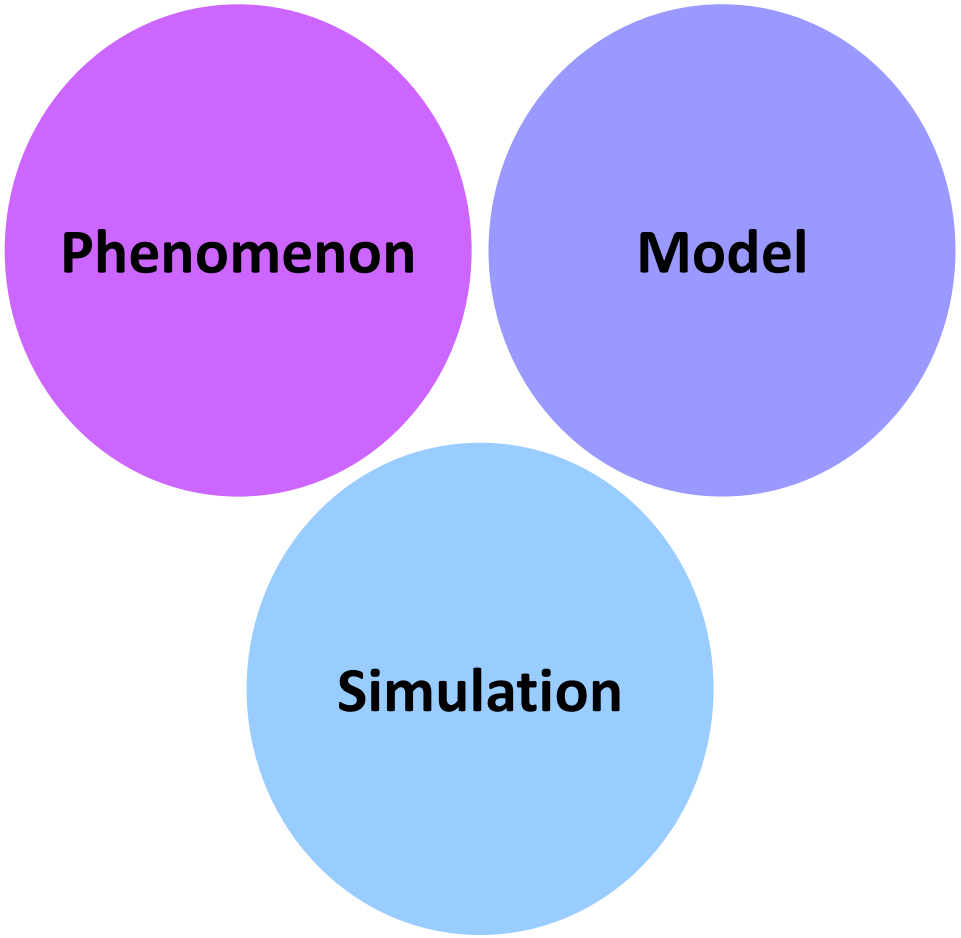
Teaching Parallel Programming in Interdisciplinary Studies

Eduardo Cesar, Ana Cortés, Antonio Espinosa, Tomàs Margalef, Juan Carlos Moure, Anna Sikora and Remo Suppi

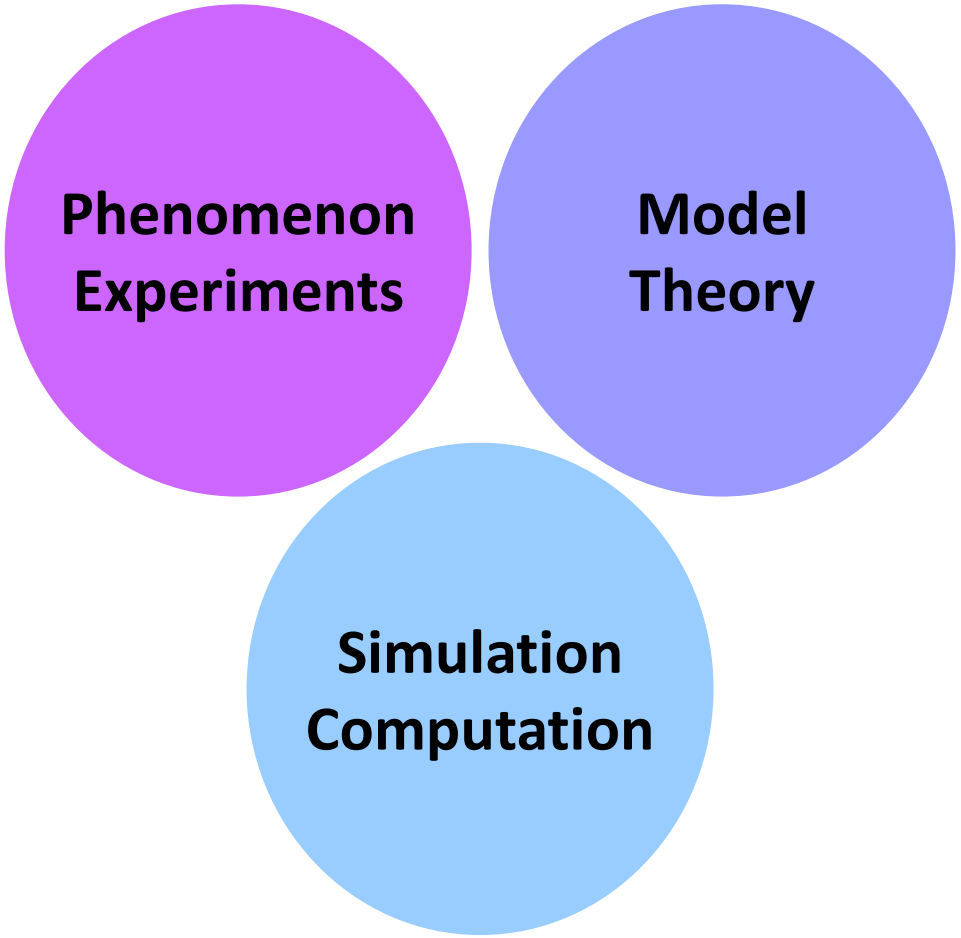
Computer Architecture and Operating Systems Department

Universitat Autònoma de Barcelona

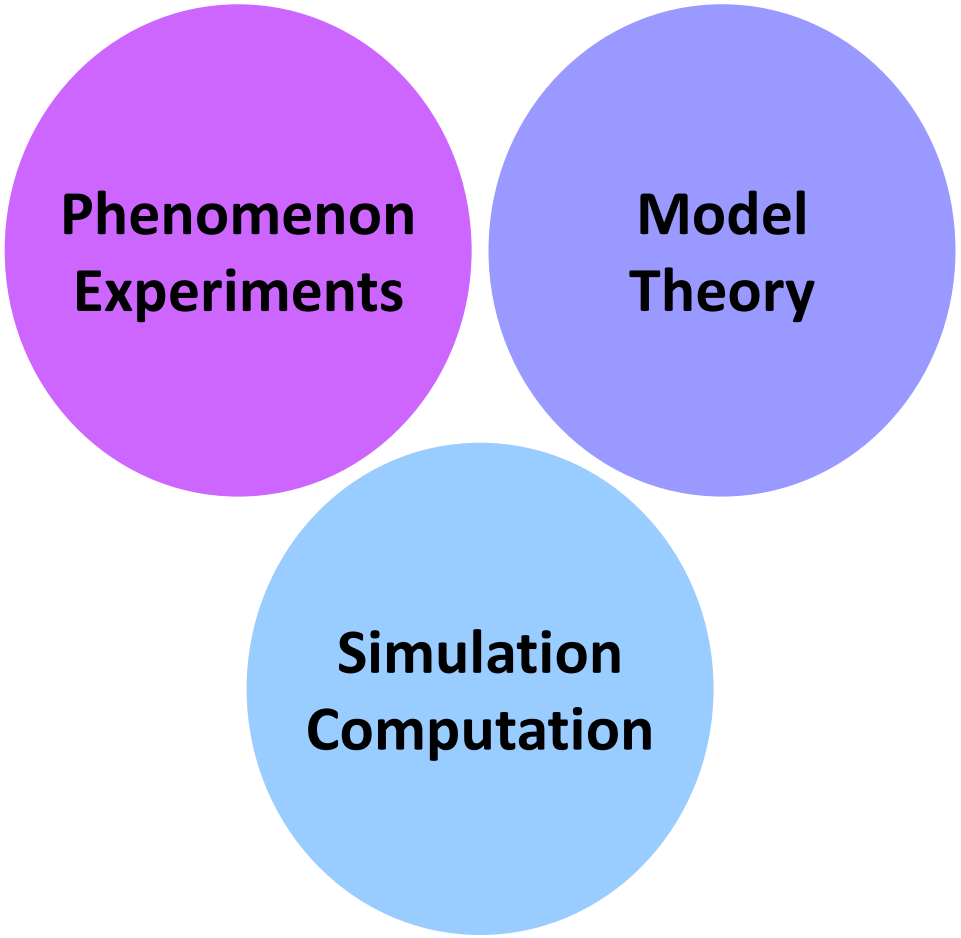
The three pillars of science



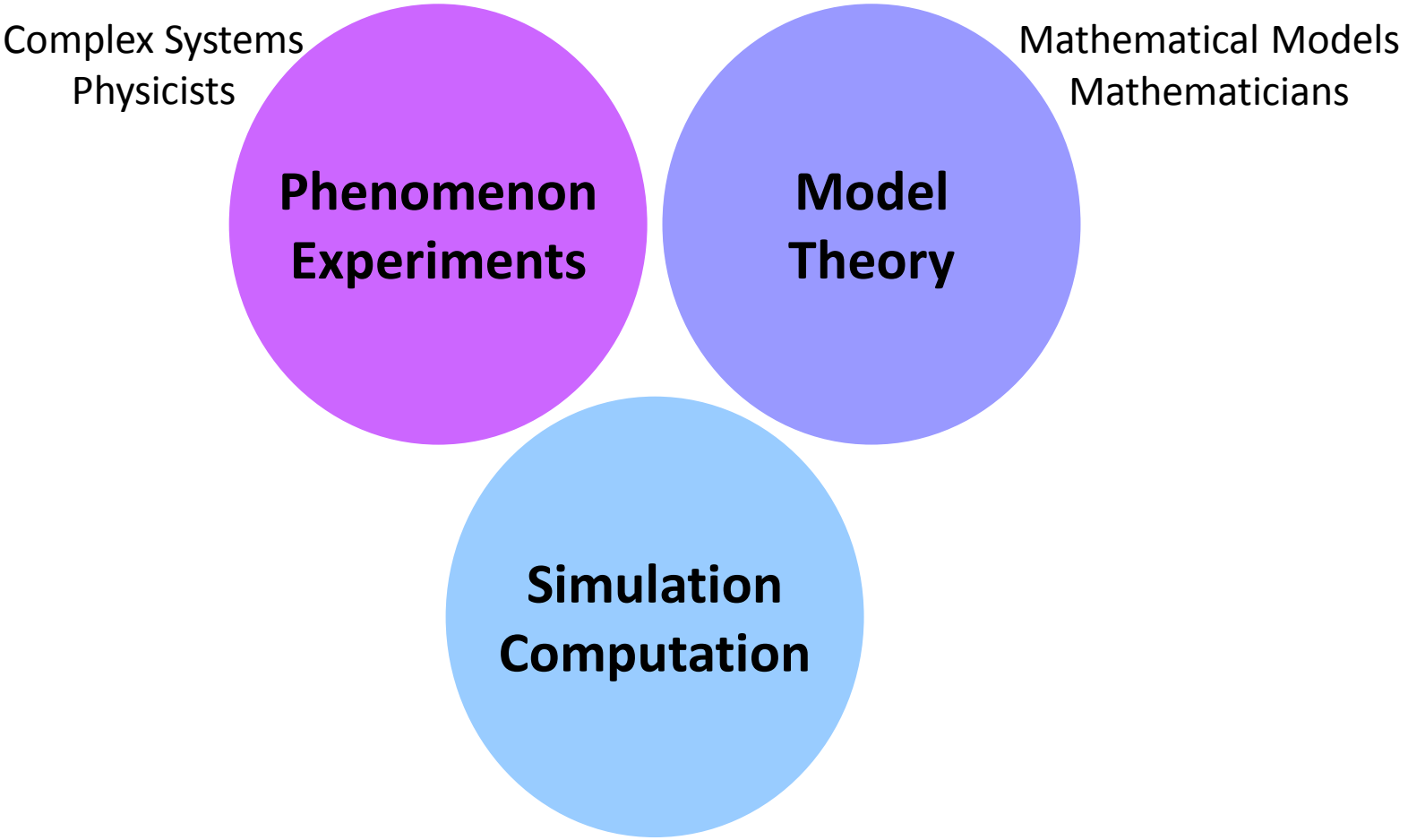
The three pillars of science



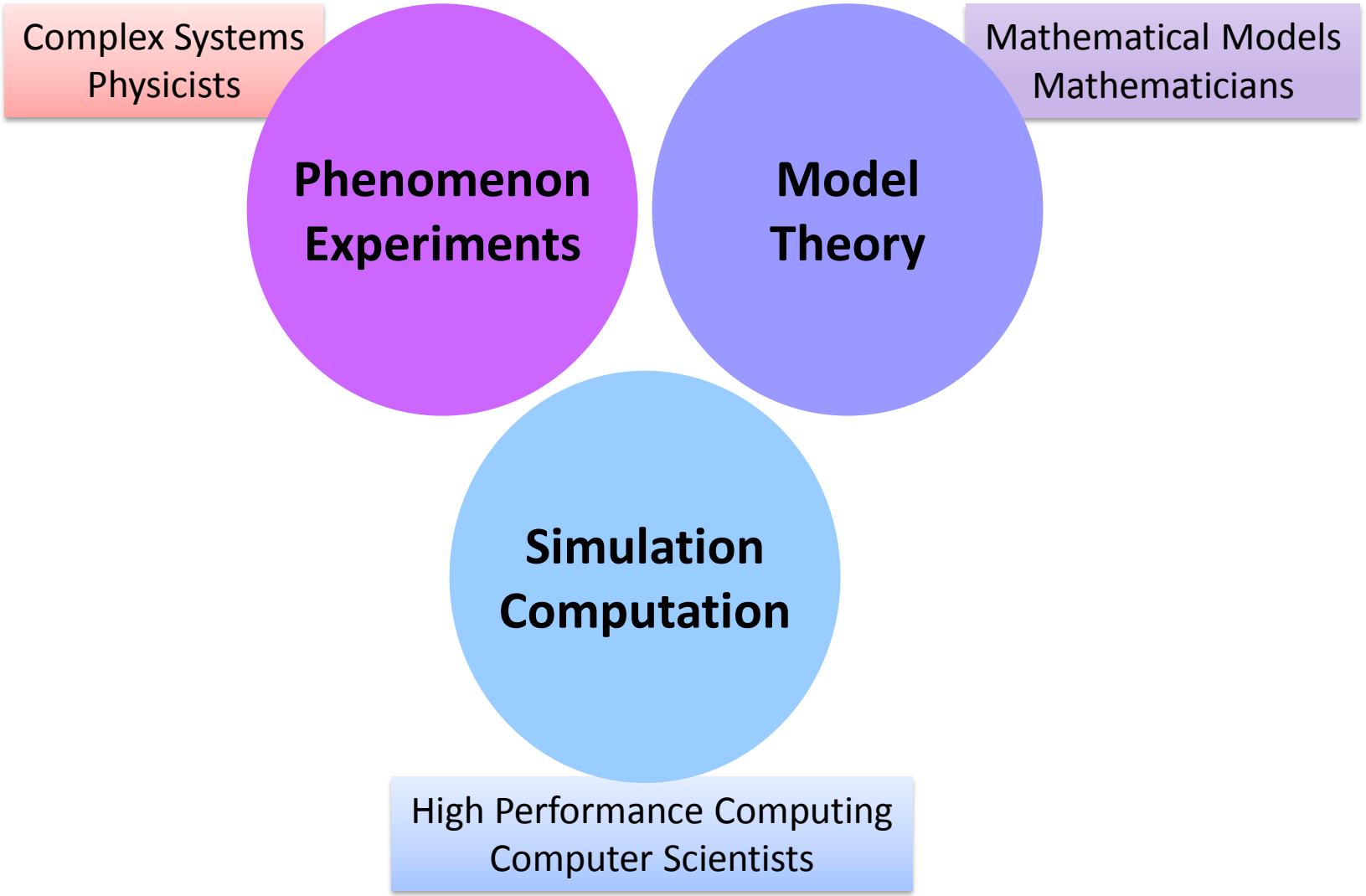
Computational Science and Engineering



Computational Science and Engineering



MSc: Modelling for Science and Engineering



MSc: Modelling for Science and Engineering

Interdisciplinary Master

Teachers

- Complex Systems Physicists
- Mathematical Models Mathematicians
- High Performance Computing Computer Scientists

Students

- Physics
- Mathematics
- Chemistry
- Biology
- Geology
- Engineering

MSc: Modelling for Science and Engineering

Interdisciplinary Master

High Performance Computing
Computer Scientists

Students

Physics

Mathematics

Chemistry

Biology

Geology

Engineering

Different background on computing

- Some programming background
- No background on Parallel Programming
- No background on performance analysis

MSc: Modelling for Science and Engineering

Interdisciplinary Master

High Performance Computing

Parallel Programming

Applied Modelling and Simulation

caos

UAB
Universitat Autònoma de Barcelona

Parallel Programming

- C programming language
- Shared Memory
  - OpenMP
- Message Passing
  - MPI
- Accelerators programming
  - CUDA
- Performance Analysis

caos

UAB
Universitat Autònoma de Barcelona

Parallel Programming

- C programming language
  - Establish a common basic level
  - Main features of C programming
  - Lab exercises
    - Editing
    - Compiling
    - Running and debugging in a cluster
    - NFS
    - Submitting to a queue system

caos

UAB
Universitat Autònoma de Barcelona

Parallel Programming

- Parallel Algorithms
  - Parallel Thinking
  - Example algorithms:
    - Matrix multiplication
    - Parallel Prefix
  - Programming paradigms
    - Master/Worker
    - SPMD
    - Pipeline

Parallel Programming

- Shared Memory: OpenMP

- **Introduction**. Concept of thread, shared and private variables, and need for synchronization.
- **Fork-join model.** The `#pragma omp parallel` clause. Introducing parallel regions.
- **Data parallelism: parallelizing loops.** The `#pragma omp for` clause. Data management clauses (`private`, `shared`, `firstprivate`).
- **Task parallelism: sections.** The `#pragma omp sections` and `#pragma omp section` clauses.
- **OpenMP runtime environment function calls.** Getting the number of threads of a parallel region, getting the thread id, and other functions.
- **Synchronization.** Implicit synchronization, `nowait` clause. Controlling executing threads, `master`, `single`, and `barrier` clauses. Controlling data dependencies, `atomic` and `reduction` clauses.
- **Performance considerations.** Balancing threads' load, `schedule` clause. Eliminating barriers and critical regions.

Parallel Programming

- Shared Memory: OpenMP

Simple example: adding two vectors.

```
for ( i = 0 ; i < N; i++ )
        c [ i ] = a [ i ] + b [ i ] ;
```

OpenMP: adding two vectors.

```
#pragma omp parallel for
for ( i = 0 ; i < N; i++ )
        c [ i ] = a [ i ] + b [ i ] ;
```

- Shared Memory: OpenMP

String simulation main computation loop.

```
for (t=1; t<=T; t++) {
    for (x=1; x<X; x++)
        U3[x] = L2*U2[x] + L*(U2[x+1] + U2[x-1]) - U1[x];
    double *TMP = U3 ;
    // rotate usage of vectors
    U3=U1 ; U1=U2 ; U2=TMP;
}
```

Parallelized string simulation main computation loop.

```
#pragma omp parallel first private (T,U1,U2,U3)
for (t=1; t<=T; t++) {
    #pragma omp for
    for ( x=1; x<X; x++)
        U3[x] = L2*U2[x] + L*(U2[x+1] + U2[x-1]) - U1[x];
    double TMP =U3 ;
    // rotate usage of vectors
    U3=U1 ; U1=U2 ; U2=TMP;
}
```

Parallel Programming

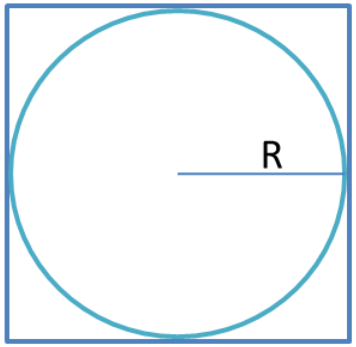
- Message Passing: MPI

- **Message passing paradigm.** Distributed memory parallel computing, the need for a mechanism for interchanging information. Introducing MPI history.
- **MPI program structure.** Initializing and analyzing the environment `MPI_Init` and `MPI_Finalize`. Communicator's definition (`MPI_COMM_WORLD`), getting the number of processes in the application (`MPI_Comm_size`) and the process rank (`MPI_Comm_rank`). General structure of an MPI call.
- **Point-to-point communication.** Sending and receiving messages (`MPI_Send` and `MPI_Recv`). Sending modes: standard, synchronous, buffered and ready send.
- **Blocking and non-blocking communications.** Waiting for an operation completion (`MPI_Wait` and `MPI_Test`).
- **Collective communication.** Barrier, broadcast, scatter, gather and reduce operations.
- **Performance considerations.** Overlapping communication and computation. Measuring time (`MPI_Time`). Discussion on the communication overhead. Load balancing.

Parallel Programming

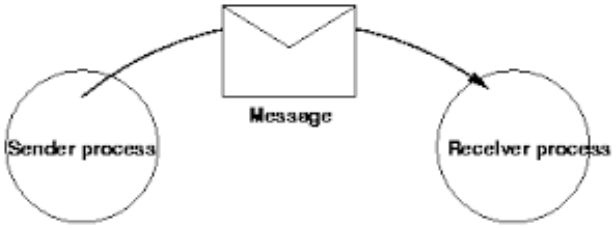
- Message Passing: MPI

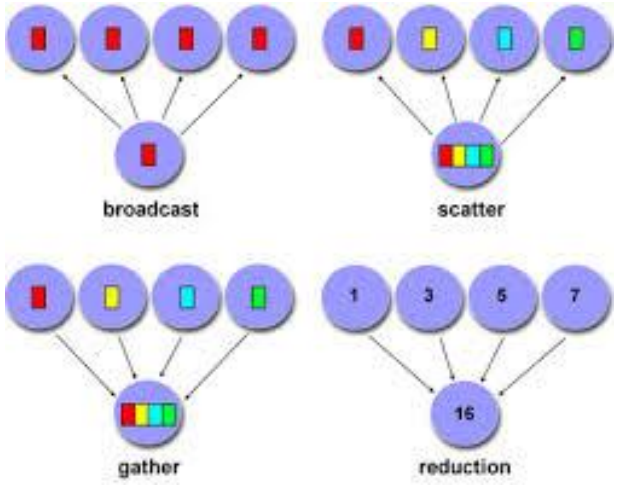
Computing π aproximation using the dartboard approach



Parallel implementation using MPI:

- Point-to-point communication
- Collective communication





Parallel Programming

UAB
Universitat Autònoma de Barcelona

- Accelerators programming: CUDA
  - Awarded Nvidia GPU Education and Research Center
- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Extensions to enable heterogeneous programming
  - APIs to manage devices, memory etc.

caos

UAB
Universitat Autònoma de Barcelona

Parallel Programming

- Accelerator Programming: CUDA

- **Introduction**. Massive data-level parallelism.
  Hierarchy of threads: warp, CTA and grid

- **Host and Device.** Move data and allocate memory.

- **Architectural restrictions.** Warp size, CTA and grid dimensions

- **Memory Space.** Global, Local and Shared Memory.

- **Synchronization.** Warp-level and CTA-level.

- **Performance considerations.** Excess of threads.
  Increasing Work per Thread.

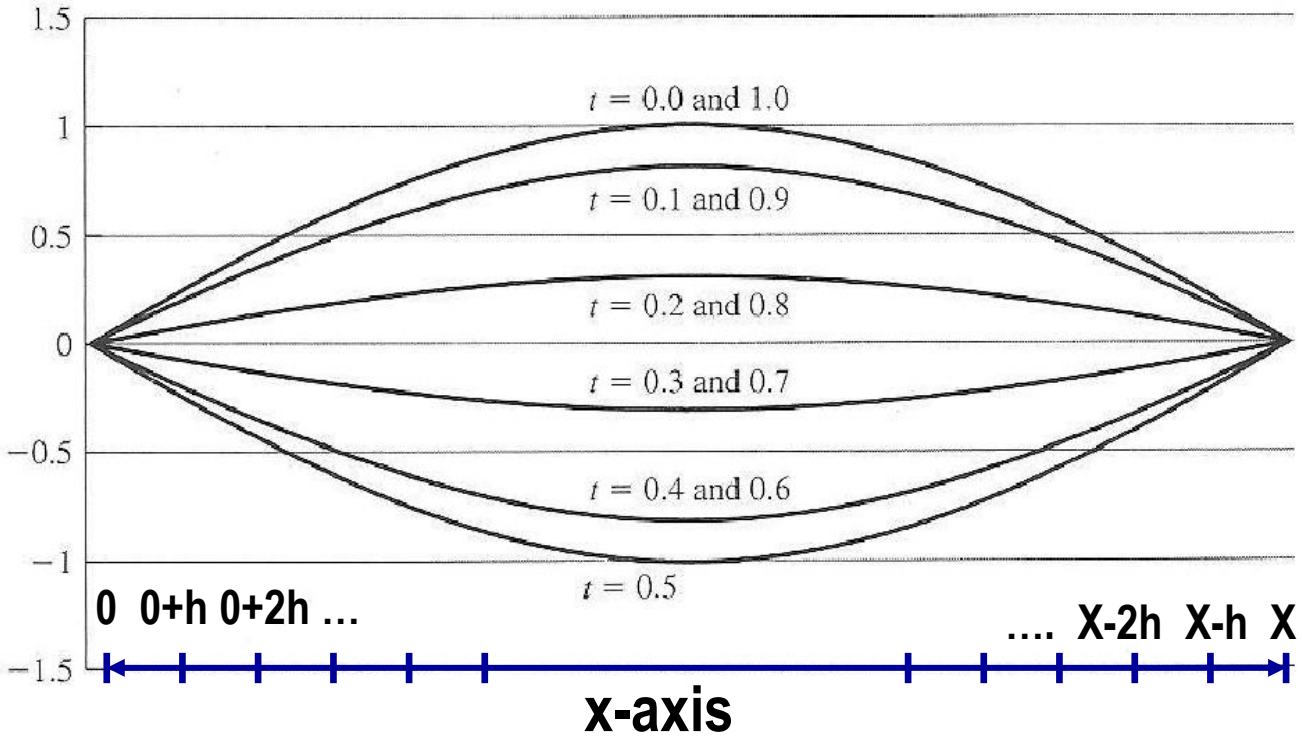
Finite Difference Method

Vibrating String

$U_{x,t}$: describes string movement in point *x* & time *t*

X+1: finite points

T: time intervals of ***k*** seconds

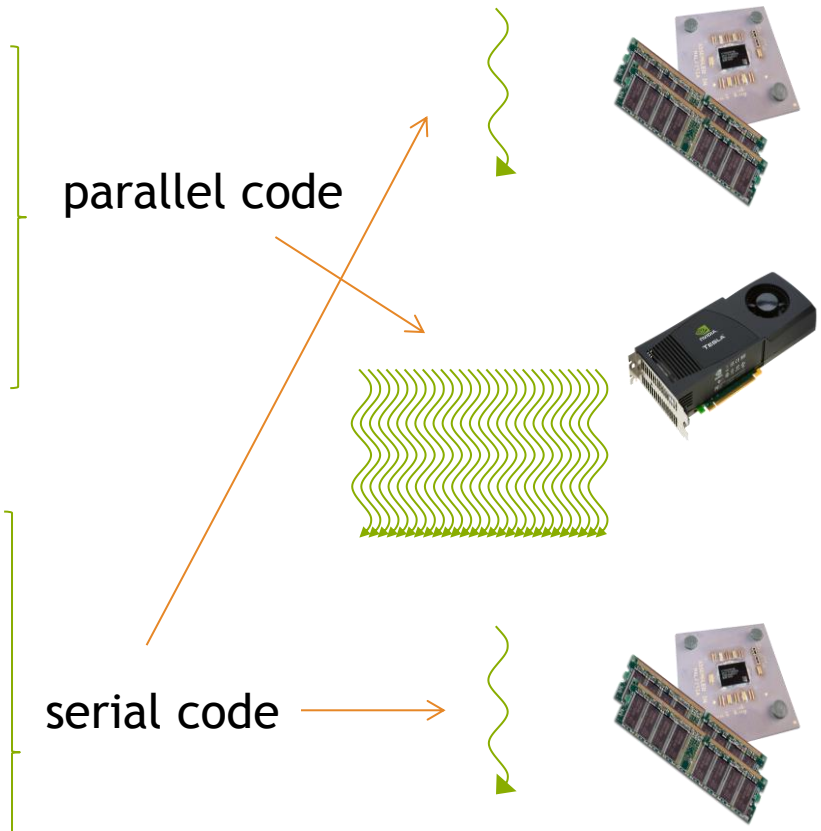


Finite Difference Equation describing system evolution along time:

$$U_{x,t+1} = 2\times(1-L)\times U_{x,t} + L\times U_{x+1,t} + L\times U_{x-1,t} - U_{x,t-1} \qquad L = (k\times C/h)^2$$

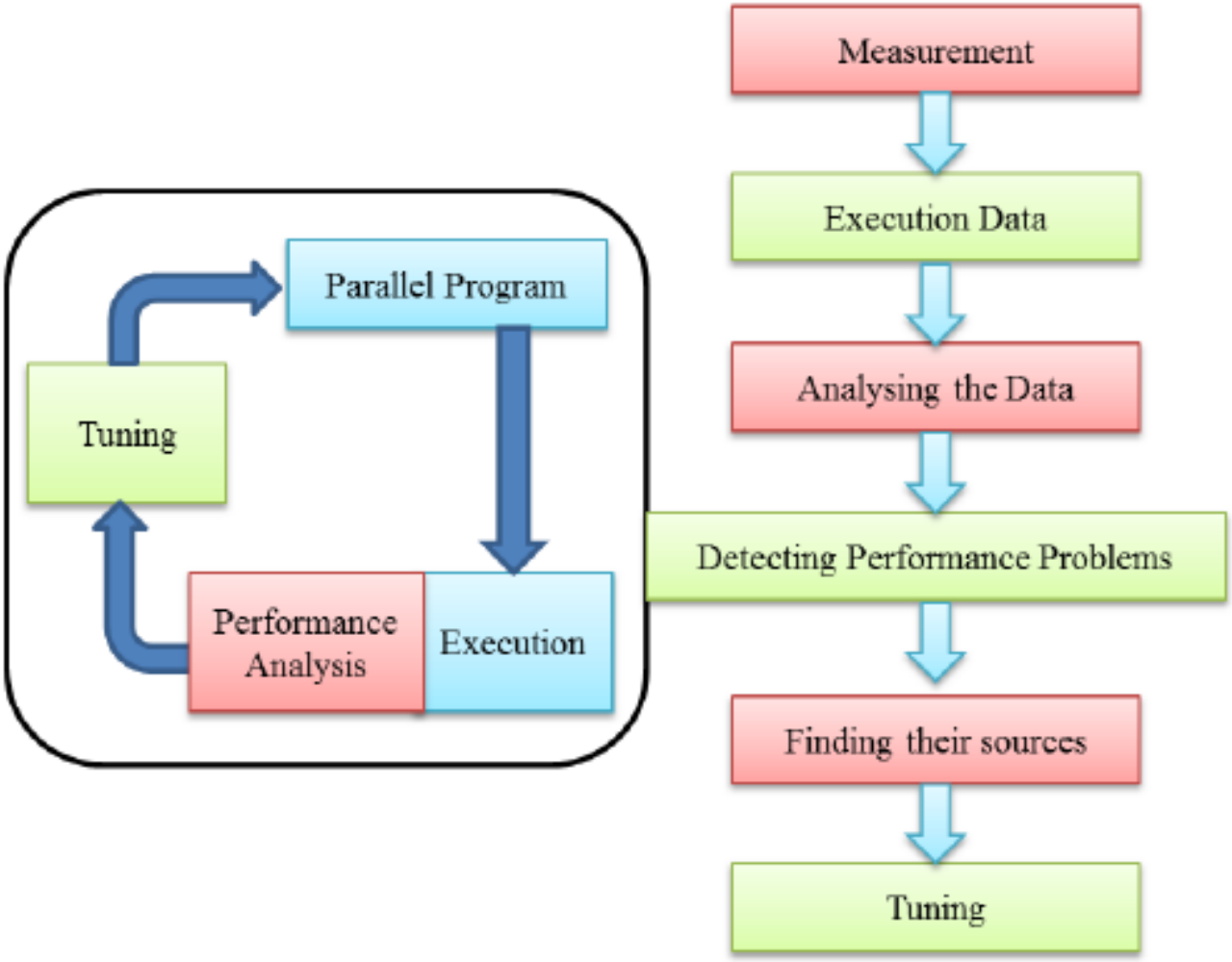
String Simulation in CUDA

```
void strCUDA( const double* U1,
       const double* U2, double* U3,
       double L, double L2, int X )
{
  int x = 1 + threadIdx.x +
          blockDim.x * blockIdx.x;

  if ( x < X )
    U3[x] = L*( U2[x-1] + U2[x+1] ) +
            L2*U2[x] - U1[x];
}

int main()  {
…
 // Alloc space for device copies
  cudaMalloc((void **)&d_U1  size); …
  // Copy to device
  cudaMemcpy(d_U1, U1, size,
             cudaMemcpyHostToDevice); …

  strCUDA<<<32,512>>>( d_U1, d_U2, … );

  // Copy result back to host
  cudaMemcpy(U3, d_U3, size,
             cudaMemcpyDeviceToHost);
  …
}
```



Parallel Programming

- Performance Analysis

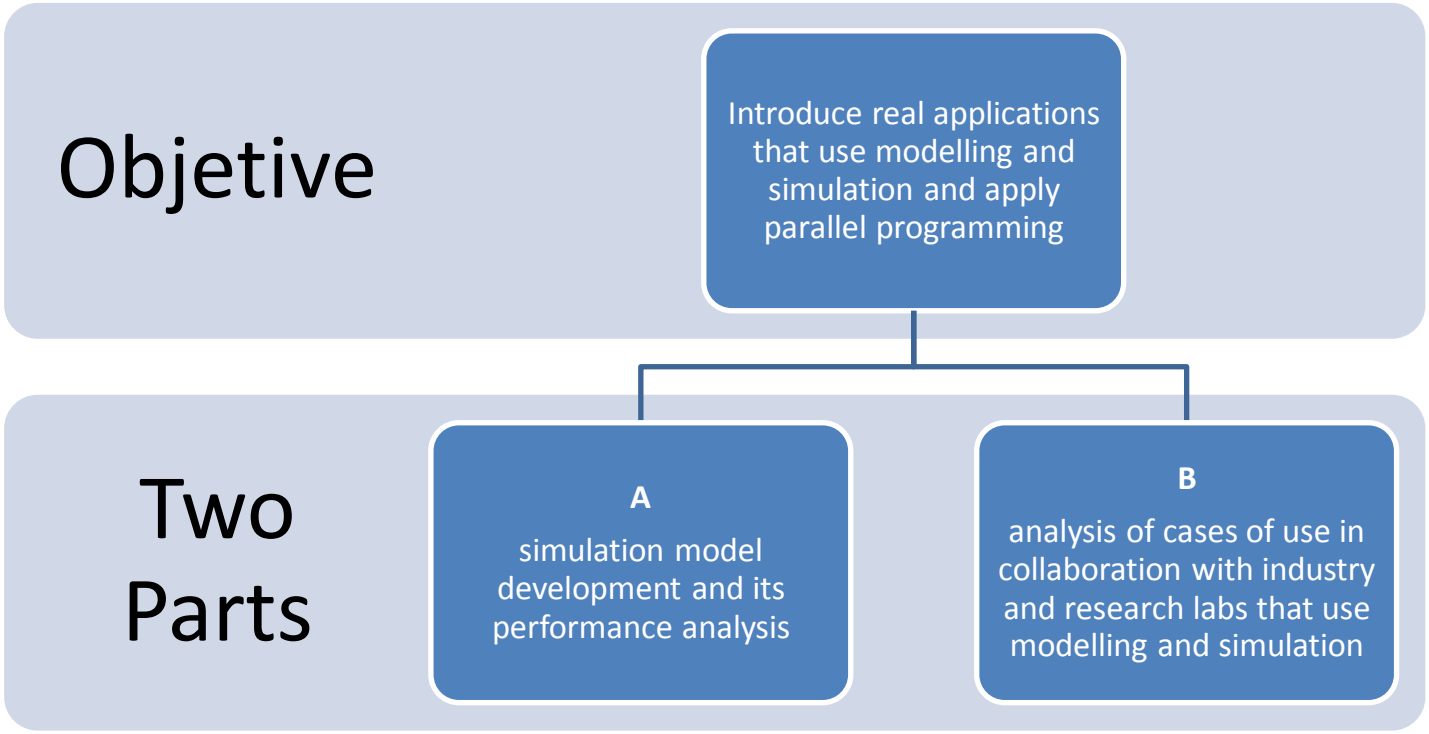


- Performance Analysis

Basic tools:
nvprof, perf command, jumpshot, likwid

Advanced tools:
Measurements – PAPI, Dyninst
Analysis and Visualization – TAU, Scalasca, Paraver
Analysis and tuning – PTF, MATE, Elastic

Applied Modelling and Simulation



Applied Modelling and Simulation

Universitat Autònoma de Barcelona

Part A. Simulation model development and performance analysis

Case study: model of emergency evacuation using Agent Based Modelling
The model includes:

- the environment and the information (doors and exit signals),
- policies and procedures for evacuation,
- social characteristics of individuals that affect the response during the evacuation.

Students receive a partial model that includes management of the evacuation

→

The model also includes individuals who should be evacuated to safe areas.

→

Parameters of the model: individuals, ages, No of people in each area, exits, safe areas, probability of exchanging information.

→

1st work:

use a single-core architecture to carry out a performance analysis.

→

2nd work:

modify the previous model to incorporate new features: overcrowding in exit zones. Carry out a new performance analysis.

Applied Modelling and Simulation

Universitat Autònoma de Barcelona

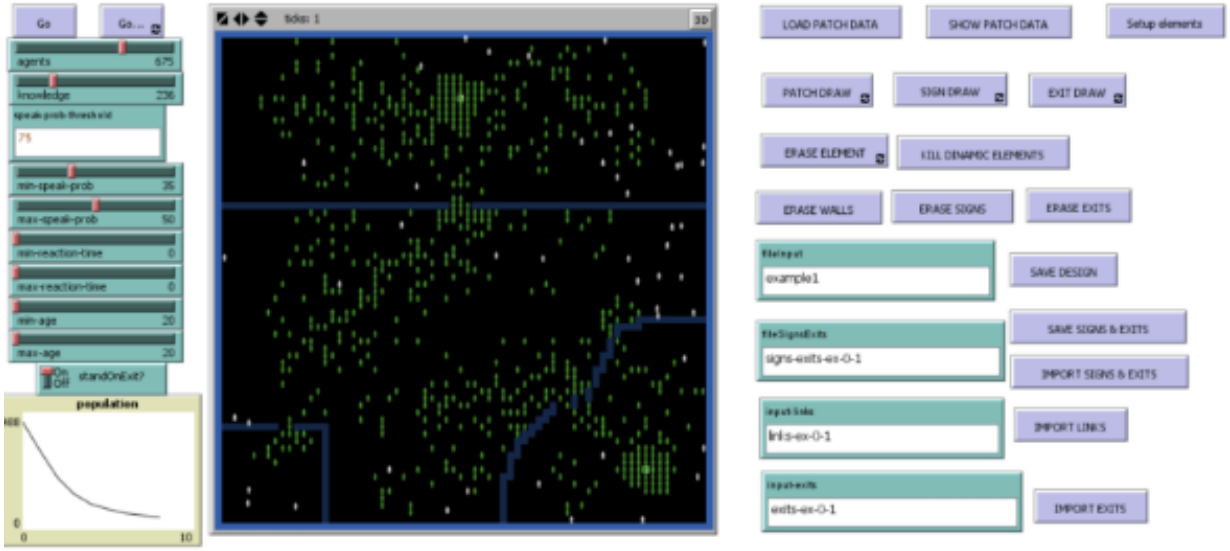
Considering the variability of each individual in the model a stability analysis is required.

Using Chebyshov Theorem the analysis indicates that 720 simulations must be made at least to obtain statistically reliable data.

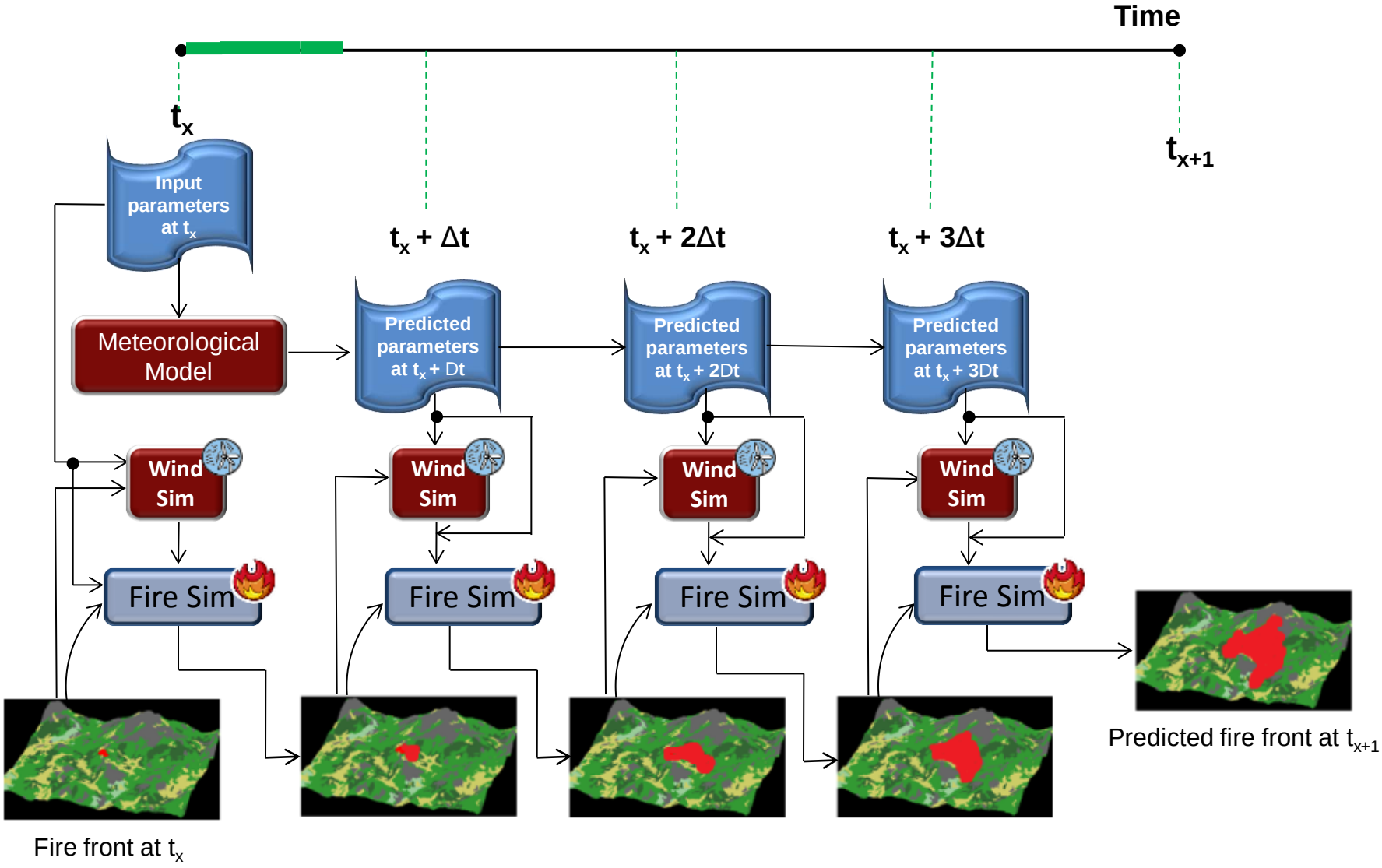
The execution time of the 720 executions on one core processor is 27 hours for 1,500 individuals scenario.

In order to use this tool as DSS, the students are instructed of necessary HPC techniques and the **embarrassingly parallel computing model** is presented to reduce the execution time and the decision-making process time.

Students must learn how to execute multiple parametric Netlogo model runs in a multi-core system and how to make a performance analysis to evaluate the efficiency and scalability of the proposed method.



Applied Modelling and Simulation



MSc: Modelling for Science and Engineering

- Internship at research centres and industries:
  - Barcelona Supercomputing Center
  - Meteocat
  - Climate Science Institute

- Master Thesis

Conclusions

- Students come from different fields
- It is necessary to establish a common basic level
- After one semester the students are able to understand the need and main features of parallel program development
- In the second semester the students develop more complex models and simulators and apply their knowledge