ELIXIRION Scientific School

Hands-on session on container orchestration and distributed computation

Oriol Martínez Acon
Research Engineer @ Barcelona Supercomputing Center

Barcelona

September 17, 2025

A bit about me

- **Oriol Martínez Acon** — Research Engineer at **Barcelona Supercomputing Center (BSC)**
- Currently working in the **Predictable Parallel Computing group**.
- **PhD student** at UPC: scheduling strategies for data-intensive workflows on heterogeneous infrastructures.
- **Master's degree** in Computer Networks and Distributed Systems (UPC).
- **Bachelor's degree** in Computer Engineering (UAB).
- Professor at ENTI-UB in Operating Systems and Distributed systems subjects.
- Actively involved in European research projects: **PROXIMITY**, **EXTRACT**, **VERGE**, **ASCENDER**, **ELIXIRION**, **AIR URBAN**.
- Expertise: task scheduling, parallel computing, container orchestration, monitoring (Prometheus/Grafana).

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Agenda

Parallel Programming in the Compute Continuum

- The **compute continuum** spans from edge devices to clusters, clouds, and HPC systems.
- Modern applications need to exploit parallelism efficiently across this heterogeneous landscape.
- Developers require programming models that **hide complexity** while still enabling scalability and performance.
- Traditional approaches (MPI, OpenMP, Spark) only address parts of this challenge → motivating new models.

Traditional Approaches

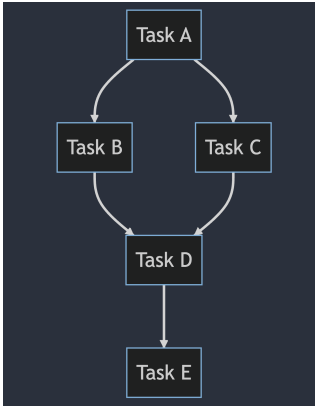
- **MPI (Message Passing Interface)**
  - Explicit send/receive calls between processes.
  - High performance and portable, but requires manual management of communication.
  - Example (C): `MPI_Send(buf,...); MPI_Recv(buf,...);`
- **OpenMP**
  - Directive-based parallelism for shared-memory systems.
  - Easier than MPI, but limited to one node (no distributed support).
  - Example (C): `#pragma omp parallel for`
- **MapReduce / Spark**
  - Functional model: tasks expressed as `map` and `reduce` operations.
  - Simple programming model for big data, but less control over low-level scheduling.
  - Example (Python Spark): `rdd.map(lambda x: x*2).reduce(lambda a,b: a+b)`
- **Limitation:** none of these approaches provide transparent support for heterogeneous, distributed, and elastic environments.
  This motivates the need for task-based programming models.

Need for Task-based Models

- Many scientific and data analytics applications can be expressed as **workflows**.
- Natural representation: a **Directed Acyclic Graph (DAG)** of tasks with dependencies.
- Example: matrix multiplication split into blocks $\rightarrow$ each block multiplication is a task.
- A task-based model can automatically exploit parallelism while respecting dependencies.
- The runtime system dynamically schedules tasks across CPUs, GPUs, and nodes.
- **Benefit:** developers focus on expressing tasks, not on low-level parallelization details.

What is a DAG?

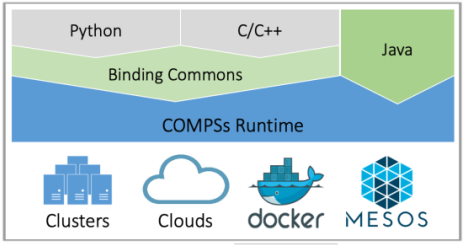
- A **Directed Acyclic Graph (DAG)** is a graph with directed edges and no cycles.
- Widely used to represent workflows in parallel and distributed computing.
- **Nodes**: computational tasks.
- **Edges**: data or control dependencies between tasks.
- Guarantees that execution always progresses forward (no circular dependencies).
- Example: block matrix multiplication, genome sequencing pipeline.



Introducing COMP Superscalar (COMPSs)

- **General-purpose + annotations/hints**: Python, C/C++, Java (*bindings compartidos*).
- **Sequential programming** with no explicit API calls.
- **Task-based**: functions/methods are the *unit of work*.
- **Infrastructure-agnostic**: clusters, clouds, containers (*Kubernetes/Docker*), batch systems.
- **Runtime builds a task graph at runtime**:
  - Dependencies inferred from *in/out/inout* parameters.
  - *Implicit workflow*: programmer writes sequential code.
- **Exploitation of parallelism**:
  - Out-of-order task execution (superscalar paradigm).
  - *Distant parallelism*: tasks can run across heterogeneous nodes.

Takeaway: Write sequential code ⇒ runtime extracts concurrency and distributes tasks.



Layers in practice:
▷ *Language bindings* (Python / C/C++ / Java)
▷ *Binding Commons* (*shared runtime API*)
▷ *COMPSs Runtime* (graph & scheduling)
▷ *Backends*: Clusters / Clouds / Containers

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

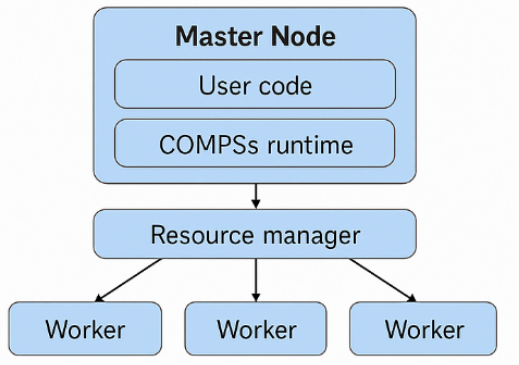
Key Features of COMPSs

- **Superscalar paradigm:** tasks run out-of-order as soon as their inputs are ready → programmer writes sequential code, COMPSs extracts parallelism.
- **Transparent data management:** runtime handles transfers, storage, and persistence → no explicit MPI calls or file copies needed.
- **Dynamic scheduling:** tasks are mapped to available resources across clusters, clouds, or federated infrastructures.
- **Integration:** works with workflow managers (e.g., PyCOMPSs), and container platforms like Docker/Kubernetes.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

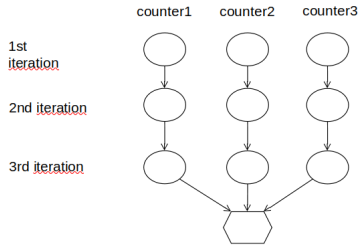
COMPSs Runtime Architecture

- **Master**: runs the user application, builds the dependency graph and coordinates workers.
- **User code** annotated with tasks.
- **COMPSs runtime**: builds dependency graph and schedules tasks.
- **Resource manager**: allocates resources across nodes/clusters.
- **Workers**: execute tasks, report back results.



Programming model example (Java)

```
public class Simple {

    public static void increment(String counterFile) {
        // Parse the content of a file and increment the
value
        …
    }

    public static void main(String[] args) {
        for (i = 0; i < 3; i++) {
            increment(counter1);
            increment(counter2);
            increment(counter3);
        }
        printCounters(counter1, counter2, counter3);
    }
}
```




```
public interface SimpleItf {

    @Method(declaringClass = "es.bsc.Simple")
    void increment(
        @Parameter(type = FILE, direction =
INOUT)
        String counterFile
    );

}
```

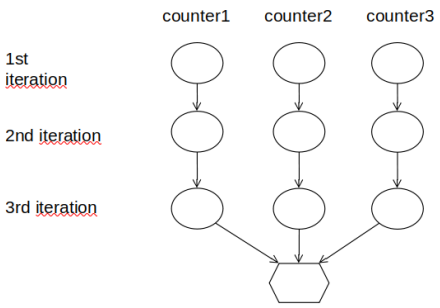
Implementation

Parameter metadata

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Programming model example (Python)

```
class Simple(object):

    @task( counterFile = FILE_INOUT)
    def increment(counterFile) :
        // Parse the content of a file and increment the
value
        …

    def main():
        for (i = 0; i < 3; i++) {
            increment(counter1);
            increment(counter2);
            increment(counter3);
        }
        printCounters(counter1, counter2, counter3);
    }
}
```



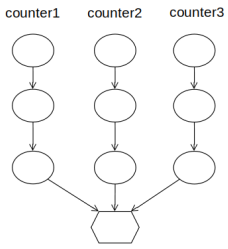
Programming model example (C++)

```
class Simple {

public:
        static void increment(string counterFile) {
        // Parse the content of a file and increment the
value
        …
    }

    void main(int argc, char* argv[]) {
        for (i = 0; i < 3; i++) {
            increment(counter1);
            increment(counter2);
            increment(counter3);
        }
        printCounters(counter1, counter2, counter3);
    }
}
```

1st iteration

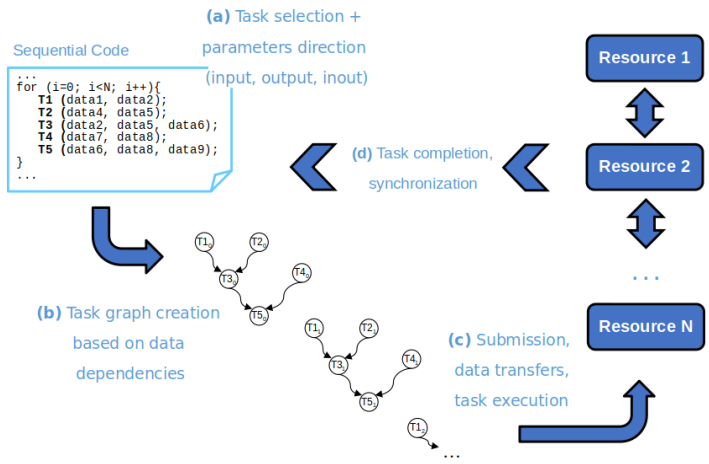
2nd iteration

3rd iteration



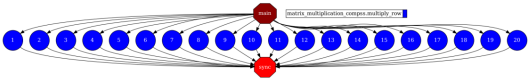

```
class ISimpleItf {

    #pragma omp task inout(counterFile)
    void increment(string counterFile) {
      //computation
    }

}
```

Programming model runtime: summary



Programmability with COMPSs

- Example: row-wise matrix multiplication in PyCOMPSs.
- **Minimal effort**: only add a `@task` decorator.
- Runtime builds the **Task Dependency Graph (TDG)** at execution.
- Developer focuses only on the *application logic*.



Task Dependency Graph (runtime-generated)

```python
@task(returns=1)
def multiply_row(row, B):
    # COMPSs task
    return np.dot(row, B)

def main():
    # Generate two random square matrices A and B
    A = np.random.rand(MATRIX_SIZE, MATRIX_SIZE)
    B = np.random.rand(MATRIX_SIZE, MATRIX_SIZE)

    # Submit one task per row of matrix A
    results = []
    for i in range(MATRIX_SIZE):
        results.append(multiply_row(np.copy(A[i, :]), B))

    # Wait for all parallel tasks to complete and retrieve results
    final_rows = compss_wait_on(results)

    # Stack rows to form the final result matrix
    final_matrix = np.vstack(final_rows)
```

PyCOMPSs annotated code

Manual Master/Worker vs COMPSs (1/2)

```python
def send_task(ip, task_idx, row, B):
    # Create and connect a TCP socket to the worker IP
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((ip, PORT))  # Establish connection to worker

        # Serialize task data (index, row, full matrix B)
        data = pickle.dumps((task_idx, row, B))
        s.sendall(struct.pack('>I', len(data)) + data)  # Send message length + data

        # Receive 4 bytes indicating the length of the response
        raw_len = s.recv(4)
        total_len = struct.unpack('>I', raw_len)[0]

        # Receive the full result data
        result_data = s.recv(total_len)

        return pickle.loads(result_data)  # Deserialize and return result
```

```python
def main():
    A = np.random.rand(20, 20)  # Generate random matrix A
    B = np.random.rand(20, 20)  # Generate random matrix B
    results = [None] * 20       # Initialize list for result rows

    for row_idx in range(20):
        ip = worker_ips[row_idx % num_workers]  # Assign row to worker (round-robin)
        task_idx, result = send_task(ip, row_idx, A[row_idx], B)  # Send task to worker

        if result is not None:
            results[task_idx] = result  # Store received result at correct index

    # Reconstruct the final matrix C = A × B
    final_matrix = np.vstack([
        row if row is not None else np.zeros(20)  # Fill missing rows with zeros (fallback)
        for row in results
    ])
```

Manual Master (TCP Sockets)

```python
def main():
    # Create a TCP/IP socket
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((HOST, PORT))     # Bind to specified host and port
        server_socket.listen()               # Start listening for incoming connections

        while True:
            conn, _ = server_socket.accept()  # Accept an incoming connection
            with conn:
                raw_len = conn.recv(4)  # Receive 4 bytes indicating message length
                total_len = struct.unpack('>I', raw_len)[0]  # Unpack length value

                data = conn.recv(total_len)  # Receive the full message
                task_idx, row, B = pickle.loads(data)  # Deserialize task inputs

                result = np.dot(row, B)  # Perform row × matrix multiplication

                result_data = pickle.dumps((task_idx, result))  # Serialize result
                conn.sendall(struct.pack('>I', len(result_data)) + result_data)  # Send length + result
```

Manual Worker (TCP Sockets)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Manual Master/Worker vs COMPSs (2/2)

- Manual distributed implementation (via TCP sockets):
  - Master handles connections, data serialization, and dispatch.
  - Workers must implement custom servers for receiving and sending results.
  - $\sim$120 lines of code (master + workers).
- COMPSs implementation:
  - $\sim$25 lines of code in a single script.
  - Distribution, communication, and fault tolerance handled by runtime.
- **Result:** $\sim$79% reduction in lines of code, higher maintainability and productivity.

Challenges in Heterogeneous Systems

- Applications today run across **clusters**, **clouds**, and **edge devices**.
- Each environment may differ in:
  - Operating systems and libraries.
  - Hardware architectures (x86, ARM, GPUs, FPGAs).
  - Software dependencies and versions.
- Ensuring that the same code runs correctly and efficiently everywhere is **difficult**.
- This heterogeneity motivates the use of **container technologies**.

Why Containers in Distributed Computing?

- Containers package the **application** + **dependencies** + **environment** in a portable unit.
- Benefits:
  - **Uniform execution**: same image runs anywhere.
  - **Dependency management**: no library conflicts.
  - **Isolation & reproducibility**: reproducible, conflict-free runs.
  - **Elasticity**: easy to deploy and scale dynamically.
- Containers provide a **standard abstraction layer** across heterogeneous infrastructures.

Portability and Interoperability

- COMPSs supports multi-architecture containers:
  - ARM (e.g., Jetson Nano) and x86 nodes.
  - Same code runs without modification.
- Runtime abstracts execution environment:
  - Transparent scheduling across cloud, edge, and HPC.
  - Exploits heterogeneous resources (CPUs, GPUs, accelerators).
- **Takeaway:** write once, run anywhere across the compute continuum.

```
omartinez@workstation:~/Helm-compss-app-verge$ kubectl get nodes -o custom-columns=NAME:.metadata.name,ARCH:.status.nodeInfo.arc
hitecture
NAME          ARCH
nano1         arm64
workstation   amd64
omartinez@workstation:~/Helm-compss-app-verge$ kubectl get pods -o wide
NAME                                    READY   STATUS    RESTARTS   AGE     IP             NODE          NOMINATED NODE   READIN
ESS GATES
compss-app-master-578df89677-mflvm      3/3     Running   0          5m33s   10.244.0.27    nano1         <none>           <none>
compss-app-worker-0-8494c8774f-tz87q    1/1     Running   0          5m33s   10.244.0.28    nano1         <none>           <none>
compss-app-worker-1-6dd5b8bc59-vzq2l    1/1     Running   0          5m33s   10.244.1.219   workstation   <none>           <none>
```

Adapting COMPSs to Container Technologies

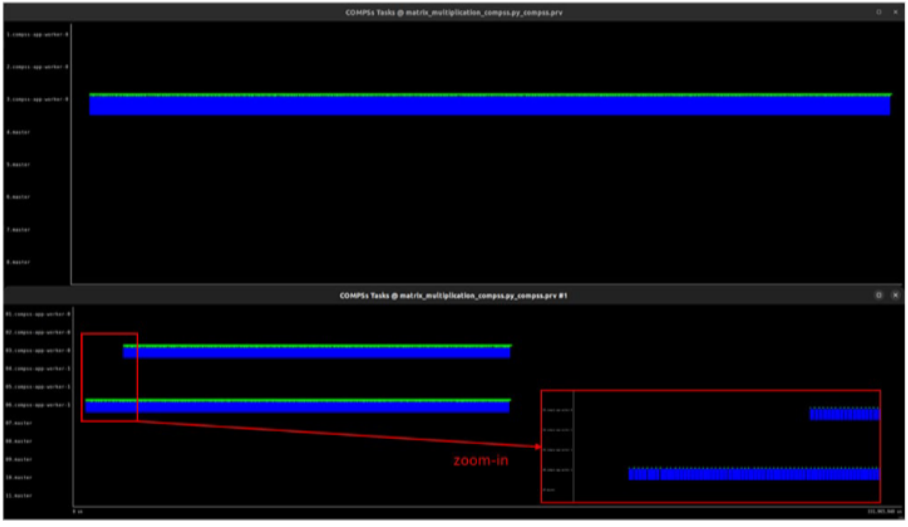
- Modern infrastructures are increasingly **cloud-native**, with containers as the unit of deployment.
- COMPSs' **task-based architecture** maps naturally to containerized workloads:
  - Each task $\rightarrow$ a container with its runtime environment.
  - Dependency graph $\rightarrow$ orchestrated execution across nodes.
- Orchestrators (e.g., **Kubernetes**, **Docker Swarm**) allow COMPSs to:
  - Achieve portability and scalability across the continuum.
  - Exploit elasticity for dynamic task scheduling.
  - Increase resilience in heterogeneous environments.

Performance Gains from Distributed Execution (1/2)

- Comparison: sequential vs distributed execution of matrix multiplication.
- Setup:
  - Sequential: 1 worker, 1 core.
  - Distributed: 2 workers, 1 core each.
  - 1000 parallelizable tasks.
- Results:
  - Sequential: 325.32 seconds.
  - Distributed: 172.99 seconds.
  - **Speedup: 1.88× (46.8% reduction in runtime).**
- **Conclusion:** COMPSs transparently exploits parallelism, reducing execution time and balancing load.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Performance Gains from Distributed Execution (2/2)



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

COMPSs in European Projects

- COMPSs has been applied and validated in several EU-funded projects:
  - **VERGE** – Edge-to-cloud orchestration for smart cities.
  - **EXTRACT** – AI-driven data-mining across the compute continuum.
  - **ASCENDER** – Next-generation programming models for extreme-scale systems.
  - **PROXIMITY** – Data-intensive workflows seamlessly deployed across edge and cloud.
  - **ELIXIRION** – Scientific school and training on advanced distributed computing.
  - **AIR URBAN** – Smart urban mobility and environmental monitoring.
- These projects showcase COMPSs' flexibility, scalability, and relevance in heterogeneous environments.

Hands-on overview

- Now that we've seen the concepts, let's deploy COMPSs in practice.
- Use a local Kubernetes cluster (`minikube`, already running).
- Recap Helm basics to prepare for deployment.
- Deploy a COMPSs demo chart (Master + Workers).
- Validate the deployment: ensure tasks run and the application executes correctly.

Goal: run a COMPSs application on Kubernetes using Helm.

Check Minikube Resources

- Before running heavy workloads, check how many CPUs and how much memory your Minikube node has.

```
# From Kubernetes view
kubectl describe node minikube | grep -i allocatable -A5

# From inside the Minikube VM
minikube ssh -- nproc          # number of CPU cores
minikube ssh -- free -h        # available memory
```

- Compare this with the requirements of COMPSs and monitoring tools.

Adjust Minikube Resources

- If resources are insufficient, update Minikube's configuration and restart it.

```
# Stop the current cluster
minikube stop

# Set CPUs (example: 4 cores)
minikube config set cpus 4

# (Optional) set memory (example: 8 GB)
minikube config set memory 8192

# Restart with new config
minikube start
```

- This ensures COMPSs apps and monitoring tools have enough resources.

Get the demo code

- In order to follow the hands-on session and launch COMPSs, we will first clone the repository containing the main sources.
- Step 1: Clone the repository (preferred way):

```
git clone https://github.com/oriolmartinezac/elixirion-school.git
cd elixirion-school
```

- Step 2 (alternative): if `git` is not available, download the archive with `wget` or from the browser.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Helm recap (quick)

- Before using Helm to deploy COMPSs, it is important to understand the basic structure it works with:
- **Helm** is the package manager for Kubernetes.
- It manages **Charts**, which are deployable packages (templates + default values).
- Each chart can be customized with a **values.yaml** file containing your configuration.
- Finally, with `helm install` or `helm upgrade`, the templates and values are rendered and applied to the cluster.

Configure the app (values.yaml) — Volume

- **What it is**: local volume for the COMPSs *master* pod (hostPath on the Minikube node).
- **Why it matters**: it is **highly recommended** to configure a volume in order to access the outputs and results of the COMPSs application execution inside the Kubernetes cluster.
- **Minikube mount**: expose a host directory into the VM, then reference it in `values.yaml`.

```
# 1) Mount volume. Left side (/home/$USER/data) → The path on your host machine (your
Right side (/home/minikube) → The path inside the Minikube VM.

minikube mount /home/$USER/data:/home/minikube

# 2) In values.yaml, set the localPath (and optionally the node name)
compss:
  master:
    volume:
      localPath: /home/minikube            # path mounted above
      node: minikube       # default node name in Minikube
```

- Without a volume, results stay inside the container filesystem. With you can directly access them.

Configure the app (values.yaml) — Resources

- **worker.number**: how many worker pods to launch in the K8s cluster.
- **worker.resources.cpu**: total CPUs *per worker* to be used by the runtime.
- **worker.resources.memory**: memory *per worker* in GiB.
- **Important**: do not exceed your Minikube VM capacity. As a rule of thumb: `number` $\times$ `cpu` $\leq$ `minikube -cpus`, and `number` $\times$ `memory` $\leq$ VM memory (leave some headroom for master/overhead).

```
compss:
  worker:
    number: 2                 # total worker pods
    labels:
      node: worker
    resources:
      cpu: 1                  # CPUs per worker
      memory: 2               # GiB per worker
```

- Example: with `minikube start -cpus=4 -memory=8192`, a safe setup is `number: 2`, `cpu: 1`, `memory: 2` (GiB).

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Configure the app (values.yaml) — Context

- **Context (path + entry file)**
  - With the default image `oriolmac/compss-elixirion:latest`, the app is expected at: `folderPath: /home/omartinez/elixiron-school/apps` and `file: app_name.py` (e.g., `matmul.py`, `cholesky.py`, `gen.py`).
  - If you built a *custom image*, set: `folderPath` to the path *inside the container* where your sources were copied (usually the same path you used in `-context-dir`), and `file` to your app's entry script (e.g., `main.py`).

```
app:
  context:
    folderPath: /home/omartinez/elixiron-school/apps   # change if custom image
    file: app.py                                       # entry script
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Configure the app (values.yaml) — Parameters

- **Parameters (`app.params`)**
  - Uncomment and set only the parameters your application needs.
  - These values are passed to the COMPSs app at runtime (treat them as strings unless specified).
  - Examples provided (Matmul, Genetic Algorithm, Cholesky) are templates—adapt to your app.

```
app:
  params:
    # num_blocks: "3"
    # elems_per_block: "1024"
    # number_iterations: "1"
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Deploy with Helm

```
# Move into the cloned Helm folder
cd helm-compss

# (Optional) Create a namespace if you want to isolate the deployment
kubectl create namespace compss

# Deploy the COMPSs application (default namespace)
helm install compss-app .

# Or deploy into a specific namespace (e.g. compss)
helm install compss-app . -n compss
```

- Run `helm install` inside the `helm-compss` folder to launch the deployment.
- You can isolate deployments in a namespace (recommended for multiple apps).

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Verify the deployment

```
# Check that everything has been deployed correctly
kubectl get pods -n compss

# Debug if something is wrong
kubectl describe pod <pod-name> -n compss
kubectl logs <pod-name> -n compss
```

- Use `kubectl get pods` to verify that all pods are `Running` or `Completed`.
- If some pods stay `Pending` or go into `Error`, use `kubectl describe` or `kubectl logs` to debug.

```
NAME                                READY   STATUS    RESTARTS   AGE
compss-master-6f58fbf85d-mgksx      3/3     Running   0          20s
compss-worker-0-5646fdb6b6-w74lk    1/1     Running   0          20s
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Results of the execution

- Because the volume is enabled, results are stored on the host and visible after execution.
- Outputs are generated in **debug mode**, so expect many logs and files.
- Main folders and files:
  - **jobs**: tasks launched inside the container (note: one job is not always equal to one task).
  - **monitor**: contains the application graph (DAG) to visualize dependencies.
  - **trace**: execution trace viewable with `wxparaver`.
  - **workers**: logs and outputs from worker containers.
  - **runtime.log**: main output from the COMPSs master.
  - Other logs (`pycompss.log`, `resources.log`, etc.) give detailed execution info.

Generate the execution graph

- To obtain a **complete graph view** of the application execution, process the monitor output with COMPSs tools inside a Docker container.
- Steps:
  1. Go to the `monitor` folder generated in the results.
  2. Run a Docker container mounting the current directory as a volume.
  3. Inside the container, move into `/framework` and run `compss_gengraph`.

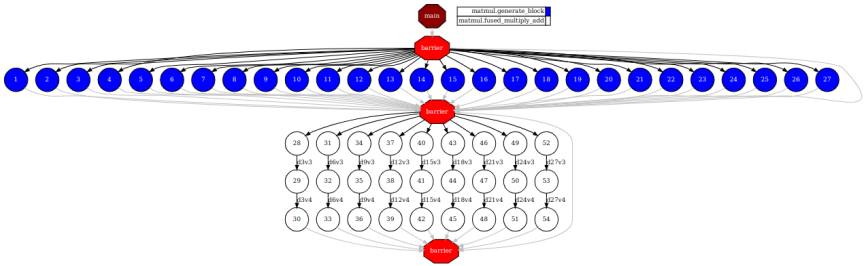
```
cd results/matmul.py_01/monitor

docker run -v $(pwd):/framework -it oriolmac/compss-elixirion \
    bash -c "compss_gengraph /framework/complete_graph.dot"
```

- This generates `complete_graph.dot`, which you can visualize with tools such as `xdot` or Graphviz.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Example: Complete Graph (Matmul)

- After running `compss_gengraph`, the file `complete_graph.dot` can be visualized with tools like `xdot` or Graphviz.
- Below is the resulting DAG for the **Matmul** application.



Why traces and wxParaver?

- Besides the DAG graph, COMPSs produces an **execution trace** (stored in the `trace/` folder).
- The trace provides a detailed timeline of the application run:
  - Tasks executed by each worker.
  - CPU and memory usage.
  - Synchronization and dependencies.
- To explore this trace, we use **wxParaver**, a tool that allows:
  - Visualizing task scheduling and runtime behavior.
  - Detecting bottlenecks and inefficiencies.
  - Understanding how the application scales across workers.
- This makes **wxParaver essential** for debugging and performance analysis of COMPSs applications on Kubernetes.

Install wxParaver (trace viewer)

- wxParaver is available for **Windows, Linux, and macOS**.
- Download it directly from: `https://tools.bsc.es/downloads`
- If you are not on Linux, simply download the package for your OS and follow the installer instructions.
- Example for Linux: `wxparaver-4.12.0-Linux_x86_64.tar.bz2`

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Install wxParaver (Linux example)

```
# Download
cd /tmp
wget https://ftp.tools.bsc.es/wxparaver/wxparaver-4.12.0-Linux_x86_64.tar.bz2

# Install system-wide (requires sudo)
sudo mkdir -p /usr/bin/wxparaver
sudo tar -xjf wxparaver-4.12.0-Linux_x86_64.tar.bz2 \
    -C /usr/bin/wxparaver --strip-components=1

# Add to PATH (session)
export PATH=/usr/bin/wxparaver/bin:$PATH

# Verify
wxparaver --version
```

- Folder layout: `/usr/bin/wxparaver/{bin,cfgs,include,lib64,share}`.
- Open COMPSs traces from the `trace/` folder (files: `*.prv`, `*.pcf`, `*.row`).

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Open a trace with wxParaver

- After execution, traces are stored in the `trace/` folder of your results.
- To visualize them, run `wxparaver` pointing to the `.prv` file.
- Example (Matmul application):

```
cd results/matmul.py_01/trace
wxparaver matmul.py_compss.prv
```

- This will open wxParaver with the execution timeline.
- You can then load configurations (`*.pcf`, `*.cfg`) to analyze parallelism, resource usage, and dependencies.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Analyze a trace with wxParaver

- Once the trace is open in wxParaver:
  - Go to `File → Load Configuration`.
  - Navigate to the `paraver_configs/` folder from the cloned repository.
- Recommended configurations:
  - `compss_tasks.cfg` → view tasks across workers.
  - `compss_tasks_id.cfg` → view tasks with their identifiers.
  - `compss_runtime.cfg` → inspect runtime internals (only if needed).
- These visualizations help to:
  - Understand task scheduling and execution order.
  - Detect idle times, bottlenecks, and resource usage.

Advanced trace analysis in wxParaver

- wxParaver allows advanced exploration of traces:
  - **Multiple views**: open several windows (e.g., tasks timeline, runtime info) at the same time.
  - **Synchronization**: align all views so they show the same time unit, making it easier to correlate events.
  - **Chopping**: select and cut a region of the trace to analyze only a subset of the execution.
  - **Histograms**: generate statistics (e.g., time spent in specific tasks, communication vs. computation, idle times).
- This enables:
  - Comparing task execution with runtime events in parallel.
  - Zooming into specific moments of interest (e.g., bottlenecks).
  - Quantifying performance through metrics (task duration, load balance).

Prepare the pod builder

- One of the strengths of containers is that we can **layer** applications on top of existing images.
- In our case: we will take a COMPSs-enabled base image and extend it with our own applications.
- Before doing so inside Kubernetes, it is essential to prepare the **pod builder**.
- Requirements:
  - **Minikube** must be running.
  - From the project `Makefile`, run:
    - `make prepare`
- This sets up the pod builder in your Kubernetes cluster, which will later be used to build custom COMPSs application images.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Build a COMPSs image

- With the pod builder ready, we can now create a **new container layer**.
- This new image:
  - Starts from a base image with COMPSs already installed.
  - Adds our application code and dependencies.
  - Can be reused to test different applications on top of COMPSs.
- **Multi-arch support**: the build system already supports generating images for multiple architectures (ARM and AMD).
- For simplicity in this hands-on session, we will only build the image for our current architecture (already set by default).
- The **base image** (`oriolmac/compss-elixirion`) already provides both architectures.

Build a COMPSs image — Command

- The script `compss_docker_gen_image` automates the process:
  - **–image-base**: base image with COMPSs (always make sure is `oriolmac/compss-elixirion`).
  - **–image-name**: name for your new image (dockerhub-username/docker-image).
  - **–context-dir**: path with your application sources.
  - **–python-packages**: path to requirements file with Python dependencies.
  - **–push**: optionally push to a Docker registry.

Example command with my docker hub repository:

```
./compss_docker_gen_image \
  --image-base="oriolmac/compss-elixirion:latest" \
  --image-name="oriolmac/compss-elixirion-custom:latest" \
  --context-dir="/home/omartinez/elixiron-school/apps" \
  --python-packages="/home/omartinez/elixiron-school/apps/requirements.txt" \
  --push
```

Why Monitoring Matters

- Traces (via wxParaver) give a detailed view of **what happened** in a single execution: task timeline, dependencies, bottlenecks.
- But to understand **how the system behaves continuously**, we also need metrics from tasks, workers, and runtime.
- This requires:
  - Collecting metrics across executions and runs.
  - Detecting bottlenecks, failures, or resource saturation as they happen.
  - Visualizing system performance at different levels (task, worker, cluster).
- Therefore, after exploring traces, we will complement them with **monitoring tools** for a higher-level and ongoing view.

Introduction to Prometheus

- Open-source monitoring and alerting toolkit, widely used in cloud-native systems.
- Designed for reliability and scalability.
- Key features:
  - Time-series database: stores metrics over time.
  - Pull-based model: scrapes metrics from exporters (apps, schedulers, containers).
  - Query language (PromQL) for flexible analysis.
- Perfect match for containerized and distributed environments.

Introduction to Grafana

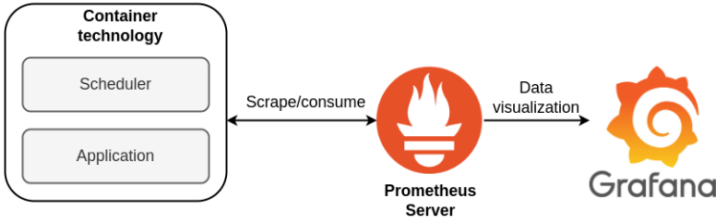
- Open-source platform for data visualization and analytics.
- Connects with Prometheus and many other data backends.
- Key features:
  - Customizable dashboards with charts, alerts, and panels.
  - Real-time visualization of application and system metrics.
  - Helps operators and developers detect anomalies quickly.
- Complements Prometheus by turning raw metrics into actionable insights.

Why Prometheus for COMPSs?

- Prometheus follows a **distributed scrape model**: periodically pulls metrics from HTTP endpoints.
- A natural fit with COMPSs:
  - **Master exporter**: runtime state, task queues, scheduling latencies.
  - **Worker exporters**: per-task metrics (duration, I/O, errors), CPU and memory usage.
- **Lightweight**: minimal overhead, no central brokers needed.
- **Scalable and federated**: multiple Prometheus instances across zones/clusters.
- Together with Grafana, it completes the monitoring stack.

Monitoring the COMPSs Master & Workers

- The **Master exporter** exposes runtime and scheduler metrics.
- Each **Worker exporter** publishes task execution and resource metrics.
- **Prometheus** scrapes all endpoints, **Grafana** visualizes and triggers alerts.
- *Note*: for very short-lived tasks a **Pushgateway** can be used, but scrape-based monitoring is preferred.



Infrastructure Awareness with Prometheus

- Beyond COMPSs metrics, Prometheus can monitor the **underlying infrastructure**.
- Examples of infrastructure metrics:
  - Node CPU and memory utilization.
  - Network latency and bandwidth.
  - GPU usage or accelerator availability.
  - Cluster load and failures.
- Provides a single source of truth for both **application** and **system** state.

COMPSs Scheduler + Prometheus

- The COMPSs scheduler can query Prometheus to obtain the **current state of the cluster**.
- Enables **adaptive scheduling**:
  - If nodes are overloaded → assign fewer tasks.
  - If resources are idle → assign more tasks.
  - Prefer nodes with high data locality or GPU availability.
- Benefits:
  - Better resource utilization.
  - Faster workflow execution.
  - Improved reliability and efficiency across the compute continuum.
- **Takeaway:** Prometheus is not only for monitoring, but also for *driving smarter scheduling decisions*.

Prometheus vs Grafana: Why Both?

Prometheus UI

- Focused on querying and inspecting raw metrics.
- Useful for debugging with PromQL.
- Limited visualization.

Grafana Dashboard

- Rich, customizable dashboards.
- Real-time monitoring with alerts.
- Shareable views for operators and developers.

Same data source, two different purposes.

Deploy Prometheus + Grafana

- To deploy monitoring tools we will use the **Helm chart** provided by the community (`kube-prometheus-stack`).
- Reminder: before launching, make sure Minikube allows enough inotify watches:

```
minikube ssh -- "sudo sysctl -w fs.inotify.max_user_watches=524288 \
                 fs.inotify.max_user_instances=8192"
```

- Then add the Helm repo and deploy Prometheus + Grafana:

```
helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts

helm repo update

helm install prometheus prometheus-community/kube-prometheus-stack \
  -n monitoring --create-namespace
```

- This will install Prometheus, Grafana, and related exporters in the `monitoring` namespace.

Verify Prometheus + Grafana deployment

- After installing with Helm, check the health of the monitoring components.
- As with COMPSs pods, use `kubectl get pods`, but this time in the `monitoring` namespace:

```
kubectl get pods -n monitoring
```

- All pods should be in `Running` or `Completed` state.
- If any pod is stuck in `Pending` or `Error`:
  - Inspect with: `kubectl describe pod <pod-name> -n monitoring`
  - Check logs with: `kubectl logs <pod-name> -n monitoring`
- Also check the exposed **services** to ensure Prometheus and Grafana are accessible:

```
kubectl get svc -n monitoring
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Monitoring components: Pods and Services

- After deployment, check that:
  - All **pods** are running correctly.
  - The required **services** (Prometheus, Grafana, Alertmanager) are exposed.

Pods status

```
omartinez@bsc-dell:~/Escritorio/BSC/elixirion/elixirion-school$ kubectl get pods -n monitoring
NAME                                                     READY   STATUS    RESTARTS      AGE
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   4 (9h ago)    29h
prometheus-grafana-75fb5cd659-577p8                      3/3     Running   6 (9h ago)    28h
prometheus-kube-prometheus-operator-78649d5c7d-gkfnm     1/1     Running   4 (147m ago)  29h
prometheus-kube-state-metrics-d687c76d7-926l2            1/1     Running   4 (147m ago)  29h
prometheus-prometheus-kube-prometheus-prometheus-0       2/2     Running   4 (9h ago)    29h
prometheus-prometheus-node-exporter-s6vnk                1/1     Running   2 (9h ago)    29h
```

Services exposed

```
NAME                                      TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
    AGE
alertmanager-operated                     ClusterIP   None            <none>        9093/TCP,9094/TCP,9094/UD
P   29h
prometheus-grafana                        ClusterIP   10.105.48.156   <none>        80/TCP
    29h
prometheus-kube-prometheus-alertmanager   ClusterIP   10.96.159.30    <none>        9093/TCP,8080/TCP
    29h
prometheus-kube-prometheus-operator       ClusterIP   10.105.123.109  <none>        443/TCP
    29h
prometheus-kube-prometheus-prometheus     ClusterIP   10.99.239.99    <none>        9090/TCP,8080/TCP
    29h
prometheus-kube-state-metrics             ClusterIP   10.101.137.45   <none>        8080/TCP
    29h
prometheus-operated                       ClusterIP   None            <none>        9090/TCP
    29h
prometheus-prometheus-node-exporter       ClusterIP   10.98.220.54    <none>        9100/TCP
    29h
```

Access Prometheus and Grafana UIs

- The monitoring stack runs inside the cluster. To open the web interfaces, use **port forwarding** from your local machine.
- Services of interest:
  - **prometheus-kube-prometheus-prometheus** → Prometheus UI (port 9090).
  - **prometheus-grafana** → Grafana UI (port 80).

```
# Forward Prometheus to localhost:9090
kubectl -n monitoring port-forward \
  svc/prometheus-kube-prometheus-prometheus 9090:9090

# Forward Grafana to localhost:3000
kubectl -n monitoring port-forward \
  svc/prometheus-grafana 3000:80
```

- Prometheus UI → http://localhost:9090
- Grafana UI → http://localhost:3000

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Login to Grafana

- Once Grafana is accessible at `http://localhost:3000`, you need to log in.
- Default credentials:
  - **User**: `admin`
  - **Password**: prom-operator (by default, stored in a Kubernetes secret).

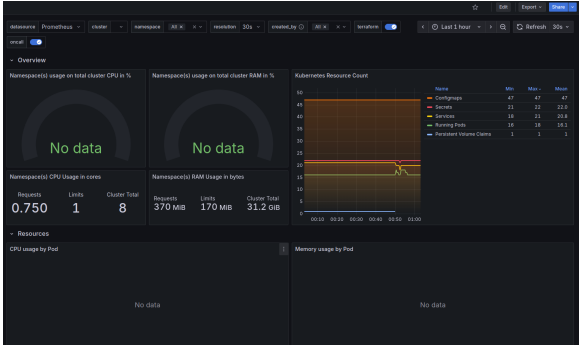
```
# Retrieve Grafana admin password
kubectl -n monitoring get secret prometheus-grafana \
  -o jsonpath="{.data.admin-password}" | base64 -d; echo
```

Dashboards in Grafana

- A **dashboard** is a set of panels and visualizations that present metrics in a clear, interactive way.
- Grafana allows importing dashboards shared by the community (ready-to-use templates for common systems).
- To import:
  1. Go to `Dashboards → Import` in Grafana.
  2. Enter the dashboard ID from `https://grafana.com/grafana/dashboards`.
  3. Example: **15758** (Kubernetes cluster monitoring).
- Once imported, panels will start displaying data collected by Prometheus.

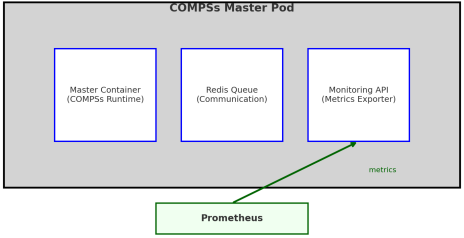
Example Imported Dashboard

- After importing the community dashboard (ID: **15758**), Grafana will display a set of prebuilt panels.
- These panels typically include:
  - CPU and memory usage of nodes and pods.
  - Cluster resource allocation and availability.
  - Pod status and health.



Inside the Master Pod

- The **COMPSs master pod** runs multiple containers, each with a specific role:
  - **Master container** → runs the COMPSs runtime and coordinates execution.
  - **Redis Queue** → lightweight message broker for communication between master and workers.
  - **Monitoring API** → exposes runtime metrics to Prometheus (scraped automatically via the `ServiceMonitor`).
- This design keeps the runtime, communication, and monitoring components isolated, but co-located in the same pod.



Why Multiple Containers in a Pod?

- In Kubernetes, a pod can include **multiple containers** that share the same lifecycle and network.
- For COMPSs, this allows us to **separate concerns**:
  - Runtime logic (COMPSs master).
  - Communication layer (Redis Queue).
  - Observability (Monitoring API).
- This layered approach provides:
  - **Flexibility**: containers can be updated or replaced independently.
  - **Isolation**: monitoring does not interfere with runtime or communication.
  - **Extensibility**: easy to add new services (e.g., logging, sidecars).
- In short: multiple containers in one pod let us **delimit the behavior** of COMPSs while keeping everything co-located.

Test Deployment: Matmul Application

- As a test case, deploy the **Matmul application** with Helm:

```
# Deploy a new instance (e.g. matmul)
helm install compss-app .
```

- After deployment, check that the **ServiceMonitor** (needed for Prometheus scraping) is created:

```
kubectl get servicemonitor -n monitoring
```

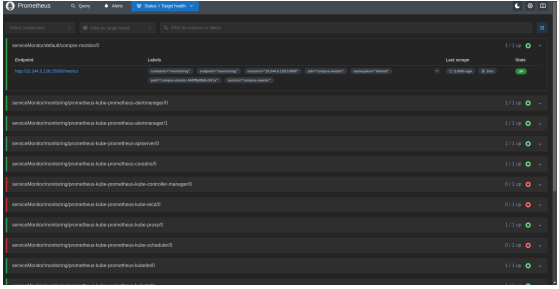
- Finally, confirm that the Prometheus server detects the new ServiceMonitor:

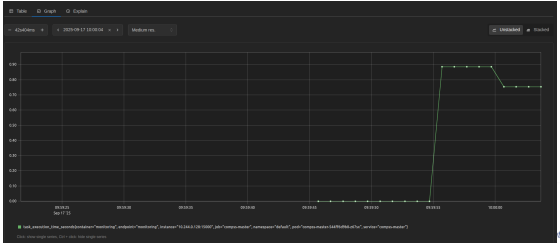
```
# Port-forward Prometheus (if not already done)
kubectl -n monitoring port-forward \
  svc/prometheus-kube-prometheus-prometheus 9090:9090

# Open in browser: http://localhost:9090
# Check under: Status → Service Discovery
```

- If everything is correct, Prometheus will show the COMPSs endpoints being scraped.

Monitoring Results

- If everything is correctly deployed and configured:
  - Prometheus should detect the **ServiceMonitor** for COMPSs and list it under `Status → Targets`.
  - Prometheus should display metrics such as **task execution time** and other runtime values from the COMPSs monitoring API.
- These confirm that the COMPSs application is being monitored in real-time. IMPORTANT: the cholesky app is not prepared to report task execution times like genetic and matmul applications.





References

- **COMPSs Programming Model:** `https://compss.bsc.es/`
- **PyCOMPSs Documentation:** `https://pycompss.readthedocs.io/`
- **Kubernetes Documentation:** `https://kubernetes.io/docs/`
- **Helm Documentation:** `https://helm.sh/docs/`
- **Minikube Documentation:** `https://minikube.sigs.k8s.io/docs/`
- **Docker Documentation:** `https://docs.docker.com/`
- **Prometheus Documentation:** `https://prometheus.io/docs/introduction/overview/`
- **Grafana Documentation:** `https://grafana.com/docs/`
- **Grafana Dashboards (Community):** `https://grafana.com/grafana/dashboards`
- **wxParaver Downloads:** `https://tools.bsc.es/downloads`