Parallelism and Concurrency (PACO)

Introduction and motivation

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera
modified by Eva Marín

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)

Outline

Motivation

Concurrency and parallelism

Examples and potential problems

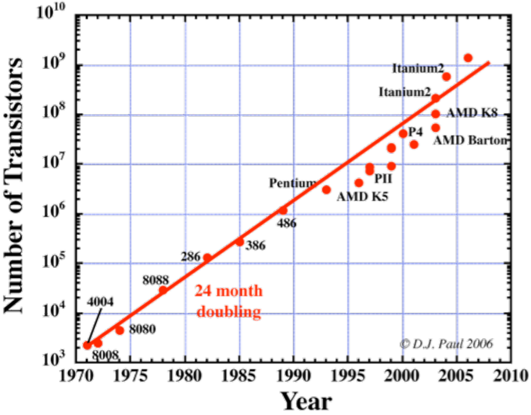
Processors, threads and processes

Programming evolution and Moore's law

- 60s and 70s
  - Assembly language was used
  - Computers able to handle large and complex programs
  - Need to get abstraction and portability without losing performance
  - High-level languages: FORTRAN and C
- 80s and 90s
  - Inability to build and maintain complex and robust applications requiring multi-million lines of code developed by hundreds of programmers
  - Computers could handle even larger more complex programs
  - Needed to get composability and maintainability
  - Object Oriented Programming: C++, C# and Java
  - Performance was not an issue (compilers and Moore's Law)

Moore's law

The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years.



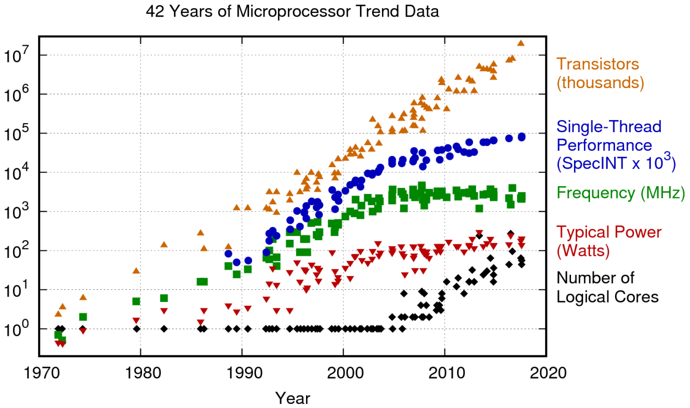
Why parallelism on 2000-present?

- Power consumption is putting a hard technological limit
- Diminishing returns when trying to use transistors to exploit more instruction-level parallelism
- To scale performance, put many processing cores (CPU) on the microprocessor chip instead of increasing clock frequency and architecture complexity
  - Each generation of Moore's law potentially and inexpensively doubles 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 (use 100s or 1000s)

Uniprocessor and multicore performance evolution

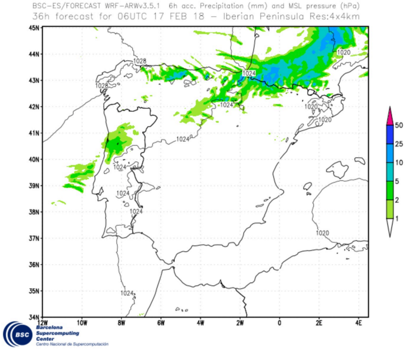
The solution to the power consumption: more than one core



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

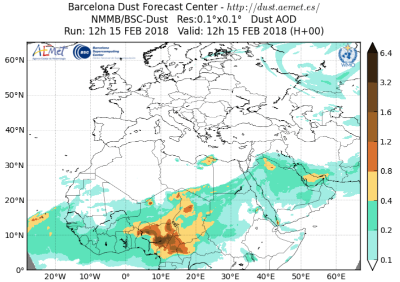
Give me an example!

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)	

Not only your mobile, tablet and laptop are parallel ...

Top500.org ranking the most powerful supercomputers (June 2019)

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Texas Advanced Computing Center/Univ. of Texas United States	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR Dell EMC	448,448	23,516.4	38,745.9	
...						
29	Barcelona Supercomputing Center Spain	**MareNostrum** - Lenovo SD530, Xeon Platinum 8160 24C 2.1GHz, Intel Omni-Path Lenovo	153,216	6,470.8	10,296.1	1,632
...						

Outline

Motivation

Concurrency and parallelism

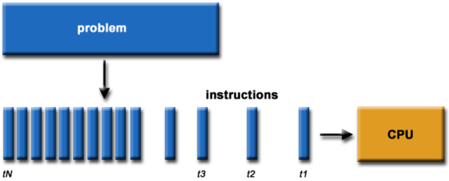
Examples and potential problems

Processors, threads and processes

Serial execution

Traditionally, programs have been written serial for sequential execution, i.e.

- Serial: with a single instruction stream
- Sequential: to be run on a computer with a single processor (CPU[2])



[2] Here we mean in-order and not pipelined/superscalar processor

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

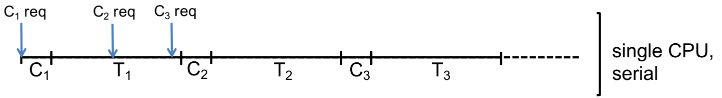
Need to manage and coordinate the execution of tasks, ensuring correct access to shared resources

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:

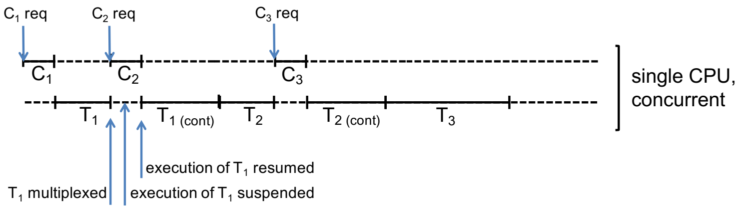
Task C_k: receives client requests
Task T_k: executes a single bank transaction (e.g. withdraw/deposit some money in bank account)



Example: client/server application

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

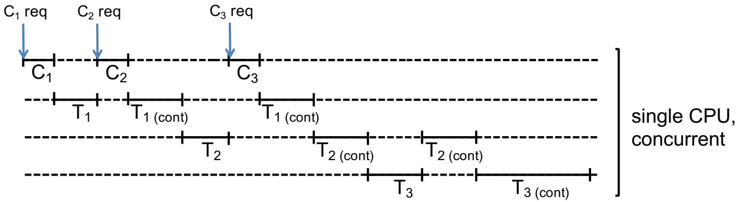
Concurrent execution of client and server tasks, but server tasks serialized



Example: client/server application

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

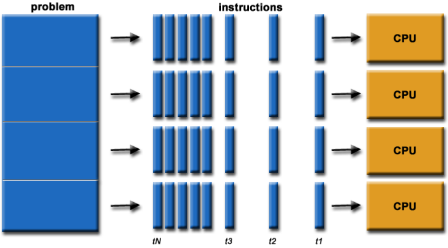
Concurrent execution of client and multiple server tasks



Parallel execution

In the simplest sense, parallelism is when we use multiple processors (CPU) to simultaneously execute the tasks identified for concurrent execution

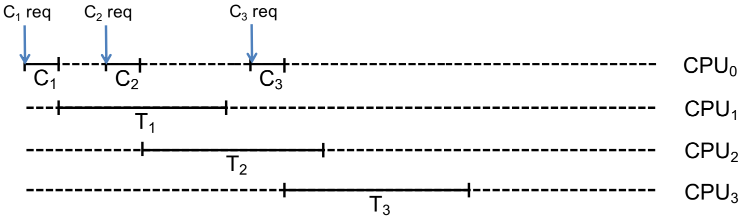
- Ideally, each CPU could receive $\frac{1}{p}$ of the program, reducing its execution time by p



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

Notice that this is not the same as *parallel computing*, whose objective is to reduce the execution (response) time of a single program:

- Parallel computing: multiple, related, interacting instruction streams (single program) that execute simultaneously
- 1 program on p processors, each processor executes $\frac{1}{p}$ of it

Outline

Motivation

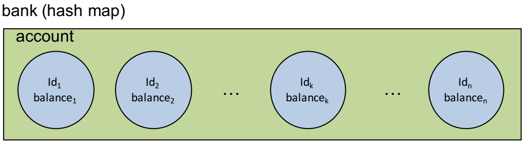
Concurrency and parallelism

Examples and potential problems

Processors, threads and processes

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, ...

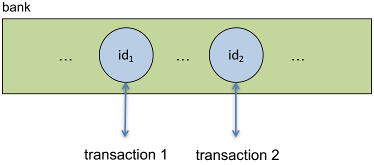
First example: simplified C code, not complete

Deposit/withdraw task

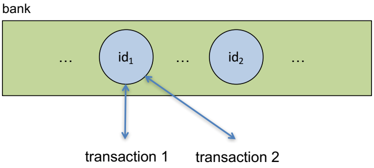
```
// code executed to process each transaction
// val contains the amount of money to deposit (positive) or to withdraw (negative)

#pragma omp task firstprivate(acc, 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 firstprivate(acc, 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

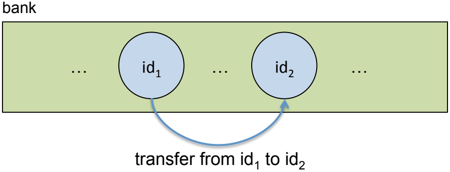
Using omp_set_lock and omp_unset_lock to protect the execution of account balance update

```
// code executed to process each transaction
#pragma omp task firstprivate(acc, 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

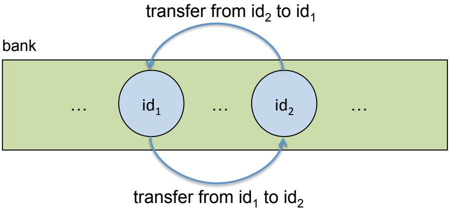


We need to protect the update of the two accounts

```
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

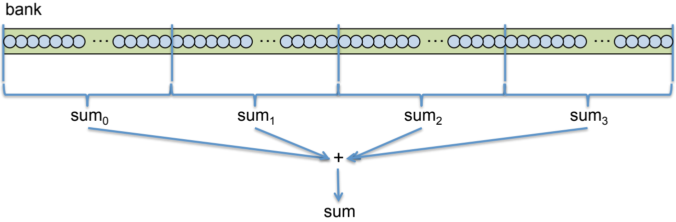
- Two or more tasks continuously change their state in response to changes in the other tasks without doing any useful work

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`)



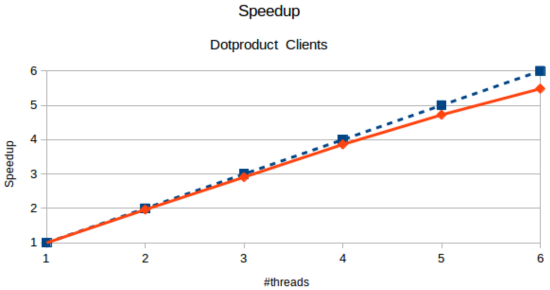
  - Computation time approx. divided by P

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);
}
```

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

Concurrency and parallelism

Examples and potential problems

Processors, threads and processes

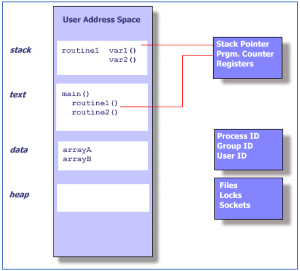
Processors vs. processes/threads

- Processes/threads are logical computing agents, offered by the OS (Operating System), that execute tasks
- Processors[3] 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

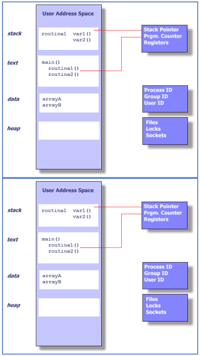
[3] CPU, processor and core refer to the same concept during this course and may be used interchangeably.

Processes vs. threads

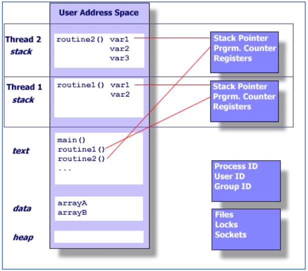
Sequential program (process)



Parallel program (processes)



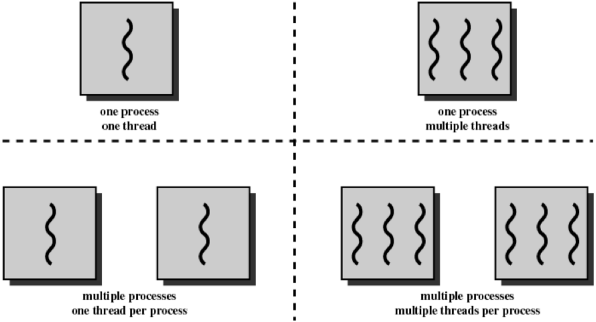
Parallel program (threads)



Processes do not share memory.
Threads do!

Processes and threads

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



Parallelism and Concurrency (PACO)

Introduction and motivation

Eduard Ayguadé, José Ramón Herrero and Gladys Utrera
modified by Eva Marín

Computer Architecture Department
Universitat Politècnica de Catalunya

Course 2019/20 (Fall semester)