

# Parallelism (PAR)

## Unit 2: Understanding parallelism

Eduard Ayguadé, José Ramón Herrero (Q1),  
Daniel Jiménez and Gladys Utrera

Computer Architecture Department  
Universitat Politècnica de Catalunya

Course 2022/23 (Spring semester)

# Outline

## Introduction

### Defining tasks and their implications

Video lesson 2

Vector sum example

Granularity and overheads

### Speed-up and efficiency

Video lesson 3

Other sources of overhead: data sharing overheads

# Expressing and understanding parallelism

- ▶ Can the computation be divided in parts?<sup>1</sup>
  - ▶ Task decomposition: based on the processing to do (e.g. functions, loop iterations)
  - ▶ Data decomposition: based on the data to be processed (e.g. elements of a vector, rows of a matrix) (implies task decomposition)
  - ▶ There may be (data or control) dependencies between tasks
- ▶ Metrics to understand how our task/data decomposition can potentially behave
- ▶ Factors: granularity and overheads

---

<sup>1</sup>Different strategies covered during this course

## Learning material for this lesson

- ▶ Atenea: Unit 2.1 Understanding parallelism I
  - ▶ Video lesson 2: expressing tasks
  - ▶ Questions after video lesson 2
  - ▶ Going further: Excel to explore the effect of task decomposition overheads
- ▶ Atenea: Unit 2.2 Understanding parallelism II
  - ▶ Video lesson 3: speed-up and efficiency
  - ▶ Questions after video lesson 3
  - ▶ Going further: Excel to explore the effect of data sharing overheads
- ▶ These slides to dive deeper into the concepts in Unit 2
- ▶ Collection of Exercises: problems in Chapter 2

# Outline

## Introduction

## Defining tasks and their implications

- Video lesson 2

- Vector sum example

- Granularity and overheads

## Speed-up and efficiency

- Video lesson 3

- Other sources of overhead: data sharing overheads

# Outline

## Introduction

## Defining tasks and their implications

### Video lesson 2

### Vector sum example

### Granularity and overheads

## Speed-up and efficiency

### Video lesson 3

### Other sources of overhead: data sharing overheads

## Concepts in video lesson 2

- ▶ TDG: directed acyclic graph to represent tasks and dependencies between them
- ▶ Metrics:
  - ▶  $T_1 = \sum_{i=1}^{nodes} (work\_node_i)$
  - ▶  $T_\infty = \sum_{i \in critical\_path} (work\_node_i)$ , assuming sufficient (infinite) resources
  - ▶  $Parallelism = T_1 / T_\infty$
  - ▶  $P_{min}$  is the minimum number of processors necessary to achieve  $Parallelism$
- ▶ Task granularity vs. number of tasks

# Outline

## Introduction

## Defining tasks and their implications

Video lesson 2

Vector sum example

Granularity and overheads

## Speed-up and efficiency

Video lesson 3

Other sources of overhead: data sharing overheads



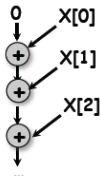
## Example 1: vector sum

Compute the sum of elements  $X[0] \dots X[n-1]$  of a vector  $X$

```
sum = 0; for ( i=0 ; i< n ; i++ ) sum += X[i];
```

**Task definition:** each iteration of the  $i$  loop is a task.

► **TDG** (with input data):



► **Metrics:**

$$T_1 \propto n$$

$$T_\infty \propto n$$

$$Parallelism = 1$$

How can we design an algorithm which leads to a TDG with more parallelism?

## Example 1: vector sum

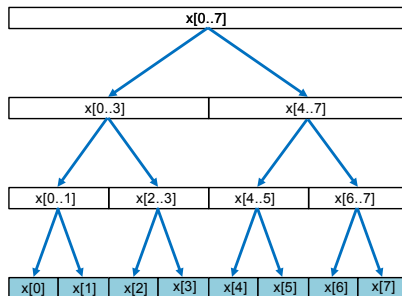
Writing a **recursive version** of the sequential program to compute the sum of elements  $X[0] \dots X[n-1]$  of a vector  $X$ , following a *divide-and-conquer* strategy:

```
int recursive_sum(int *X, int n) {
    int ndiv2 = n/2;
    int sum=0;

    if (n==1) return X[0];

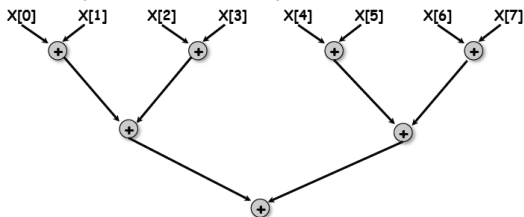
    sum1 = recursive_sum(X, ndiv2);
    sum2 = recursive_sum(X+ndiv2, n-ndiv2);
    return sum1+sum2;
}

void main() {
    int sum, X[N];
    ...
    sum = recursive_sum(X,N);
    ...
}
```



## Example 1: vector sum

- ▶ **Task definition:** each invocation to `recursive_sum`
- ▶ **TDG** (with input data):



- ▶ **Metrics:**  
 $T_1 \propto n$ ;  $T_\infty \propto \log_2(n)$ ;  $Parallelism \propto (n \div \log_2(n))$
- ▶ Same problem can be expressed with different algorithms/implementations leading to different metrics

# Outline

## Introduction

## Defining tasks and their implications

Video lesson 2

Vector sum example

Granularity and overheads

## Speed-up and efficiency

Video lesson 3

Other sources of overhead: data sharing overheads

# Granularity and parallelism

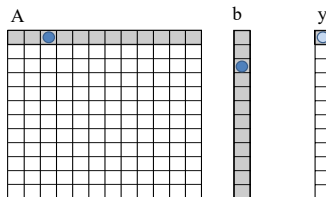
Given a sequential program, the number of tasks that one can generate and the size of the tasks (what is called **granularity**) are related one to the other.

- ▶ Fine-grained tasks vs. coarse-grained tasks
- ▶ The parallelism increases as the decomposition becomes finer in granularity (small tasks) and vice versa

## Granularity and parallelism: fine-grained decomposition

Example: matrix-vector product ( $n$  by  $n$  matrix):

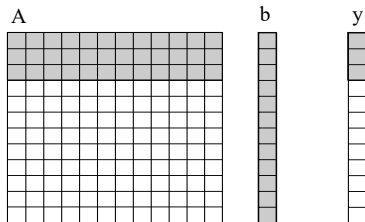
- ▶ A task could be each individual  $\times$  and  $+$  in the dot product that contributes to the computation of an element of  $y$   
 $(y[i] = y[i] + A[i][j] * b[j])$



- ▶ A task could also be each complete dot product to compute an element of  $y$  ( $y[i] = y[i] + \sum_{j=1}^n (A[i][j] * b[j])$ )

## Granularity and parallelism: coarse-grained decomposition

- ▶ A task could be in charge of computing a number of consecutive elements of  $y$  (e.g. three elements)



- ▶ A task could be in charge of computing the whole vector  $y$

## Granularity and parallelism: fine vs. coarse-grained

- ▶ It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity but...
  - ▶ Inherent bound on how fine the granularity of a computation can be
    - ▶ e.g. *matrix-vector multiply*:  $(n^2)$  concurrent tasks.
  - ▶ Tradeoff between the granularity of a decomposition and associated overheads (sources of overhead: creation of tasks, task synchronization, exchange of data between tasks, ...)
  - ▶ The granularity may determine performance bounds



## Example 2: stencil computation using Jacobi solver

Stencil algorithm that computes each element of matrix utmp using 4 neighbor elements of matrix u, both matrices with  $n \times n$  elements

```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];                  // element u[i][j]
            utmp[n*i + j] = tmp / 4;                // element utmp[i][j]
        }
    }
}
```

## Example 2: stencil computation using Jacobi solver

What tasks can be? Assume: 1) the innermost loop body takes  $t_{body}$  time units; and 2)  $n$  is very large, so that  $n - 2 \simeq n$

Task is ... (granularity)	Num. tasks	Task cost	$T_1$	$T_\infty$	Parallelism
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	1
Each iteration of i loop	$n$	$n \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot t_{body}$	$n$
Each iteration of j loop	$n^2$	$t_{body}$	$n^2 \cdot t_{body}$	$t_{body}$	$n^2$
$r$ consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$n^2 \cdot t_{body}$	$n \cdot r \cdot t_{body}$	$n \div r$
$c$ consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$n^2 \cdot t_{body}$	$c \cdot t_{body}$	$n^2 \div c$
A block of $r \times c$ iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$n^2 \cdot t_{body}$	$r \cdot c \cdot t_{body}$	$n^2 \div (r \cdot c)$

Finer grain task decomposition  $\rightarrow$  higher parallelism, but ...

## Example 2: stencil computation using Jacobi solver

... what if each task creation takes  $t_{create}$ ?

Task is ... (granularity)	Num. tasks	Task cost	Task creation ovh
All iterations of i and j loops	1	$n^2 \cdot t_{body}$	$t_{create}$
Each iteration of i loop	n	$n \cdot t_{body}$	$n \cdot t_{create}$
Each iteration of j loop	$n^2$	$t_{body}$	$n^2 \cdot t_{create}$
r consecutive iterations of i loop	$n \div r$	$n \cdot r \cdot t_{body}$	$(n \div r) \cdot t_{create}$
c consecutive iterations of j loop	$n^2 \div c$	$c \cdot t_{body}$	$(n^2 \div c) \cdot t_{create}$
A block of r x c iterations of i and j, respectively	$n^2 \div (r \cdot c)$	$r \cdot c \cdot t_{body}$	$(n^2 \div (r \cdot c)) \cdot t_{create}$

Trade-off between task granularity and task creation overhead

## Example 3: stencil computation using Gauss-Seidel solver

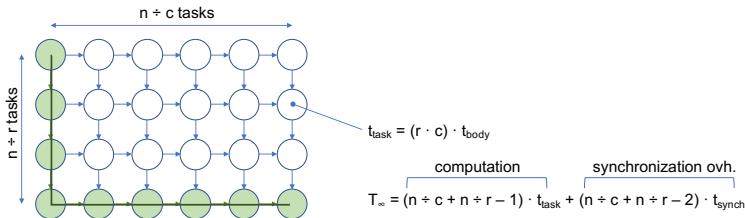
Stencil algorithm that **updates** (in place) each element of matrix  $u$  using its 4 neighbors, matrix size  $n \times n$  elements.

```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];                  // element u[i][j]
            u[n*i + j] = tmp / 4;                    // element u[i][j]
        }
    }
}
```

## Example 3: stencil computation using Gauss-Seidel solver

Assuming: 1) each task computes a block of  $r \times c$  iterations of the  $i$  and  $j$  loops, respectively; and 2) each task synchronization takes  $t_{synch}$



Again, trade-off between task granularity and task synchronisation overhead

# Outline

## Introduction

## Defining tasks and their implications

Video lesson 2

Vector sum example

Granularity and overheads

## Speed-up and efficiency

Video lesson 3

Other sources of overhead: data sharing overheads

# Outline

## Introduction

## Defining tasks and their implications

Video lesson 2

Vector sum example

Granularity and overheads

## Speed-up and efficiency

Video lesson 3

Other sources of overhead: data sharing overheads

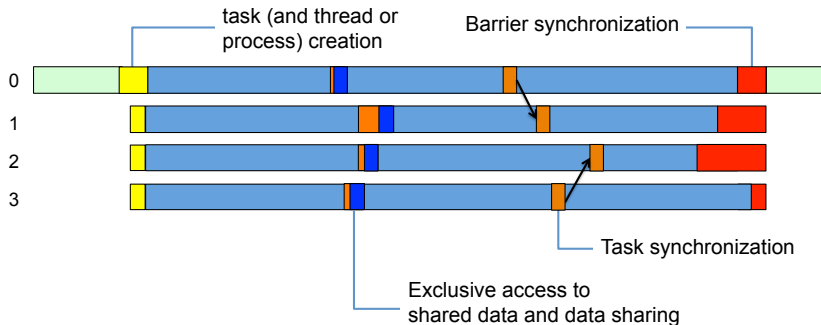
## Concepts in video lesson 3

- ▶ Task scheduling (mapping) on  $P$  processors
- ▶ Metrics:
  - ▶  $T_p$ : execution time on  $P$  processors
  - ▶ Speed-up  $S_p = \frac{T_1}{T_p}$
  - ▶ Efficiency  $E_p = \frac{S_p}{P}$
- ▶ Strong vs. weak scaling
- ▶ Amdahl's law, or the negative effect of the serial parts in your parallel application
  - ▶ parallel fraction  $\phi$
  - ▶  $S_p = \frac{1}{(1-\phi)}$  when  $p \rightarrow \infty$  (ideally, assuming all parallel regions scale to  $\infty$  processors)



## Sources of overhead until now ...

Parallel computing is not for free, we should account overheads (i.e. any cost that gets added to a sequential computation so as to enable it to run in parallel)



# Outline

## Introduction

## Defining tasks and their implications

Video lesson 2

Vector sum example

Granularity and overheads

## Speed-up and efficiency

Video lesson 3

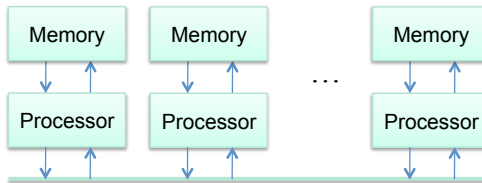
Other sources of overhead: data sharing overheads

## Other sources of overhead

- ▶ **Data sharing:** can be explicit via messages, or implicit via a memory hierarchy (caches)
- ▶ **Idleness:** thread cannot find any useful work to execute (e.g. dependences, load imbalance, poor communication and computation overlap or hiding of memory latencies, ...)
- ▶ **Computation:** extra work added to obtain a parallel algorithm (e.g. replication)
- ▶ **Memory:** extra memory used to obtain a parallel algorithm (e.g. impact on memory hierarchy, ...)
- ▶ **Contention:** competition for the access to shared resources (e.g. memory, network)

## How to model data sharing overhead?

We start with a simple architectural model in which each processor  $P_i$  has its own memory, interconnected with the other processors through an interconnection network.



- ▶ Processors access to local data (in its own memory) using regular load/store instructions
- ▶ We will assume that local accesses take zero overhead.

## How to model data sharing overhead?

- ▶ Processors can access remote data (in other processors) using a message-passing model (remote load instruction<sup>2</sup>)
- ▶ To model the time needed to access remote data we will use two components:
  - ▶ Start up: time spent in preparing the remote access ( $t_s$ )
  - ▶ Transfer: time spent in transferring the message (number of bytes  $m$ , time per byte  $t_w$ ) from/to the remote location

$$t_{access} = t_s + m \times t_w$$

- ▶ Synchronization between the two processors involved may be necessary to guarantee that the data is available

---

<sup>2</sup>Remote store is also possible, not used in our model.

# How to model data sharing overhead?

Assumptions (to make simpler the model)

- ▶ At a given moment, a processor  $P_i$  can only perform one remote memory access to another processor  $P_j$
- ▶ At a given moment, a processor  $P_i$  can only serve one remote memory access from another processor  $P_k$
- ▶ Both actions can be performed simultaneously:  $P_i \rightarrow P_j$  and  $P_i \leftarrow P_k$

## Back to example 2: Jacobi solver

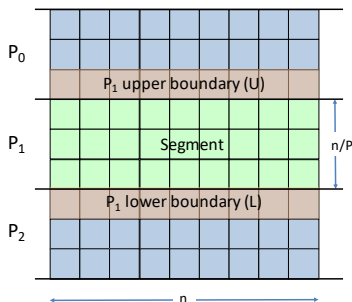
Stencil algorithm: each element of matrix `utmp` computed using 4 neighbor elements of matrix `u`, both matrices with  $n \times n$  elements

```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];                // element u[i][j]
            utmp[n*i + j] = tmp / 4;                // element utmp[i][j]
        }
    }
}
```

*Task definition:*  $n \div P$  consecutive iterations of  $i$  loop, being  $P$  the number of processors

## Example 2: data decomposition and movement



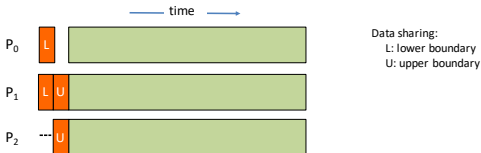
- ▶ Row decomposition, each processor with  $n^2/P$  elements of  $u$  matrix and  $n^2/P$  elements of  $utmp$  matrix (segment)
- ▶ Upper and lower boundaries, each with  $n$  elements
- ▶ No need to gather  $utmp$  in one of the processors at the end of the computation



## Example 2: parallel execution timeline

### ► Parallelization strategy:

1. Exchange boundaries with the two adjacent processors
2. Each processor computes the elements of its utmp segment



### ► Questions:

1. What is the data sharing time per segment assuming each boundary is accessed using a single message?
2. What is the total time (computation and data sharing)?
3. Obtain the expression for the speed-up  $S_P$

## Example 2: parallel execution time and speedup

With the same assumptions as before:

- ▶ The sequential execution time is  $T_1 = n^2 \times t_{body}$
- ▶ The parallel execution time, considering both computation and data movement, is:

$$T_P = \frac{n^2}{P} \times t_{body} + 2 \times (t_s + n \times t_w)$$

- ▶ The corresponding expression for the speedup is

$$S_P = \frac{T_1}{T_P} = \frac{n^2 \times t_{body}}{\frac{n^2}{P} \times t_{body} + 2 \times (t_s + n \times t_w)}$$

## Back to example 3: Gauss-Seidel solver

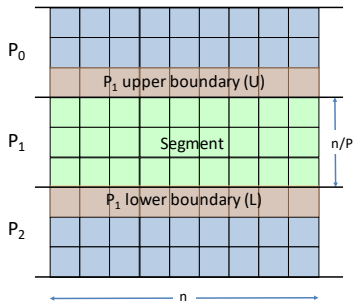
Stencil algorithm that **updates** (in place) each element of matrix  $u$  using its 4 neighbours, matrix size  $n \times n$  elements.

```
void compute(int n, double *u, double *utmp) {
    int i, j;
    double tmp;

    for (i = 1; i < n-1; i++) {
        for (j = 1; j < n-1; j++) {
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]
                  4 * u[n*i + j];                // element u[i][j]
            u[n*i + j] = tmp / 4;                  // element u[i][j]
        }
    }
}
```

*Task definition 1:*  $n \div P$  consecutive iterations of  $i$  loop, being  $P$  the number of processors

## Example 3: data decomposition and movement

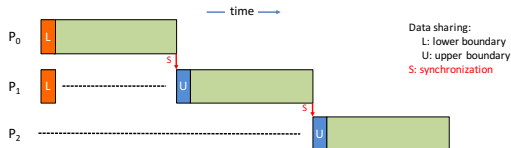


- ▶ Row distribution, each processor with  $n^2/P$  elements (segment)
- ▶ Upper and lower boundaries needed to compute segment, each with  $n$  elements (data sharing)
- ▶ No need to gather final  $u$  in one of the processors

## Example 3: parallel execution timeline

### ► Parallelization strategy:

1. Access to lower boundary L in next processor (if any)
2. Wait for termination of task in previous processor, if any (dependence)
3. Access to upper boundary U computed by that task (if any)
4. Apply stencil algorithm to segment



### ► Questions:

1. What is the data sharing time per segment?
2. What is the total time (computation and data sharing)?

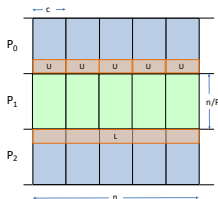
## Example 3: Gauss-Seidel solver

Stencil algorithm that **updates** (in place) each element of matrix  $u$  using its 4 neighbours, matrix size  $n \times n$  elements.

```
void compute(int n, double *u, double *utmp) {  
    int i, j;  
    double tmp;  
  
    for (i = 1; i < n-1; i++) {  
        for (j = 1; j < n-1; j++) {  
            tmp = u[n*(i+1) + j] + u[n*(i-1) + j] + // elements u[i+1][j] and u[i-1][j]  
                  u[n*i + (j+1)] + u[n*i + (j-1)] - // elements u[i][j+1] and u[i][j-1]  
                  4 * u[n*i + j];                // element u[i][j]  
            u[n*i + j] = tmp / 4;                  // element u[i][j]  
        }  
    }  
}
```

*Task definition 2:* block of  $n \div P$  by  $c$  consecutive iterations of  $i$  and  $j$  loops, respectively;  $P$  is the number of processors

## Example 3: blocking parallelization



### ► Data decomposition:

- Row distribution, each processor with  $n^2/P$  elements
- Tasks compute segments of  $n \div P$  rows by  $c$  columns

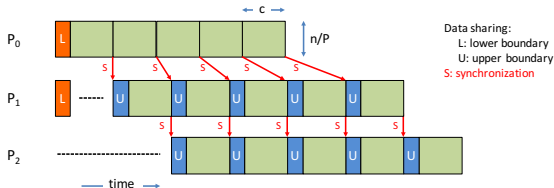
### ► Parallelization strategy:

**Once:** Access to  $n$  elements for lower boundary  $L$  (if any)

**For each block (task):**

- Wait for termination of task computing the same block in previous processor, if any (dependence)
- Access to the  $c$  elements in that block that belong to the upper boundary  $U$  (if any)
- Apply stencil algorithm to the block

## Example 3: parallel execution timeline with blocking



### Questions:

1. What is the overhead of data sharing (before and during the parallel computation)?
2. What is the total time (computation and data sharing)?
3. Is there an optimum value for  $c$ ?



## Example 3: parallel execution time and speedup

Assuming that  $t_{body}$  is the computation time for the innermost loop body and  $n$  is very large, so that  $n - 2 \simeq n$

$$T_P = \left(\frac{n}{c} + P - 1\right) \times \left(\frac{n}{P} \times c\right) \times t_{body} +$$

$$(t_s + n \times t_w) + \left(\left(\frac{n}{c} + P - 2\right) \times (t_s + c \times t_w)\right)$$

The corresponding expression for the speedup would be

$$S_P = \frac{T_1}{T_P} = \frac{n^2 \times t_{body}}{T_P}$$

## Example 3: optimum blocking factor

Assuming  $P \gg 2$ :

$$T_P \simeq \left(\frac{n}{c} + P\right)\left(\frac{n}{P} \times c\right)t_{body} + (t_s + n \times t_w) + \left(\frac{n}{c} + P\right)(t_s + c \times t_w)$$

The optimum block size  $c_{opt}$  is obtained applying the derivative to  $T_P$  and equal it to zero

$$\frac{\partial T_P}{\partial c} = n \times t_{body} - t_s \frac{n}{c^2} + P \times t_w = 0$$

$$c_{opt} = \sqrt{\frac{n \times t_s}{n \times t_{body} + P \times t_w}} = \sqrt{\frac{t_s}{t_{body} + t_w \frac{P}{n}}}$$

If  $n \gg P$  then

$$c_{opt} \simeq \sqrt{\frac{t_s}{t_{body}}}$$

# Parallelism (PAR)

## Unit 2: Understanding parallelism

Eduard Ayguadé, José Ramón Herrero (Q1),  
Daniel Jiménez and Gladys Utrera

Computer Architecture Department  
Universitat Politècnica de Catalunya

Course 2022/23 (Spring semester)