

# Parallelism (PAR)

## Unit 1: Why parallel computing?

Eduard Ayguadé, José Ramón Herrero,  
Daniel Jiménez and Gladys Utrera

Computer Architecture Department  
Universitat Politècnica de Catalunya

Course 2021/22 (Fall semester)

## Learning material for this lesson

- ▶ Atenea: Unit 1 "Why parallel computing?"
  - ▶ Video lesson 1: serial vs. parallel
  - ▶ Questions after video lesson 1
  - ▶ Going further: dining philosophers problem
- ▶ These slides to dive deeper into the concepts in Unit 1
- ▶ Collection of Exercises: problems in Chapter 1

# Outline

Motivation (video lesson 1)

Concurrency and parallelism

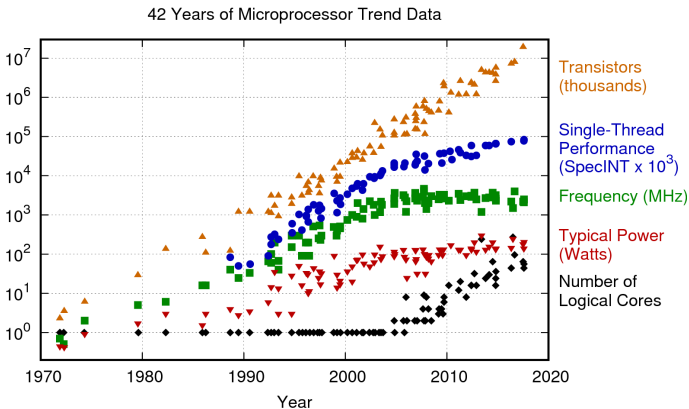
Examples and potential problems

Processors, threads and processes

# Concepts in video lesson 1

- ▶ Serial execution
  - ▶ instructions are executed one after another, only one at any moment in time
  - ▶  $T = N \div F$
  - ▶ To run faster, increase  $F$  (instruction per time unit executed)  
→ technology and architecture
- ▶ Parallel execution on  $P$  processors
  - ▶ Each processor executes  $\frac{1}{P}$  instructions of the program
  - ▶  $T = (N \div P) \div F$
- ▶ Throughput computing on  $P$  processors
  - ▶  $k$  programs executed on  $P$  processors
  - ▶ OS multiplexes their execution on time, fairness

# Uniprocessor and multicore performance evolution



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp

## Uniprocessor and multicore performance evolution (cont.)

- ▶ Moore's law: the number of transistors on an integrated circuit potentially doubles approximately every two years
  - ▶ Higher clock frequencies
  - ▶ More complex architectural designs (ability to exploit Instruction-Level Parallelism, ILP)
- ▶ But ...
  - ▶ Diminishing returns when using transistors to exploit more ILP
  - ▶ Power consumption/heat dissipation: hard technological limit
- ▶ As an alternative to scale performance, each generation of Moore's law allows to potentially double the number of cores
  - ▶ This vision creates a desperate need for all computer scientists and practitioners to be aware of parallelism<sup>1</sup>

---

<sup>1</sup> Parallelism and parallel computing has been taught for several decades in some master and PhD curricula, oriented to solve computationally intensive applications in science and engineering with problems too large to solve on one computer.

# Scaling beyond multicores ... servers ... supercomputers

Top500.org ranking the most powerful supercomputers (June 2021)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	<b>Supercomputer Fugaku</b> - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, <b>Fujitsu</b> RIKEN Center for Computational Science <b>Japan</b>	7,630,848	442,010.0	537,212.0	29,899
2	<b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, <b>IBM DOE/SC/Oak Ridge National Laboratory United States</b>	2,414,592	148,600.0	200,794.9	10,096
3	<b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, <b>IBM/NVIDIA/Mellanox DOE/NNSA/LLNL United States</b>	1,572,480	94,640.0	125,712.0	7,438
4	<b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, <b>NRCPC National Supercomputing Center in Wuxi China</b>	10,649,600	93,014.6	125,435.9	15,371
5	<b>Perlmutter</b> - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, <b>HPE DOE/SC/LBNL/NERSC United States</b>	706,304	64,590.0	89,794.5	2,528
...					
63	<b>MareNostrum</b> - Lenovo SD530, Xeon Platinum 8160 24C 2.1GHz, Intel Omni-Path, <b>Lenovo Barcelona Supercomputing Center Spain</b>	153,216	6,470.8	10,296.1	1,632
...					

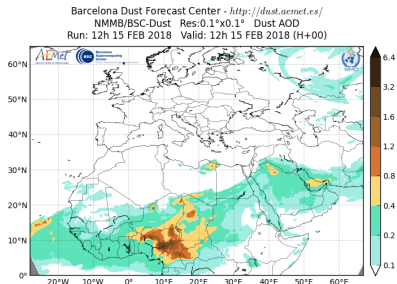
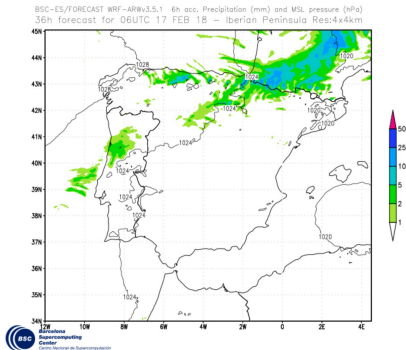
# Give me an example of their use!

Rainfall 60 hour forecast		
Machine	Parallel	Sequential
MN3	32 min (128 cores)	2.5 days approx.
MN4	23 min (128 cores)	

Figure 1: Accumulated rainfall

Dust Disp. 70 hour forecast		
Machine	Parallel	Sequential
MN3	49 min (260 cores)	8 days approx.
MN4	32 min (248 cores)	

Figure 2: Sahara dust dispersion





# Outline

Motivation (video lesson 1)

Concurrency and parallelism

Examples and potential problems

Processors, threads and processes

## Concurrent execution

Exploiting concurrency consists in breaking a problem into discrete parts, to be called tasks, to ensure their correct simultaneous execution

- ▶ Each (serial) task is sequentially executed on a single CPU ...
- ▶ ... but multiple tasks interleave their execution on the CPU, so that the CPU time is multiplexed among tasks. The SO is in charge of preempting the currently running task, for example after its time quantum expires, starting or resuming the execution of another task

Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources

## Example: client/server application

Client connection implies the execution of the client task (C). As a response, the server task (T) is executed

Task C<sub>k</sub>: receives client requests (duration 1 time unit)

Task T<sub>k</sub>: executes a single bank transaction (e.g. withdraw/deposit some money in bank account, with a duration of 4 time units)

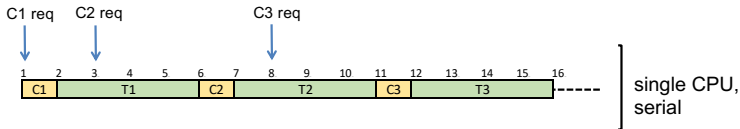


Figure 3: Sequential execution of client and server tasks

## Example: client/server application

Client connection implies the execution of the client task (C). As a response, the server task (T) is executed

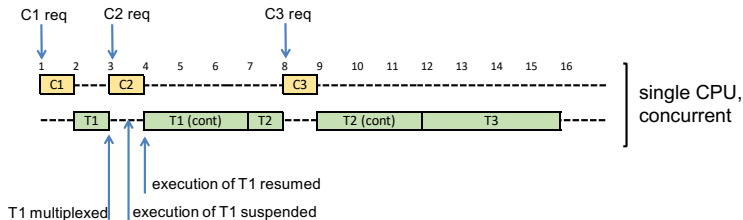


Figure 4: Concurrent execution of client and server tasks, but server tasks serialized

## Example: client/server application

Client connection implies the execution of the client task (C). As a response, the server task (T) is executed

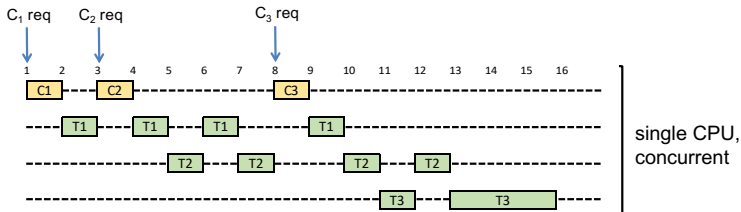


Figure 5: Concurrent execution of client and multiple server tasks

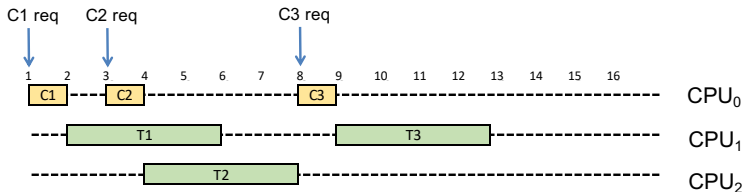
## Parallel execution

Parallel execution: to reduce the execution (response) time of a program. Parallelism is when we use multiple processors (CPU) to simultaneously execute the tasks identified for concurrent execution

- ▶ 1 program on  $p$  processors
- ▶ Ideally, each CPU could receive  $\frac{1}{p}$  of the program, reducing its execution time by  $p$

## Example: client/server application

Client connection implies the execution of the client task (C). As a response, the server task (T) is executed



**Figure 6:** Parallel execution of client and server tasks on **several processors**

## Throughput vs. parallel computing

*Throughput computing*: multiple processors can also be used to increase the number of programs executed per time unit

- ▶ *Multiprogrammed* execution of multiple, unrelated, instruction streams (programs) at the same time on multiple processors
- ▶  $n$  programs on  $p$  processors; if  $(n \geq p)$  each program receives  $\frac{p}{n}$  processors, one processor otherwise



# Outline

Motivation (video lesson 1)

Concurrency and parallelism

Examples and potential problems

Processors, threads and processes

## Examples and potential problems

bank (hash map)

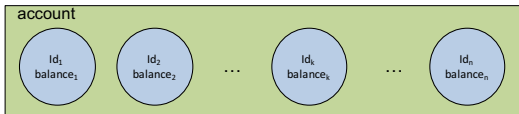


Figure 7: Bank with several accounts

Three different cases and potential problems:

- ▶ Example 1: two simultaneous deposit/withdraw operations
  - ▶ Correctness: data race, starvation
- ▶ Example 2: two simultaneous money transfers
  - ▶ Correctness: deadlock
- ▶ Example 3: simple bank statistics
  - ▶ Efficiency: lack or dependency of work, overheads, ...

# First example: simplified C code, not complete

## Deposit/withdraw task

```
// code executed to process each transaction
// val contains the amount of money to deposit (positive) or to withdraw (negative)

#pragma omp task shared(acc) firstprivate(val)
if ((acc.balance + val) < 0)
    Error("Not enough money in account %d", acc.id);
else {
    acc.balance = acc.balance + val;
    Correct("New balance in account %d: %d\n", acc.id, acc.balance);
}
```

# First example: two simultaneous withdraw operations

Figure 8: No problem if  $id_1 \neq id_2$

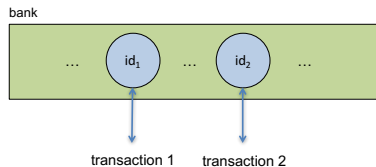
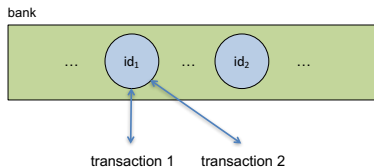


Figure 9: Concurrent execution of code on same account if  $id_1 = id_2$ :  
**data race**



## First example: data race – free money

Problem: **Data race** in the access to balance

Assume  $\text{acc.balance} = 105$ , initially

Time	Transaction 1 (val1=-100)	Transaction 2 (val2=-10)
1	if ((acc.balance+val1)>0)	
2		if ((acc.balance+val2)>0)
	acc.balance=acc.balance+val1	acc.balance=acc.balance+val2
3	Step1: read acc.balance (105)	
4	Step2: sum = 105 + (-100)	
5		Step1: read acc.balance (105)
6	Step3: write acc.balance with 5	
7		Step2: sum = 105 + (-10)
8		Step3: write acc.balance with 95

# Simplified C code, not complete

Using `omp_set_lock` and `omp_unset_lock` to protect the execution of account balance update

```
// code executed to process each transaction
#pragma omp task shared(acc) firstprivate(val)
if ((acc.balance + val) < 0)
    Error("Not enough money in account %d", acc.id);
else {
    omp_set_lock(&acc.lock);
    acc.balance = acc.balance + val;
    omp_unset_lock(&acc.lock);
    Correct("New balance in account %d: %d\n", acc.id, acc.balance);
}
```

# First example: data race – money but negative balance

Problem: Still **data race** in the access to balance

Time	Transaction 1 (val1=-100)	Transaction 2 (val2=-10)
1	if ((acc.balance+val1)>0)	
2	Set lock	
	acc.balance=acc.balance+val1	
3	Step1: read acc.balance (105)	
4		if ((acc.balance+val2)>0)
5		Set lock failed
6	Step2: sum = 105 + (-100)	Set lock failed
7	Step3: write acc.balance with 5	Set lock failed
8	Unset lock	Set lock
		acc.balance=acc.balance+val2
9		Step1: read acc.balance (5)
10		Step2: sum = 5 + (-10)
11		Step3: write acc.balance with -5
12		Unset lock

# Simplified C code, not complete

Using `omp_set_lock` and `omp_unset_lock` to protect the execution of account balance update

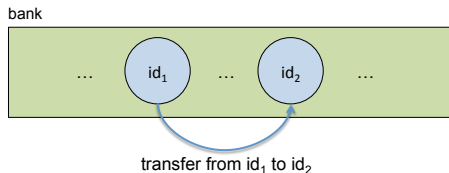
```
// code executed to process each transaction
#pragma omp task shared(acc) firstprivate(val)
{
    omp_set_lock(&acc.lock);
    if ((acc.balance + val) < 0)
        Error("Not enough money in account %d", acc.id);
    else {
        acc.balance = acc.balance + val;
        Correct("New balance in account %d: %d\n", acc.id, acc.balance);
    }
    omp_unset_lock(&acc.lock);
}
```



## First example: correct execution

Time	Transaction 1 (val1=-100)	Transaction 2 (val2=-10)
1	Set lock	
2	if ((acc.balance+val1)>0)	
	acc.balance=acc.balance+val1	
3	Step1: read acc.balance (105)	
4	Step2: sum = 105 + (-100)	Set lock failed
5	Step3: write acc.balance with 5	Set lock failed
6	Unset lock	Set lock
7		if ((acc.balance+val2)>0)
8		Error: not enough money in account
9		Unset lock

## Second example: transfer between two accounts

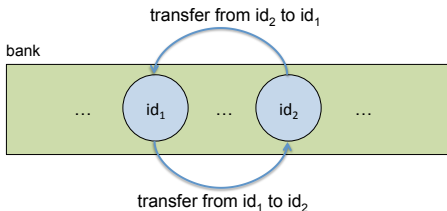


We need to protect the update of the two accounts

```
int transfer(account * from, account * to, int val) {  
    int status = 0;  
    omp_set_lock(&from.lock);  
    omp_set_lock(&to.lock)  
  
    if (from->balance > val) {  
        from->balance -= val;  
        to->balance += val;  
        status = 1;  
    }  
  
    omp_unset_lock(&to.lock);  
    omp_unset_lock(&from.lock);  
    return status;  
}
```

## Second example: two simultaneous transfers, same account

But, what if "John wants to transfer \$10 to Peter's account" while "Peter wants to also transfer \$20 to John's account"?



Both get blocked when invoking `omp_set_lock`

- Cycle in locking graph = **deadlock**

## Second example: deadlock

Time	Transaction 1 (John to Peter)	Transaction 2 (Peter to John)
1	Set lock on John account	
2		Set lock on Peter account
3		Set lock on John account failed
4	Set lock on Peter account failed	
5	DEADLOCK	DEADLOCK

## Second example: ordering lock acquisition

Standard solution: canonical order for locks (e.g. acquire in decreasing order)

```
int transfer(account * from, account * to, int val) {
    int status = 0;
    if (from->id > to->id) {
        omp_set_lock(&from.lock);
        omp_set_lock(&to.lock);
    } else {
        omp_set_lock(&to.lock);
        omp_set_lock(&from.lock);
    }

    if (from->balance > val) {
        from->balance -= val;
        to->balance += val;
        status = 1;
    }

    omp_unset_lock(&to.lock);
    omp_unset_lock(&from.lock);
    return status;
}
```

## Other potential concurrency problems

### Race Condition

- ▶ Multiple tasks read and write some data and the final result depends on the relative timing of their execution

### Deadlock

- ▶ Two or more tasks are unable to proceed because each one is waiting for one of the others to do something

### Starvation

- ▶ A task is unable to gain access to a shared resource and is unable to make progress

### Livelock

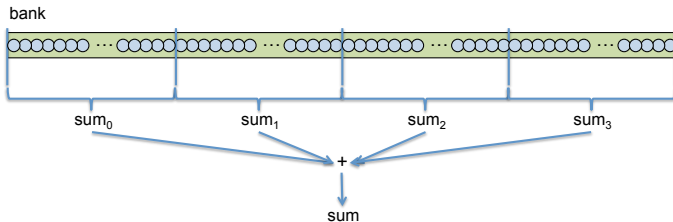
- ▶ Two or more tasks continuously change their state in response to changes in the other tasks without doing any useful work

## Third example: bank statistics

- ▶ Imagine that every day the bank needs to compute the total interest (*sum*) that has to pay to all its customers (hundred thousands, or more!)
  - ▶  $sum = \sum_{i=1}^{number\_clients} balance_i \times interest_i$
- ▶ For simplicity, if we assume that *balance* and *interest* are vectors with vector elements *i* associated to client *i*, then the computation of *sum* implies a **Dot Product** of two vectors:  
 $sum = balance \times interest$

## Third example: bank statistics

- ▶ The computation and data can be partitioned among multiple processors ( $P$ ). each working with  $1/P$  elements and accumulating the result in a "shared" variable (e.g. `sum`). This is shown in the following figure:



- ▶ Computation time approx. divided by  $P$



## Simplified C code, not complete

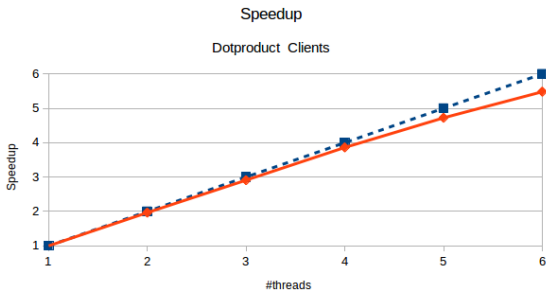
```
float balance[MAX_CLIENTS];
float interest[MAX_CLIENTS];

float DotProduct (float * balance, float * interest, long number_clients) {
    float sum = 0.0;
    // This distributes iterations among participating processors
    #pragma omp parallel for reduction(+: sum)
    for (int client = 0 ; client < number_clients; client++) {
        sum += balance[client] * interest[client];
    }
    return(sum);
}
```

**Note:** this is the so-called **work-sharing model** in OpenMP, only for loops; in this course we will be using the **tasking model** in OpenMP that applies to a wider set of code structures.

## How much faster is the parallel version?

- ▶ **Parallel version** is almost  $P$  **times faster** than the **sequential version**



- ▶ Results are shown for
  - ▶ Boada machine using up to 6 cores of one node
  - ▶ A set of 100,000,000 clients
  - ▶ Dashed line shows ideal linear speedup

## Potential parallelism problems

- ▶ Lack of work or work dependency
  - ▶ Coverage or extent of parallelism in algorithm
  - ▶ Dependencies (sequential is an extreme case)
  - ▶ Hard to equipartition the work
    - ▶ Load imbalance
  - ▶ Due to the parallelization strategy and parallel programming model
- ▶ Overheads of the parallelization
  - ▶ Granularity of partitioning among processors
    - ▶ Work generation and synchronization
  - ▶ Locality of computation and communication

# Outline

Motivation (video lesson 1)

Concurrency and parallelism

Examples and potential problems

Processors, threads and processes

## Processors vs. processes/threads

- ▶ Processes/threads are logical computing agents, offered by the OS (Operating System), that execute tasks
- ▶ Processors<sup>2</sup> are the hardware units that physically execute those logical computing agents
- ▶ In most cases, there is a one-to-one correspondence between processes/threads and processors, but not necessarily (it is a OS decision)
- ▶ Tasks are created by the parallel runtime that supports the execution of a parallel language (e.g. `#pragma omp task`) and are not known by the OS

---

<sup>2</sup>CPU, processor and core refer to the same concept during this course and may be used interchangeably.

# Processes vs. threads

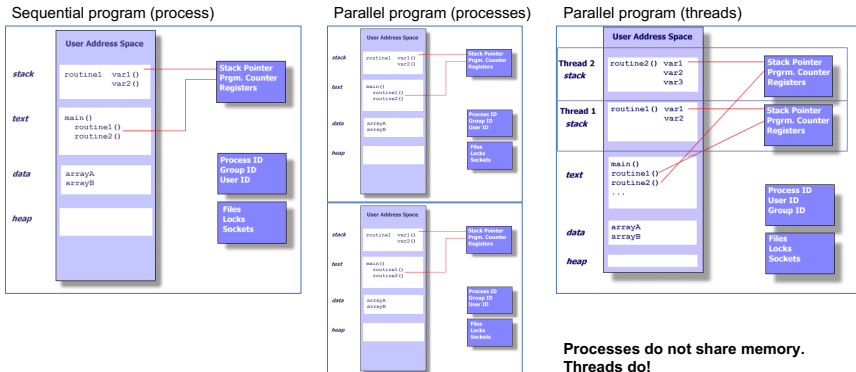


Figure 10: Processes vs threads: User Address Space and other resources

# Processes and threads

Processes and threads co-exist in the same parallel program:

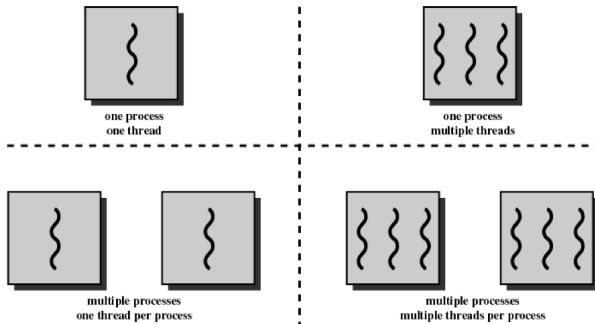


Figure 11: Processes and threads

# Parallelism (PAR)

## Unit 1: Why parallel computing?

Eduard Ayguadé, José Ramón Herrero,  
Daniel Jiménez and Gladys Utrera

Computer Architecture Department  
Universitat Politècnica de Catalunya

Course 2021/22 (Fall semester)