Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based Parallel Programming Models: The Convergence of High-Performance and Cyber-Physical Computing Domains

Eduardo Quiñones
{eduardo.quinones@bsc.es}

ACACES, Fiuggi, 2021, 12-18 Sep

Welcome!

What are your expectations on this course?
(or why did you choose this course?)

Goal of this course:

Understanding the benefits and the research challenges of applying task-based parallel programming models (OpenMP and COMPSs) when developing Cyber-Physical Systems (CPS)

Structure of the Course

1. **Introduction**
   - Cyber-Physical Systems (CPS)
   - Task-Based Parallel Programming Models: OpenMP and COMPSs
2. **Lesson 2: OpenMP**
   - API, execution model and memory model
   - Challenges of applying OpenMP to CPS
3. **Lesson 3: OpenMP and CPS**
   - Functional correctness and time predictability
     - OpenMP tracing
   - Model driven engineering and OpenMP
4. **Lesson 4: Distribution across the compute continuum: COMPSs**
   - API, execution model and memory model
   - Functional correctness and time predictability
     - COMPSs tracing
   - A real CPS: A smart mobility application

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based Parallel Programming Models: The Convergence of High-Performance and Cyber-Physical Computing Domains

Lesson 1: Introduction

Eduardo Quiñones
{eduardo.quinones@bsc.es}

ACACES 2021, Fiuggi

Outline

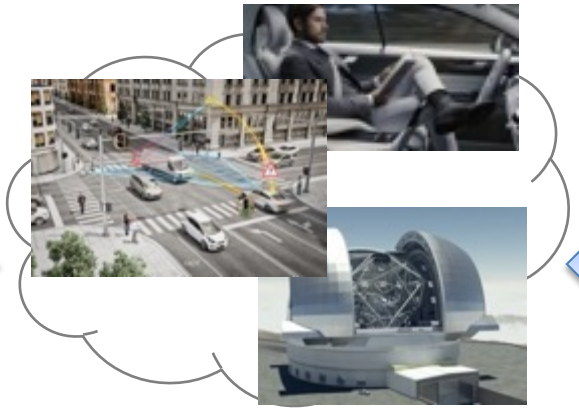
- **Cyber-Physical Systems (CPS)**
  - **Requirements and computing infrastructure**
  - **Types of CPS**
  - **Software development complexity**
- Task-based Parallel Programming Models
  - Introduction to parallel programming models
  - OpenMP and COMPSs
  - Model Driven Engineering

The Convergence of High-Performance and Real-Time Computing Domains

High Performance Computing (HPC) Systems

Computing Spectrum

Cyber-Physical Systems (CPS)



Massive parallel systems that operates **<u>as fast as possible</u>**

Network of HW/SW components (*cyber*) that **must** operate **correctly** in response to its (*physical*) inputs from a **<u>functional</u>** and **<u>non-functional</u>** perspective

<u>Performance</u> becomes as important as other non-functional requirements!

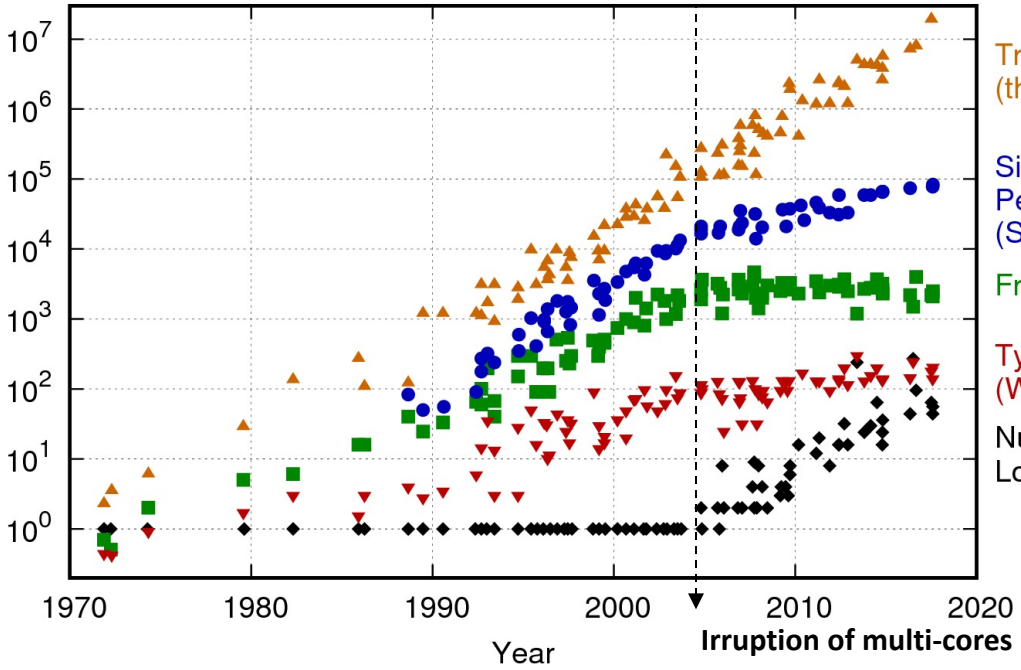
CPS vs. Embedded Systems

- **CPS** integrates the computation, networking and physical processes
  - **Non-functional requirements (NFR)** are inherited from the Cyber-physical interactions
- **Embedded systems** are typically responsible of the control part of the CPS
  - Embedded systems must fulfil the NFR from CPS

Non-functional Requirements (NFR)

- Inherited due to the cyber-physical interactions, e.g.,
  - **Real-time:** The end-to-end response time (from sensor to actuator) must be within a given time budget
  - **Power/Thermal:** The energy/temperature of the computing elements must be within a given budget due to power supply/operational environment limitations
  - **Safety:** CPS must be built guaranteeing the correctness and integrity of its operation
  - **Security:** CPS must prevent external elements not to affect the correctness and integrity of the system
- **Performance:** CPS must provide the computing power to implement advanced functionalities

Processor Design Trend



Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

Heterogeneous and Parallel computing becomes key to cope with performance requirements

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

Converging HPC and CPS: Parallel and Heterogeneous Computing

HPC Domain (~300W)

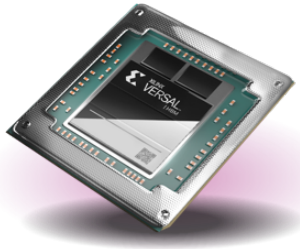
NVIDIA A100
(GPU-based)

Intel Xeon Series
(40-core fabric)

Embedded Domain (~10-20W)

NVIDIA Jetson Family
(GPU-based)

Kalray MPPA Coolidge
(80-core fabric)

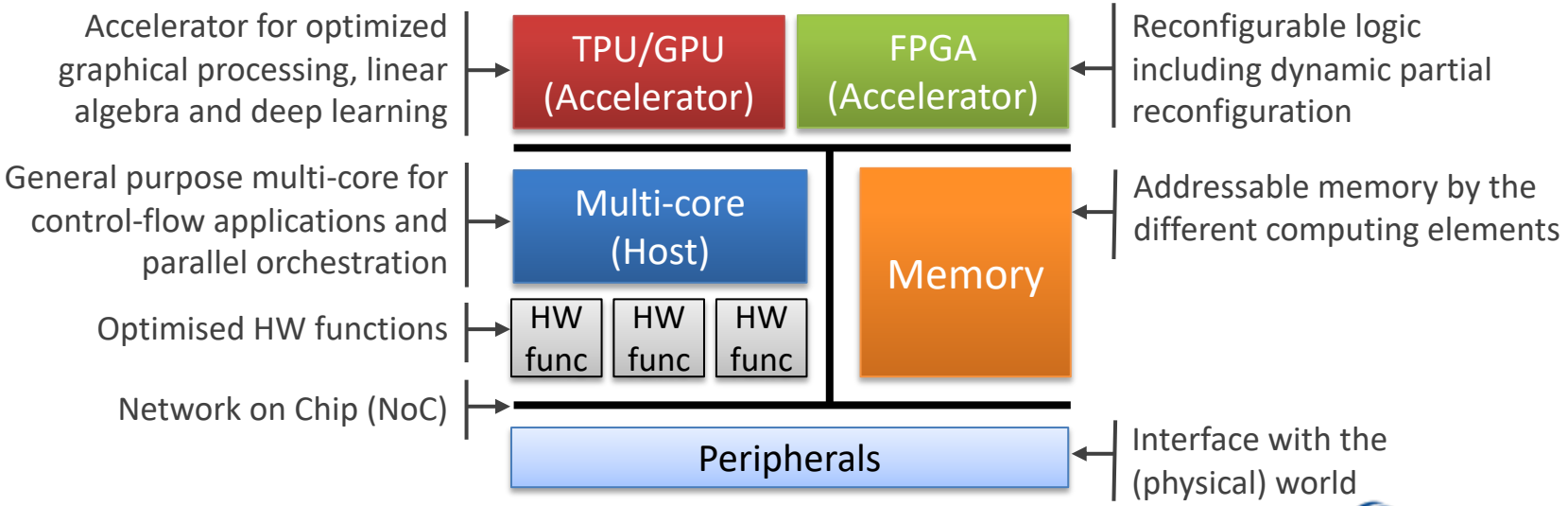


Xilinx Versal
(GPU- and FPGA-based with dynamic partial reconfiguration)

CPS takes full benefit of heterogeneous computing due to the dedicated accelerators and low power consumption

Heterogeneous and Parallel Processor Architectures

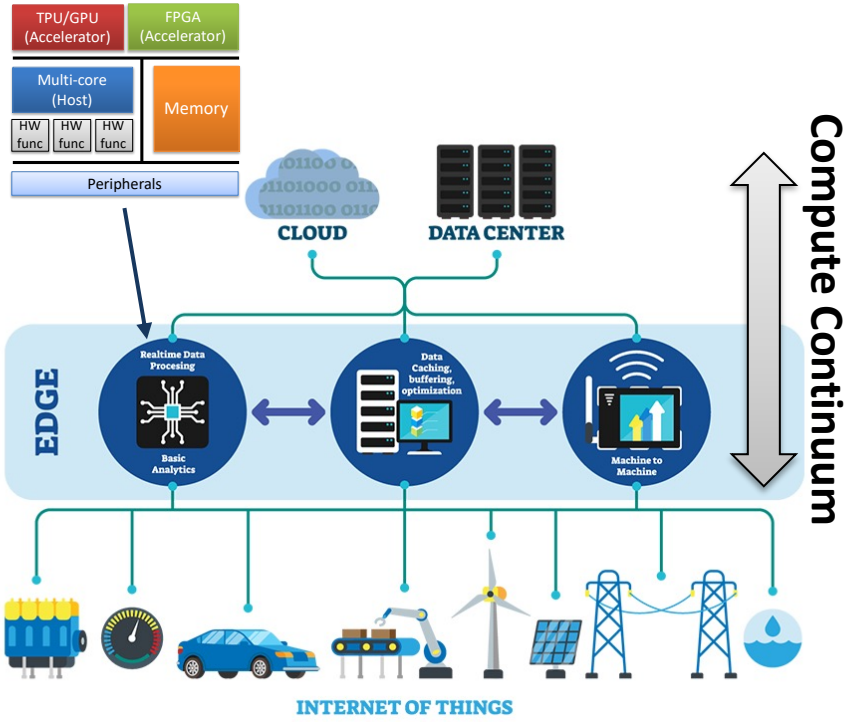
Host-centric paradigm: The parallel computation is orchestrated by the general-purpose multi-core



Compute Continuum: From Edge to Cloud

CPS is **suitable for IoT** and **edge** computing paradigms

- The computation is selectively move close to the data-sources so decision-making occurs as close as possible
  - Enables faster real-time processing, higher privacy control and lower network costs
  - The use of powerful heterogeneous embedded processor architectures becomes fundamental
- Cloud computing is used to execute computational intensive and batch processes

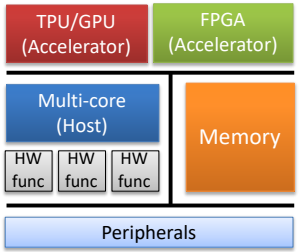


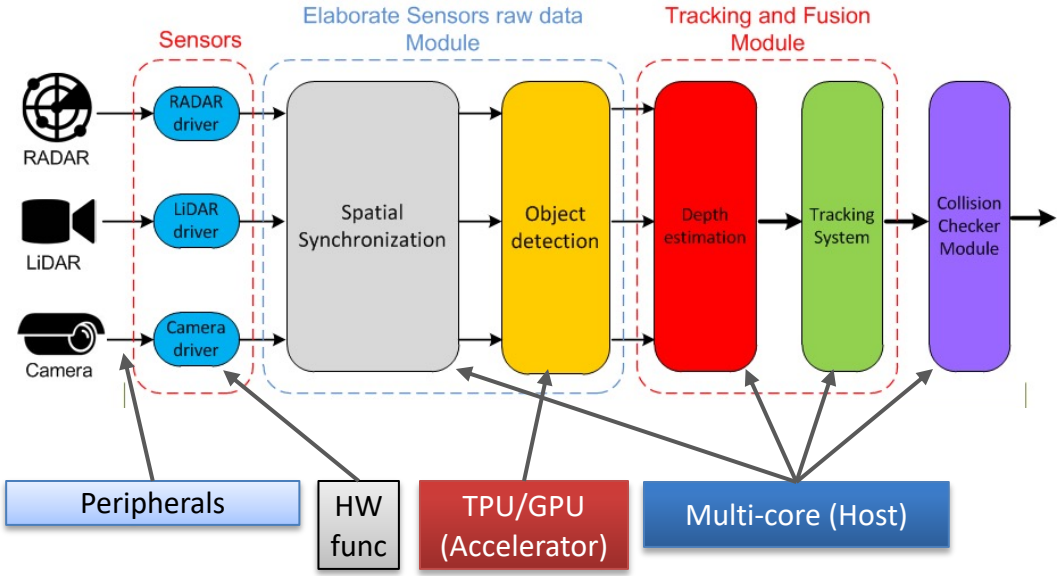
Two types of CPS

- Tighly-couple CPS
  - Subsystems execute within the same processors or in a controlled (and reliable) network of few processors (e.g., *automotive domain*)
- Loosely-couple CPS
  - Subsystems execute within a non-reliable network of heterogenous computing elements, i.e., the compute continuum (e.g., *smart cities*)
  - Some subsystems may implement tighly-couple CPS

Tighly Couple CPS Example: Vehicle Collision Detection

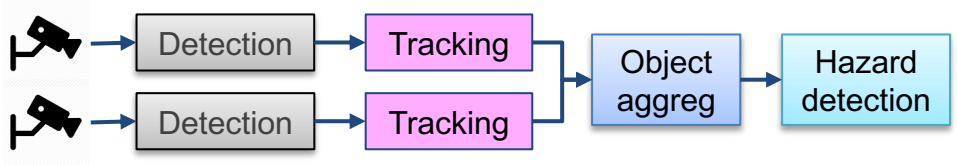
An Adavanced Driving Assistant System (ADAS) used to identify objects in front of vehicles and detect potential collisions

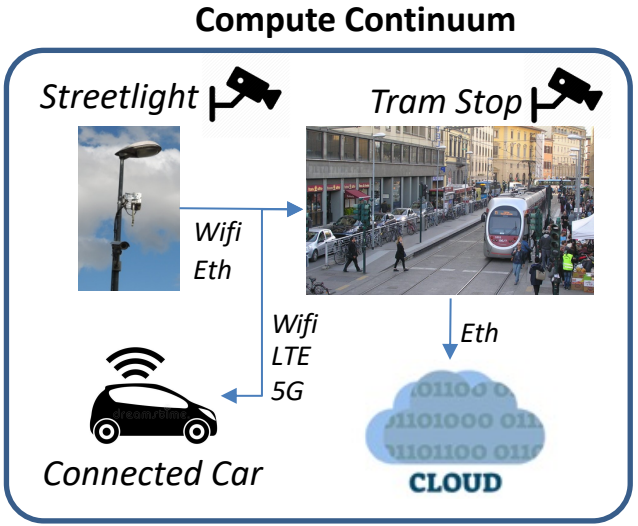




Loosely-couple CPS Example : Smart City Collision Detection

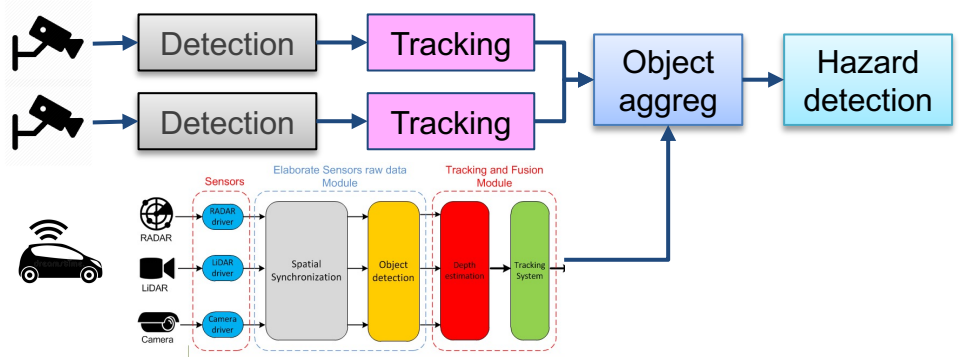
Sensing capabilities of vehicles and cities can be combined to identify hazardous situations

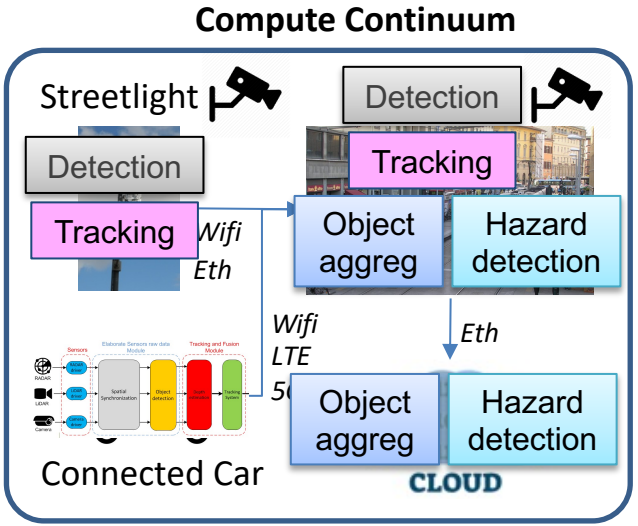




Loosely-couple CPS Example: Smart City Collision Detection

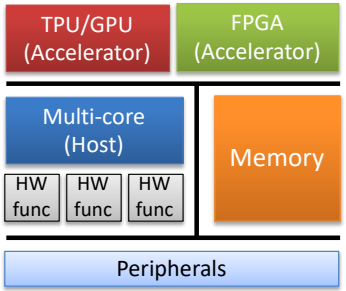
Sensing capabilities of vehicles and cities can be combined to identify hazardous situations

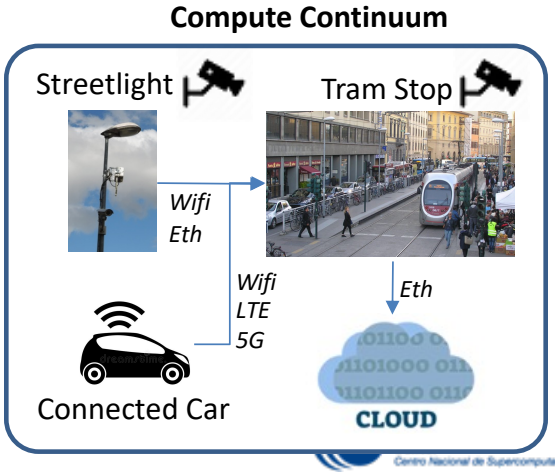




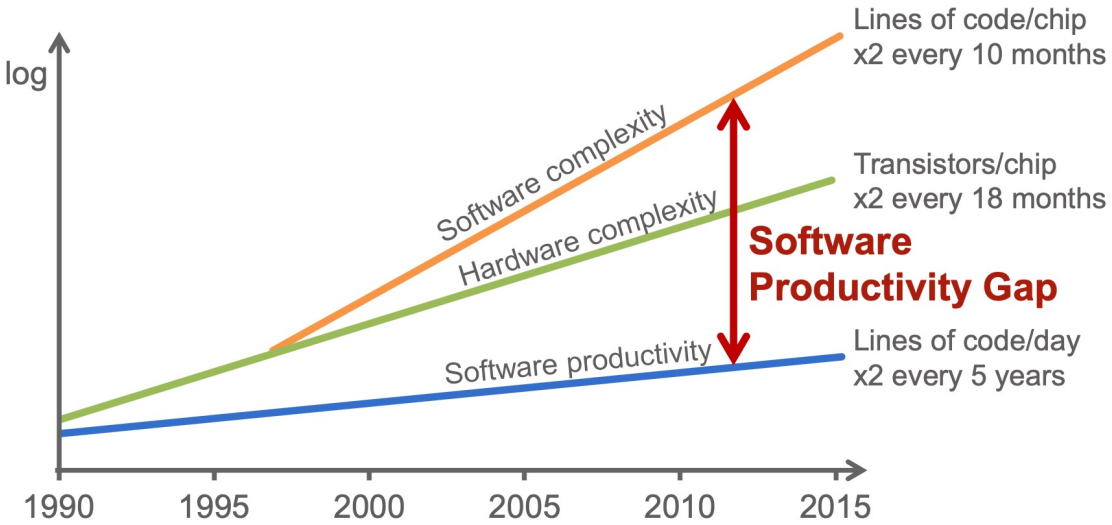
How would you develop such a CPS?

1. **Exploit the parallel performance capabilities** of the (different) processor architectures
2. **Efficiently distribute** the data-analytics workflow across the compute continuum
3. **Guarantee functional correctness and the non-functional requirements** of the CPS





SW Development Complexity



Source: ITRS & Hardware-dependent Software, Ecker et al., Springer

This course will present the task-based parallel programming model to efficiently:

1. **Exploit parallelism**
2. **Distribute computation** across the compute continuum
3. **Reason about** the functional and non-functional correctness

Outline

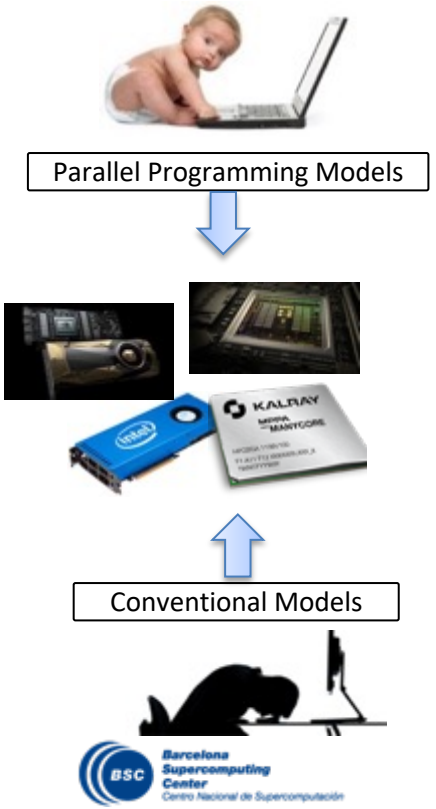
- Cyber-Physical Systems (CPS)
  - Requirements and computing infrastructure
  - Types of CPS
  - Software development complexity
- **Task-based Parallel Programming Models**
  - **Introduction to parallel programming models**
  - **OpenMP and COMPSs**
  - **Model Driven Engineering**

Parallel Programming Models

- A set of programming elements to **describe** the parallel behaviour of an application and **abstract** the complexities of the underlying parallel platform
  - **Granularity level of parallelism** exploited: instruction, statement, loop, procedural
  - **Synchronization** model: coarse-grain, fine-grain
  - **Execution model**: fork-join, thread-pool, etc.
  - **Memory model**: Shared, distributed
- Commonly built on top of a base programming language

Parallel Programming Models

- Mandatory to enhance **productivity**
  - **Programmability.** Abstracts the parallelism while hiding the underlying computing platform complexities
  - **Portability/scalability.** The same source code is valid in different parallel platforms
  - **Performance.** Rely on run-time mechanisms to exploit the performance capabilities of parallel platforms



Types of Parallel Programming Models

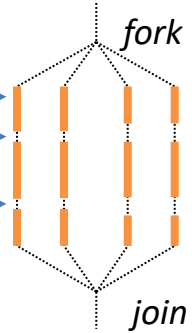
- **Hardware-centric**
  - Provide a user-friendly interface to tune the application to native platform features, e.g., *NVIDIA CUDA*
  - **None portable**
- **Application-centric**
  - The application must fulfill the execution model to exploit parallelism, e.g. *OpenCL*
  - May require a full rewriting process of the application, **impacting on programmability**
- **Parallelism-centric**
  - Parallelism is expressed by means of constructs various levels of abstraction, e.g. *POSIX threads, OpenMP, OpenACC, MPI, COMPSs, Spark, Ray*
  - This approach allows flexibility and expressiveness, while decoupling design from implementation

Type of Parallelism

- ***Structured parallelism:*** The parallel execution follows a regular pattern
  - Very suitable for parallel loops
  - E.g., fork-join, pipeline
- ***Unstructured parallelism:*** The parallel execution does not fit within a pattern, or it change dynamically
  - Suitable for procedural-level parallelism
  - E.g., **tasking**

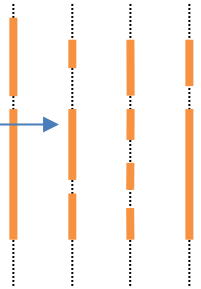
```
parallel for (i=LB; i<UB; ++i)
    do_computation();
endfor
```

Distribute the loop iteration among parallel units (threads)

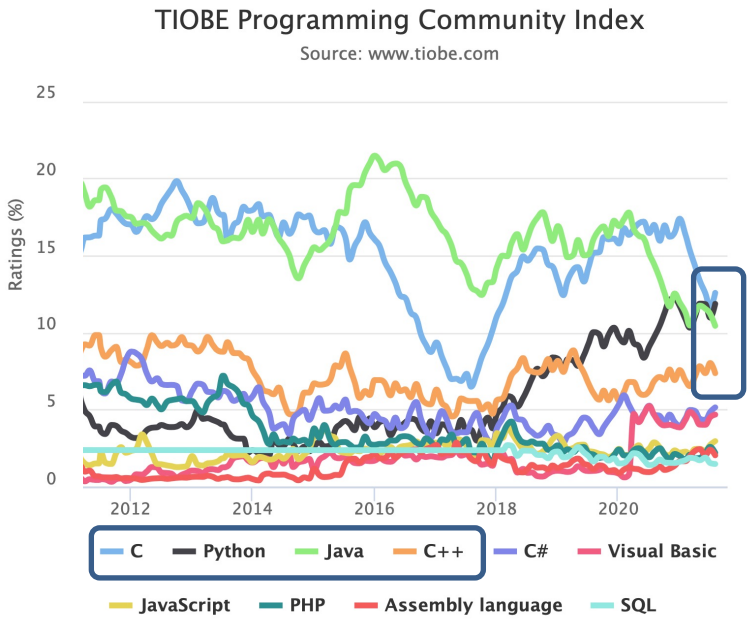


Distribute tasks among parallel units (threads)

```
task
    do_computation_1();
endtask
…
task
    do_computation_N();
endtask
```



Parallel Programming Models and Programming Languages



Model	Base Language	Type of PPM	Type of architect	Type of Parallelism
CUDA	C/C++, Python	HW-centric	NVIDIA GPU	Struct/ Unstruct
OpenCL	C/C++	App-centric	GPU/ FPGAs	Struct
OpenMP	C/C++	Parallel-centric	Shared mem	Struct/ **Unstruct**
Pthreads	C/C++	Parallel-centric	Shared mem	Unstruct
MPI	C/C++, Python	Parallel-centric	Distributed mem	Unstruct
COMPSs	C++, Java Python	Parallel-centric	Distributed mem	**Unstruct**
Spark	Java, Python	Parallel-centric	Distributed mem	Struct
Ray	C++,Java Python	Parallel-centric	Distributed mem	Unstruct

Why OpenMP?

- **Mature language** constantly reviewed (last release Nov 2020, v5.1)
  - Defacto industrial standard in HPC
  - Active research community with an **increasing interest** on the CPS domain
- **Productivity** in parallel programming
  - Performance
    - Exploitation of structured and unstructured fine-grain parallelism coupled with an advanced accelerator model
    - Powerful task-based model supporting fine-grain synchronization mechanisms based on data-dependencies among tasks
    - Performance analysis tools of the parallel execution
  - Portability
    - Supported by many chip vendors used in CPS (Intel, IBM, ARM, NVIDIA, TI, Gaisler, Kalray)
  - Programmability
    - Interoperability with other programming models (e.g., CUDA, OpenCL)
    - Allows incremental parallelization (#pragma omp) that can be easily compiled sequentially

Why COMPSs?

- **Programming distribute framework** highly inspired in the OpenMP tasking model
  - Programs are written sequentially in Python, Java or C++
  - The code is annotated to describe asynchronous procedures (task) than can execute in parallel
    - Includes a fine-grain synchronization mechanism based on data dependencies among tasks
- **Productivity** in distributed programming
  - Performance
    - Exploitation of distributed computation in heterogeneous HPC and edge/cloud environments
    - Powerful performance analysis tool of the distributed execution
  - Portability
    - Supports many HPC and cloud technologies: DFS, Docker, Kubernetes, Serverless, etc.
  - Programmability
    - Interoperability with other programming models (e.g., OpenMP)
    - Currently available for Python, Java and C++
    - Allows incremental parallelization (`@task`) to easily execute sequentially

OpenMP Tasking Model

Sequential version

```
void main() {
     int x,y;
     f1(&x,&y);
     f2(x);
     f3(y);
}
```

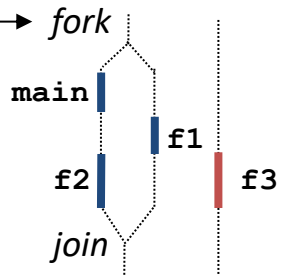
Executes on the host

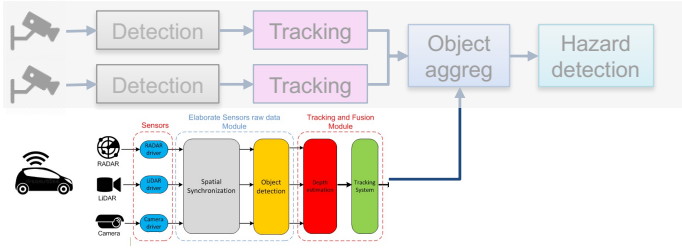
Executes on the accelerator

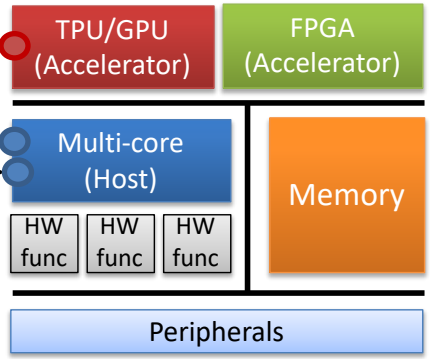
OpenMP version

```
void main() {
     #pragma omp parallel
     #pragma omp master
     {
          int x,y;
          #pragma omp task depend(out:x,y)
          { f1(&x,&y); }
          #pragma omp task depend(in:x)
          { f2(x); }
          #pragma omp target map(to:y) depend(in:y)
          { f3(y); }
     }
}
```

1. Open parallelism







2. Tasks executed on the host

3. Tasks executed on the host and accelerator when f1 completes

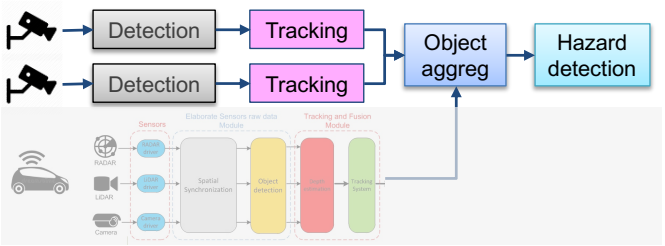
COMPSs Tasking Model

Sequential version

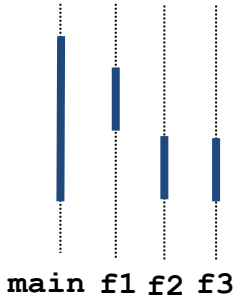
```
def f1():
    …
    return x,y
def f2(x):
    …
def f3(y)
    …
def main():
    x,y=f1()
    f2(x)
    f3(y)
```

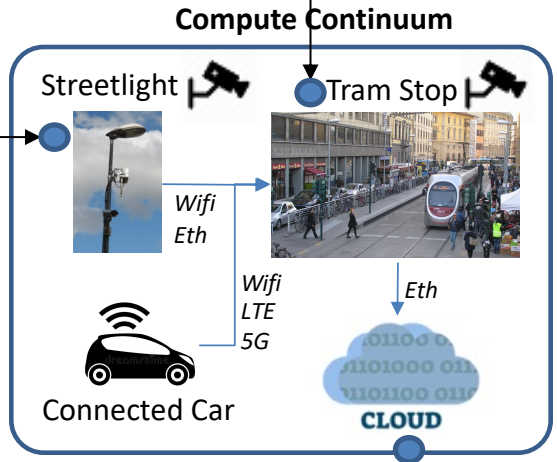
COMPSs version

```
@task(x=OUT,y=OUT)
def f1():
    …
    return x,y
@task(x=IN)
def f2(x):
    …
@task(y=IN)
def f3(y)
    …
def main():
    x,y=f1()
    f2(x)
    f3(y)
```



Tasks executed across the compute continuum





Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Principle behind Tasking Models

- Tasking provides a great expressiveness to describe the parallel nature of applications
  - Developers specify **what** the application does and not **how** it is done
  - The parallel framework is responsible of orchestrating the execution
- Tasking facilitates programmability, but …
  - … complicates deriving **functional** and **non-functional correctness**

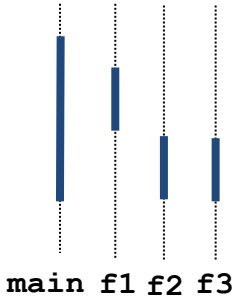
Computation is not fully controlled by the programmer but by the parallel framework

"I'm a software engineer, so I can confirm it works by magic."

Main Factors Impacting Parallel Execution

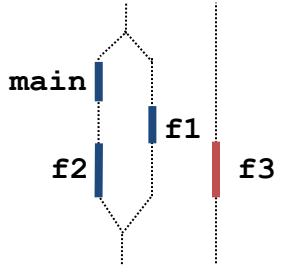
COMPSs version

```
@task(x=OUT,y=OUT)
def f1():
    …
    return x,y
@task(x=IN)
def f2(x):
    …
@task(y=IN)
def f3(y)
    …
def main():
    x,y=f1()
    f2(x)
    f3(y)
```

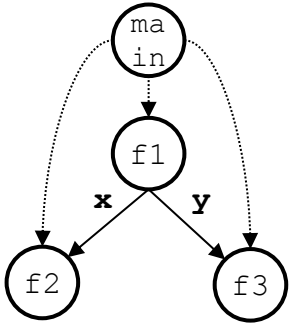


OpenMP version

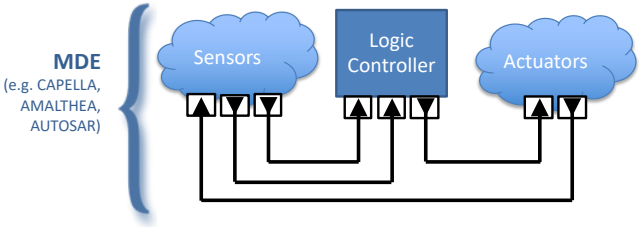
```
void main() {
    #pragma omp parallel
    #pragma omp master
    {
        int x,y;
        #pragma omp task depend(out:x,y)
        { f1(&x,&y); }
        #pragma omp task depend(in:x)
        { f2(x); }
        #pragma omp target map(to:y) depend(in:y)
        { f3(y); }
```



1. Parallel structure of the application (including data usage): **Task Dependency Graph (TDG)** or **Direct Acyclic Graph (DAG)**
2. The execution and memory model: The **Runtime Scheduler** responsible of mapping task to parallel units



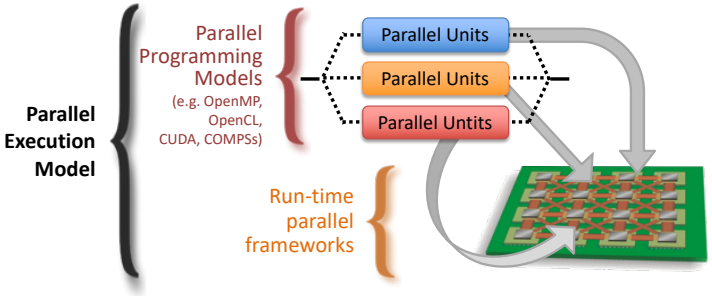
Model Driven Engineering and Parallel Programming Models



Model Driven Engineering (MDE) in CPS

1. Construction of complex systems
2. **Formal verification** of functional and non-functional requirements with **composability** features
3. **Correct-by-construction paradigm** by means of code generation
   - Suitable only for single-core execution or with very limited multi-core support

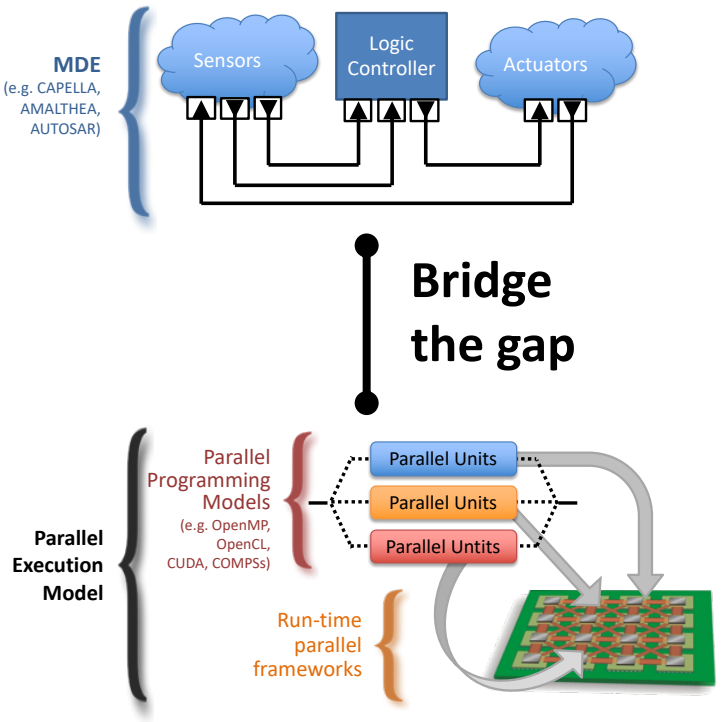
Gap between the MDE used for CPS and the PPM supported by parallel platforms



Parallel Programming Models

1. Mandatory for **SW productivity** in terms of
   - Programmability: Parallel abstraction while hiding HW complexities
   - Portability: Compatibility multiple HW platforms
   - Performance: Exploiting parallel capabilities of underlying HW
2. **Efficiet offloading** to HW acceleration devices for an energy-efficient parallel execution

Model Driven Engineering and Parallel Programming Models



1. **Synthesis methods** for an efficient generation of parallel source code, while keeping non-functional and composability guarantees
2. **Run-time parallel frameworks** that guarantee system correctness and exploit the performance capabilities of parallel architectures
3. **Integration** of parallel frameworks into MDE frameworks

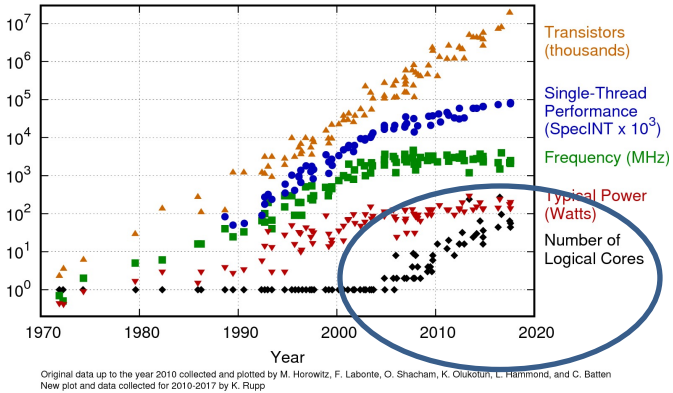
Conclusion: CPS and Parallel Computation

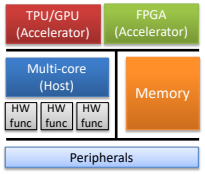
1. CPS **requires parallel computation** to cope with the performance requirements of the most advanced functionalities, but…
2. … current parallel frameworks remove from developers the responsibilty of managing the parallel execution, **difficulting deriving guarantees** of functional and non-functional correctness
3. … most CPS are implemented using **model driven engineering** approaches

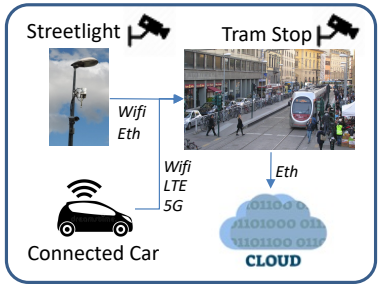
This course will present the benefits and challenges of applying tasking parallel programming model

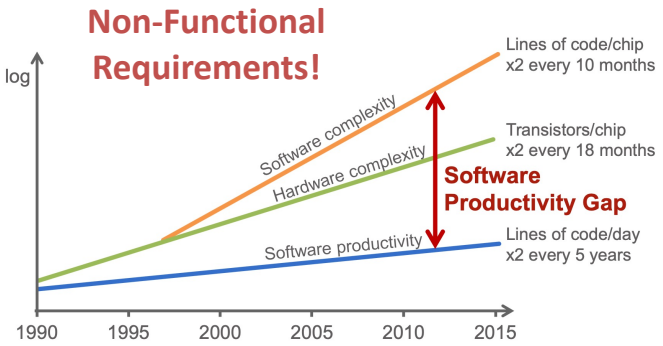
- Focus on two specific parallel programming languages: OpenMP and COMPSs
- The same concepts applies to other tasking languages

Recap of Lesson 1 in one slide









Tasking Model

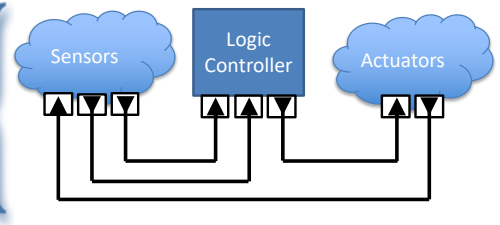
OpenMP

COMPSs

Challenge!

Computation is not fully controlled by the programmer but by the parallel framework

MDE (e.g. CAPELLA, AMALTHEA, AUTOSAR)



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based Parallel Programming Models: The Convergence of High-Performance and Cyber-Physical Computing Domains

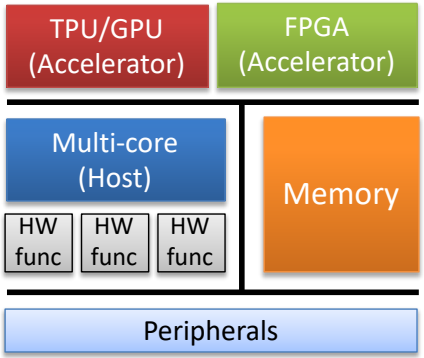
Lesson 2: OpenMP

Eduardo Quiñones
{eduardo.quinones@bsc.es}

ACACES 2021, Fiuggi

Outline

- OpenMP API
- Execution Model and Memory Model
- Spawning and Distributing Parallelism
- Synchronization and Data-Sharings
- Challenges of applying OpenMP to CPS
- Conclusions



OpenMP API

Compiler directives

```
#pragma omp parallel num_threads(4)
{…}
```

- Annotations in the source code
- Can be easily ignored by the compiler, allowing for incremental parallelization
- Directives can include **clauses** to define properties of the directive

Runtime library routines

```
omp_set_num_threads(4);
```

- Get/Set runtime information from source code

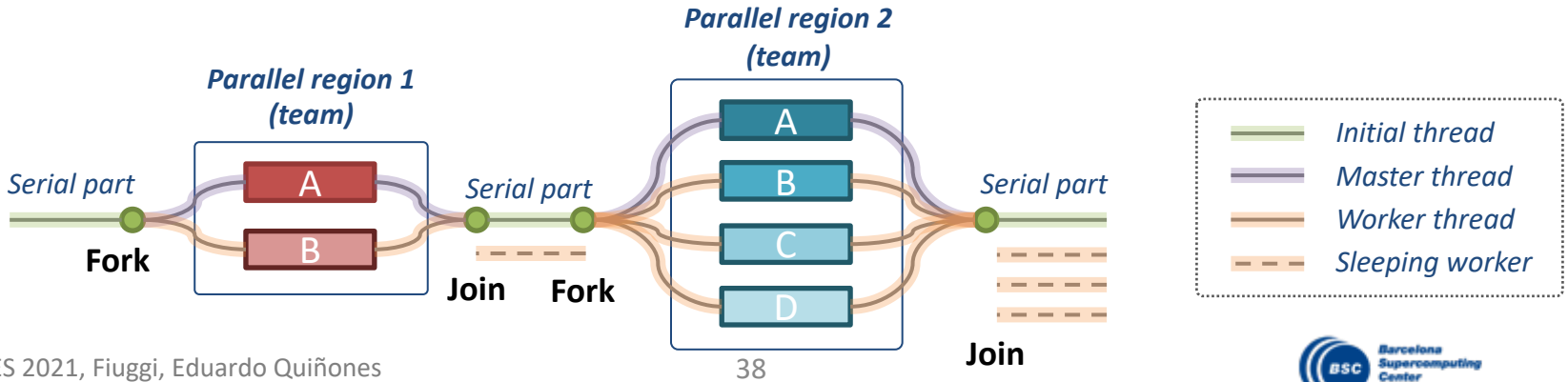
Environment variables

```
sh$ OMP_NUM_THREADS=4 ./openmp_exec
```

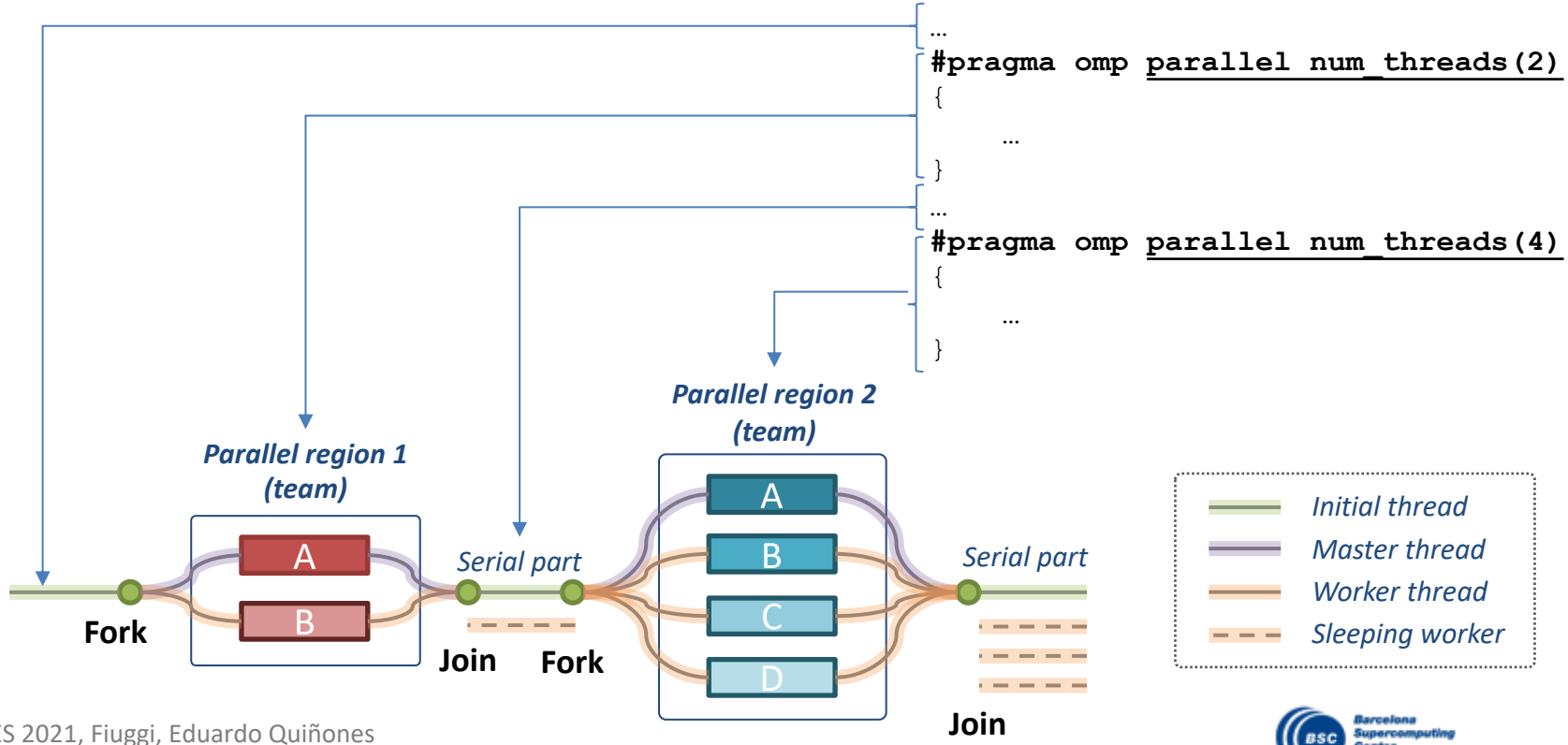
- Set runtime information at execution time

Execution Model: Fork-Join

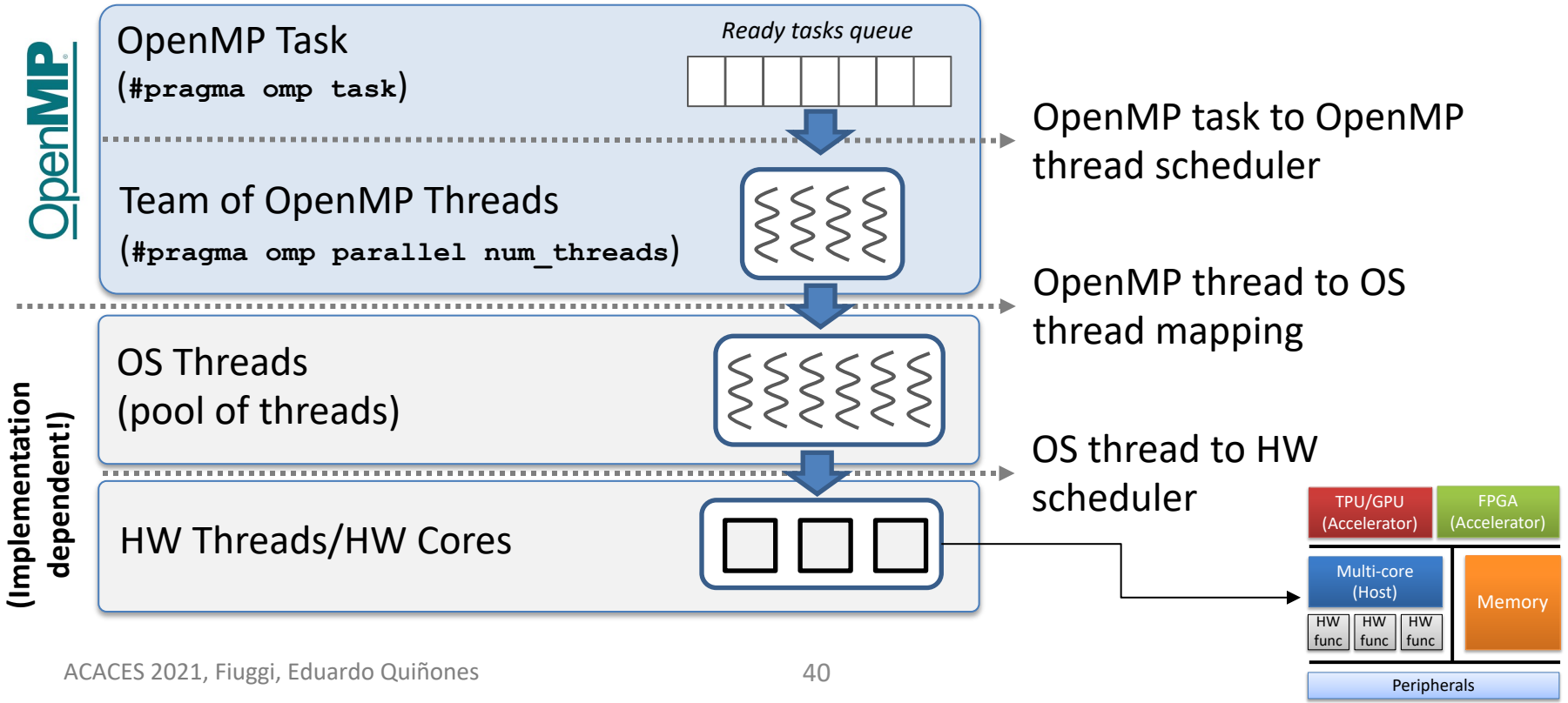
- OpenMP programs start execution with a unique **initial thread**
- Worker threads are spawned in **parallel regions** (`#pragma omp parallel`)
- The thread encountering a parallel region becomes the master thread, and together with worker threads, form a **team**
- Worker threads are destroyed (or put to sleep) between parallel regions



Execution Model: Fork-Join



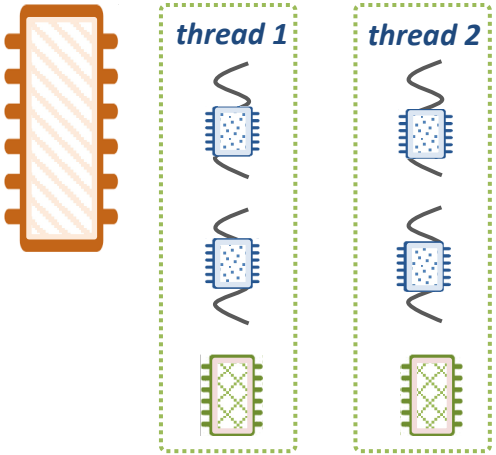
Execution Model: Abstraction Layers



Memory Model: Relaxed-consistency shared-memory

- Different views of the memory

Main memory	Shared for all threads
Temporary view	Copy of the main memory for a given thread and a region of the execution
Threadprivate memory	Particular to each thread (**not recommended to be used!**)

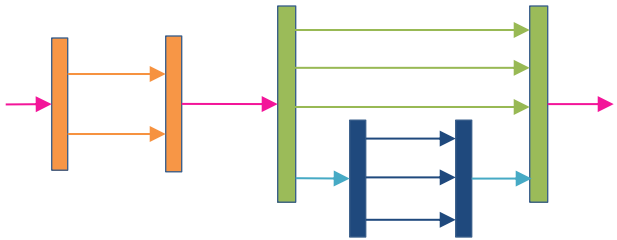


- Designed for shared memory nodes (UMA/NUMA)
  - Extended for heterogeneous computing nodes, i.e., host + accelerator(s)
- The access to variables can be **shared** or **private**
  - `shared, firstprivate, private, lastprivate` clauses
- Memory consistency is enforced by (implicit/explicit) flush operations

Spawning Parallelism

- Parallelism is spawned when a **`parallel`** construct is found
  - Threads of a team are synchronized when a **`barrier`** construct is found
  - There is an implicit barrier at the end of a parallel region.
- Parallel regions can be nested.
- The number of threads suitable for each region can be defined by the programmer
  - Construct clause: **`num_threads(4)`**
  - Runtime library routine: **`omp_set_num_threads(4)`**
  - Environment variable: **`OMP_NUM_THREADS=4`**

```
…
#pragma omp parallel num_threads(2)
{…}
…
#pragma omp parallel num_threads(4)
{
  …
  #pragma omp parallel num_threads(3)
  {…}
  …
}
…
```



Hands-on: Hello World Program

```
#include <iostream>
#include <omp.h>

int main (int argc, char* argv[])
{
    #pragma omp parallel num_threads(4)
    {
        std::cout << "Hello world" << std::endl;
        std::cout << "I am thread" << omp_get_thread_num()
                  << " of " << omp_get_num_threads()
                  << std::endl;
    }
    return 0;
}
```

- ***How many messages?***
- ***In which order?***

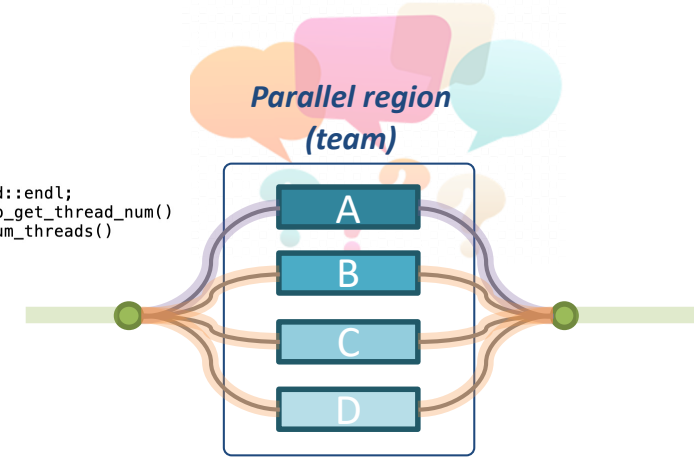
Hands-on: Hello World Program

```
Hello worldHello world
Hello worldI am thread 0 of Hello world4

I am thread 2 of 4
I am thread 1 of 4

I am thread 3 of 4
```

```
Hello world
I am thread 0 of 4
Hello world
I am thread Hello world2 of 4

I am thread 3 of 4
Hello world
I am thread 1 of 4
```

```
Hello worldHello world
I am thread 0Hello world
I am thread 1 of 4
 of 4
Hello world
I am thread 2 of 4

I am thread 3 of 4
```

```
#include <iostream>
#include <omp.h>

int main (int argc, char* argv[])
{
    #pragma omp parallel num_threads(4)
    {
        std::cout << "Hello world" << std::endl;
        std::cout << "I am thread" << omp_get_thread_num()
                  << " of " << omp_get_num_threads()
                  << std::endl;
    }
    return 0;
}
```



- ***How many messages? 4***
- ***In which order? UNDETERMINED***

Distributing Parallelism

- **Thread-centric model**
  - Conceptual abstraction of user-level threads
  - Structured data-parallelism
  - Representative constructs: **`for`** and **`sections`**

- **Task-centric model** *(introduced in v3.0, May 2008)*
  - Oblivious of the physical layout
  - Structured and unstructured data- and task- parallelism
  - Representative constructs: **`task`** and **`taskloop`**

Focus of this course!

- ➢ **Prescriptive**
- ✓ Less overhead
- ✓ Highly tunable

- ➢ **Descriptive**
- ✓ Dynamic parallelism
- ✓ Fine-grain synchronization

Distributing parallelism with the thread model

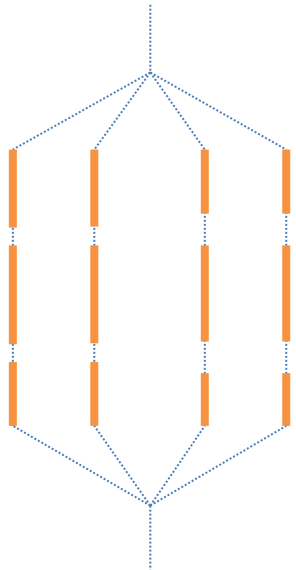
Parallel loops:

```
#pragma omp parallel num_threads(4)
#pragma omp for
for (i=LB; i<UB; ++i) {
     do_computation();
}
```

Parallel sections:

```
#pragma omp parallel num_threads(4)
#pragma omp sections
{
    #pragma omp section
    { do_computation_1(); }
    …
    #pragma omp section
    { do_computation_N(); }
}
```

Distribute [*loop-chuncks* / *structured blocks*] *among threads*



for loop schedule clause

static:

Assign a consecutive block of iterations to each thread in a round-robin fashion

static,n:

Define chunk size to enhance load balance although introducing overhead

dynamic,n:

Allow threads to fetch chunks as they are idle; chunck size can be defined as well

guided,n:

Chunk size is proportional to the number of unassigned iterations divided by the number of threads; chunck size can be defined as well

Hands-on: `for` loop

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 20

int main (int argc, char* argv[])
{
    int i;
    #pragma omp parallel num_threads(4)
    #pragma omp for schedule(static)
    for(i=0; i<N; i++)
    {
        printf("Thread ID %d - Iter %d\n",
                omp_get_thread_num(), i);
    }
    return 0;
}
```

- ***How many messages?***
- ***In which order?***

Hands-on: `for` loop

static

```
Thread ID 0  - Iter 0
Thread ID 0  - Iter 1
Thread ID 0  - Iter 2
Thread ID 0  - Iter 3
Thread ID 0  - Iter 4
Thread ID 3  - Iter 15
Thread ID 3  - Iter 16
Thread ID 3  - Iter 17
Thread ID 3  - Iter 18
Thread ID 3  - Iter 19
Thread ID 2  - Iter 10
Thread ID 2  - Iter 11
Thread ID 2  - Iter 12
Thread ID 2  - Iter 13
Thread ID 2  - Iter 14
Thread ID 1  - Iter 5
Thread ID 1  - Iter 6
Thread ID 1  - Iter 7
Thread ID 1  - Iter 8
Thread ID 1  - Iter 9
```

static

```
Thread ID 0  - Iter 0
Thread ID 0  - Iter 1
Thread ID 0  - Iter 2
Thread ID 0  - Iter 3
Thread ID 0  - Iter 4
Thread ID 2  - Iter 10
Thread ID 2  - Iter 11
Thread ID 2  - Iter 12
Thread ID 2  - Iter 13
Thread ID 2  - Iter 14
Thread ID 3  - Iter 15
Thread ID 3  - Iter 16
Thread ID 3  - Iter 17
Thread ID 3  - Iter 18
Thread ID 3  - Iter 19
Thread ID 1  - Iter 5
Thread ID 1  - Iter 6
Thread ID 1  - Iter 7
Thread ID 1  - Iter 8
Thread ID 1  - Iter 9
```

static,2

```
Thread ID 3  - Iter 6
Thread ID 3  - Iter 7
Thread ID 3  - Iter 14
Thread ID 3  - Iter 15
Thread ID 2  - Iter 4
Thread ID 2  - Iter 5
Thread ID 2  - Iter 12
Thread ID 2  - Iter 13
Thread ID 0  - Iter 0
Thread ID 0  - Iter 1
Thread ID 0  - Iter 8
Thread ID 0  - Iter 9
Thread ID 0  - Iter 16
Thread ID 0  - Iter 17
Thread ID 1  - Iter 2
Thread ID 1  - Iter 3
Thread ID 1  - Iter 10
Thread ID 1  - Iter 11
Thread ID 1  - Iter 18
Thread ID 1  - Iter 19
```

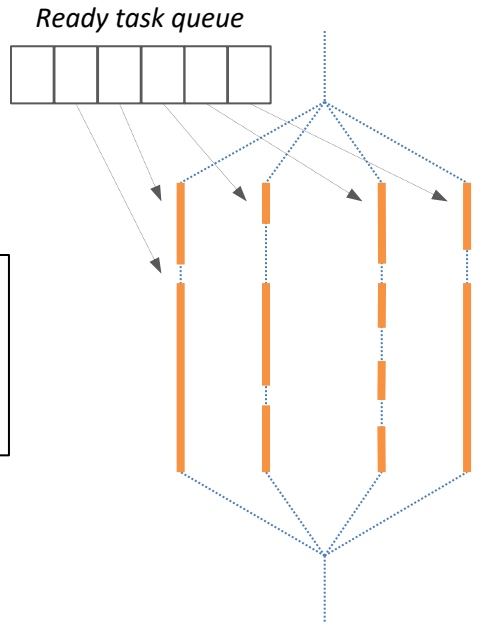
- ***How many messages? 20***
- ***In which order? UNDETERMINED***

Distributing parallelism with the task model

```
#pragma omp parallel num_threads(4)
#pragma omp master
{
    #pragma omp task
    { do_computation_1(); }
    …
    #pragma omp task
    { do_computation_N(); }
}
```

*A **task** (i.e., a task region and its data environment) is generated when a thread encounters a `task` construct*

*The **team of threads** executes a set of ready tasks **Scheduling** is implementation defined*



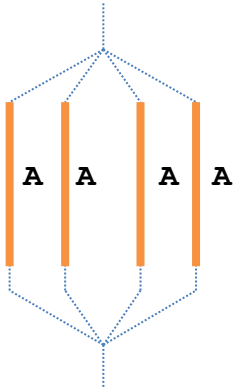

```
#pragma omp parallel num_threads(4)
#pragma omp master
{
    #pragma omp taskloop
    for (i=LB; i<UB; ++i)
        do_computation();
}
```

*A **taskloop** distributes iterations across tasks generated by the construct*

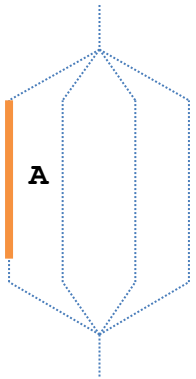
Distributing parallelism with the task model

- The OpenMP fork-join model is not suitable for the tasking model
  - The **`parallel`** construct replicates the encapsulated code to all threads
- The **`master`** and **`single`** constructs assigns the code within the parallel region to a single thread

```
#pragma omp parallel \
        num_threads(4)
{
    A;
}
```




```
#pragma omp parallel \
        num_threads(4)
#pragma omp master
{
    A;
}
```



Hands-on: `task`

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    #pragma omp parallel num_threads(4)
    {
        #pragma omp task
        printf("Thread ID %d\n", omp_get_thread_num());
    }
    return 0;
}
```

- ***How many messages?***
- ***In which order?***

Hands-on: `task`

```
Thread ID 0
Thread ID 3
Thread ID 1
Thread ID 2
```

```
Thread ID 3
Thread ID 0
Thread ID 2
Thread ID 1
```

```
Thread ID 0
Thread ID 2
Thread ID 1
Thread ID 3
```

- ***How many messages? 4***
- ***In which order? UNDETERMINED***

```
#pragma omp master
```

```
Thread ID 0
```

```
Thread ID 2
```

```
Thread ID 3
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[])
{
    #pragma omp parallel num_threads(4)
    {
        #pragma omp task
        printf("Thread ID %d\n", omp_get_thread_num());
    }
    return 0;
}
```

Hands-on: `task`

```
void saxpy(double *x, double *y)
{
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < N; i+=BS1) {
            for(int j = 0; j < BS1; j+=BS2) {
                for(int k = 0; k < BS2; ++k) {
                    y[i+j+k] += a * x[i+j+k];
                }
            }
        }
    }
}
```

- ***Which units can run concurrently?***
- ***Where does `task` can be inserted?***

Granularity of the parallel execution

- Amount of execution done by each task

Degree of parallelism

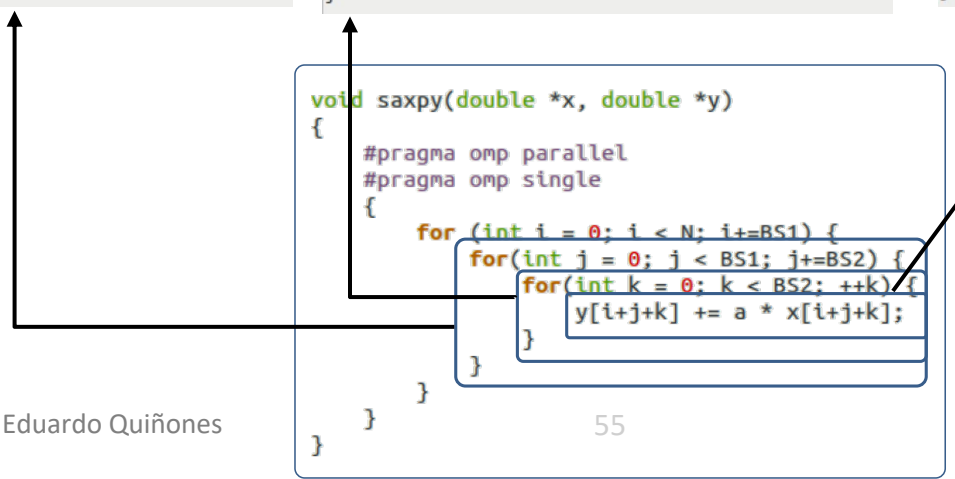
- How many tasks can be potentially executed simultaneously

Hands-on: `task`

```
void saxpy(double *x, double *y)
{
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < N; i+=BS1) {
            #pragma omp task
            for(int j = 0; j < BS1; j+=BS2) {
                for(int k = 0; k < BS2; ++k) {
                    y[i+j+k] += a * x[i+j+k];
                }
            }
        }
    }
}
```

```
void saxpy(double *x, double *y)
{
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < N; i+=BS1) {
            for(int j = 0; j < BS1; j+=BS2) {
                #pragma omp task
                for(int k = 0; k < BS2; ++k) {
                    y[i+j+k] += a * x[i+j+k];
                }
            }
        }
    }
}
```

```
void saxpy(double *x, double *y)
{
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < N; i+=BS1) {
            for(int j = 0; j < BS1; j+=BS2) {
                for(int k = 0; k < BS2; ++k) {
                    #pragma omp task
                    y[i+j+k] += a * x[i+j+k];
                }
            }
        }
    }
}
```



Hands-on: `task` and `taskloop`

```
void saxpy(double *x, double *y)
{
    #pragma omp parallel
    #pragma omp single
    {
        for (int i = 0; i < N; i+=BS1) {
            #pragma omp task
            for(int j = 0; j < BS1; j+=BS2) {
                for(int k = 0; k < BS2; ++k) {
                    y[i+j+k] += a * x[i+j+k];
                }
            }
        }
    }
}
```

```
void saxpy(double *x, double *y)
{
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskloop
        for (int i = 0; i < N; i+=BS1) {
            for(int j = 0; j < BS1; j+=BS2) {
                for(int k = 0; k < BS2; ++k) {
                    y[i+j+k] += a * x[i+j+k];
                }
            }
        }
    }
}
```

Equivalent if single iterations are distributed across threads: **`grainsize(strict:1)`**

Synchronization and data-sharings

- Mechanisms to define the order and the type of access to data
  - Prevents **data races**: two threads access the same **object** and at least one of them is a write
- **Synchronization** imposes an order of execution of parallel units
- **Data-sharings** define the scope at which a change in a variable is visible

Memory fences

Thread-centric model:
- `barrier`
- `nowait` (clause)

Task-centric model:
- `taskwait`
- `taskgroup`
- `depend` (clause)

Memory consistency

- `private` (clause)
- `firstprivate` (clause)
- `Lastprivate` (clause)
- `shared` (clause)

Mutual exclusion

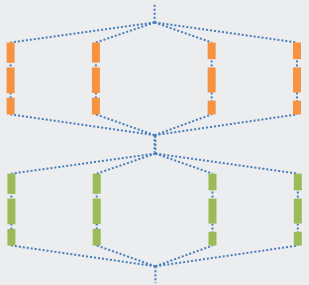
- `atomic`
- `critical`

Memory Fences for the Thread Model

`barrier`

- All threads of the team must execute the barrier and any pending work before proceeding

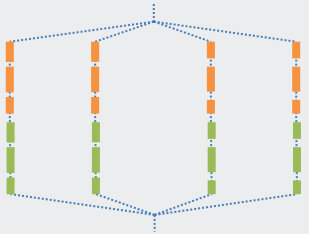
```
#pragma omp parallel
{
    #pragma omp for
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for
    for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
}
```



`nowait` (clause)

- Avoid unnecessary implicit synchronizations

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
    for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
}
```

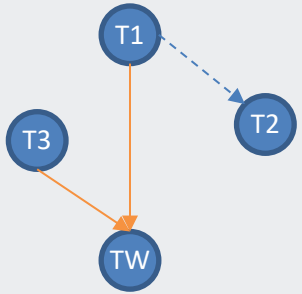


Memory Fences for the Task Model

`taskwait`

- The encountering task is suspended until all previous child tasks have executed

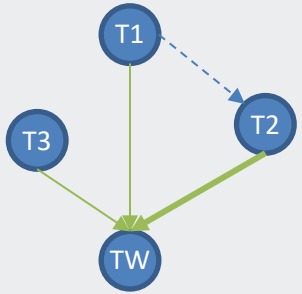
```
#pragma omp task // T1
{
    #pragma omp task // T2
    {…}
}
#pragma omp task // T3
{…}
#pragma omp taskwait
```



`taskgroup`

- Generates a new region where all inner tasks have to finish before the encountering thread can proceed

```
#pragma omp taskgroup
{
    #pragma omp task // T1
    {
        #pragma omp task // T2
        {…}
    }
    #pragma omp task // T3
    {…}
}
```



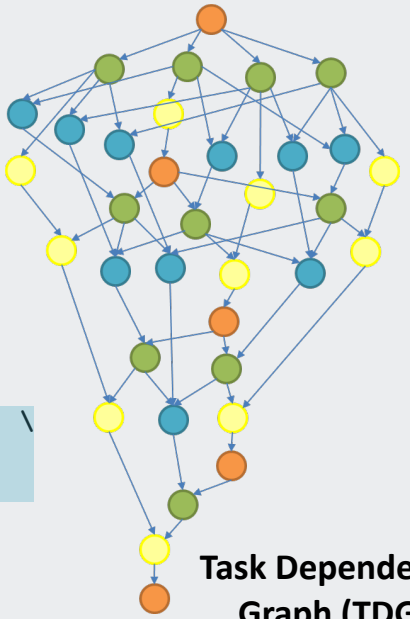
Memory Fences for the Task Model

depend (clause)

- Enforce ordering constraints on the scheduling of tasks
- Defines data-flow execution

Cholesky Factorization

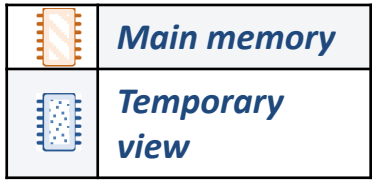
```
for (int k = 0; k < nt; k++) {
   #pragma omp task depend (inout: Ah[k][k])
   potrf (Ah[k][k]);

   for (int i = k + 1; i < nt; i++) {
      #pragma omp task depend (in: Ah[k][k]) \
                      depend (inout: Ah[k][i])
      trsm (Ah[k][k], Ah[k][i]);
   }
   for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++)
         #pragma omp task depend (in: Ah[k][i], Ah[k][j]) \
                         depend (inout: Ah[j][i])
         gemm (Ah[k][i], Ah[k][j], Ah[j][i]);
      #pragma omp task depend (in: Ah[k][i]) \
                      depend (inout: Ah[i][i])
      syrk (Ah[k][i], Ah[i][i]);
}}
```

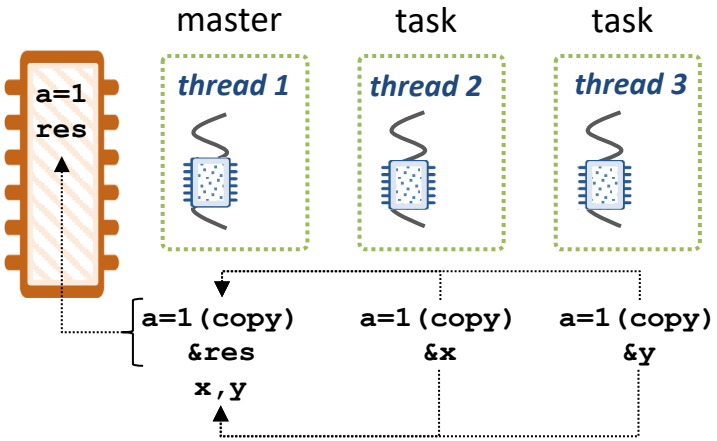


Task Dependency Graph (TDG)

Data-sharing attributes

- Defines the visibility of variable across parallel regions
  - **`shared`, `private`, `firstprivate`** and **`lastprivate`** clauses






```
int a = 1, res;
#pragma omp parallel shared(res) firstprivate(a)
#pragma omp master
{
    int x,y;
    #pragma omp task shared(x) firstprivate(a)
    x = a*a;
    #pragma omp task shared(y) firstprivate(a)
    y = a+a;
    #pragma omp taskwait
    res = x+y;
}
```

Hands-on: data-sharing + coarse grain synchronizations

```c
#include <stdio.h>

#define N 20

long long fib (int n)
{
    long long x, y;
    if (n < 2) return n;

    x = fib(n - 1);
    y = fib(n - 2);

    return x + y;
}

int main()
{
    int res = fib(N);

    printf("Fibonacci number %d is %d\n", N, res);
    return 0;
}
```

- ***Which portions can be concurrent?***
- ***Which synchronizations are needed?***
- ***Which are the data-sharing attributes?***

Hands-on: data-sharing + coarse grain synchronizations

```c
#include <stdio.h>

#define N 20

long long fib (int n)
{
    long long x, y;
    if (n < 2) return n;

    #pragma omp task shared(x) firstprivate(n)
    x = fib(n - 1);
    #pragma omp task shared(y) firstprivate(n)
    y = fib(n - 2);

    #pragma omp taskwait
    return x + y;
}

int main()
{
    int res = -1;
    #pragma omp parallel shared(res)
    #pragma omp single
    {
        res = fib(N);
    }

    printf("Fibonacci number %d is %d\n", N, res);
    return 0;
}
```

Data-sharings:

- **`x`** and **`y`** are shared variables
- **`n`** is not shared among tasks

Concurrent functions

Synchronization
(**`x`** and **`y`** are shared variables!)

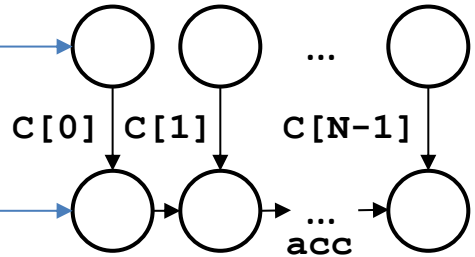
Hands-on: data-sharing + fine grain synchronizations

```
double dot_product (long N, long CHUNK_SIZE,
                    double A[N], double B[N])
{
    long N_CHUNKS;
    long actual_size;
    int j;
    double acc;

    N_CHUNKS = N/CHUNK_SIZE;
    N_CHUNKS = (N_CHUNKS*CHUNK_SIZE<N)?
           N_CHUNKS+1 : N_CHUNKS;
    double *C = malloc (N_CHUNKS*sizeof(double));

    acc=0.0;
    j=0;
    for (long i=0; i<N; i+=CHUNK_SIZE) {

        actual_size = (N-CHUNK_SIZE>=CHUNK_SIZE)?
                CHUNK_SIZE:(N-CHUNK_SIZE);

        C[j]=0;
        for (long ii=0; ii<actual_size; ii++)
            C[j]+= A[i+ii] * B[i+ii];

        acc += C[j];

        j++;
    }

    return(acc);
}
```

- ***Which portions can be concurrent?***
- ***Which synchronizations are needed?***
- ***Which are the data-sharing attributes?***

Hands-on: data-sharing + fine grain synchronizations

```
double dot_product (long N, long CHUNK_SIZE, double A[N], double B[N])
{
    long N_CHUNKS;
    long actual_size;
    int j;
    double acc;

    N_CHUNKS = N/CHUNK_SIZE;
    N_CHUNKS = (N_CHUNKS*CHUNK_SIZE<N)? N_CHUNKS+1 : N_CHUNKS;
    double *C = malloc (N_CHUNKS*sizeof(double));

    acc=0.0;
    j=0;
    for (long i=0; i<N; i+=CHUNK_SIZE) {

        actual_size = (N-CHUNK_SIZE>=CHUNK_SIZE)?CHUNK_SIZE:(N-CHUNK_SIZE);

        #pragma omp task firstprivate(j, i, actual_size) depend(out:C[j])
        {
            C[j]=0;
            for (long ii=0; ii<actual_size; ii++)
                C[j]+= A[i+ii] * B[i+ii];
        }

        #pragma omp task shared(acc) firstprivate(j) depend(inout:acc) depend(in:C[j])
        acc += C[j];

        j++;
    }

    #pragma omp taskwait

    return(acc);
}
```



Synchronization
(`acc` is a shared variable)

Mutual Exclusion

atomic

- Ensures a specific storage location is accessed atomically
- Only specific operations are allowed
- Decorators specify the type of access (e.g., update, read, write,…)

```
#pragma omp parallel for \
        shared(x, y, index, n)
for (i=0; i<n; i++) {
   #pragma omp atomic update
   x[index[i]] += work1(i);
   y[i] += work2(i);
}
```

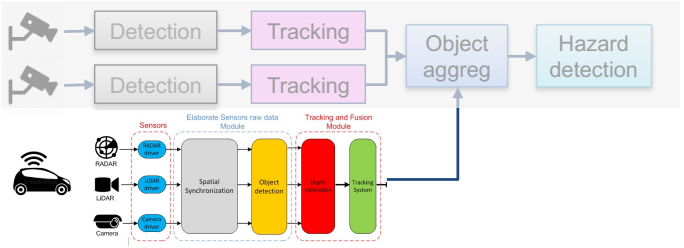
critical

- Restricts execution of a structured block to a single thread at a time
- Can be named
- Might perform worse than atomic but is more flexible

```
#pragma omp parallel shared(x, y) \
    private(ix_next, iy_next)
{
   #pragma omp critical(xaxis)
   ix_next = dequeue(x);
   work(ix_next, x);
   #pragma omp critical(yaxis)
   iy_next = dequeue(y);
   work(iy_next, y);
}
```

Summary

Spawn parallelism		- **parallel** - **num_threads** (clause) - **master/single**
Distribute parallelism	Thread-centric	- **for** - **section**
	Task-centric	- **task** - **taskloop**
Synchronize & Data-sharing	Memory fence	- **barrier** - **nowait** (clause) - **taskwait** - **taskgroup** - **depend** (clause)
	Memory consistency	- **private** - **firstprivate** - **lastprivate** - **shared**
	Mutual exclusion	- **atomic** - **critical**

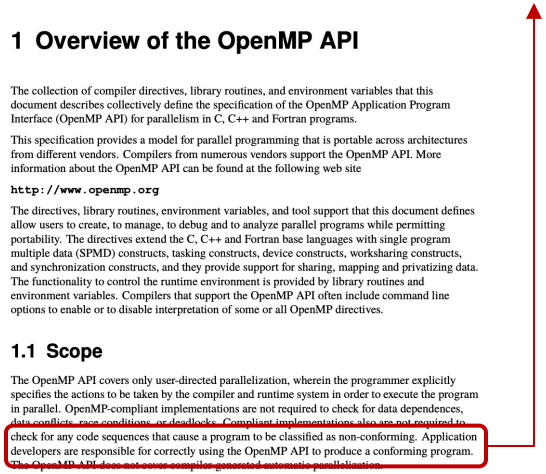


Let's **develop CPS with OpenMP** and so **cope with the the performance requirements** of the most advanced CPS functionalities!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Implementing CPS with OpenMP

- **Page 1 of the OpenMP specification** document says:
  - **Application developers are responsible** for correctly using the OpenMP API to produce a conforming program



1 Overview of the OpenMP API

The collection of compiler directives, library routines, and environment variables that this document describes collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for parallelism in C, C++ and Fortran programs.

This specification provides a model for parallel programming that is portable across architectures from different vendors. Compilers from numerous vendors support the OpenMP API. More information about the OpenMP API can be found at the following web site

`http://www.openmp.org`

The directives, library routines, environment variables, and tool support that this document defines allow users to create, to manage, to debug and to analyze parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing, mapping and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include command line options to enable or to disable interpretation of some or all OpenMP directives.

1.1 Scope

The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependences, data conflicts, race conditions, or deadlocks. Compliant implementations also are not required to check for any code sequences that cause a program to be classified as non-conforming. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. The OpenMP API does not cover compiler-generated automatic parallelization

CPS correctness cannot rely on magic!

Implementing CPS with OpenMP

- The complexity of parallel programming increases if guarantees on functional and non-functional correctness must be provided

1. **Functional correctness (safety)** ensure a correct system operation in response to its inputs guaranteeing system integrity
   - **Reliability**: The property that ensures the system correctness
   - **Resiliency**: The property that guarantees the system recovery if an unexcepted event impacts on system correctness, e.g., a soft transient error
2. **Non functional correctness**
   - **Time predictability:** Reasoning about the *timing behaviour* of the parallel execution to ensure the execution completes within a given *deadline*

Conclusions

- OpenMP provides a **great expressiveness** to describe parallelism, but…
  - … puts all **responsibility on functional correctness** on the software developer (*not always the best option, even for HPC…*)
  - … does not provide any support to guarantee **time predictability**

Next lesson will **analyse OpenMP from a functional correctness and time predictability** perspective to enable its applicability on the development of CPS

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based Parallel Programming Models: The Convergence of High-Performance and Cyber-Physical Computing Domains

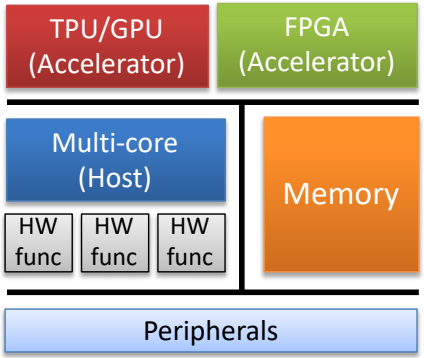
Lesson 3: OpenMP and CPS

Eduardo Quiñones
{eduardo.quinones@bsc.es}

ACACES 2021, Fiuggi

Outline

- Reliability and resiliency on parallel execution
- Task Dependency Graph (TDG)
- Time predictability
  - OpenMP task scheduler
  - Schedulability analysis
- OpenMP Tracing
  - Extrae and Paraver
- Model Driven Engineering
  - Amalthea and OpenMP
- Conclusions



Reliability: Parallel Execution and Correctness

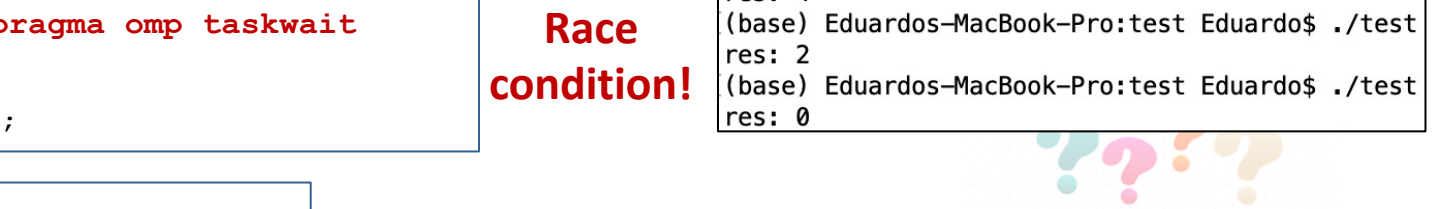
1. **Data races**
   - Occur when two threads access the same shared object and at least one of them is a write
   - Data races result in undefined behavior
2. **Wrong data sharing definition**
   - Occur when the visibility of the variables is not properly setup, resulting in an incorrect execution
3. **Deadlocks**
   - Occur when the program is waiting for an event that cannot happen
     - Two threads are waiting in the same critical region
   - Deadlock blocks the execution of the program forever

Hands-on: Parallel Correctness

```
int a=2, b=2, res=0;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) firstprivate(a)
    x = a;
    #pragma omp task shared(y) firstprivate(b)
    y = b;
    #pragma omp taskwait
    res = x+y;
}
printf("res: %d\n",res);
```

Race condition!

```
(base) Eduardos-MacBook-Pro:test Eduardo$ ./test
res: 4
(base) Eduardos-MacBook-Pro:test Eduardo$ ./test
res: 2
(base) Eduardos-MacBook-Pro:test Eduardo$ ./test
res: 0
```



```
int a=2, b=2, res=0;
#pragma omp parallel shared(res,a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) shared(a)
    x = a;
    #pragma omp task shared(y) shared(b)
    y = b;
    #pragma omp taskwait
    res = x+y;
}
printf("res: %d\n",res);
```

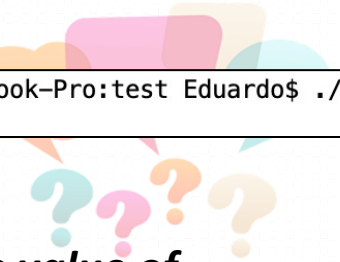
- ***Which is the value of `res` printed?***

```
(base) Eduardos-MacBook-Pro:test Eduardo$ ./test
res: 4
```

Hands-on: Parallel Correctness

```
int a = 2, b=2, res=0;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) firstprivate(a)
    x = a;
    #pragma omp taskwait
    #pragma omp task shared(y) firstprivate(b)
    y = b;
    #pragma omp taskwait
    res = x+y;
}
printf("res: %d\n",res);
```

```
(base) Eduardos-MacBook-Pro:test Eduardo$ ./test
res: 4
```



- ***Which is the value of `res` printed?***

```
int a = 2, b=2, res=0;
#pragma omp parallel firstprivate(res) firstprivate(a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) firstprivate(a)
    x = a;
    #pragma omp task shared(y) firstprivate(b)
    y = b;
    #pragma omp taskwait
    res = x+y;
}
printf("res: %d\n",res);
```

shared(res)

Race condition!

```
(base) Eduardos-MacBook-Pro:test Eduardo$ ./test
res: 0
```

Hands-on: Parallel Correctness

```
int a = 2, b=2, res;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x, y, factor=0;
    #pragma omp task shared(x,factor) firstprivate(a)
    {
        x = a;
        if(cond(x)) factor++;
    }
    #pragma omp task shared(y) firstprivate(b)
    y = b;

    factor++;
    #pragma omp taskwait
    res = (x+y)*factor;
}
printf("res: %d\n",res);
```

- ***Is this code functionally correct?***

Race condition!

Hands-on: Parallel Correctness

```
int a = 2, b=1, res;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x, y,factor=0;
    #pragma omp task shared(x,factor) firstprivate(a)
    {
        x = a*a;
        if(cond(x))
            #pragma omp critical(factor_update)
            factor++;
    }
    #pragma omp task shared(y) firstprivate(b)
    y = b*b;

    #pragma omp critical(factor_update)
    {
        factor++;
        #pragma omp taskwait
        res = (x+y)*factor;
    }
}
printf("res: %d\n",res);
```

- ***What is wrong with this code?***

Deadlock!

Hands-on: Parallel Correctness

```
int a = 2, b=1, res;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x, y,factor=0;
    #pragma omp task shared(x,factor) firstprivate(a)
    {
        x = a*a;
        if(cond(x))
            #pragma omp critical(factor_update)
            factor++;
    }
    #pragma omp task shared(y) firstprivate(b)
    y = b*b;

    #pragma omp critical(factor_update)
    factor++;

    #pragma omp taskwait
    res = (x+y)*factor;
}
printf("res: %d\n",res);
```

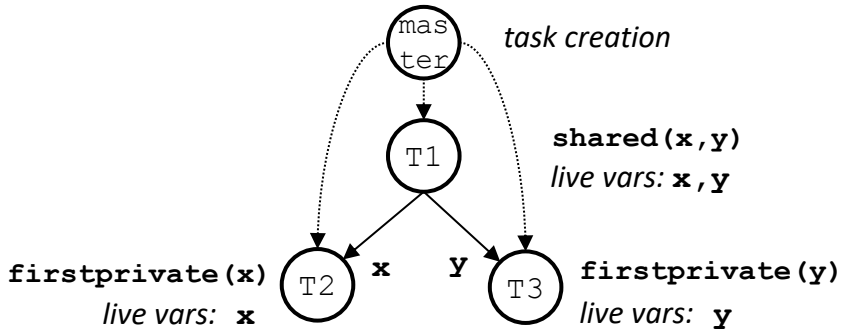
- The usage of critical mutex is not recommended!
- If needed, better use **`atomic`**

Task Dependency Graph (TDG)

A representation of the parallel nature of a given OpenMP region, extracted by means of compilation and runtime methods [1]

- Includes all the information for functional and non-funcional correctness
  - **Parallel units** and **synchronization** dependencies
  - **Liveness analysis of variables** and **data-sharings** involved in the parallel execution
- Independent from the targeted parallel platform (but can include HW dependent information)
  - **Execution characterisation** of parallel units (e.g., time, energy, memory behaviour)

```
#pragma omp parallel
#pragma omp master
{
    int x,y;
    #pragma omp task depend(out:x,y) shared(x,y) // T1
    { f1(&x,&y); }
    #pragma omp task depend(in:x) firstprivate(x) // T2
    { f2(x); }
    #pragma omp task depend(in:y) firstprivate(y) // T3
    { f3(y); }
}
```

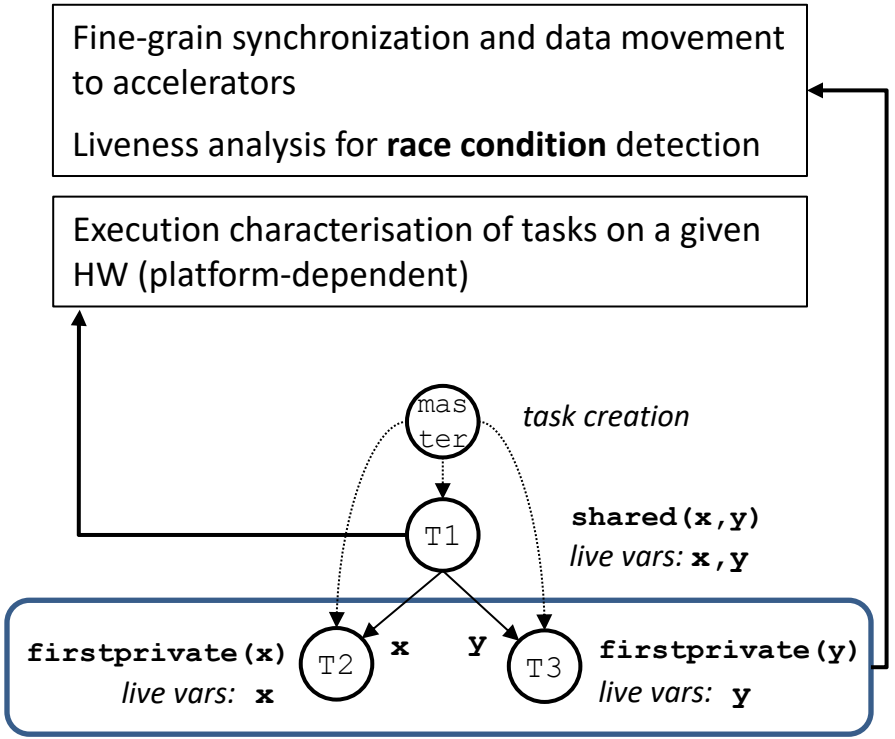


[1] Vargas, et.al. *A Lightweight OpenMP Run-time for Embedded Systems*, in ASP-DAC 2016;
Vargas, et.al., *OpenMP and Timing Predictability: A Possible Union?*, in DATE 2015

Task Dependency Graph (TDG)

A representation of the parallel nature of a given OpenMP region, extracted by means of compilation and runtime methods [1]

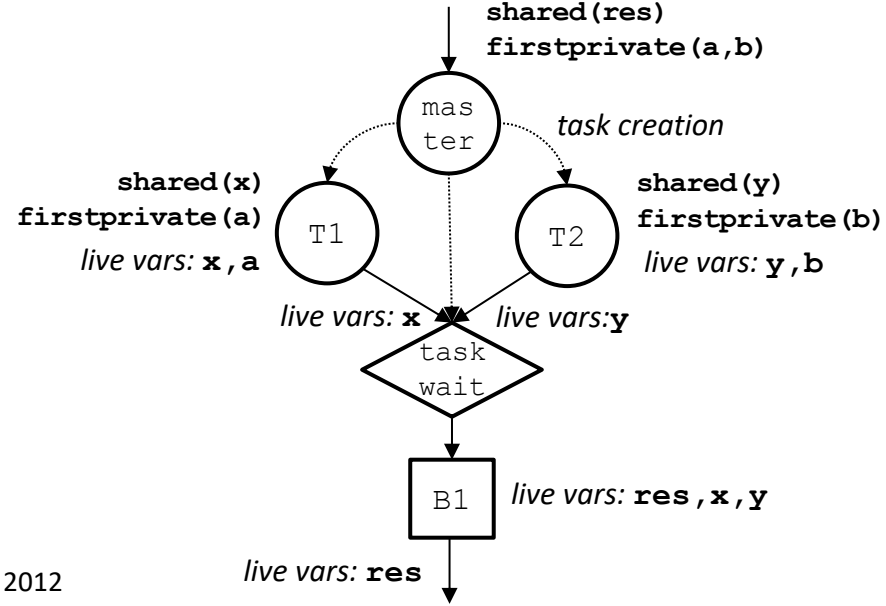
- Includes all the information for functional and non-funcional correctness
  - **Parallel units** and **synchronization** dependencies
  - **Liveness analysis of variables** and **data-sharings** involved in the parallel execution
- Independent from the targeted parallel platform (but can include HW dependent information)
  - **Execution characterisation** of parallel units (e.g., time, energy, memory behaviour)



[1] Vargas, et.al. *A Lightweight OpenMP Run-time for Embedded Systems*, in ASP-DAC 2016;
Vargas, et.al., *OpenMP and Timing Predictability: A Possible Union?*, in DATE 2015

TDG and Functional Correctness[1]

```
int a = 2, b=2, res=0;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) firstprivate(a) // T1
    x = a;
    #pragma omp task shared(y) firstprivate(b) // T2
    y = b;
    #pragma omp taskwait
    res = x+y; // B1
}
printf("res: %d\n",res);
```

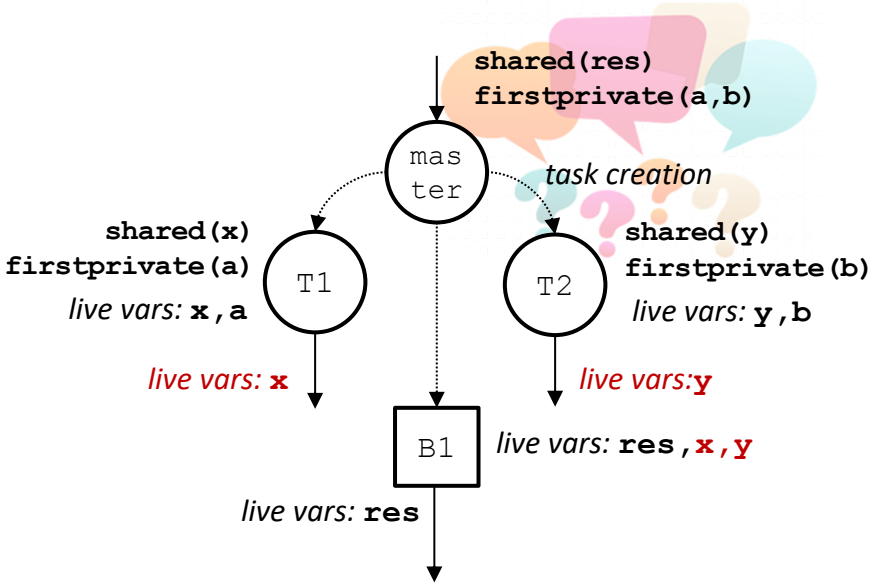


[1] Royuela, Duran, Liao, Quinlan, ***Auto-scoping for OpenMP tasks***, in IWOMP 2012

[2] Lin, ***Static nonconcurrency analysis of openmp programs***, in IWOMP 2008

TDG and Functional Correctness

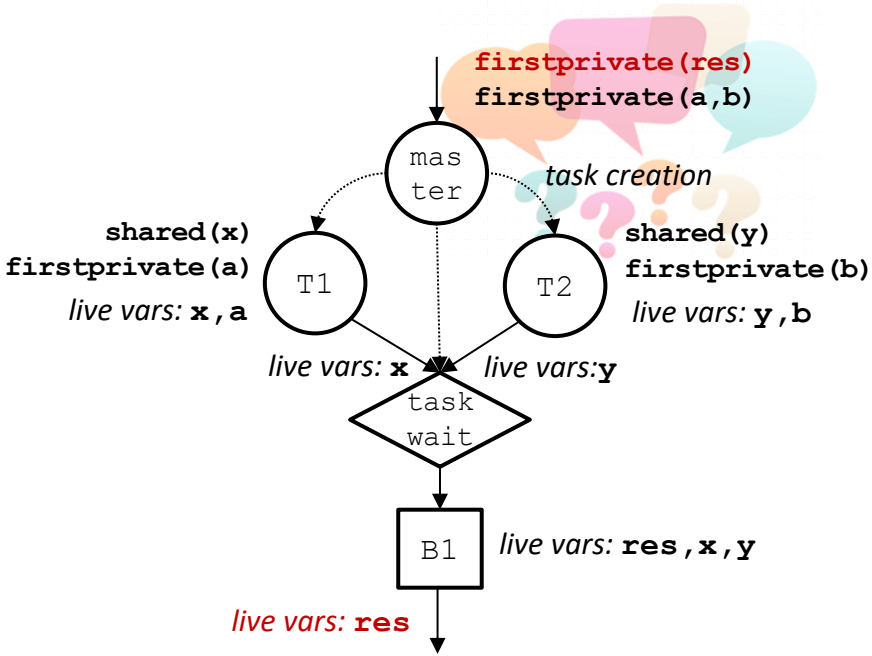
```
int a = 2, b=2, res=0;
#pragma omp parallel shared(res) firstprivate(a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) firstprivate(a) // T1
    x = a;
    #pragma omp task shared(y) firstprivate(b) // T2
    y = b;
    #pragma omp taskwait
    res = x+y; // B1
}
printf("res: %d\n",res);
```



x and **y** are write/read without a predefined order

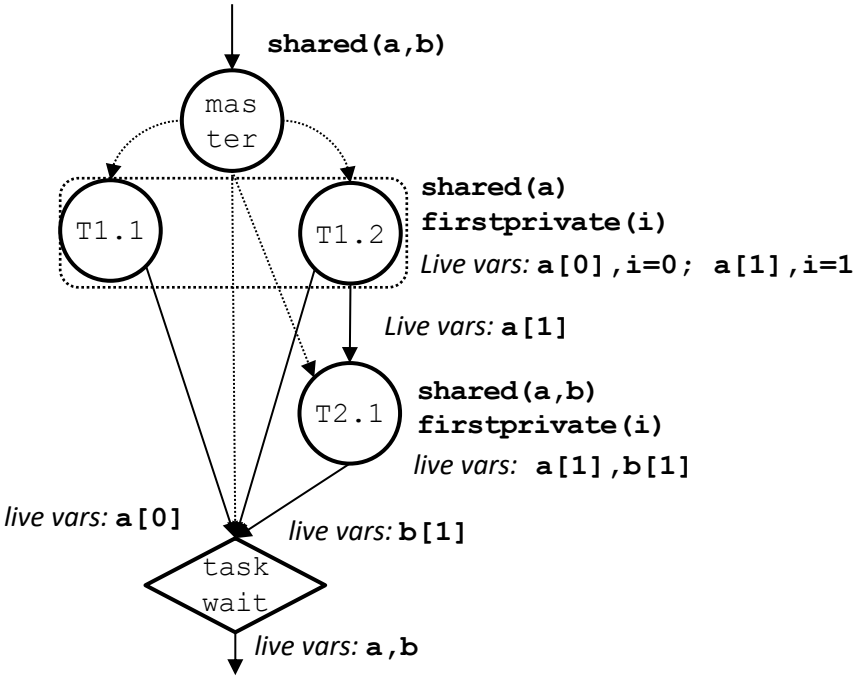
TDG and Functional Correctness

```
int a = 2, b=2, res=0;
#pragma omp parallel firstprivate(res) firstprivate(a,b)
#pragma omp master
{
    int x=0,y=0;
    #pragma omp task shared(x) firstprivate(a) // T1
    x = a;
    #pragma omp task shared(y) firstprivate(b) // T2
    y = b;
    #pragma omp taskwait
    res = x+y; // B1
}
printf("res: %d\n",res);
```



The *liveness analysis* and the *data-sharing* of **`res`** does not match!

Hands-on: TDG

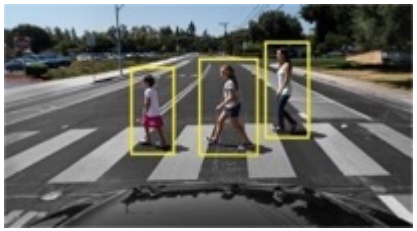


- ***How does the TDG look like?***

```
#define N 2

int a[N], b[N];
#pragma omp parallel shared(a,b)
#pragma omp master
{
    for(int i=0; i<N; i++) {
        // T1
        #pragma omp task shared(a) firstprivate(i) \
                         depend(out:a[i])
        a[i] = init(i);
        if(!(i%2))
            // T2
            #pragma omp task shared(a,b) firstprivate(i) \
                             depend(in:a[i], out:b[i])
            b[i] = compute(a[i]);    {
    }
    #pragma omp taskwait
}
```

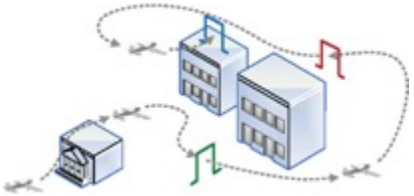
Examples of OpenMP-TDGs

Pedestrian detector (automotive)



Infra-red sensor pre-processing (space)

3D Path Planning (avionics)

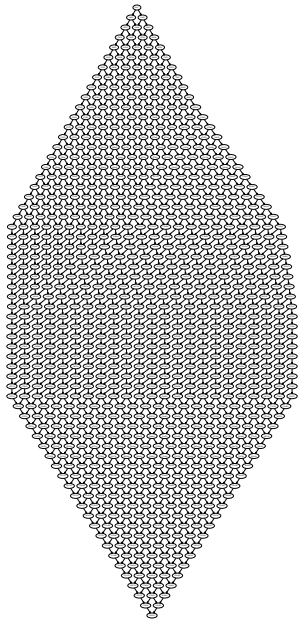


Cholesky Factorization (HPC)

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix}$$
$$= \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix},$$

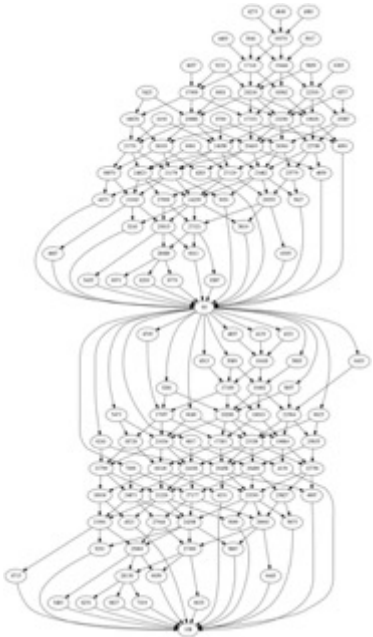
Examples of OpenMP-TDGs

Pedestrian detector (automotive)

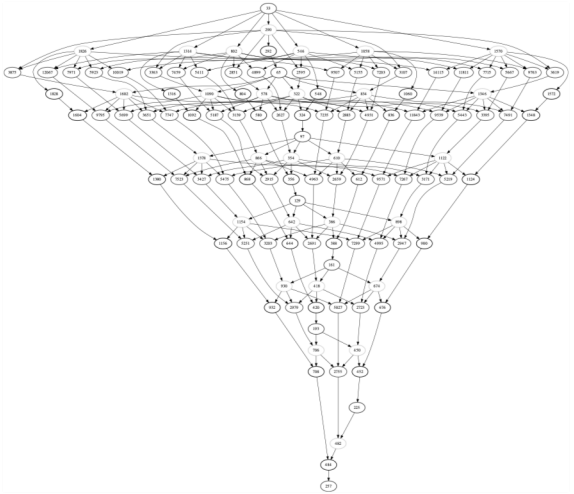


Infra-red sensor pre-processing (space)

3D Path Planning (avionics)



Cholesky Factorization (HPC)



Resiliency

- The OpenMP specification does not include error handling mechanisms to safely recover from errors
  - Relies on those provided by the base programming language, e.g., exceptions in case of C++
- OpenMP includes directives (**`cancel`** and **`cancellation point`**) to cancel the parallel execution of **`parallel`**, **`sections`**, **`for`** and **`taskgroup`**

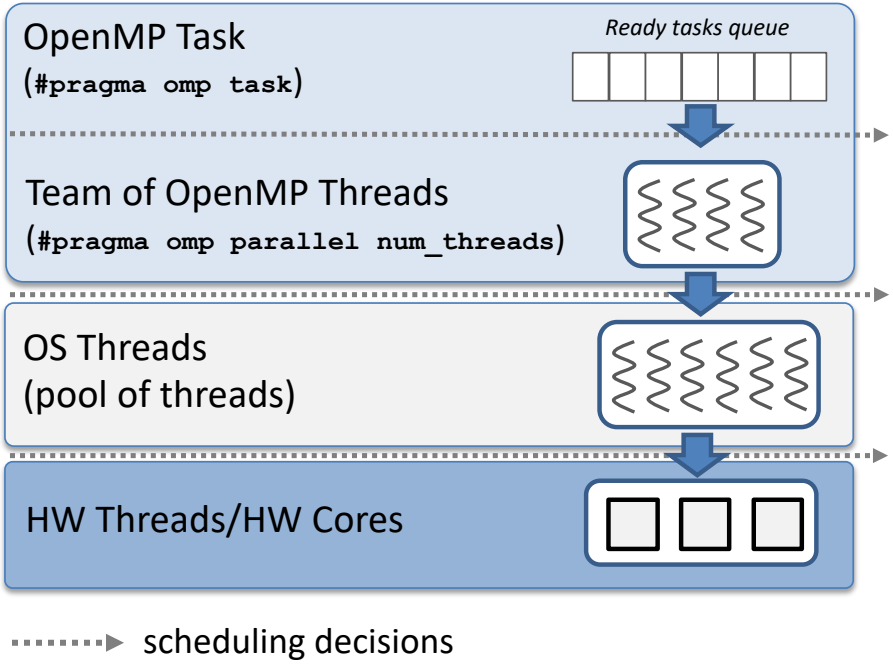
```
std::exception *ex = NULL;
#pragma omp parallel shared(ex)
{
    #pragma omp for
    for (int i=0;i<N;i++){
        try {
            iteration();
        }
        catch (std::exception *e) {
            #pragma omp atomic write
            ex = e;
            #pragma omp cancel for
        }
    }
    if (ex)
        #pragma omp cancel parallel
}
if (ex)
    handle_exception();
```

Time predictability

- The timing behaviour of parallel execution depends on the **allocation of parallel units** to **computing resources**
  1. The parallel structure of the application
     - The Task Dependency Graph (TDG)
  2. The scheduler(s) responsible of allocating parallel units (OpenMP tasks) to computing resources (cores/acceleration devices)
     - The execution profile of the parallel units into the computing resources

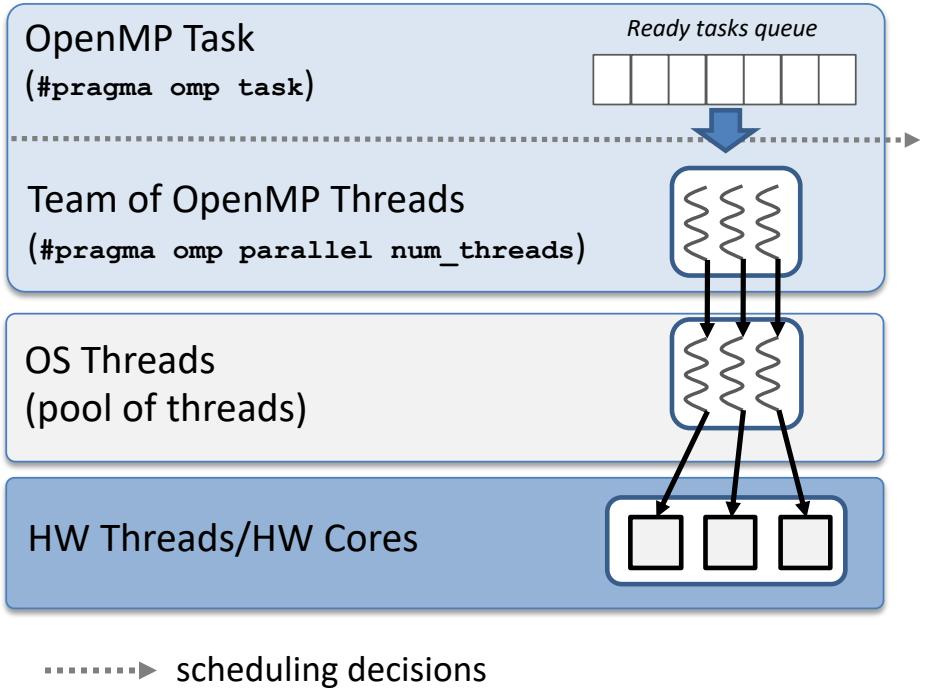
Time predictability: Task Scheduler

- The OpenMP framework includes multiple levels of scheduling that **dificults the time predictability**



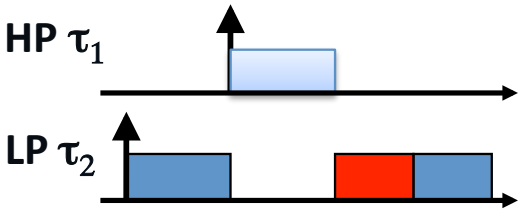
Time predictability: Task Scheduler

- The **OpenMP Thread Affinity** allows fixing the OpenMP threads of a team to the available HW threads on a device (places)
  - `OMP_PLACES`
  - `OMP_PROC_BIND`
- The parallel execution is **only managed** by the OpenMP task to OpenMP thread scheduler increasing time predictability

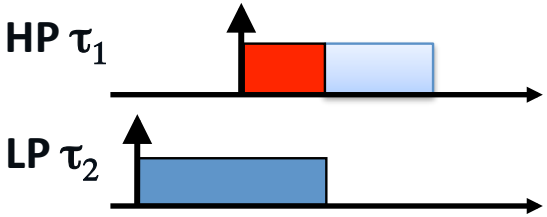


The OpenMP Scheduler

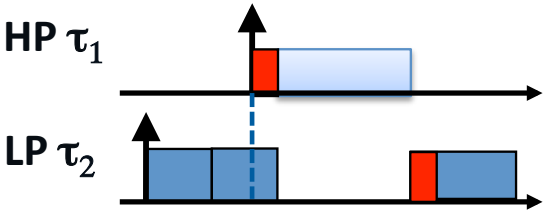
- Given two tasks with different priorities, there exist three preemption strategies



Fully preemptive scheduling



Non-preemptive scheduling



Limited preemption (cooperative) scheduling

OpenMP Enabling HPC since 1997

Barcelona Supercomputing Center Centro Nacional de Supercomputación

The OpenMP Scheduler

- The OpenMP tasking execution model defines a **limited preemption scheduling strategy**
  - OpenMP task-based program can only be preempted at predefined points of the execution (a.k.a. preemption points or *task scheduling points*)
    - *Task creation and completion,* **`taskwait, taskgroup`**
    - Tasks cannot be preempted at any other point and must execute until completion
  - Tasks includes a **`priority`** clause that can be used by the scheduler
- The actual implementation of the scheduler included in the runtime is *implementation-defined*

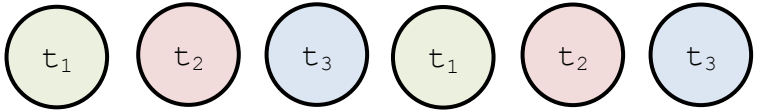
The OpenMP Task Scheduler

```
#pragma omp parallel num_threads(1)
#pragma omp master
{
    for(int i=0;i<2;i++) {
        #pragma omp task priority(3) // t1
        { … }
        #pragma omp task priority(2) // t2
        { … }
        #pragma omp task priority(1) // t3
        { … }
    }
}
```

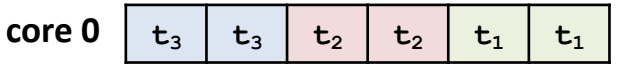
- priority

+ priority

TDG (order of creation):

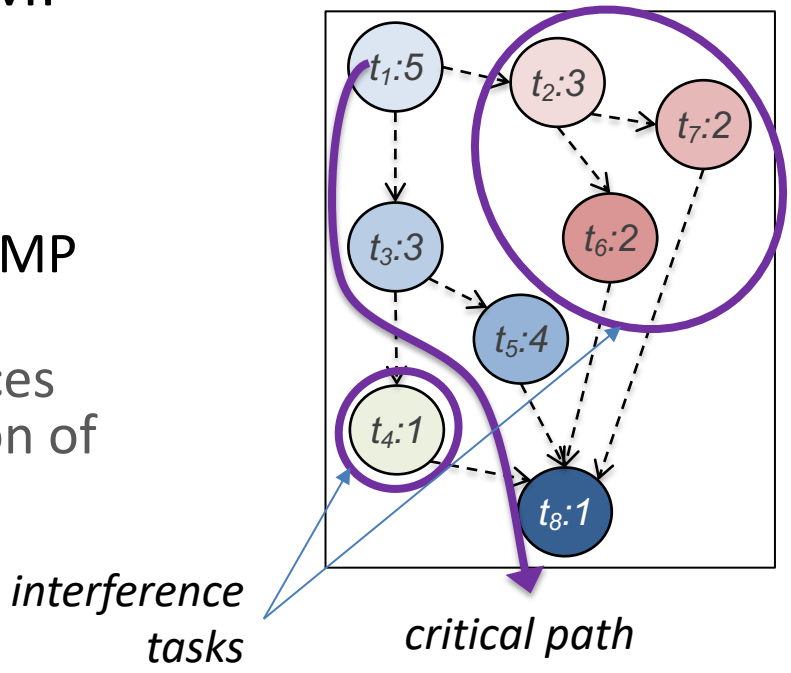


(a possible) order of execution:

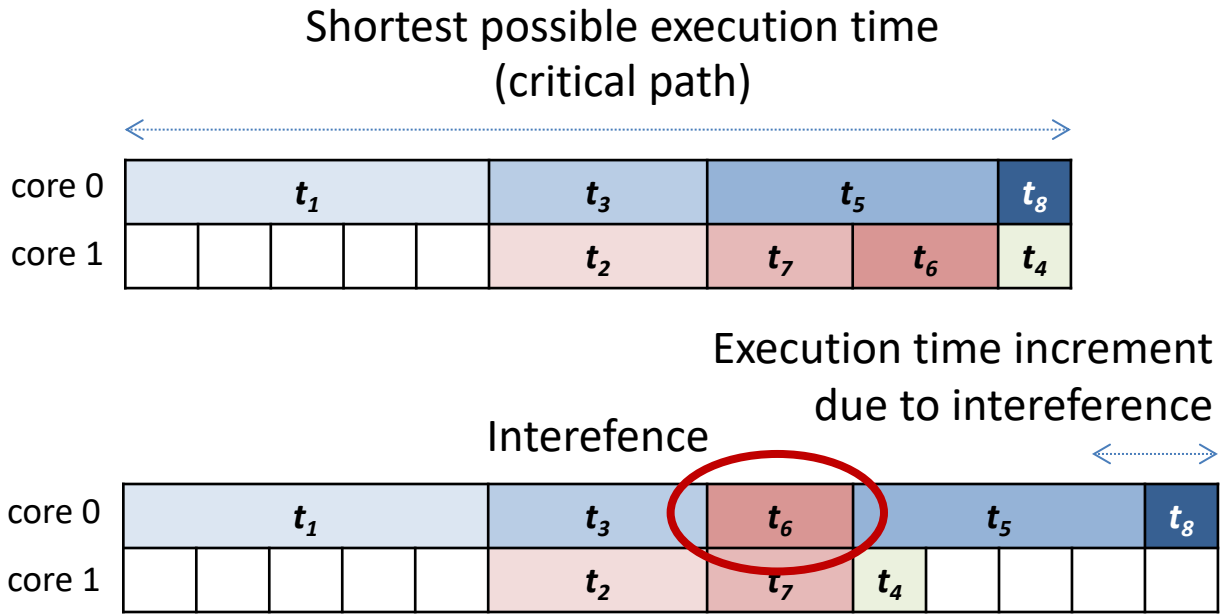


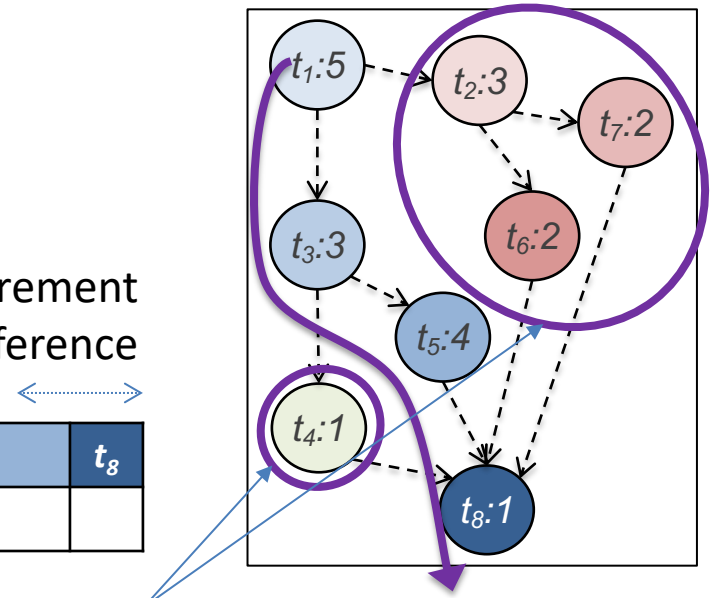
Time Predictability: TDG + Scheduler

- The execution time of an OpenMP-TDG is determined by:
  1. The execution of OpenMP tasks within the **critical path**
  2. I**nterferences** of the rest of OpenMP tasks on the critical path
  3. **Interferences** on HW/SW resources due to the simultaneous execution of OpenMP tasks
     - **Not addressed in this course!**



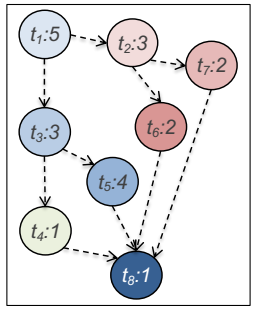
Time Predictability: TDG + Scheduler





Schedulability Analysis

- Determines a **response time upper bound (R^{ub})** of an OpenMP-TDG under a *work-conserving scheduler*[1]
  - An OpenMP application is schedulable if $R^{ub} \leq$ Deadline (D)



Critical path

$$R^{ub} = len(G) + \frac{1}{m}(vol(G) - len(G)) \leq D$$

Divided among processing units (cores)

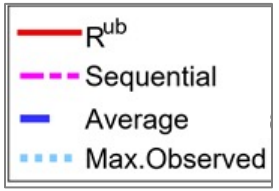
Interferences of the remaining work

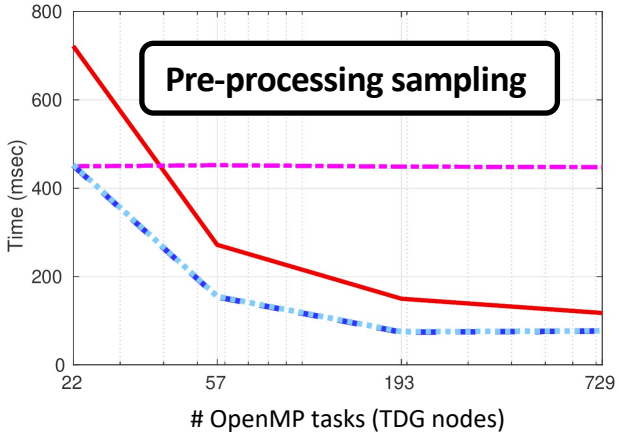
- ***G:*** TDG annotated with execution times of tasks
- ***len(G):*** critical path
- ***vol(G):*** sequential execution time
- ***D:*** deadline

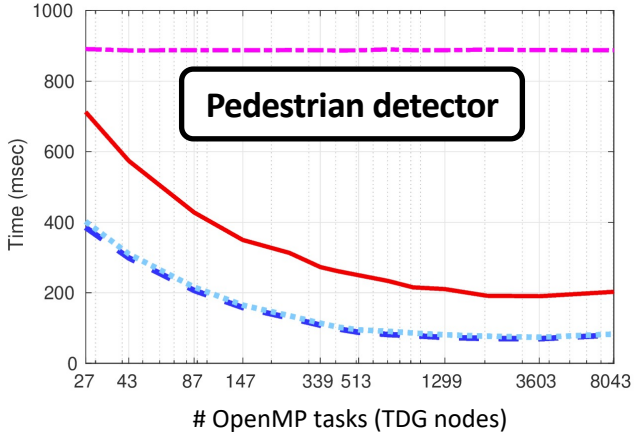
[1] *A. Melani, et.al.,* **A static scheduling approach to enable safety-critical OpenMP applications,** In ASP-DAC 2017

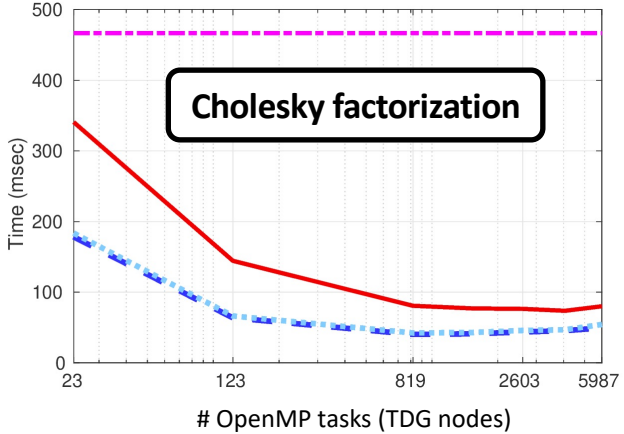
R^{ub} of Real OpenMP applications

- Executed using 16-cores only of the Intel® Xeon Platinum
  1. Improved performance parallel vs. sequential
  2. Average vs. max. observed execution times
  3. Maximum observed time over R^{ub}



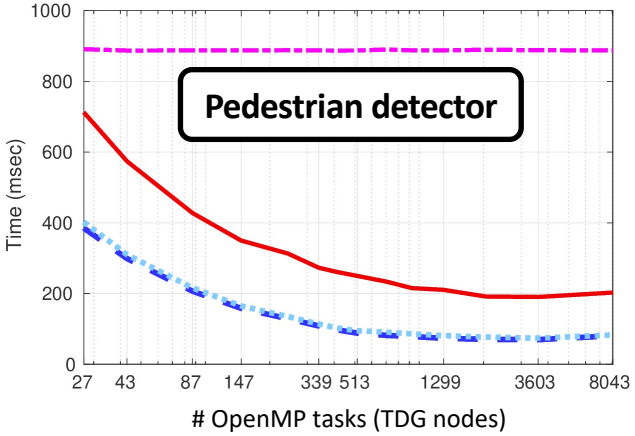


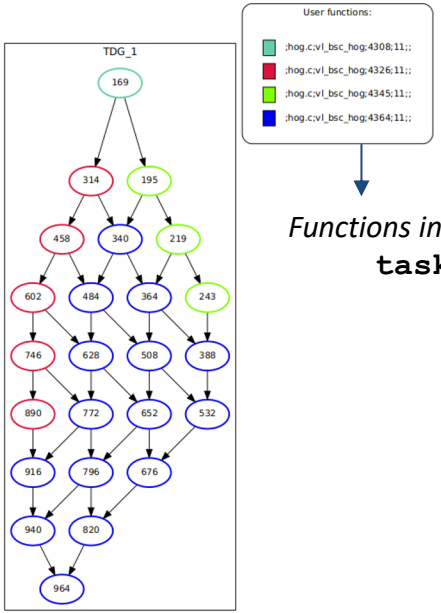




Understanding Parallel Execution

- Is schedulability analysis sufficient to understand parallel execution? **NO!**
  - No information about the **parallel execution efficiency** from a programming perspective
  - No information about the **usage** of computing resources



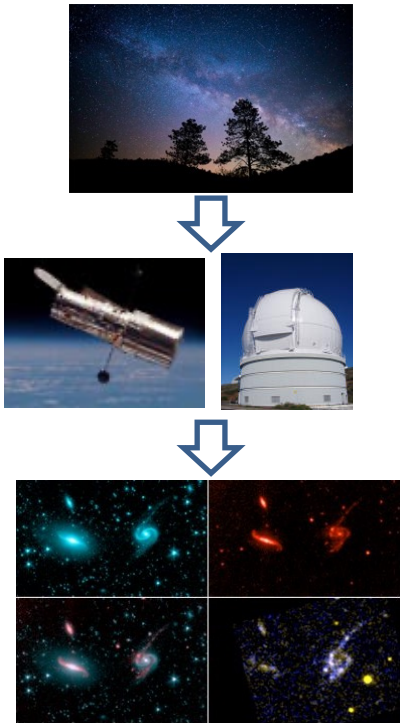


Functions included within the **`task`** *construct*

Understanding Parallel Execution

Based on how data is collected	Based on how data is stored
• **Instrumentation**: - Captures information based on events **(TDG events!)** - Requires modification of the application manual or **automatic** - Reports exact data	• **Tracing:** - Stores information in a **timeline basis** - Holds exact data - A profile can be derived from the trace
• **Sampling**: - Captures information periodically - Does not require modifying the application - Reports relative data	• **Profiling:** - Stores information in counters - Holds summarized data - A trace cannot be derived from a profile

A Multispectal Imaging of the Parallel Execution

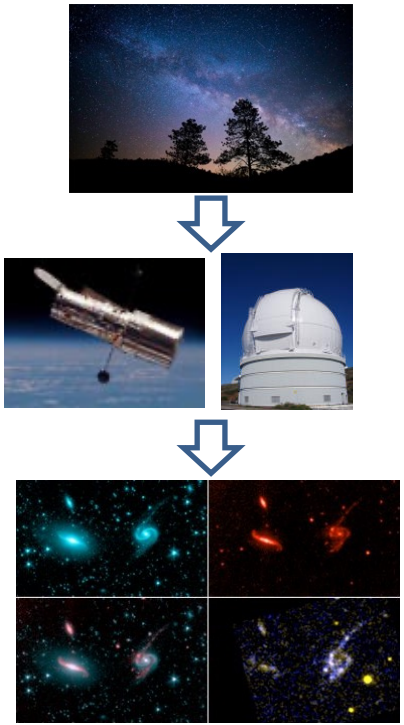


Reality

Tools for observing the reality

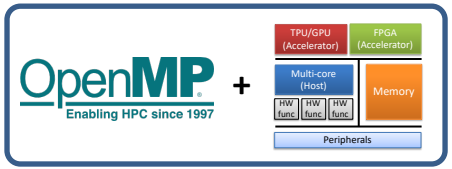
Different representations of the same reality containing **different information**

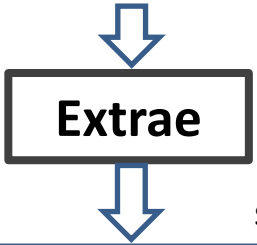
A Multispectal Imaging of the Parallel Execution



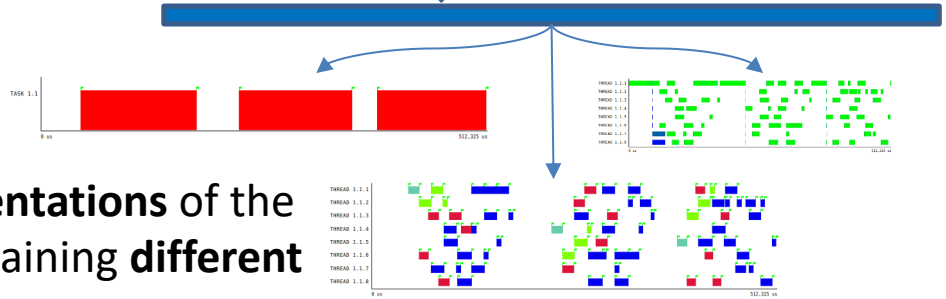
Reality

Tools for observing the reality





Sequence of time-stamped events (trace)



Different representations of the same reality containing **different information**

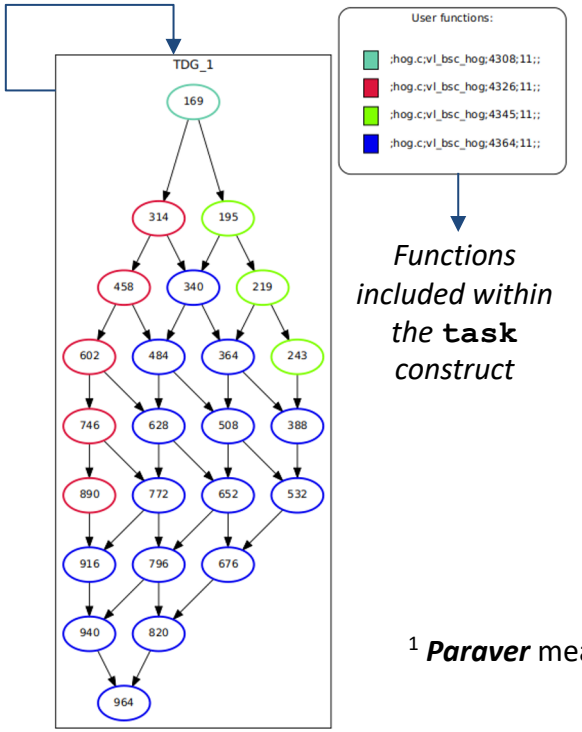
Understanding OpenMP: **Extrae**[1]

- A dynamic **instrumentation** package to **trace** parallel programs
- Capable of automatically capturing the activity of the parallel runtimes
  - No need to access the source code, recompiling, relinking, or having prior knowledge of application internals structure
- Allows reasoning about the execution behaviour of the parallel programing model
  - OpenMP support (other supported programming models are MPI, pthread, etc.)

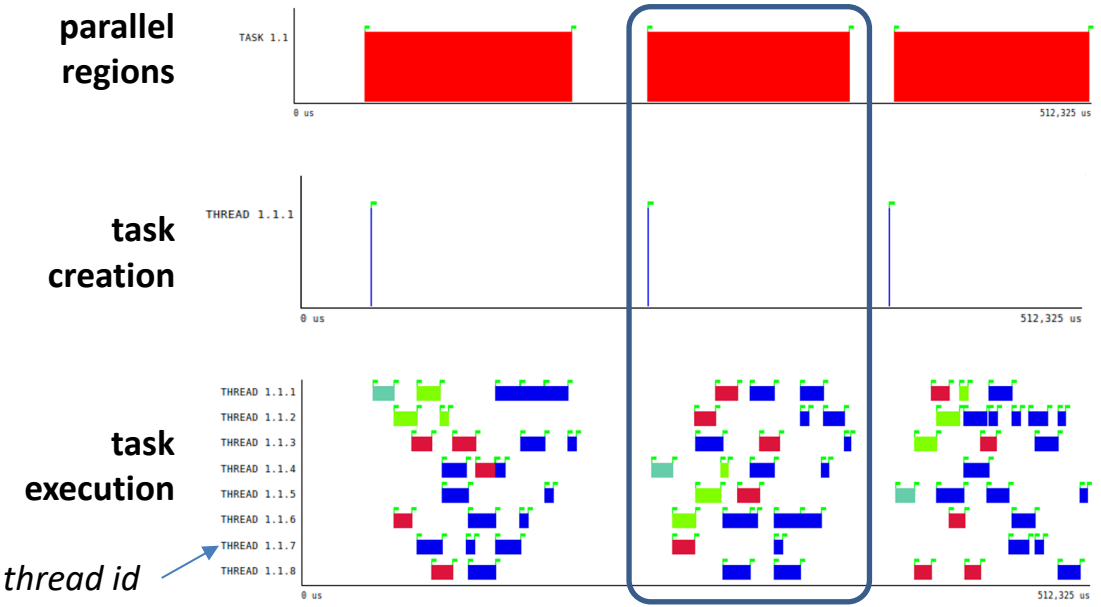
[1] ***Extrae*** means ***Extract*** in spanish. Available here: https://tools.bsc.es/extrae

A Multispectal Imaging of the Parallel Execution: Extrae + Paraver[1]

```
for (int i=0; i<3; i++)
```



Functions included within the **`task`** *construct*



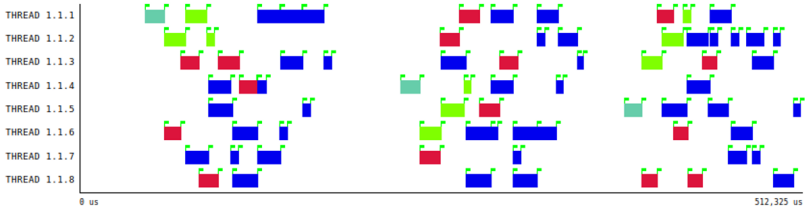
[1] ***Paraver*** means ***for seeing*** in spanish. Available here: https://tools.bsc.es/paraver

A Multispectal Imaging of the Parallel Execution: **Extrae + Paraver**

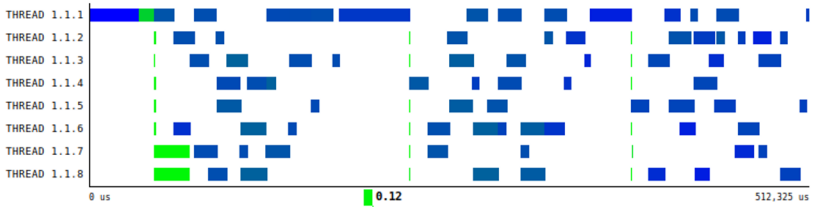
Parallel functions view
(Parallel programming level)

Hardware counters information view
(computing resources usage level)

Task execution

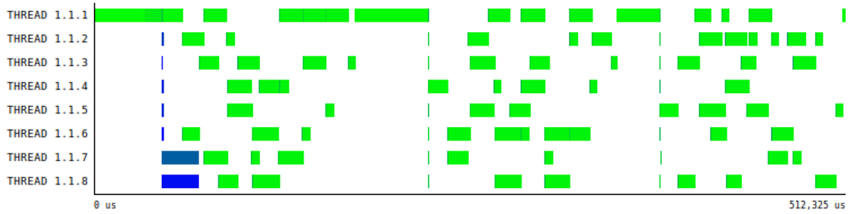


IPC (Instructions per Cycle)



Range:
0.1 to **2.0**

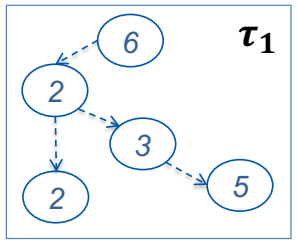
L1 miss ratio

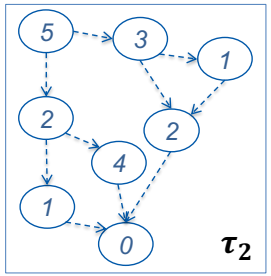


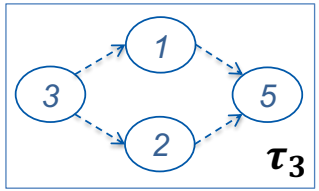
Range:
1% to **15%**

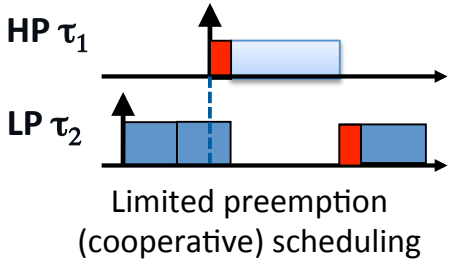
Supporting multiple CPS functionalities in OpenMP

- CPS are composed of multiple functionalities (a.k.a. real-time tasks) $\boldsymbol{\tau_k}$ (TDG $\boldsymbol{G_k}$), each characterized by a *period* ***(T)***, a *deadline* ***(D)*** and a *priority*



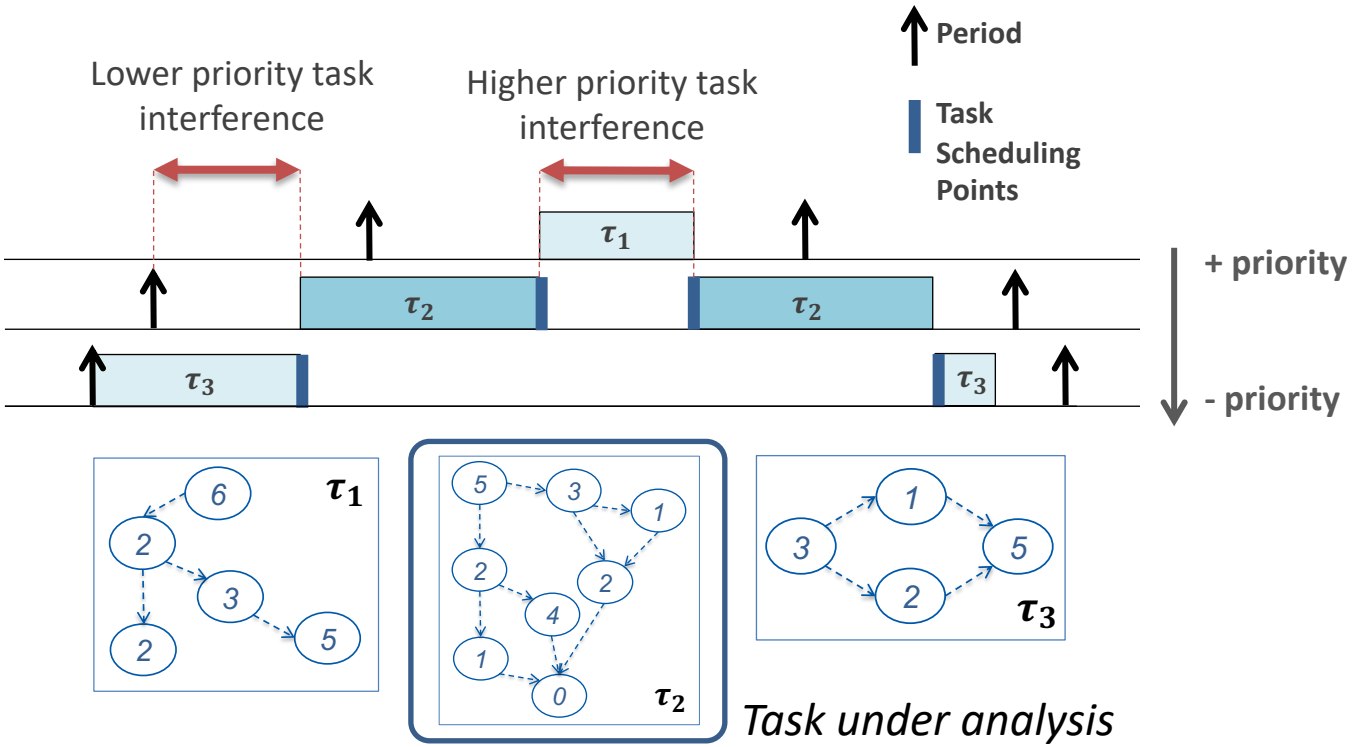




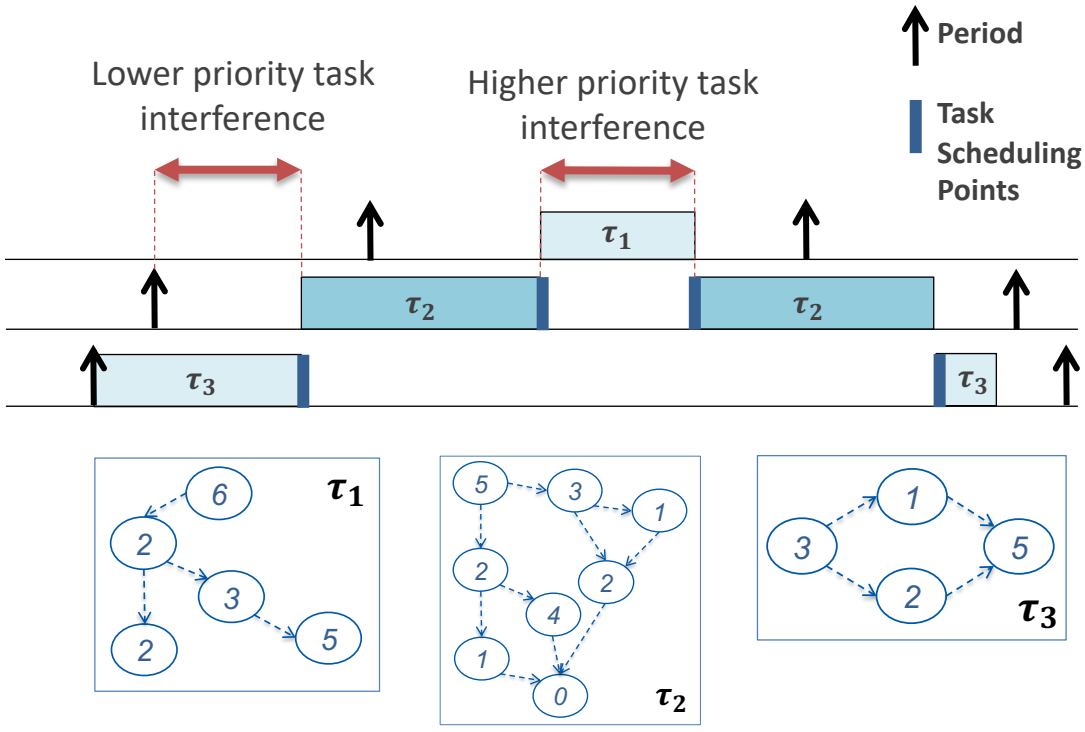


- The *limited preemption strategy* and the **`priority`** clause supported by OpenMP allows to analyse CPS with multiple functionalities implemented with OpenMP

Supporting multiple OpenMP functionalities in CPS

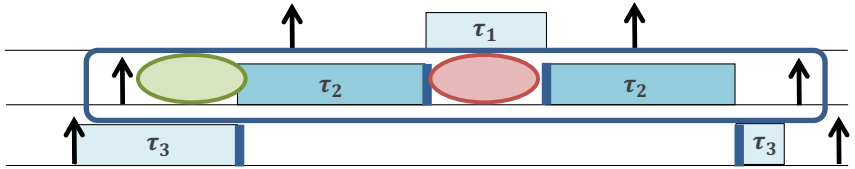


Supporting multiple OpenMP functionalities in CPS




```
#pragma omp parallel
#pragma omp single nowait
{
 while (1) {
  if (get_current_clock_t() == period_1)
    #pragma omp task priority(1)
    { τ1;
      #pragma omp taskwait
    }
  if (get_current_clock_t() == period_2)
    #pragma omp task priority(2)
    { τ2;
      #pragma omp taskwait
    }
  if (get_current_clock_t() == period_3)
    #pragma omp task priority(3)
    { τ3;
      #pragma omp taskwait
    }
  }
}
```

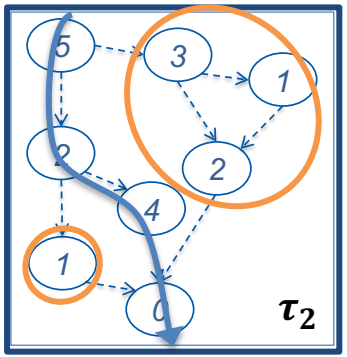
Schedulability Analysis



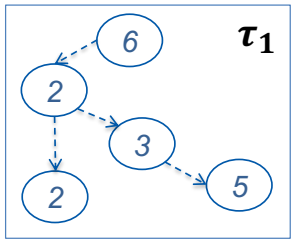
Critical path

Intra-task interference (self-interferences)

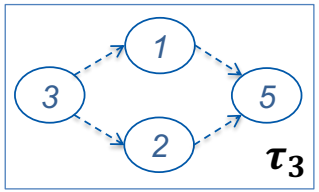
$$R_2^{ub} = len(G_2) + \frac{1}{m}\left(vol(G_2) - len(G_2)\right) + \frac{1}{m}\left(I_2^{hp} + I_2^{lp}\right) \leq D_2 \quad (1)$$



Higher priority tasks interference



Lower priority tasks interference




```
#pragma omp parallel
#pragma omp single nowait
{
 while (1) {
  if (get_current_clock_t() == period_1)
    #pragma omp task priority(1)
    { τ1;
      #pragma omp taskwait
    }
  if (get_current_clock_t() == period_2)
    #pragma omp task priority(2)
    { τ2;
      #pragma omp taskwait
    }
  if (get_current_clock_t() == period_3)
    #pragma omp task priority(3)
    { τ3;
      #pragma omp taskwait
    }
  }
}
```

[1] *M. A. Serrano, et.al.,* **An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling,** In ISORC 2017

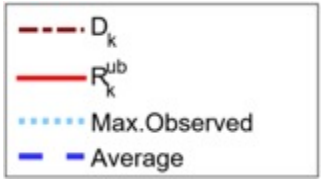
Schedulability Analysis

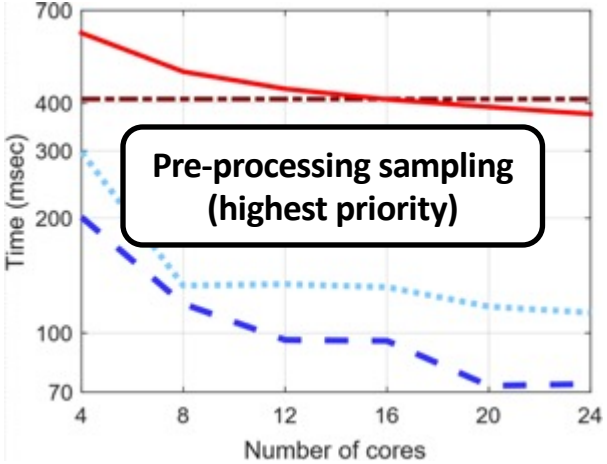
CPS composed of three concurrently OpenMP functionalities

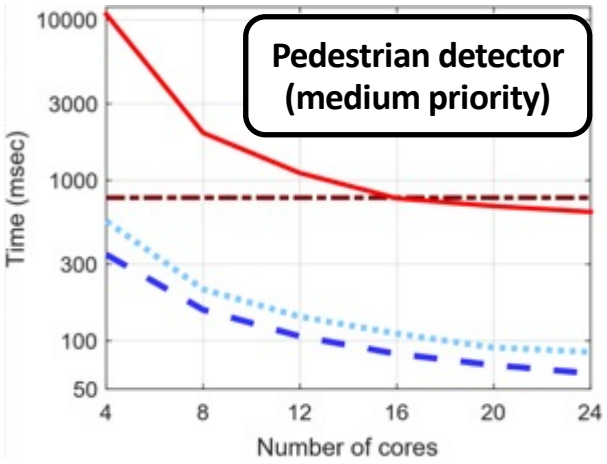
(24-core Intel Xeon)

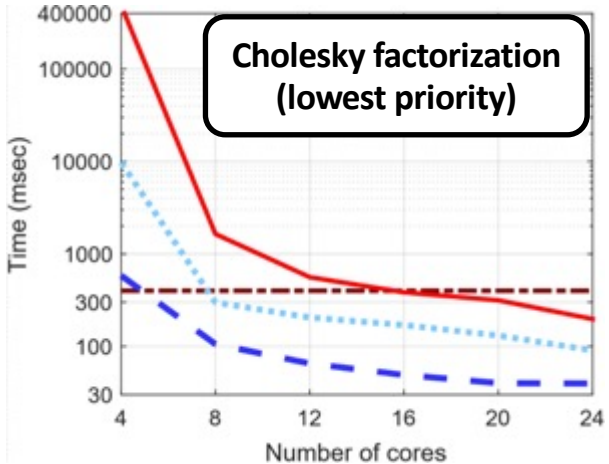
Functionality	τ_i	# nodes	P	T_k (ms)	D_k (ms)
Pre-processing sampling	τ_1	193	1	410	410
Person detector	τ_2	1299	2	780	780
Cholesky factorization	τ_3	819	3	400	400

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T = \begin{pmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{pmatrix} \begin{pmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{pmatrix} = \begin{pmatrix} L_{11}^2 & & \text{(symmetric)} \\ L_{21}L_{11} & L_{21}^2 + L_{22}^2 & \\ L_{31}L_{11} & L_{31}L_{21} + L_{32}L_{22} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{pmatrix}.$$





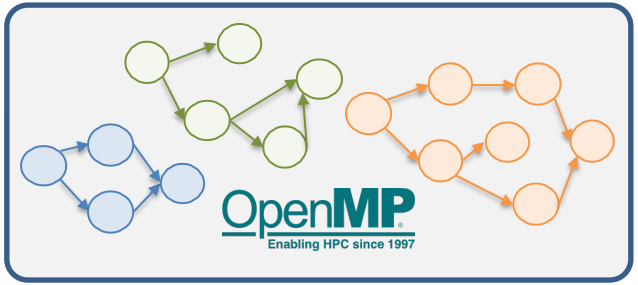


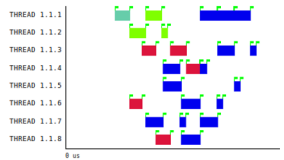


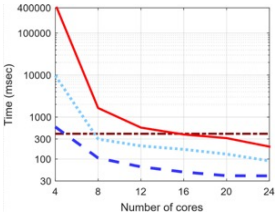
CPS and OpenMP

"I'm a software engineer, so I can confirm it works ~~by magic.~~" from a functional and timing perspective"

Cyber-Physical System (CPS)

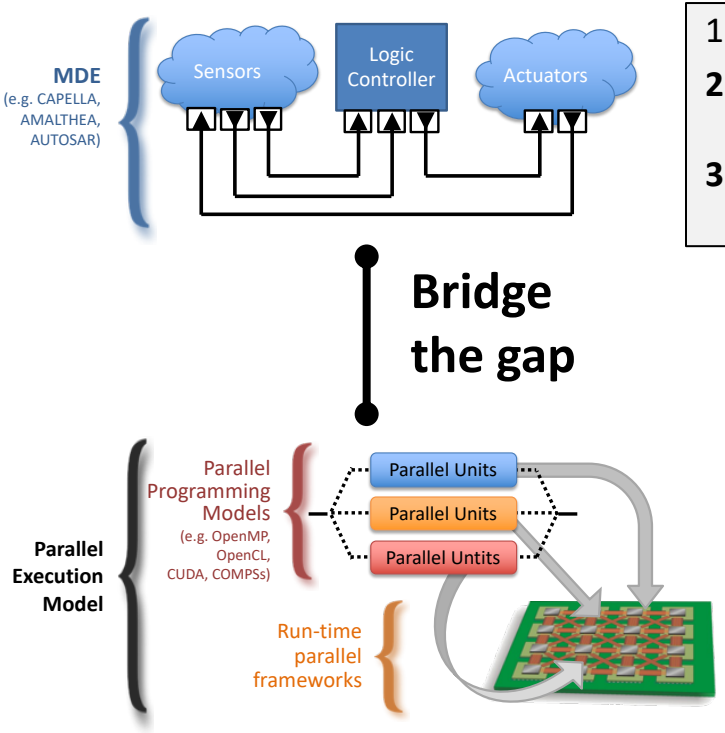






- So, can we now **develop** the most advanced CPS functionalities with **OpenMP**?...
- **… Not Really**

Model Driven Engineering and OpenMP



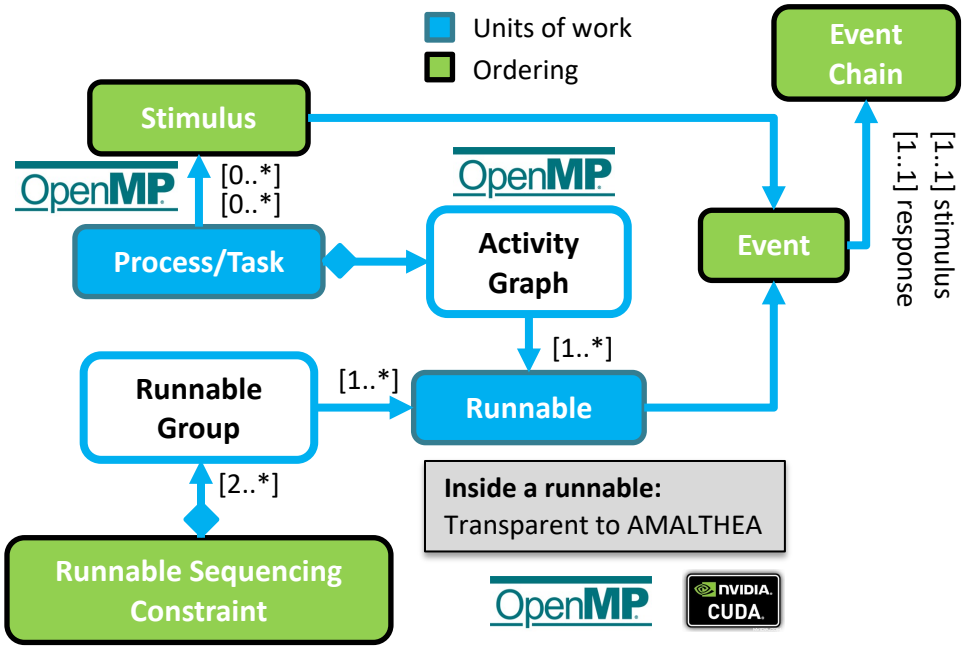
1. Construction of complex systems
2. **Formal verification** of functional and non-functional requirements with **composability** features
3. **Correct-by-construction paradigm** by means of code generation

✓ 1. **Compiler and run-time parallel frameworks** that guarantee system correctness and exploit the performance capabilities of parallel architectures

? 2. **Synthesis methods** for an efficient generation of parallel source code, while keeping non-functional and composability guarantees

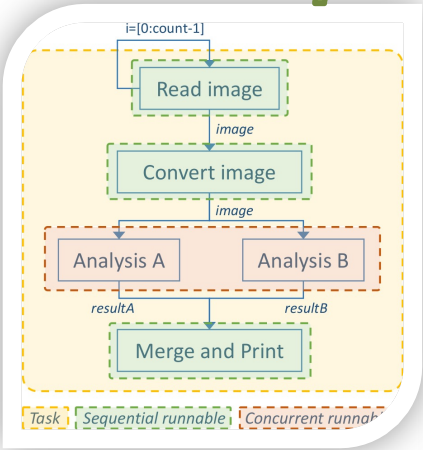
AMALTHEA/AUTOSAR and OpenMP

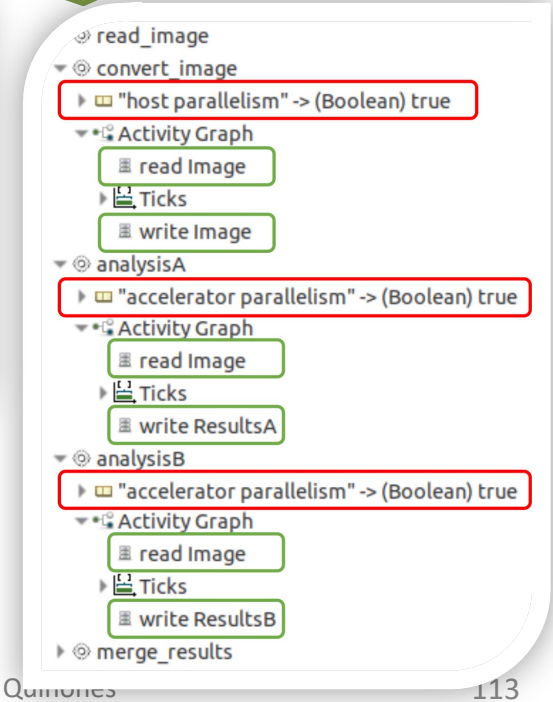
- Automotive MDE highly inspired in AUTOSAR developed by Bosch
  - Defacto standard for the development of automotive SW
  - Used by most of OEM and TIER1 and TIER2 automotive companies
- Multiple abstraction layers to define CPS SW components
  - AMALTHEA task
  - Runnable
  - Stimulus
- Compatibility between the AMALTHEA and OpenMP execution models



AMALTHEA/AUTOSAR and OpenMP

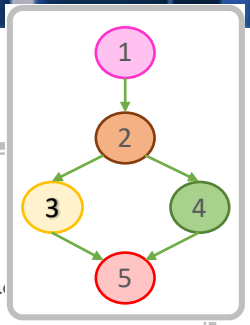
AMALTHEA DSML

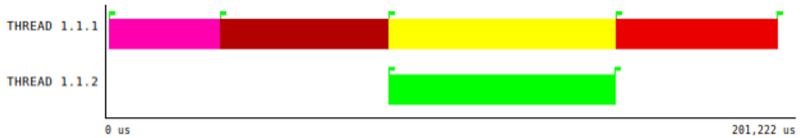




Source-code transformation

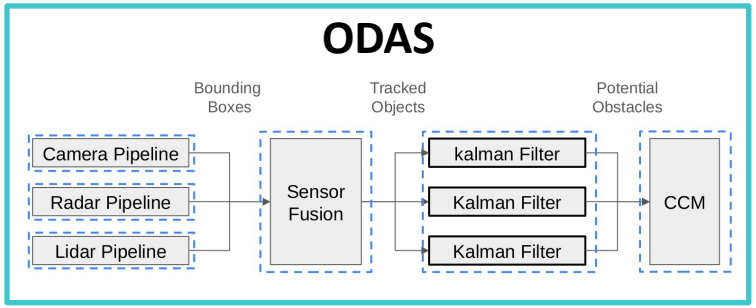
```
#pragma omp parallel
#pragma omp single
    #pragma omp task priority(x)
    {
        #pragma omp task depend(out: Ima
        run_read_image ("") ;
        #pragma omp task depend(inout:Image)
        run_convert_image ("") ;
        #pragma omp target depend(in:Image) \
                           depend(out:ResultsA)
        run_analysisA ("");
        #pragma omp target depend(in:Image) \
                           depend(out:ResultsB)
        run_analysisB ("");
        #pragma omp task depend(in:ResultsA,ResultsB)
        un_merge_results ("") ;
        #pragma omp taskwait
    }
```

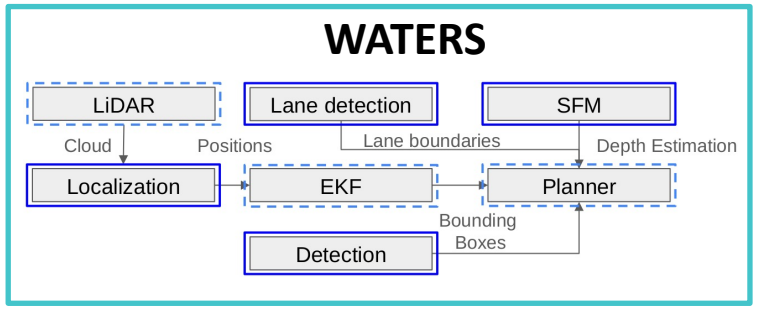


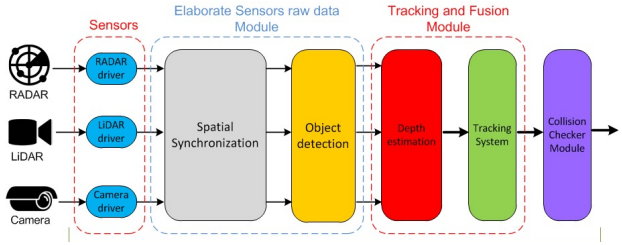


AMALTHEA and OpenMP

AMPERE
A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimisation
ampere.euproject.eu







Setup		Speedup
ODAS	2-cores	**1.88**
	4-cores	**2.62**
WATERS	4-cores + GPU	**6.21**

NVIDIA Jetson TX2 board with a GPU, a 4-core ARM CPU

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions

- The **TDG** (extracted by means of compiler and/or runtime methods) includes all the **information** needed to
  - Reason about the **timing behaviour** of OpenMP programs and so derive timing guarantees
    - The use of tracing tools (e.g., Extrae and Paraver) are needed to incorporate the required information and further understand the execution behaviour
    - The OpenMP execution model implements a **limited preemption scheduling strategy** upon which schedulability analysis can be built
  - Implement compiler mechanisms to **guarantee functional correctness** by detecting (and correcting) race conditions
- OpenMP is **compatible with the AMALTHEA DSML**, facilitating its usage in the automotive domain

Challenges we are addressing…

1. Better characterisation of the parallel execution
   - Contention on shared resources due to parallel execution
   - Overhead introduced by the run-time mechanism
   - Compiler and run-time mechanism to ensure no data-races and deadlocks
2. Modification of the OpenMP standard to better capture functional/non-functional requirements
   - Error handling mechanisms to safely recover the parallel execution from errors
   - Event-driven execution missing
3. Interoperability with different MDE

Literature of OpenMP on CPS

- **openmp.org**

Analysis of the overall OpenMP specification

1. *M. Serrano, S. Royuela and E. Quiñones.* **Towards an OpenMP Specification for Critical Real-time Systems.** In IWOMP 2018
2. *R. Vargas, E. Quinones and A. Marongiu,* **OpenMP and Timing Predictability: A Possible Union?,** In DATE 2015

Schedulability analysis for homogeneous computing

3. *M. A. Serrano, A. Melani, S. Kehr, M. Bertogna, E. Quiñones,* **An Analysis of Lazy and Eager Limited Preemption Approaches under DAG-based Global Fixed Priority Scheduling,** In ISORC 2017
4. *A. Melani, M. A. Serrano, M. Bertogna, I. Cerutti, E. Quiñones, G. Buttazzo,* **A static scheduling approach to enable safety-critical OpenMP applications,** In ASP-DAC 2017
5. *M. A. Serrano, Alessandra Melani, Marko Bertogna and Eduardo Quiñones,* **Response-Time Analysis of DAG Tasks under Fixed Priority Scheduling with Limited Preemptions,** In DATE, Dresden (Germany), March 2016
6. *Roberto E. Vargas, Sara Royuela, Maria A. Serrano, Xavier Martorell, Eduardo Quiñones,* **A Lightweight OpenMP4 Run-time for Embedded Systems**, In ASP-DAC 2016
7. *Maria A. Serrano, Alessandra Melani, Roberto Vargas, Andrea Marongiu, Marko Bertogna and Eduardo Quiñones,* **Timing Characterization of OpenMP4 Tasking Model,** In CASES 2015

Schedulability analysis for heterogeneous computing

8. *M. A. Serrano and E. Quiñones*, **Response-Time Analysis of DAG Tasks Supporting Heterogeneous Computing, i**n DAC 2018

Functional safety

9. *S. Royuela, L.M. Pinho and E. Quinones,* **Converging Safety and High-performance Domains: Integrating OpenMP into Ada,** In DATE 2018
10. *S. Royuela, A. Duran, M. A. Serrano, E. Quiñones,* **A functional safety OpenMP for critical real-time embedded systems,** In IWOMP 2017
11. *S. Royuela, X. Martorell, E. Quinones, and L. M. Pinho,* **OpenMP Tasking Model for Ada: Safety and Correctness,** In Ada-Europe 2017

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based Parallel Programming Models: The Convergence of High-Performance and Cyber-Physical Computing Domains

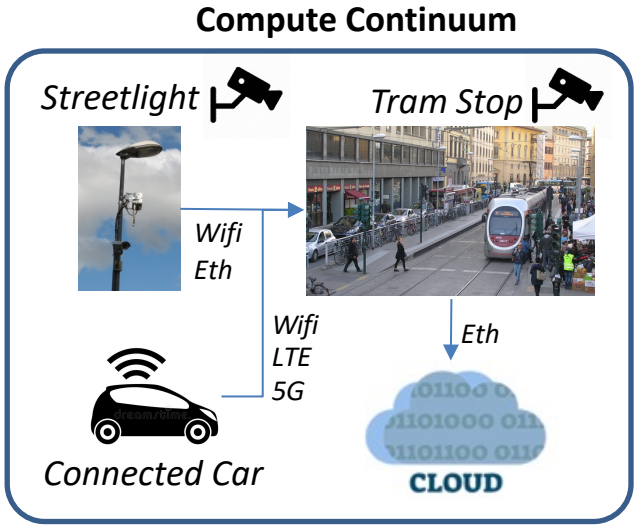
Lesson 4: Distribution across the compute continuum: COMPSs

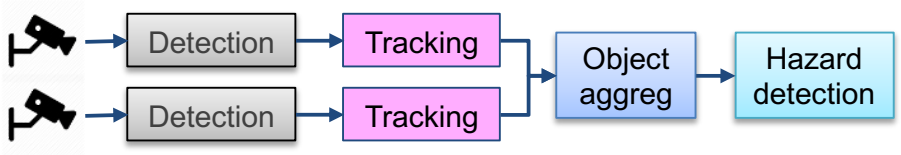
Eduardo Quiñones
{eduardo.quinones@bsc.es}

ACACES 2021, Fiuggi

Outline

- COMPSs framework
  - Execution model and memory model
  - Task model
- Reliability and resiliency
- Time predictability
  - Static allocation heuristics
- A real CPS: a smart mobility use-case
- Conclusions





COMPSs[1]

- **Programming distribute framework** highly inspired in the OpenMP tasking model
- Supports **Python, Java and C++**
  - For Python and C++, the code is annotated to describe asynchronous procedures (task) and the data dependencies among them
  - For Java, the model does not require to use any special API call, pragma or construct in the application
- **Agnostic** of the underlying distributed computing infrastructure
  - Programs do not include any infrastructure details, making applications portable
- The memory and file system space is **abstracted**, giving the illusion of a single memory space and file system
  - The runtime takes care of all the necessary data transfers.

[1] http://compss.bsc.es

Execution Model: Master-Worker

- The COMPSs runtime is composed of
  - **Master**, responsible of the execution of the *main program* and the distribution of the asynchronous tasks, honoring task data dependencies
  - **Worker**, responsible of the execution of the COMPSs tasks on the different computing resources as described in the ***resource.xml*** file, and the data transfer among workers

resources.xml

```
<ResourceList>
<ComputeNode Name="192.168.12.1">
    <Processor Name="MainProcessor">
      <ComputingUnits>6</ComputingUnits>
    </Processor>
    <Adaptors>
          <Ports>
        <MinPort>43002</MinPort>
        <MaxPort>43003</MaxPort>
</ComputeNode>
```

Native Linux

```
<Cloud>
  <CloudProvider Name="rotterdam">
  ...
  <Images>
      <Image Name="bscppc/tracking_partial:rotter
...
</Cloud>
```

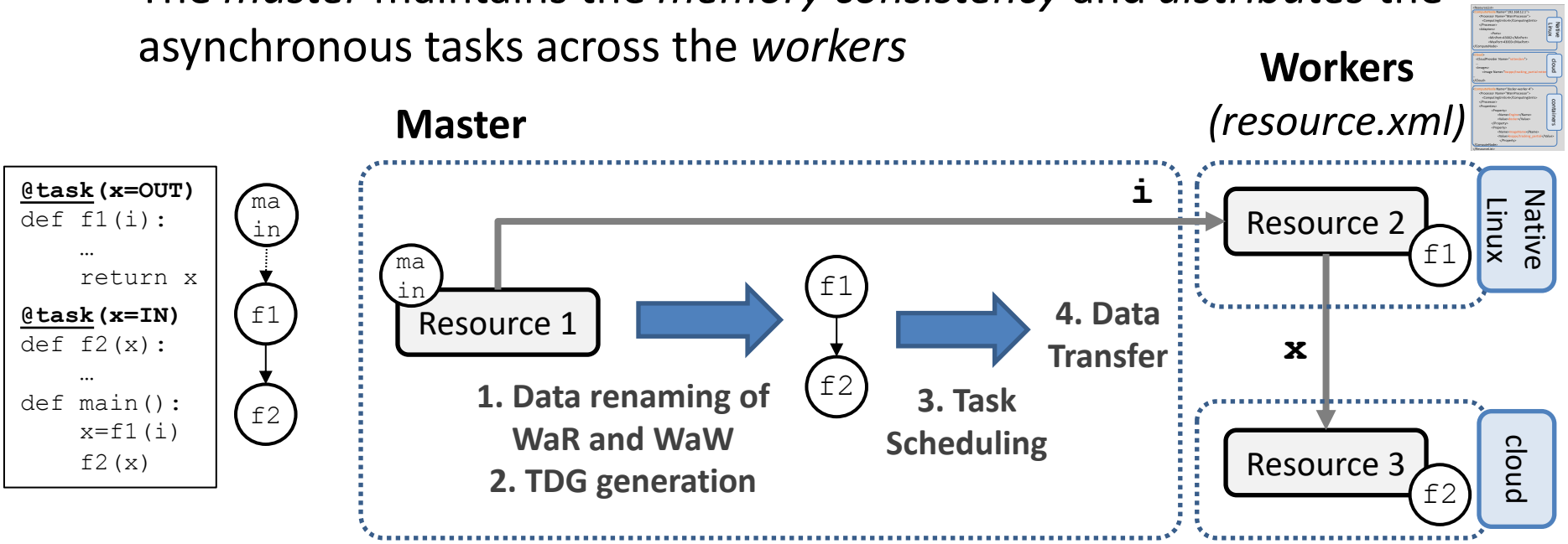
cloud

```
...
<ComputeNode Name="docker-worker-4">
    <Processor Name="MainProcessor">
      <ComputingUnits>6</ComputingUnits>
    </Processor>
    <Properties>
          <Property>
              <Name>Engine</Name>
              <Value>docker</Value>
          </Property>
          <Property>
              <Name>ImageName</Name>
              <Value>bscppc/tracking_partial</Value>
          </Property>
</ComputeNode>
</ResourceList>
```

containers

Memory Model and Parallel distribution: Task Model

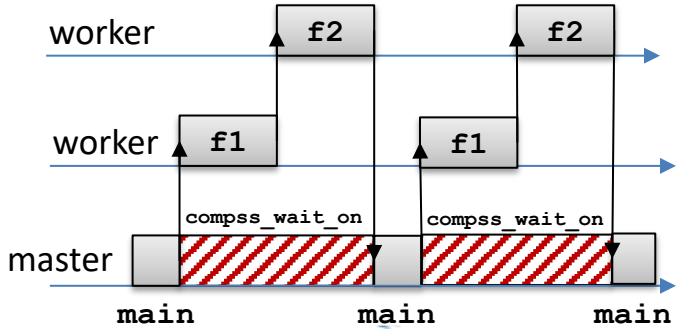
- The *master* maintains the *memory consistency* and *distributes* the asynchronous tasks across the *workers*



Memory Model and Parallel distribution: Task Model

- The COMPSs tasking model is **similar** to the OpenMP tasking model…
  - Oblivious of the underlying distributed physical layout
  - Structured and unstructured data- and task-parallelism
  - Representative construct: **`@task`** (python decorator)
  - Coarse- and fine-grain synchronization: **`compss_wait_on`** (COMPSs runtime call) and **`IN`** and **`OUT`** data dependencies (python decorator)

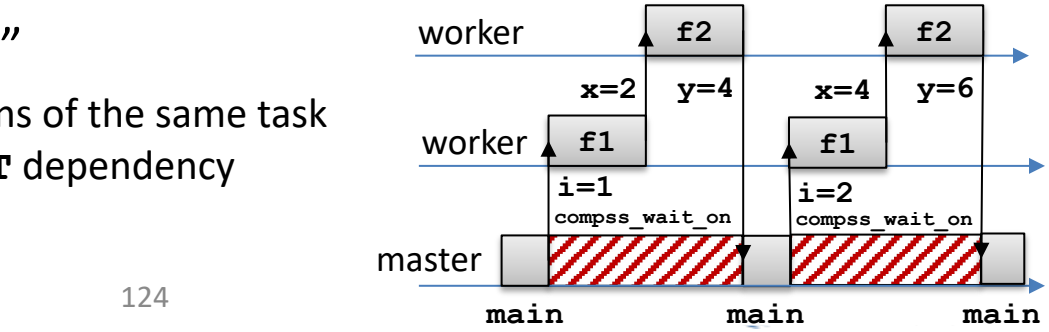
```
@task(x=OUT)
def f1(i):
    return i*2
@task(x=IN)
def f2(x):
    return x+2
def main():
    for i in [1..2]
        x=f1(i)
        y=f2(x)
        compss_wait_on(y)
```



Memory Model and Parallel distribution: Task Model

- **… but not the same!**
  - Synchronization directives implies data transfer *between workers* and *master and workers*
    - Input/output data is serialized/deserialized and stored in disk
  - There are not shared variables
  - COMPSs tasks are “stateless”
    - State across multiple executions of the same task must be included as an **INOUT** dependency

```
@task(x=OUT)
def f1(i):
    return i*2
@task(x=IN)
def f2(x):
    return x+2
def main():
    for i in [1..2]
        x=f1(i)
        y=f2(x)
        compss_wait_on(y)
```



Hands-on: COMPSs TDG

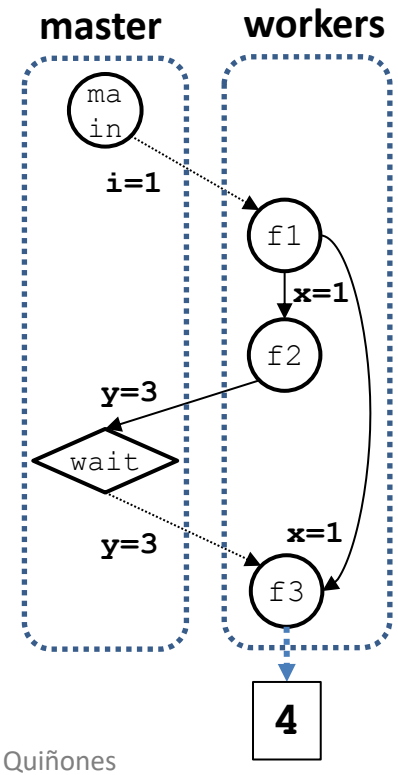
```
@task(i=IN,x=OUT)
def f1(i):
    x=i*2
    return x
@task(i=IN)
def f2(i):
    y=i+2
    return y
@task(y=IN)
def f3(x,y):
    print(x+y)
def main():
    x=f1(1)
    y=f2(x)
    compss_wait_on(y)
    f3(x,y)
```

```
@task(i=IN,x=OUT)
def f1(i):
    x=i*2
    return x
@task(i=IN,y=OUT)
def f2(i):
    y=i+2
    return y
@task(y=IN,x=IN)
def f3(x,y):
    print(x+y)
def main():
    x=f1(1)
    y=f2(x)
    f3(x,y)
```

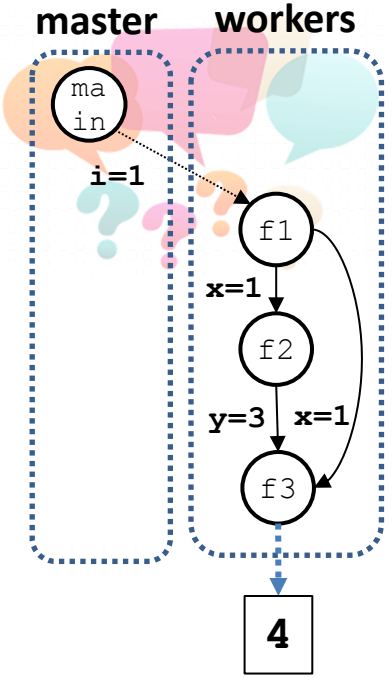
- ***Are these two source-codes equivalent from a functional and parallel perpective?***
- ***YES and NO***

Hands-on: COMPSs TDG

```
@task(i=IN,x=OUT)
def f1(i):
    x=i*2
    return x
@task(i=IN)
def f2(i):
    y=i+2
    return y
@task(y=IN)
def f3(x,y):
    print(x+y)
def main():
    x=f1(1)
    y=f2(x)
    compss_wait_on(y)
    f3(x,y)
```




```
@task(i=IN,x=OUT)
def f1(i):
    x=i*2
    return x
@task(i=IN,y=OUT)
def f2(i):
    y=i+2
    return y
@task(y=IN,x=IN)
def f3(x,y):
    print(x+y)
def main():
    x=f1(1)
    y=f2(x)
    f3(x,y)
```



Implementing CPS with COMPSs

- The complexity of parallel programming increases if guarantees on functional and non-functional correctness must be provided

1. **Functional correctness (safety)** ensure a correct system operation in response to its inputs guaranteeing system integrity
   - **Reliability**: The property that ensures the system correctness
   - **Resiliency**: The property that guarantees the system recovery if an unexcepted event impacts on system correctness, e.g., a soft transient error
2. **Non functional correctness**
   - **Time predictability:** Reasoning about the *timing behaviour* of the parallel execution to ensure the execution completes within a given *deadline*

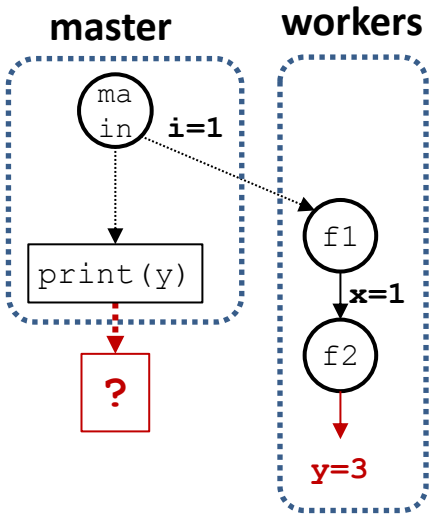
Copy-paste from the OpenMP Lesson!

Reliability and Resiliency

1. **Data races**
   - Occur when two workers access to the same memory object or file and at least one of them is a write
   - A memory object or file written by one worker cannot be read by another worker if no **synchronization** is done
   - Data races result in undefined behavior
2. **Deadlocks**
   - COMPSs does not include mutual exclusion
3. **Error handling mechanisms**
   - COMPSs does not include them to safely recover the parallel execution from errors
     - Relies on those provided by the base programming language, i.e., exceptions in case of Python or C++

Data Races and TDG

```
@task(i=IN,x=OUT)
def f1(i):
    x=i*2
    return x
@task(i=IN)
def f2(i):
    y=i+2
    return y
def main():
    x=f1(1)
    y=f2(x)
    compss_wait_on(y)
    print(y)
```



The COMPSs framework **raises an exception** when accessing **y** value!

Time predictability

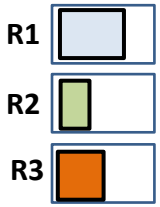
- The timing behaviour of parallel execution depends on the **allocation of parallel units** to **computing resources**
  1. The parallel structure of the application
     - The Task Dependency Graph (TDG)
  2. The scheduler responsible of allocating COMPSs tasks to workers
     - The execution profile of the parallel units into the computing resources
     - The cost of **serialization/deserialization** and **data transfers** among computing resources

Time predictability

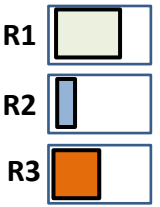
- Achivied by means of **static allocation** of COMPSs tasks to workers due to the complexity and heterogenity of the compute continuum infrastructure (including edge and cloud resources)
  - Schedulability analysis would result too pessimistic due to communication costs
- Allocation heuristics tries to minimize the computation/ communication costs
  - Based on the parallel nature of the TDG and the execution time characterisation of tasks and data transfers across the compute
  - **Extrae and Paraver supported**

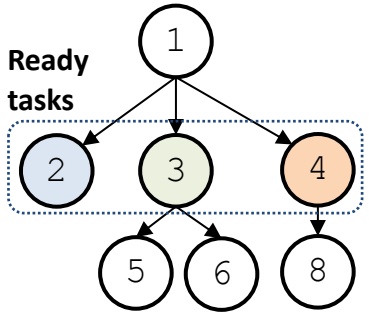
Time predictability: Static Allocation Heuristics

- Heuristics based on successors
  - *Largest Number of Successors (LNS)*
    - Order of allocation of ready tasks: **3 (R2) , 4 (R3) , 2 (R1)**

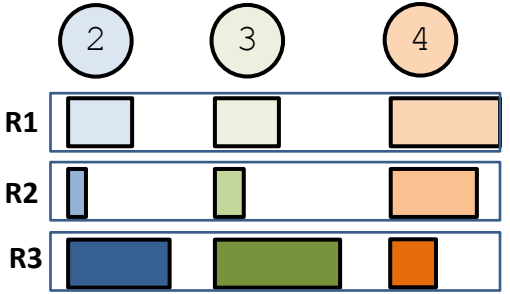


- Heuristics based on processing time
  - *Shortest Processing Time (SPT)*
    - Order of allocation of ready tasks: **2 (R2) , 4 (R3) , 3 (R1)**

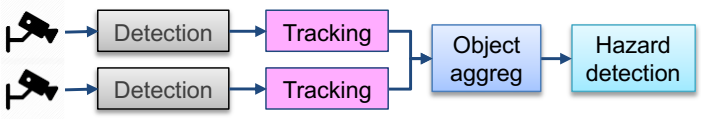




Task Profile (including communication and computation) on Resources R1, R2 and R3



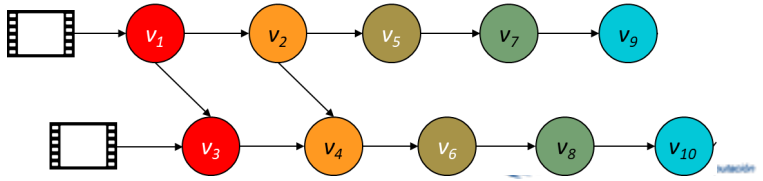
Time predictability: Static Allocation Heuristics (Example)



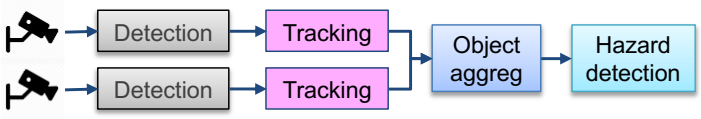

```
## Main function ##
while True:
    for i, camid in cameras:
        obj_list = get_detected_objects (camid)
        track_obj[i] = tracker(obj_list, track_obj[i])

    dedupl_obj = deduplicator(track_obj)
    snapshot = create_data_model(dedupl_obj)
    federate_to_cloud(snapshot, dC_bcknd)
```

- get_detected_objects()
- tracker()
- deduplicator()
- data_model_creation()
- federate_to_cloud()

```
def get_detected_objects (cam_id):
    return DNN_detect_obj(cam_id)

def tracker(obj_list, track_obj):
    return track(obj_list, track_obj)

def deduplicator(track_obj):
    return dedupl_obj(track_obj)

def create_data_model(dedupl_obj):
    snapshot = model.create(dedupl_obj)
    return snapshot

def federate_to_cloud(snapshot, dC_bcknd):
    snapshot.federate(backend_to_federate)
```



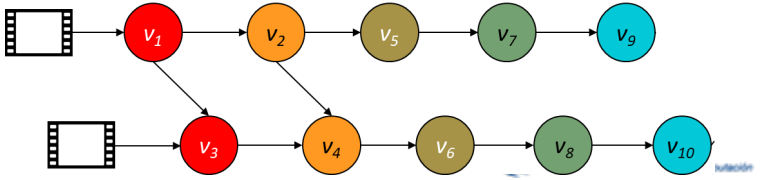
Time predictability: Static Allocation Heuristics (Example)



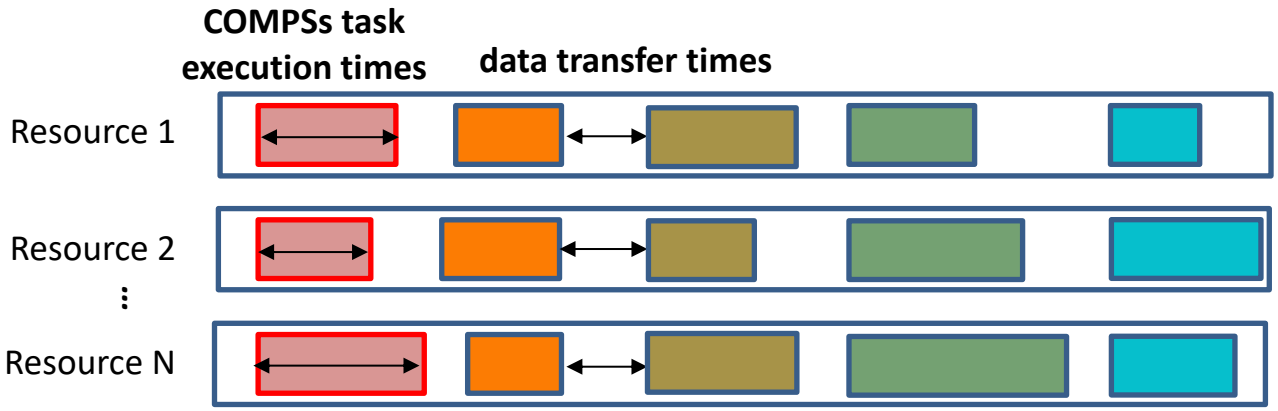

```
## Main function ##
while True:
    for i, camid in cameras:
        obj_list = get_detected_objects (camid)
        track_obj[i] = tracker(obj_list, track_obj[i])

    dedupl_obj = deduplicator(track_obj)
    snapshot = create_data_model(dedupl_obj)
    federate_to_cloud(snapshot, dC_bcknd)
```

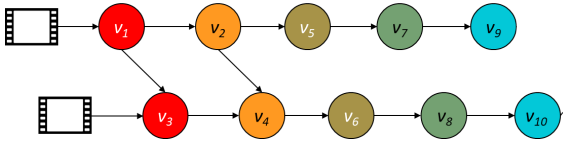
- get_detected_objects()
- tracker()
- deduplicator()
- data_model_creation()
- federate_to_cloud()

```
@task(returns=list)
def get_detected_objects (cam_id):
    return DNN_detect_obj(cam_id)

@task(obj_list=IN, track_obj=IN, returns=list)
def tracker(obj_list, track_obj):
    return track(obj_list, track_obj)

@task(obj_list=COLLECTION_IN, returns=list)
def deduplicator(track_obj):
    return dedupl_obj(track_obj)

@task(dedupl_obj=IN, model = IN)
def create_data_model(dedupl_obj):
    snapshot = model.create(dedupl_obj)
    return snapshot

@task(snapshot=IN, dC_bcknd = IN)
def federate_to_cloud(snapshot, dC_bcknd):
    snapshot.federate(backend_to_federate)
```

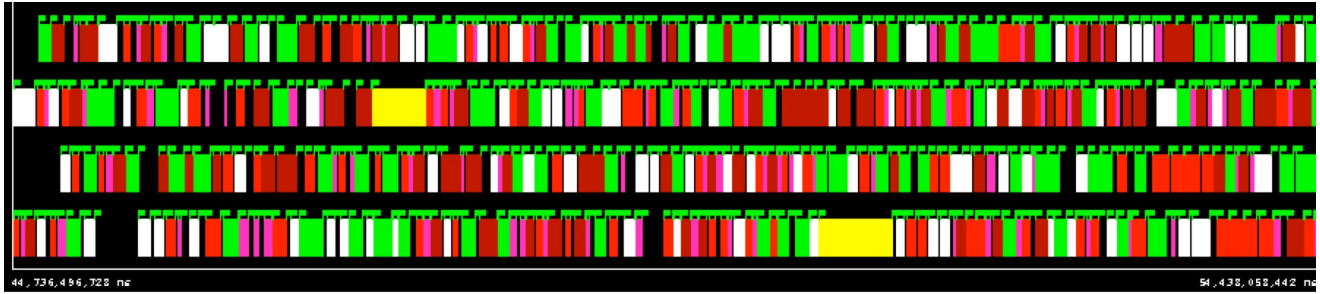


Execution time characterisation

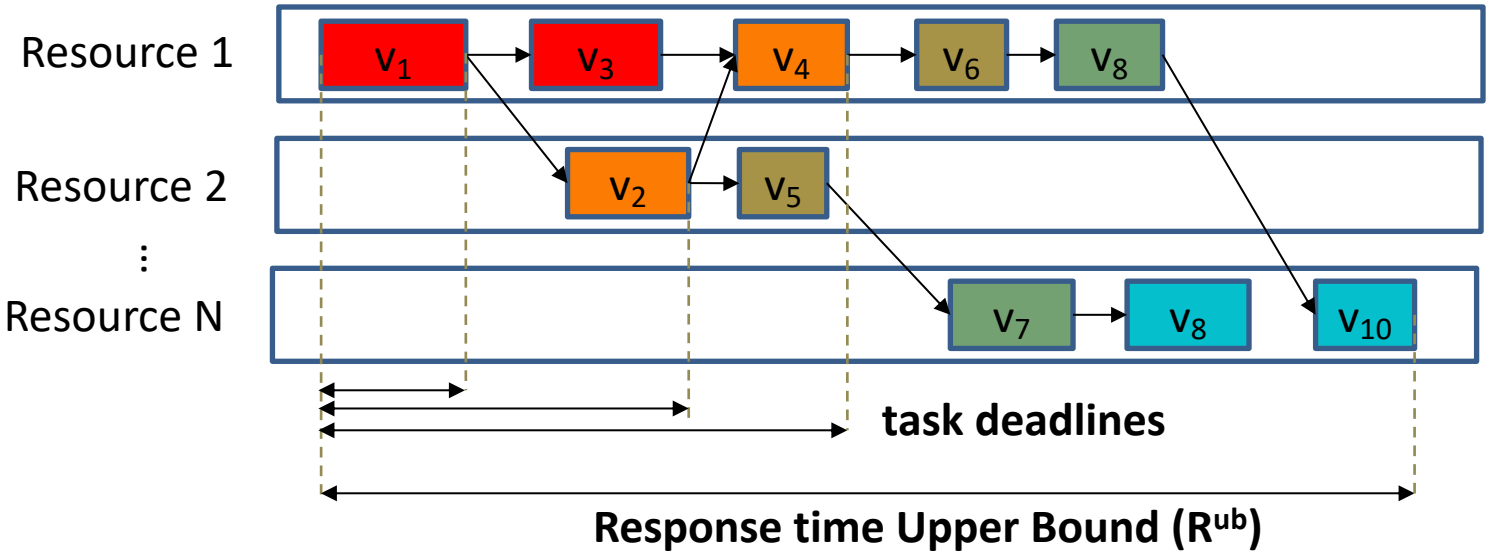




Information extracted from Extrae + Paraver!



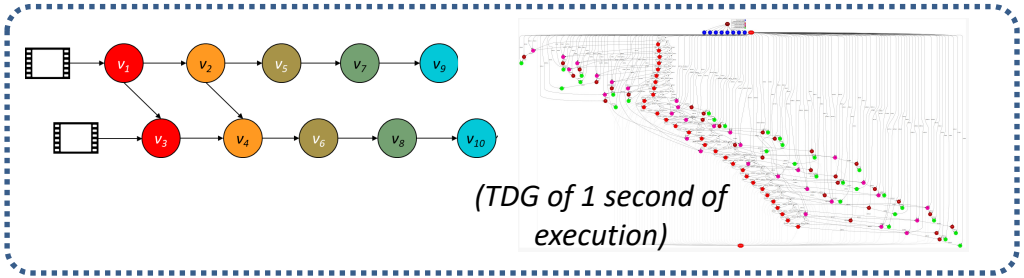
Static Allocation Heuristics



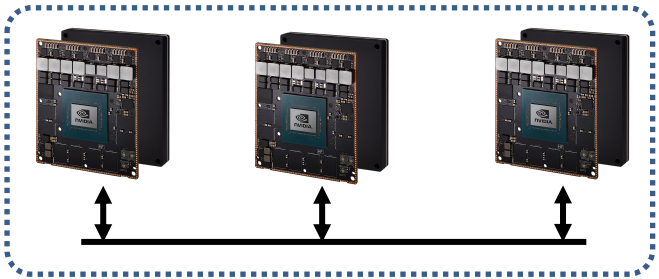
- ✓ **Heuristics** to minimize end to end response time

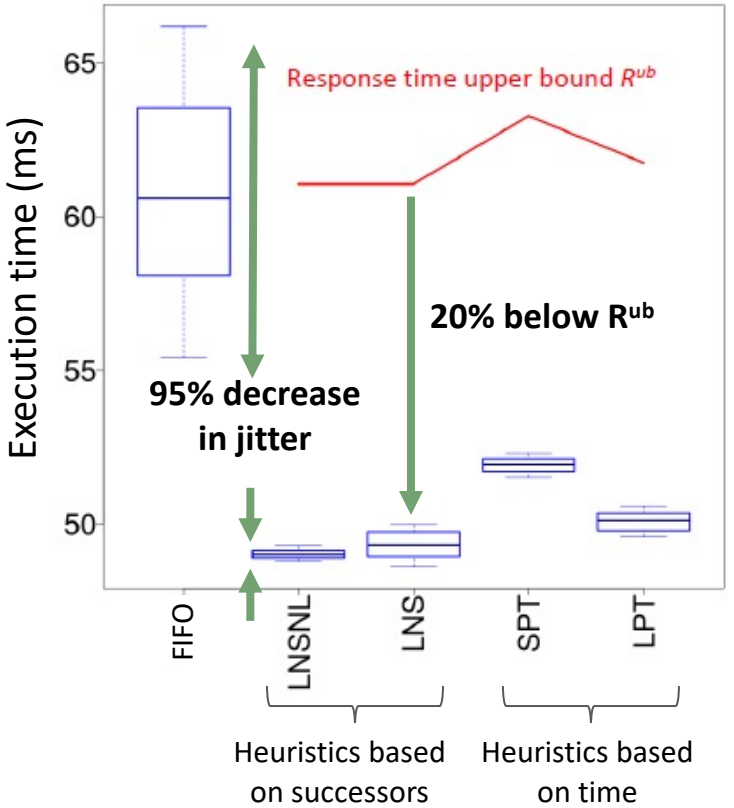
Static Allocation Heuristics

COMPSs application (5 seconds of execution)

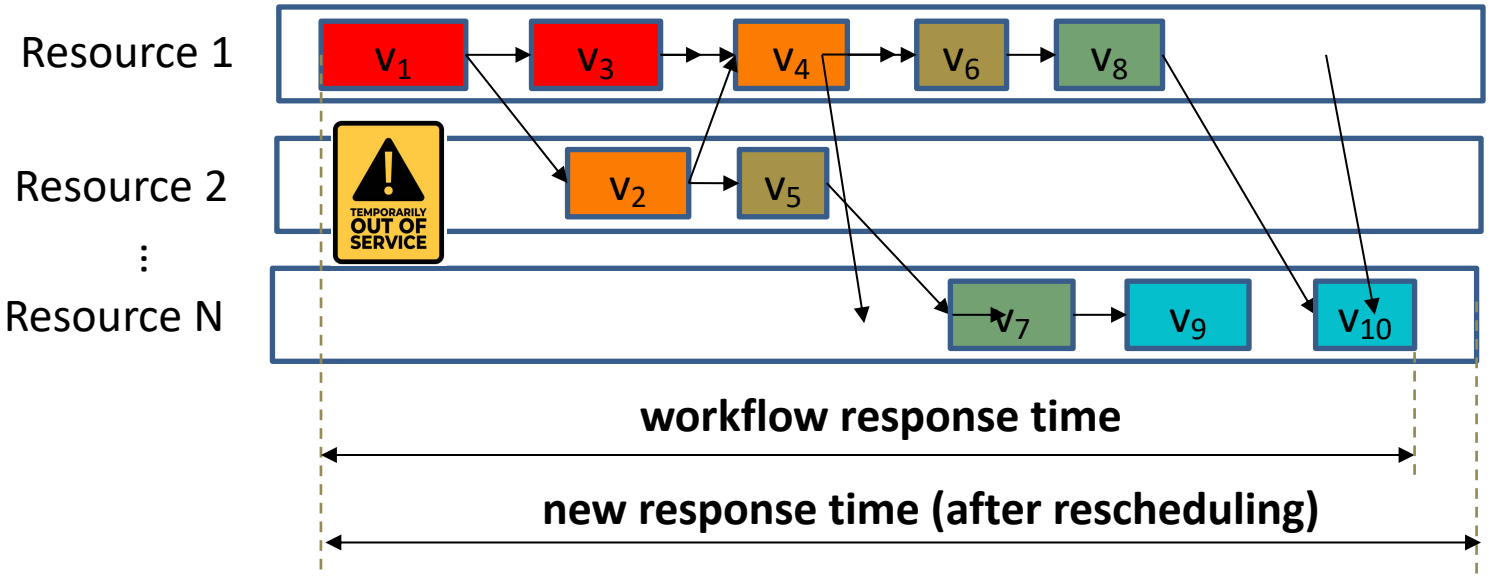


compute continuum





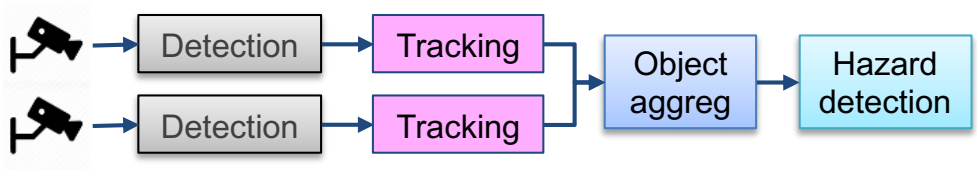
Static Allocation in a Dynamic Environment



- ✓ **Re-scheduling** based on resource availability at runtime

A Real CPS: A Smart Mobility TDG

- Extract valuable knowledge from a distributed sensing infrastructure, executed on a distributed computing infrastructure



CLASS

Edge and Cloud Computation: A Highly Distributed Software for Big Data Analytics

class-project.eu (City of Modena)

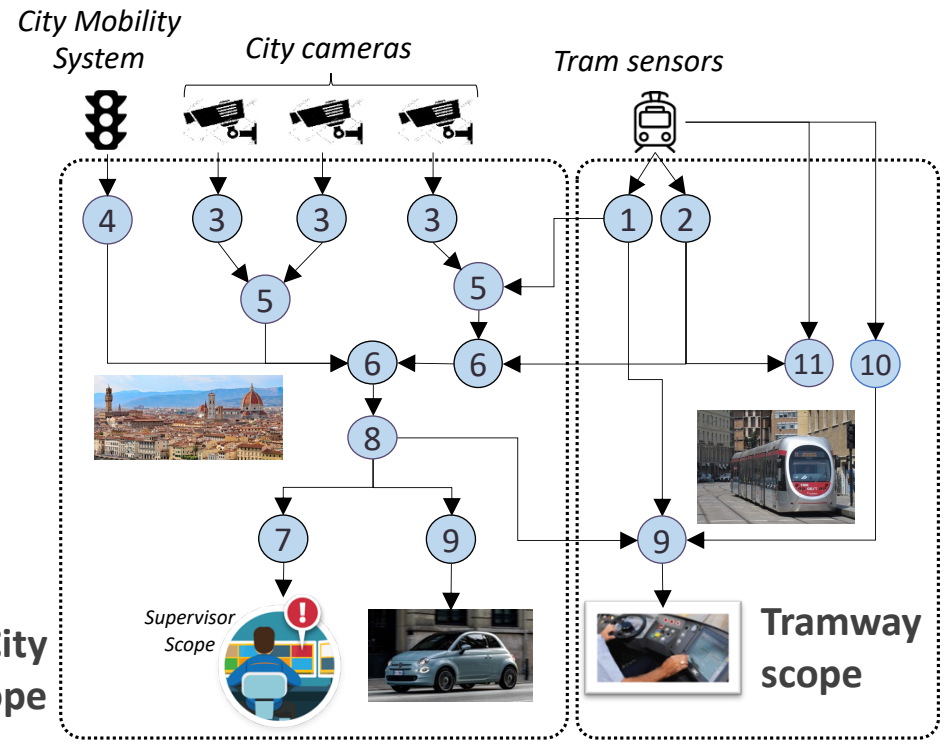
ELASTIC

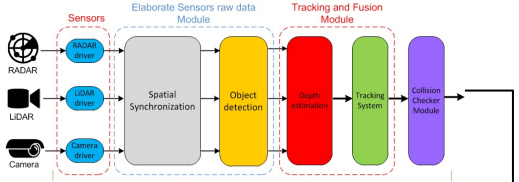
A Software Architecture for Extreme-ScaLe Big-Data AnalyticS in Fog CompuTIng Ecosystems

elastic-project.eu (City of Florence)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

A Real CPS: A Smart Mobility TDG

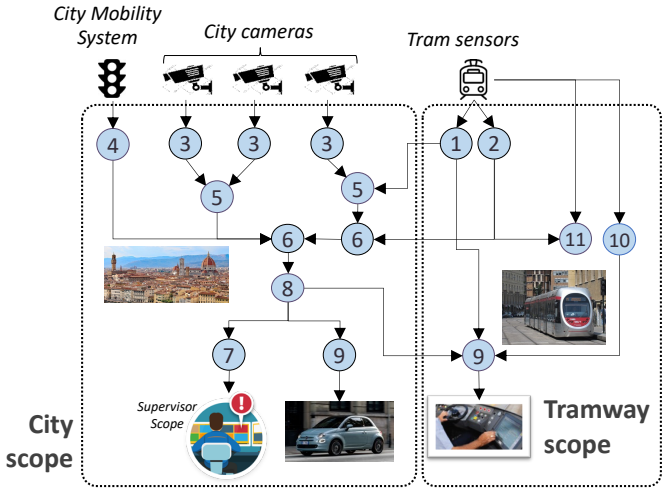


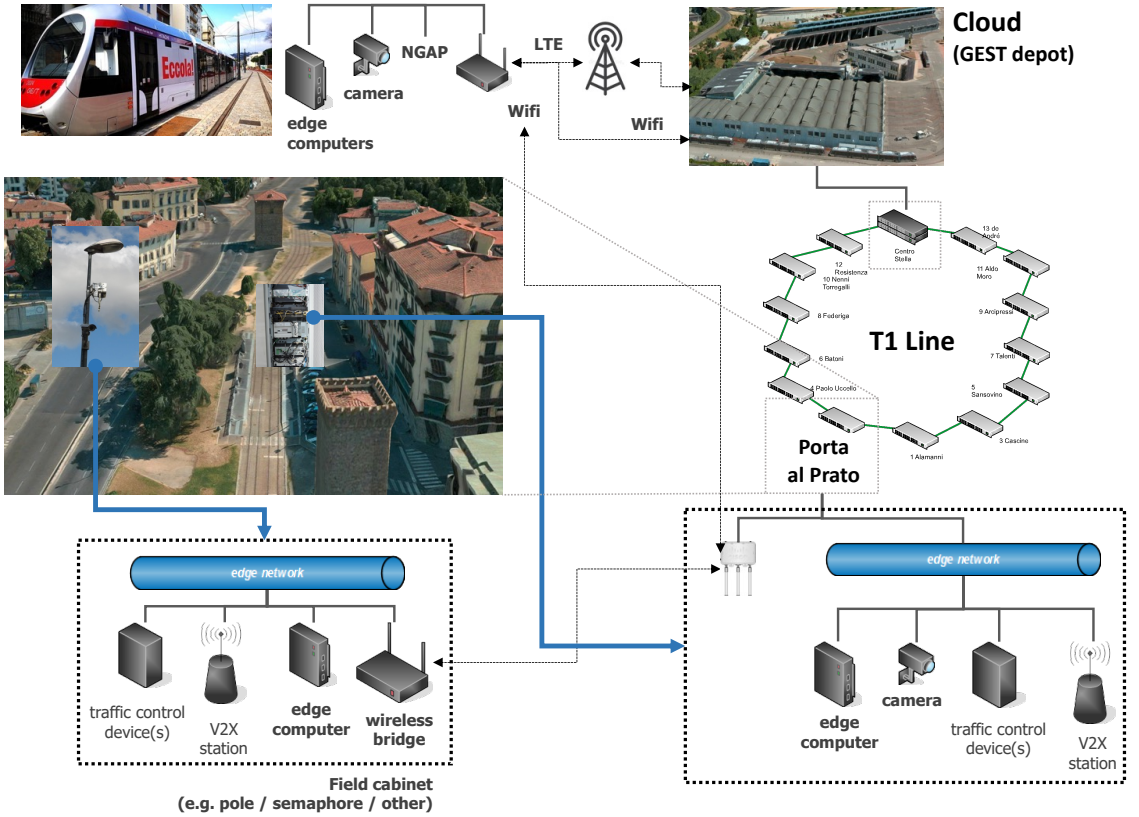


Data Analytics Methods (COMPSs tasks)

1. Sensor fusion
2. Tram position
3. Object recognition
4. UTC/Supervisor consolidation
5. Data fusion
6. Data aggregation
7. Dashboard
8. Hazard detection
9. Alert visualization (cars/trams)
10. Electric power consumption
11. Defect Detector

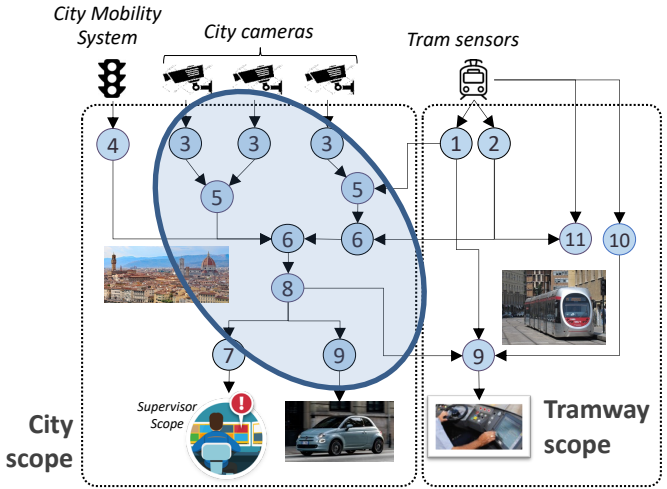
A Real CPS: A Smart Mobility TDG

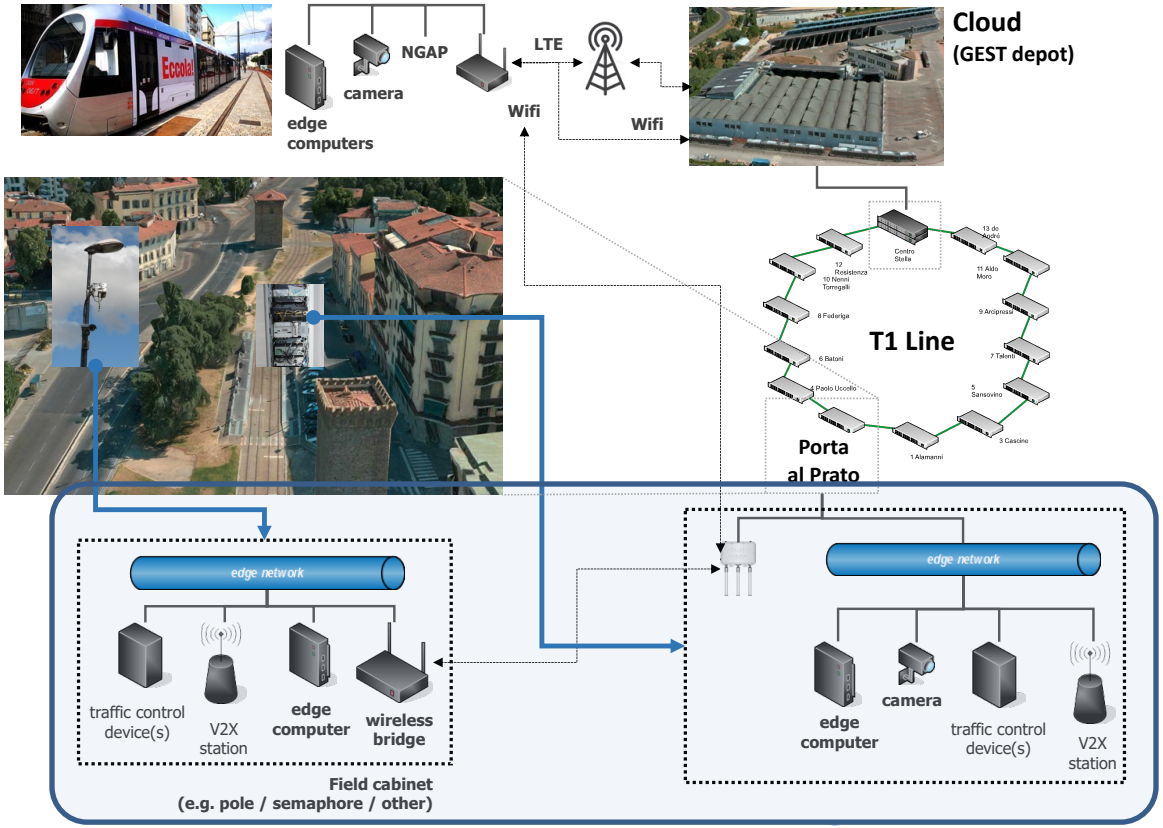




Described in the ***resources.xml*** file!

A Real CPS: A Smart Mobility TDG

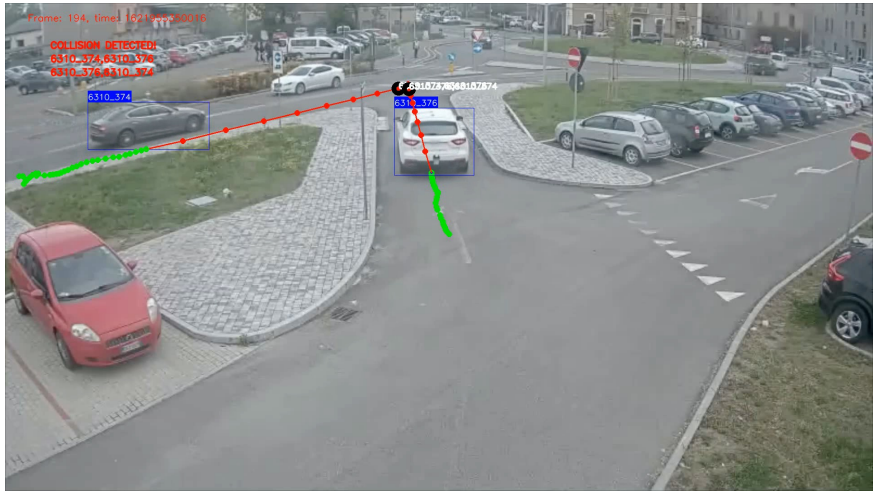


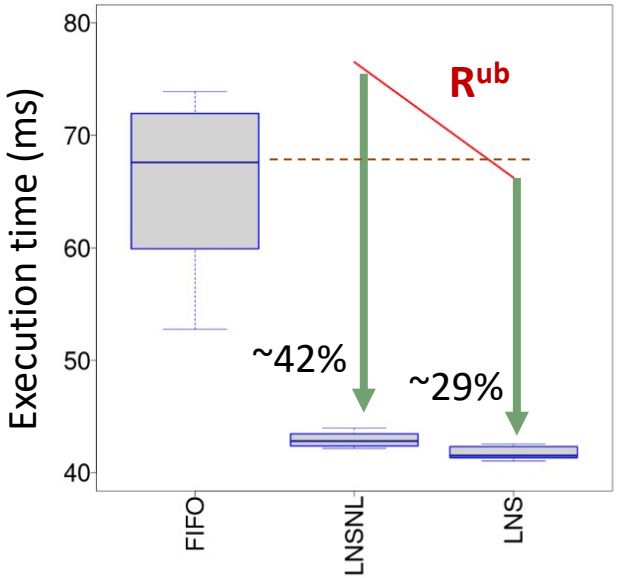


Described in the ***resources.xml*** file!

A Real CPS: A Smart Mobility TDG

Simultaneous processing of 3 video sources with the COMPSs data analytics workflow





Conclusion

- COMPSs provides a task-based framework for the development of complex data-analytics workflows
  - Similar principles of OpenMP
  - Time predictability is achieved by means of static allocation heuristics
    - The use of tracing tools (e.g., Extrae and Paraver) are needed to incorporate the required information and further understand the execution behaviour
- Currently being applied in real CPS projects

Home-take Message

1. CPS **requires parallel computation** to cope with the performance requirements of the most advanced functionalities, and…
2. … current task-based parallel programming models allows to **reasoning about functional correctness and time predictability** while removing from developers the responsibilty of managing the complexity of parallel execution
3. Unfortunately, **reasoning** is not enough… **it must be guaranteed!!!**

VERY INTERESTING RESEARCH IS STILL PENDING!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Task-based Parallel Programming Models: The Convergence of High-Performance and Cyber-Physical Computing Domains

Eduardo Quiñones
{eduardo.quinones@bsc.es}

ACACES 2021, Fiuggi