Scalable Execution Framework for R on Manycore Systems

Xiran Zhang[1], Javier Conejero[2], Sameh Abdulah[1], Jorge Ejarque[3], Ying Sun[1], Rosa M. Badia[2], David E. Keyes[1] and Marc G. Genton[1]

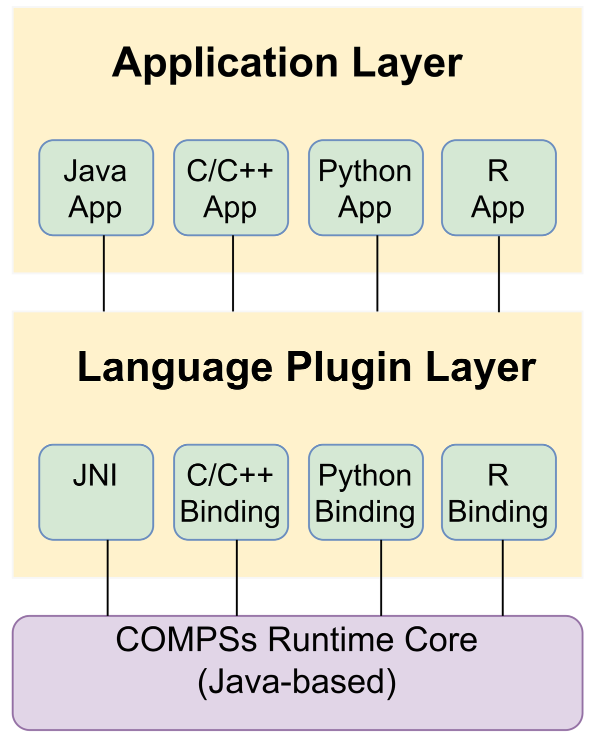
[1]Computer, Electrical and Mathematical Sciences and Engineering Division (CEMSE), King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia

[2]Barcelona Supercomputing Center, Barcelona, Spain

[3]Seqera Labs, Barcelona, Spain

https://github.com/stsds/RCOMPSs

Introduction

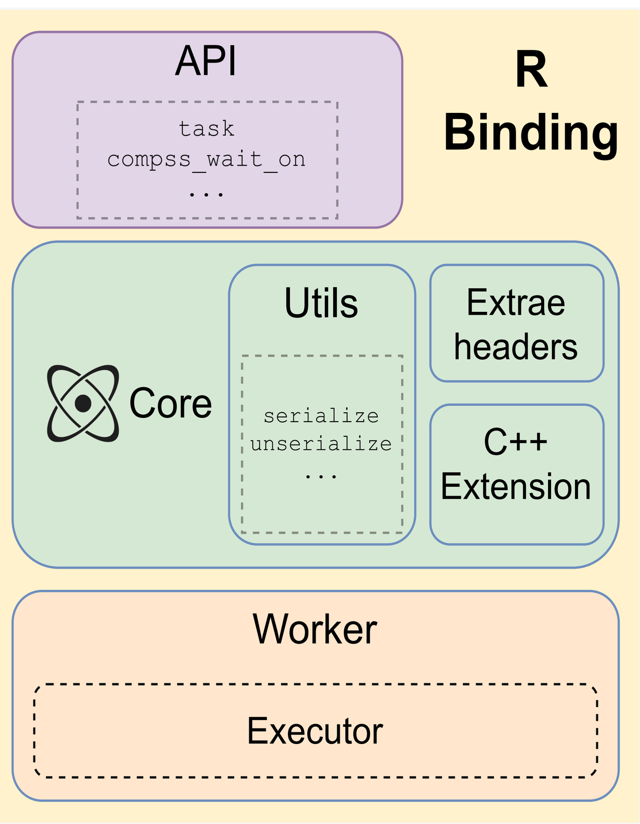


- **Task-based parallelism** is a computing paradigm in which work is decomposed into discrete, self-contained tasks that can be concurrently executed. This model abstracts parallel execution by focusing on task dependencies rather than low-level thread management
- **COMPSs** (COMP Superscalar) is a Java-based programming model and runtime system that enables developers to write simple sequential code while automatically handling parallelization, scheduling, and task distribution
- **Single memory and storage space:** The memory and file system space is also abstracted by COMPSs, giving the illusion that a single memory space and single file system is available. The runtime takes care of all the necessary data transfers
- **Advantages:** Task-based programming model, data dependencies management, dynamic task scheduling, resource agnostic, and extended to support ***R***
- **No/Minimal APIs:** In COMPSs, the model does not require any special API calls, pragmas, or constructs within the application. However, for the R bindings, a small set of API calls must be used

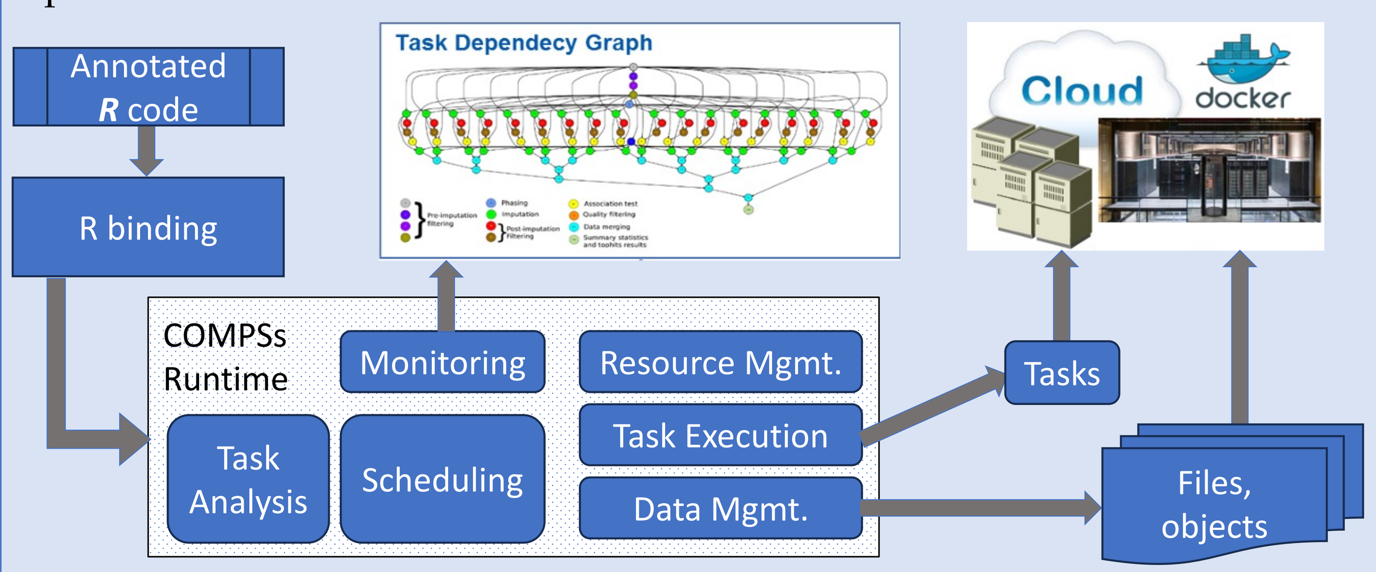
RCOMPSs library

Designed for building **task-based parallelism** workflows in ***R***:

- RCOMPSs Leverages the Java-based COMPSs runtime system
- RCOMPSs allows retaining conventional sequential workflow development while integrating task-based parallelism
- Users annotate code with parallel tasks, which are distributed and executed by the COMPSs runtime system
- RMVL, a fast R-based serialization library, reduces I/O bottlenecks, optimizes the use of computing resources, and scales effectively in distributed runtime systems, making it essential for high-performance task execution



The architecture of RCOMPSs



Example 1: Standardization of a dataset

To demonstrate the simplicity of converting a normal ***R*** code into RCOMPSs style, we now give a simple example of standardizing *X* in parallel

First, we need to define the tasks in a separate file, say `task.R`

```
get_mean <- function(x){
  mean(x)
}
get_sd <- function(x){
  sd(x)
}
standardize <- function(x, mu, sigma){
  (x - mu) / sigma
}
```

These functions are going to be defined as tasks

Then, we write the main file that defines the workflow with the dependencies automatically detected and arranged by RCOMPSs:

```
library(RCOMPSs)
source("task.R")
compss_start()
get_mean.dec <- task(get_mean, "task.R",
                     info_only = F, return_value = T)
get_sd.dec <- task(get_sd, "task.R",
                   info_only = F, return_value = T)
standardize.dec <- task(standardize, "task.R",
                        info_only = F, return_value = T)
# Task (1)
mu <- get_mean.dec(X)
# Task (2)
sigma <- get_sd.dec(X)
# Task (3)
X_standardized <- standardize.dec(X, mu, sigma)
# Get the result
X_standardized <- compss_wait_on(X_standardized)
compss_stop()
```

Each function is defined as a task

We normally invoke the decorated functions which are then submitted as tasks

Run the job with bash command:

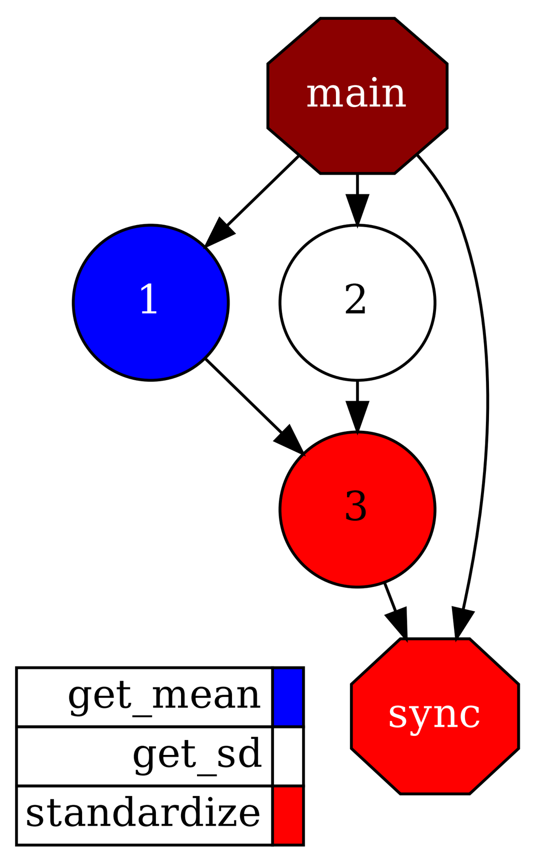
```
runcompss -g job.R
```

Then the three tasks are arranged as:

- The flag `-g` instructs the RCOMPSs runtime to generate a graphical representation of the task-dependency graph (DAG)
- Three tasks are initiated to execute the code, as illustrated by the DAG diagram
- There are three types of tasks here: (1) Compute the mean;

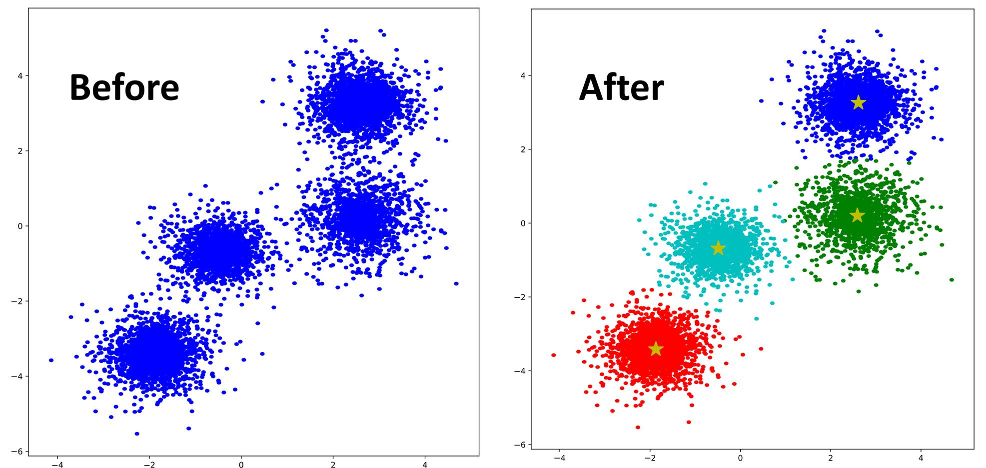
  (2) Compute the standard deviation;

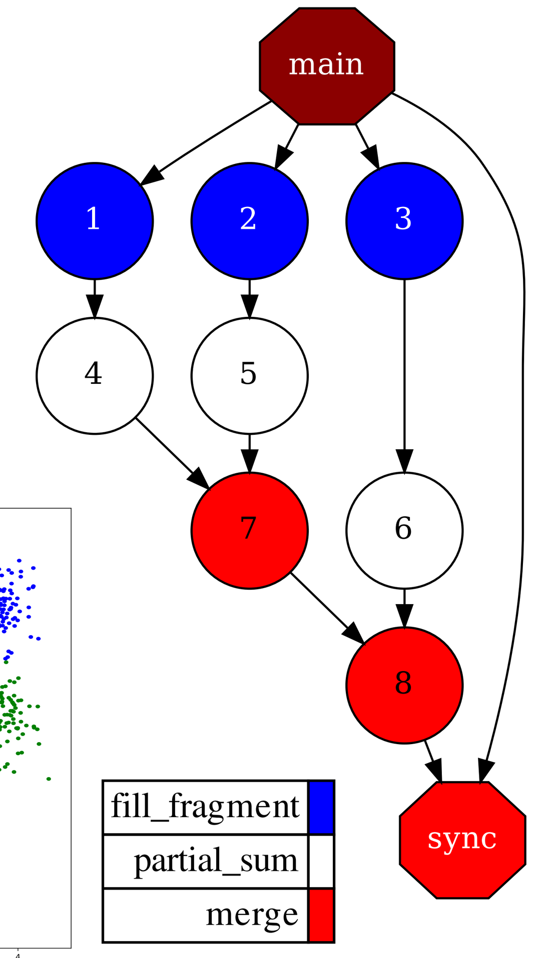
  (3) Standardize the data.
- The first two tasks can run in parallel, while ***task (3)*** must wait for the results of ***task (1)*** and ***task (2)***



Example 2: K-means clustering

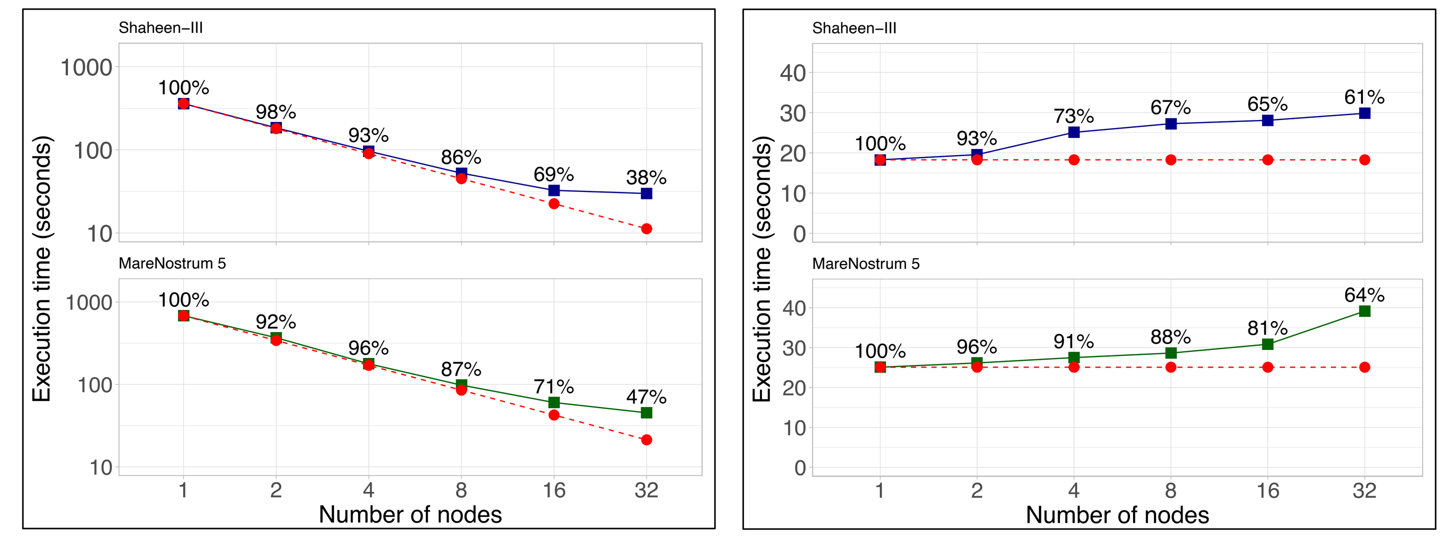
- **Step 0:** Divides dataset into fragments for concurrent processing
- **Initialization**: Starts with *k* cluster centroids
- **Data Fragment Generation:** Uses `fill_fragment` task to generate random data on-the-fly
- **Executes** `partial_sum` tasks in parallel for local sums and counts.
- **Global Centroid Update:** Combines local results with a hierarchical tree of `merge` tasks



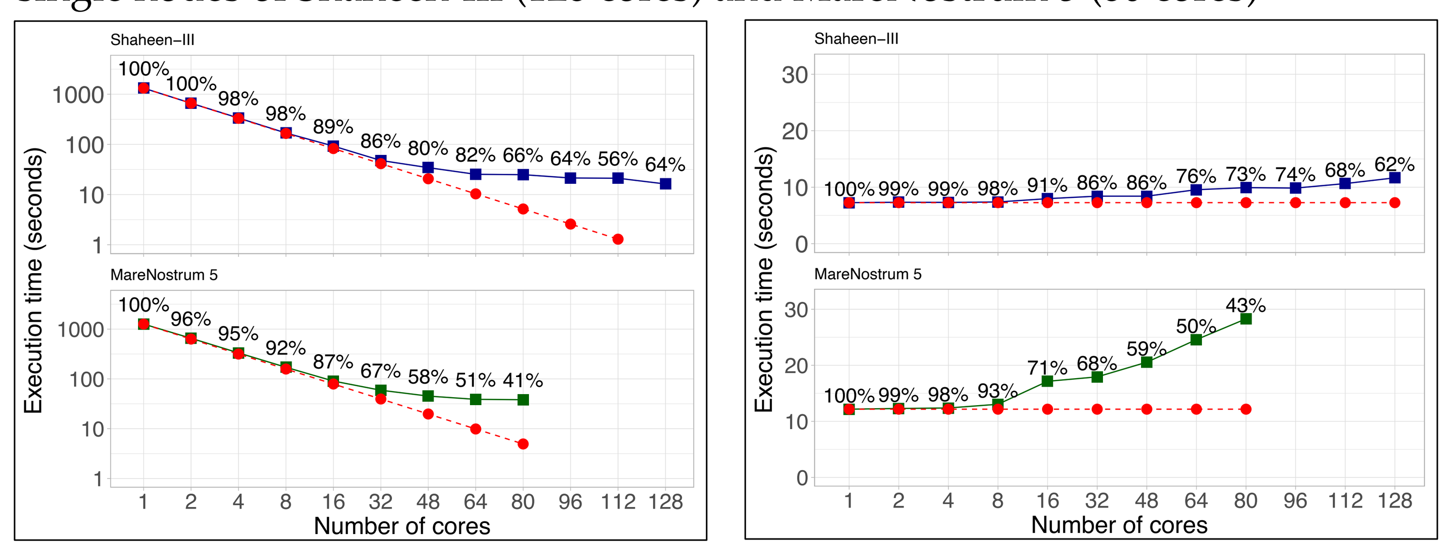


Performance

Strong (left) and weak (right) scalability of K-means clustering algorithm on up to 32 nodes of Shaheen-III and MareNostrum 5



Strong (left) and weak (right) scalability of K-means clustering algorithm on single nodes of Shaheen-III (128 cores) and MareNostrum 5 (80 cores)

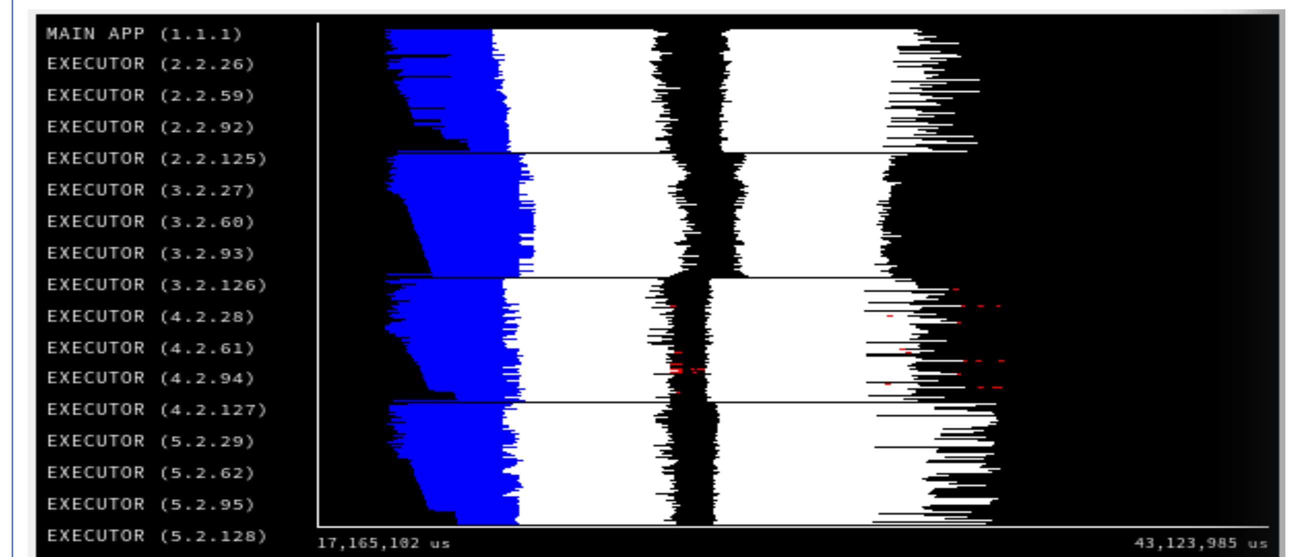


Acknowledgment

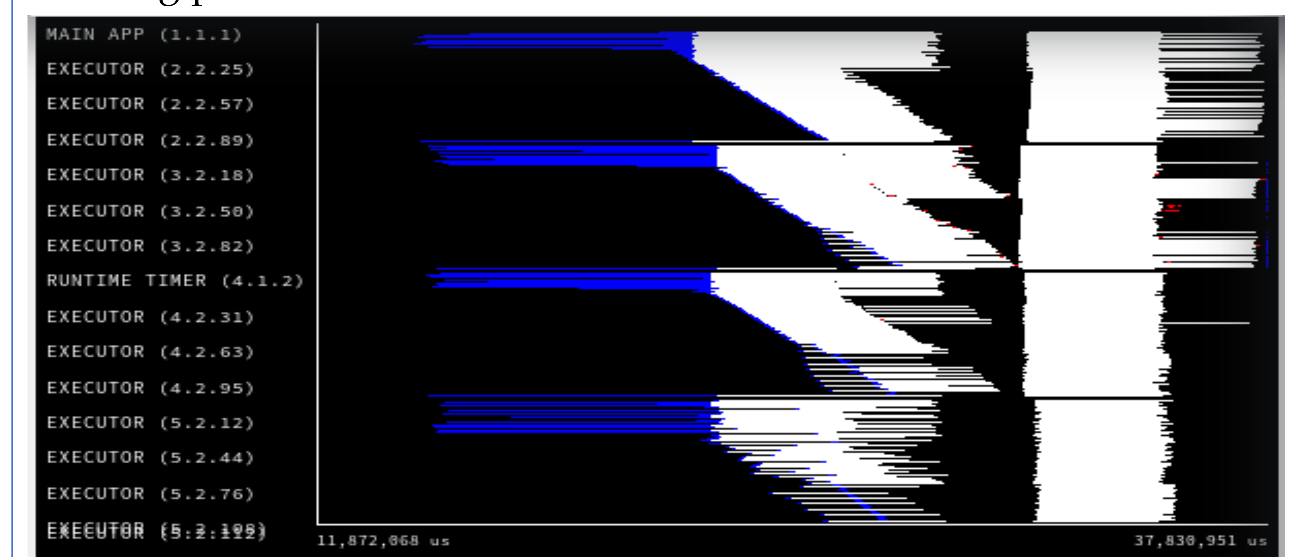
The research is partially funded by King Abdullah University of Science and Technology (KAUST). It utilized the resources at the KAUST Supercomputing Laboratory (KSL). Key resources employed in this study included the Shaheen-III supercomputer, which are pivotal assets of the KSL. R.B. and J.C. acknowledge funding from projects CEX2021-001148-S, and PID2023-147979NB-C21 from the MCIN/AEI and MICIU/AEI /10.13039/501100011033 and by FEDER, UE, by the Departament de Recerca i Universitats de la Generalitat de Catalunya, research group MPiEDist (2021 SGR 00412).

EXTRAE / Paraver tracing

- **EXTRAE/Paraver Tracing:** Performance analysis toolset widely used for trace generation and visualizing the execution of parallel applications
- EXTRAE captures execution traces. Paraver visualizes execution traces to identify performance bottlenecks, such as load imbalances and communication delays, allowing developers to optimize their algorithms for specific hardware architectures
- Tracing profile with RCOMPSs on KAUST Shaheen-III:



- Tracing profile with RCOMPSs on BSC MareNostrum 5:



- The blue blocks represent parallel `fill_fragment` tasks, followed by white `partial_sum` tasks for centroid calculation. On Shaheen-III, these tasks run concurrently, but the red `merge` tasks must complete first, causing a synchronization gap. On MN5, slower worker initialization reduces parallelism in the blue phase, making it often partly sequential, which increases overall execution time

Conclusion and future work

- **User-friendly parallel computing**: Allows R developers to parallelize code with minimal modifications via function annotations
- **Efficient runtime management**: Handles task scheduling, data dependencies, communication, and fault tolerance. It is accessible to users without deep expertise in parallel systems
- **Benchmarking results**:
  - Validated with algorithms: K-means clustering
  - Demonstrated strong scalability on Shaheen-III (@KAUST) and MareNostrum 5 (@BSC) systems
- **Integration with Performance Tools**:
  - Compatible with EXTRAE and Paraver for in-depth profiling of parallel applications
- **Future Work**:
  - Expand support for COMPSs features (task constraints, GPU-aware execution)
  - Enhance API for complex workflows and support additional data types
  - Evaluate in large-scale real-world applications and expand cloud-native compatibility