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 2020/21 (Fall semester – online)

Learning material for this lesson

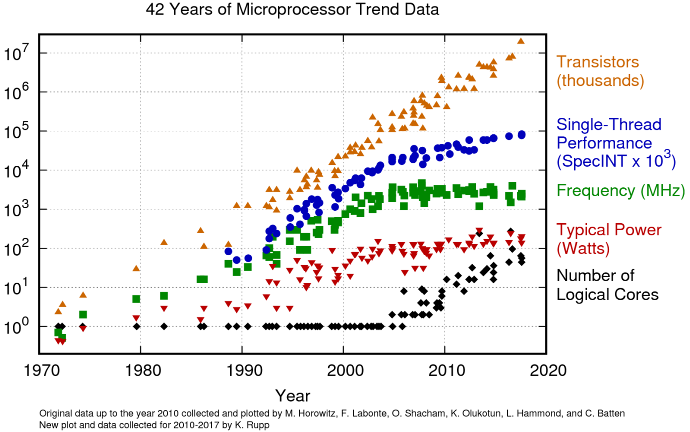
- Atenea: Unit 1 "Why parallel computing?"
  - Video lesson 1: serial vs. parallel
  - Questions after video lesson 1
  - Going further: dining philosophers problem
- These slides to dive deeper into the concepts in Unit 1
- Collection of Exercises: problems in Chapter 1

Outline

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) $\rightarrow$ 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



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[1]

[1] 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.

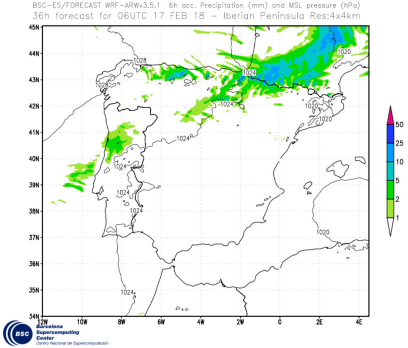
Scaling beyond multicores ... servers ... supercomputers

Top500.org ranking the most powerful supercomputers (June 2020)

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,299,072	415,530.0	513,854.7	28,335
2	**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
3	**Sierra** - IBM Power System AC922, 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
4	**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
5	**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
	...				
37	**MareNostrum** - Lenovo SD530, Xeon Platinum 8160 24C 2.1GHz, Intel Omni-Path, Lenovo Barcelona Supercomputing Center Spain	153,216	6,470.8	10,296.1	1,632

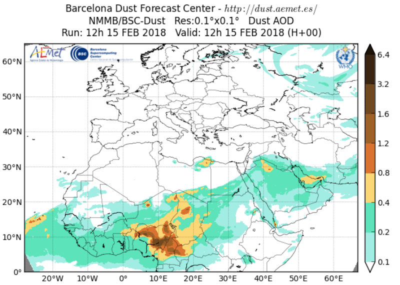
Give me an example of their use!

Accumulated rainfall (60 hour forecast)



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

Sahara dust dispersion (72 hour forecast)



Machine	Parallel	Sequential
MN3	49 min (260 cores)	8 days approx.
MN4	32 min (248 cores)	

Outline

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 multiplex/interleave their execution on the CPU

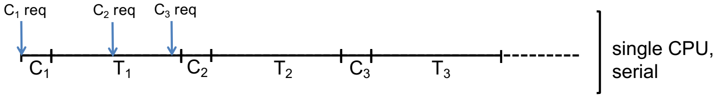
Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources

Example: client/server application

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

Sequential execution of client and server tasks:

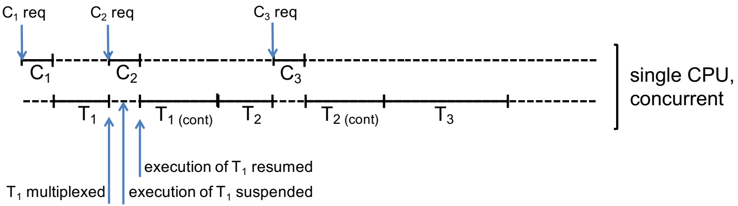
Task C_k: receives client requests
Task T_k: executes a single bank transaction (e.g. withdraw/deposit some money in bank account)



Example: client/server application

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

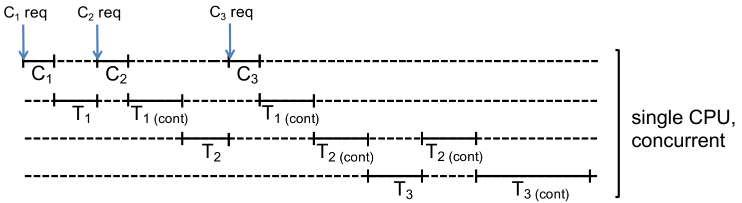
Concurrent execution of client and server tasks, but server tasks serialized



Example: client/server application

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

Concurrent execution of client and multiple server tasks



Parallel execution

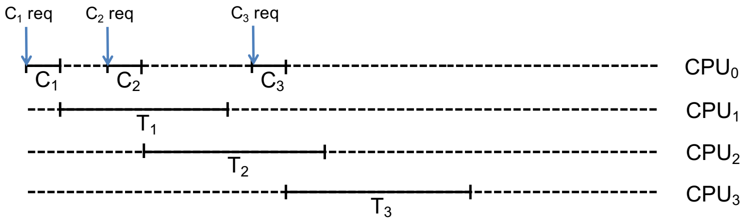
Parallel execution: to reduce the execution (response) time of a program. Parallelism is when we use multiple processors (CPU) to simultaneously execute the tasks identified for concurrent execution

- 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

Parallel execution of client and server tasks on **several processors**



Throughput vs. parallel computing

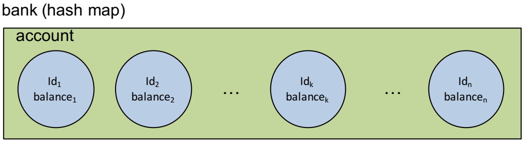
Throughput computing: multiple processors can also be used to increase the number of programs executed per time unit

- *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

Examples and potential problems

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, ...

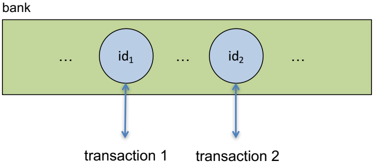
First example: simplified C code, not complete

Deposit/withdraw task

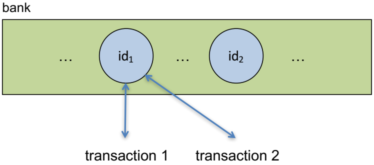
```
// code executed to process each transaction
// val contains the amount of money to deposit (positive) or to withdraw (negative)

#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

- No problem if $id_1 \neq id_2$



- 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 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	Step 1: read acc.balance → 105	
4	Step 2: sum → 105 + (-100)	
5		Step 1: read acc.balance → 105
6	Step 3: write acc.balance → 5	
7		Step 2: sum → 105 + (-10)
8		Step 3: write acc.balance → 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	Step 1: read acc.balance → 105	
4		if ((acc.balance + val2) > 0)
5		set lock failed
6	Step 2: sum → 105 + (-100)	
7	Step 3: write acc.balance → 5	
8	unset lock	set lock
		acc.balance = acc.balance + val2
15		Step 1: read acc.balance → 5
16		Step 2: sum → 5 + (-10)
17		Step 3: write acc.balance → -5
18		unset lock

Simplified C code, not complete

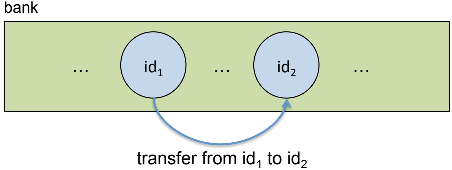
Using omp_set_lock and omp_unset_lock to protect the execution of account balance update

```
// 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	Step 1: read acc.balance → 105	
4	Step 2: sum → 105 + (-100)	set lock failed
5	Step 3: write acc.balance → 5	
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

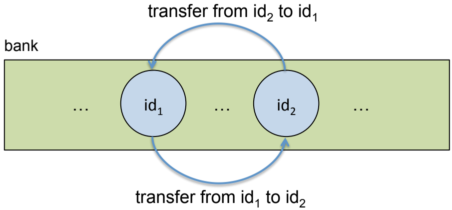


We need to protect the update of the two accounts

```
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 → Peter)	Transaction 2 Peter → 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

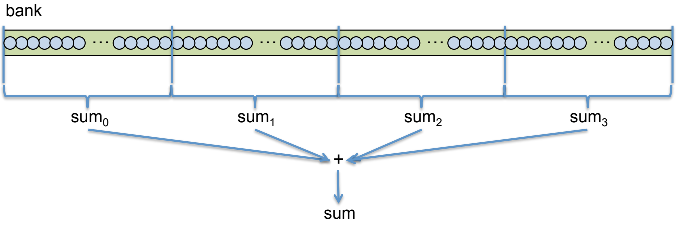
- Two or more tasks continuously change their state in response to changes in the other tasks without doing any useful work

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`)



  - Computation time approx. divided by P

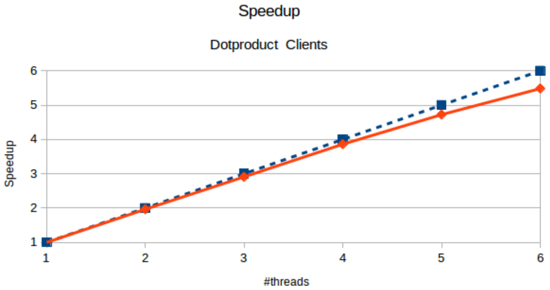
Simplified C code, not complete

```
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

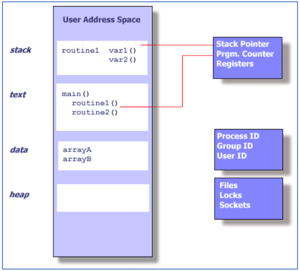
Processors vs. processes/threads

- Processes/threads are logical computing agents, offered by the OS (Operating System), that execute tasks
- Processors[2] 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

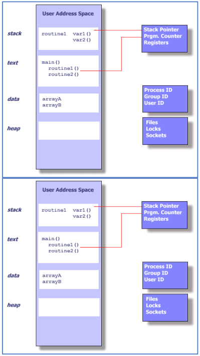
[2] CPU, processor and core refer to the same concept during this course and may be used interchangeably.

Processes vs. threads

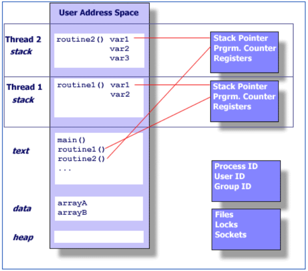
Sequential program (process)



Parallel program (processes)



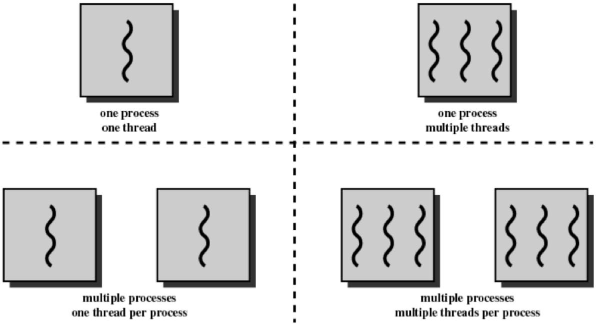
Parallel program (threads)



Processes do not share memory.
Threads do!

Processes and threads

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



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 2020/21 (Fall semester – online)