  succ ← my successor
  if my-id < succ < key-id //next hop
        call Lookup(key-id) on succ
  else                                    //done
    return succ
```

- **Corectitudinea** depinde doar de **succesori**

Imbunatatirea performantei

- **Problema:** Forwardarea prin intermediul succesorului poate fi inceata

- Stuctura de date este o lista inlantuita: O(n)

- **Idee:** Putem sa facem operatia sa arate mai mult a **binary search**?
  - Adica sa imbunatatim distanta cu fiecare pas

“Finger table” permite operatii de lookup in timp log N

Finger *i* redirecteaza catre succesorul lui $n+2^i$

Implicatii ale tabelei finger

- Avem un **binary lookup tree** cu fiecare nod
  - Cu ramuri redirectate catre alte tabele finger ale celorlalte noduri
- Este mai bine decat sa aranjam nodurile intr-un singur arbore
  - Fiecare nod actioneaza ca radacina a unui arbore
    - Deci nu exista niciun **root hotspot**
    - Nu exista **single point of failure**

Lookup with finger table

```
Lookup(key-id)
  look in local finger table for
        highest n: my-id < n < key-id
  if n exists
        call Lookup(key-id) on node n //next hop
  else
        return my successor   //done
```

Chord

Inel de identificatori peste spatiul hash 2^m

= node

= key

node id = hash(node)
key id = hash(key)

successor(id)

finger table for node at id *i*

finger	node id
1	succ(*i*)
2	succ(*i* + 2)
j	succ(*i* + 2^{j-1})

Chord

Inelul de identificatori

= node

= key

Chord

Inel de identificatori

Chord

Inel de identificatori

finger	node id
1	succ(i)
2	succ(i + 2)
3	succ($i + 2^2$)
4	succ($i + 2^3$)

Chord

Inel de identificatori

= node

= key

finger	node id
1	succ(4)
2	succ(4 + 2)
3	succ(4 + 2^2)
4	succ(4 + 2^3)

Chord

Inel de identificatori

= node

= key

finger	node id
1	5
2	8
3	8
4	1

Asadar... Chord Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n, n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n, n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

Inel de identificatori

lookup(10)

finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n, n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

```
lookup(10)
```

follow finger 3 to node id 8

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n, n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

```
lookup(10)

follow finger 3 to node id 8

node id 8 identifies as predecessor of id 10
```

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n, n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

```
lookup(10)

follow finger 3 to node id 8

node id 8 identifies as predecessor of id 10

complete lookup at successor of node id 8
```

Chord - Lookup

```
find_successor(id):
  p = find_predecessor(id)
  return p.successor

find_predecessor(id):
  n = self
  while id not between (n, n.successor]:
    n = n.closest_preceding_finger(id)
  return n
```

finger	node id
1	5
2	8
3	8
4	1

finger	node id
1	11
2	11
3	1
4	1

```
lookup(10)

follow finger 3 to node id 8

node id 8 identifies as predecessor of id 10

complete lookup at successor of node id 8
```

Nr. de hopuri?
Fiecare op. finger lookup injumatateste distanta la cheie =>
O(log N)

Joining: Linked list insert

Join (2)

Join (3)

Mesajele ***notify*** mentin predecesorii

Mesajele ***stabilize*** fixeaza succesorii

Failures may cause incorrect lookup

N80 does not know correct successor, so incorrect lookup

Exemplu: Chord - Joins + Stabilization

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization

join(), self = 6

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Joins + Stabilization

```
join():
  self.predecessor = null
  self.successor = find_successor(self)

stabilize():
  p = self.successor.predecessor
  if p between (self, self.successor):
    self.successor = p
  self.successor.notify(self)

notify(n):
  if self.predecessor == null ||
     n between (self.predecessor, self):
    self.predecessor = n
```

Chord - Failure + Replication

E mentinuta o lista a k succesori

Cheile sunt replicate pe toti k succesorii

Load balance
1st and 99th percentiles
Number of keys per node
Total number of keys (x 10,000)
0
50
100
150
200
250
300
350
400
450
500
0
20
40
60
80
100

Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
Failed Nodes (Fraction of Total)
0
0.05
0.1
0.15
0.2
0.25
0
0.05
0.1
0.15
0.2

Lookup path length
1st and 99th percentiles
Path length
Number of nodes
0
2
4
6
8
10
12
1
10
100
1000
10000
100000

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
Number of Nodes
0
100
200
300
400
500
600
700
0
20
40
60
80
100
120
140
160
180
200

Load balance
1st and 99th percentiles
Number of keys per node
Total number of keys (x 10,000)
0
50
100
150
200
250
300
350
400
450
500
0
20
40
60
80
100

Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
Failed Nodes (Fraction of Total)
0
0.05
0.1
0.15
0.2
0.25
0
0.05
0.1
0.15
0.2

Lookup path length
1st and 99th percentiles
Path length
Number of nodes
0
2
4
6
8
10
12
1
10
100
1000
10000
100000

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
Number of Nodes
0
100
200
300
400
500
600
700
0
20
40
60
80
100
120
140
160
180
200

Load balance
1st and 99th percentiles
Number of keys per node
Total number of keys (x 10,000)
0
50
100
150
200
250
300
350
400
450
500
0
20
40
60
80
100

Failure resilience
95% confidence interval
Failed Lookups (Fraction of Total)
Failed Nodes (Fraction of Total)
0
0.05
0.1
0.15
0.2
0.25
0
0.05
0.1
0.15
0.2

Lookup path length
1st and 99th percentiles
Path length
Number of nodes
0
2
4
6
8
10
12
1
10
100
1000
10000
100000

Lookup latency
5th, 50th, and 95th percentiles
Lookup Latency (ms)
Number of Nodes
0
100
200
300
400
500
600
700
0
20
40
60
80
100
120
140
160
180
200

The Impact of DHT Routing Geometry on Resilience and Proximity

K. Gummadi[*], R. Gummadi[†], S. Gribble[‡], S. Ratnasamy[§], S. Shenker[¶], I. Stoica[‖] [*]

ABSTRACT

The various proposed DHT routing algorithms embody several different underlying routing *geometries*. These geometries include hypercubes, rings, tree-like structures, and butterfly networks. In this paper we focus on how these basic geometric approaches affect the resilience and proximity properties of DHTs. One factor that distinguishes these geometries is the degree of *flexibility* they provide in the selection of neighbors and routes. Flexibility is an important factor in achieving good static resilience and effective proximity neighbor and route selection. Our basic finding is that, despite our initial preference for more complex geometries, the ring geometry allows the greatest flexibility, and hence achieves the best resilience and proximity performance.

Categories and Subject Descriptors

C.2 [**Computer Systems Organisation**]: Computer Communication Networks

General Terms

Algorithms, Performance

Keywords

1. INTRODUCTION

The unexpected and unprecedented explosion of peer-to-peer file-sharing, ignited by Napster and refueled by a succession of less legally vulnerable successors (*e.g.*, Gnutella, Kazaa), inspired the development of distributed hash tables (DHTs). DHTs offer a promising combination of extreme scalability (scaling typically as $\log n$) and useful semantics (supporting a hash-table-like `lookup` interface), and have been proposed as a substrate for many large-scale distributed applications (see, for example, [5, 12, 22]).

Our focus here is not on the uses of DHTs but on the underlying DHT routing algorithms. A wide variety of DHT routing algorithms have been proposed, and the list grows longer with each passing conference. However, the DHT routing literature is still very much in its infancy; as a result, most DHT routing papers describe new algorithms and very few provide more general insight *across* algorithms or usefully compare *between* algorithms.

In this paper we attempt to take a very small step in this direction by looking at the basic *geometry* underlying DHT routing algorithms and how it impacts their performance in two important areas: static resilience and proximity routing. While the current collection of DHT routing algorithms differs in many respects, perhaps the most fundamental distinction between them lies in their different routing geometries. For instance, CAN [19] (when the dimension is taken to be

Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., & Stoica, I. (2003, August). The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (pp. 381-394). ACM.

Topologii de rutare in spatele unor DHT

Ring (Chord)
Tree (Tapestry, PRR)
Hypercube (CAN)
Butterfly (Viceroy)
XOR (Kademlia)
Hybrid (Pastry, Bamboo)

Proximity + Resilience

Proximity – Alegerea unor rute prin vecini “geografic apropiati” reduce latenta

Resilience – Continuarea rutarii unor cereri chiar cand apar partitionari si defecte la nivelul retelei

Flexibility

Neighbor selection – optiunea de a selecta ce noduri peer sa pastram in tabela de rutare

Route selection – optiunea de a selecta unde sa rutam o anumita destinatie

Flexibility in neighbor selection -> good proximity

Flexibility in route selection -> good resilience

Topologii de rutare in spatele unor DHT

Ring (Chord)
Tree (Tapestry, PRR)
Hypercube (CAN)
Butterfly (Viceroy)
XOR (Kademlia)
Hybrid (Pastry, Bamboo)

Topologii de rutare in spatele unor DHT - Arbore

Topologii de rutare in spatele unor DHT - Arbore

Neighbor selection – un nod pentru fiecare prefix din subarborele opus

Topologii de rutare in spatele unor DHT - Arbore

Neighbor selection – un nod pentru fiecare prefix din subarborele opus

Route selection – rutarea catre vecinul din subarborele destinatiei

Good flexibility in neighbor selection

Poor flexibility in route selection

Topologii de rutare in spatele unor DHT - Hipercub

Neighbor selection –
vecinii difera printr-un bit in descompunerea binara

Topologii de rutare in spatele unor DHT - Hipercub

Neighbor selection – vecinii difera printr-un bit in descompunerea binara

Route selection – rutare catre destinatie prin corectarea oricarui bit diferit

Poor flexibility in neighbor selection

Good flexibility in route selection

Topologii de rutare in spatele unor DHT - Inel

Neighbor selection – un vecin in fiecare interval de finger

Topologii de rutare in spatele unor DHT - Inel

Neighbor selection – un vecin in fiecare interval de finger

Route selection – rutarea catre destinatie prin progresarea de-a lungul inelului

Good flexibility in neighbor selection

Good flexibility in route selection

BitTorrent

- Principalul exemplu de sistem P2P in folosinta astazi (alaturi de Boinc):
  - Dezvoltat de Cohen in ‘01
  - Mare parte din traficul de Internet astazi!
  - Folosit pentru transfer de continut legal si mai putin legal (aici intervine rolul DHT)
- Date livrate folosind “torrents”:
  - Fisierele sunt transferate in bucati, e favorizat paralelismul
  - Foloseste un tracker sau un decentralized index (DHT)

Bram Cohen (1975 -)

Exemplu: BitTorrent

1. User clicks on download link
   - Gets ***torrent*** file with content hash, IP addr of ***tracker***
2. User's BitTorrent (BT) client talks to tracker
   - Tracker tells it **list of peers** who have file
3. User's BT client downloads file from one or more peers
4. User's BT client tells tracker it has a copy now, too
5. User's BT client serves the file to others for a while

Provides huge download bandwidth, without expensive server or network links

BitTorrent Protocol (2)

- Toti peers (cu exceptia celor seed) regasesc torrenti in acelasi timp

BitTorrent Protocol (3)

- Fisierele sunt impartite in bucati, ceea ce permite descarcarea paralela (viteza de download)

BitTorrent Protocol (5)

- Indexul DHT (distribuit intre peers) este complet descentralizat

Cooperative storage with a DHT

- Aplicatiile pot fi **distribuite** pe multe noduri
- DHT **distribuie stocarea de date** pe mai multe noduri

BitTorrent si DHT

- BitTorrent poate folosi DHT in loc de (sau lucreaza impreuna cu un) tracker

- Clientii BT folosesc DHT:
  - Key = **file content hash** (“infohash”)
  - Value = **IP address of peer** willing to serve file

- Clientul:
  - `get(infohash)` pentru a gasi alti client ce pot deserve o descarcare
  - `put(infohash, my-ipaddr)` pentru a se identifica ca furnizor de content

La ce foloseste un DHT pentru BitTorrent?

- DHT e un tracker in sine, mai putin fragmentat decat alti tracker
  - Nodurile peer se pot regasi unele cu altele mai usor, chiar si in prezenta defectelor
- In plus, un tracker classic este expus unor probleme de **legalitate**

Sumar

- Sisteme peer-to-peer
- Distributed Hash Tables
  - Chord
- Topologii din spatele DHT
- Exemplificare BitTorrent