Orchestration of Software Packages in Data Science Workflows

Cristian Ramon-Cortes*, Jorge Ejarque*, Rosa M. Badia*
*Barcelona Supercomputing Center (BSC)
E-mail: {cristian.ramoncortes, jorge.ejarque, rosa.m.badia}@bsc.es

Keywords—Distributed Computing, Workflow Managers, Programming Models, Data Science pipelines, Dust Forecast

I. Extended Abstract

Nowadays, Data Science applications are complex workflows composed of binary executions, MPI simulations, multi-threaded applications, and user-defined analysis (possibly written in Java, Python, C/C++, or R). Our proposal integrates this heterogeneity into a single task-based programming model capable of orchestrating the execution of the different frameworks in a transparent way and without modifying nor its behaviour, nor its syntax. Thus, our prototype is designed for non-expert users that want to build complex workflows where some steps require a highly optimised state of the art software packages.

A. Integration with Software Packages

To integrate the execution of other frameworks transparently into a single programming model, our prototype extends the COMPSs framework [1], [2] to act as an orchestrator rather than a regular application executor. Next, we detail the modifications of the programming model annotations, although the Runtime master, and worker executors have also been modified to schedule multi-node tasks and executing the different software packages.

The COMPSs Programming Model [3], [4] defines annotations that must be added to the sequential code in order to run the applications in parallel. These annotations can be split into Method Annotations and Parameter Annotations. Our prototype extends the programming model by providing a new set of Method Annotations and Parameter annotations to support the execution of binaries, multi-threaded applications (OmpSs [5]), MPI simulations, nested COMPSs applications, and multi-node tasks inside a workflow. From now on, tasks that must execute software packages are called *Non-Native Tasks*.

On the one hand, a new Method Annotation is defined for each supported non-native task. Notice that the Method Annotation must contain framework related parameters and, thus, its content varies depending on the target framework. Next, we list the currently supported frameworks and their specific parameters.

- **Binaries:** Execution of regular binaries (e.g., BASH, SH, C, C++, FORTRAN)
  - Binary: Binary name or path to an executable
  - Working Directory: Working directory for the final binary execution
- **OmpSs:** Execution of OmpSs binaries
  - Binary: Path to the execution binary
  - Working Directory: Working directory for the final binary execution
- **MPI:** Execution of MPI binaries
  - Binary: Path to the execution binary
  - MPI Runner: Path to the MPI command to run
  - Computing Nodes: Number of required computing nodes
  - Working Directory: Working directory for the final binary execution
- **COMPSs:** Execution of nested COMPSs workflows
  - Application Name
  - Runcompss: Path to the runcompss command
  - Flags: Extra flags for the nested runcompss command
  - Computing Nodes: Number of required computing nodes
  - Working Directory: Working directory for the nested COMPSs application
- **Multi-node:** Execution of native Java/Python tasks that require more than one node
  - Computing Nodes: Number of required computing nodes

On the other hand, in order to input and output data from the execution of non-native tasks, the Parameter Annotation also needs to be enhanced. When using multi-node tasks the parameters and the return value of the task are the same than when using a regular method task. However, when executing standalone binaries, OmpSs processes, MPI processes, or COMPSs applications the exit value of the processes is used as the return value. Thus, we have decided that the COMPSs non-native tasks must use the exit value of their internal binary as the return value of the task. In this sense, our prototype allows the users to capture this value by defining the return type of the non-native task as an *int* (for implicit synchronisation), as an *Integer* (for post-access synchronisation) or to forget it (declaring the function as *void*).

However, the users do not only need the process exit value to work with this kind of applications but need to set the Standard Input (*stdIn*) and capture the Standard Output (*stdOut*) and Error (*stdErr*). For this purpose, our prototype includes a new Parameter Annotation, `stream`, that allows the users to set some parameters as I/O streams for the non-native

tasks. Stream parameters are not passed directly to the binary command but rather they are set as *stdIn*, *stdOut* or *stdErr* of the binary process. Since this kind of redirection is restricted to files in *LINUX* Operating Systems, we have decided to keep the same restrictions to the annotation. Consequently, all *stream* parameters must be files.

B. Evaluation

1) NMMB-MONARCH: The NMMB-MONARCH is a fully online multiscale chemical weather prediction system for regional and global-scale applications. The system is based on the meteorological Nonhydrostatic Multiscale Model on the B-grid [6], developed and widely verified at the National Centers for Environmental Prediction (NCEP). The model couples online the NMMB with the gas-phase and aerosol continuity equations to solve the atmospheric chemistry processes in detail. It is also designed to account for the feedbacks among gases, aerosol particles, and meteorology.

The NMMB-MONARCH workflow is composed by five main steps, namely *Initialization*, *Fixed*, *Variable*, *UMO Model*, and *Postprocess*. Although all the phases spawn several binaries, the NEMS binary soars above the rest. It is called during *UMO Model* step and its a Fortran 90 application parallelised with MPI. Therefore, NEMS can be executed in multiple cores and multiple nodes, relying on the MPI paradigm. The model uses the Earth System Modelling Framework library as the main framework [7].

2) Parallelisation design: The original NMMB-MONARCH application consists in a BASH script defining the main workflow and a set of Fortran binaries. Since our prototype supports Java and Python workflows, we have ported two different versions of the NMMB-MONARCH. Both of them parallelise the main workflow while keeping the Fortran binaries.

Regarding the parallelisation, all the binaries have been considered as tasks. For instance, Figure 1 shows the annotation of the `deeptemperature` binary in Python. We do not show the equivalent Java version due to space constraints.

```
@binary(binary='/path/to/deeptemperature.x')
@task(returns=int,
      seamask=FILE_IN,
      deep_temperature=FILE_OUT)
def deeptemperature():
    pass
```

Fig. 1. Annotation of the `deeptemperature` binary in Python

As previously explained, the NEMS simulator invoked within the *UMO Model* step is implemented using MPI. Thus, in contrast to the rest of binaries, we have annotated it as an MPI task. Figure 2 shows the annotation of the `NEMS` binary in Python. Notice that, considering that the NEMS execution can be performed with a different number of nodes, the constraint decorator attached to the MPI task may vary between executions. Hence, to ease this management to users, the constraint has been defined using an environment variable.

To conclude, the final workflow structure for both the Java and the Python versions is depicted in Figure 3. Readers can identify the *Fixed*, *Variable*, *UMO Model*, *Postprocess* and *Figures and animations creation* steps, where the *Variable*, *UMO Model* and *Postprocess* are executed thrice.

```
@constraint(computingUnits='$NEMS_CUS_PER_NODE')
@mpi(runner='mpirun',
     binary='/path/to/NEMS.x',
     workingDir='/path/to/nems/out',
     computingNodes='$NEMS_NODES')
@task(returns=int,
      stdOutFile=FILE_OUT_STDOUT,
      stdErrFile=FILE_OUT_STDERR)
def nems():
    pass
```

Fig. 2. Annotation of the `nems` MPI binary in Python

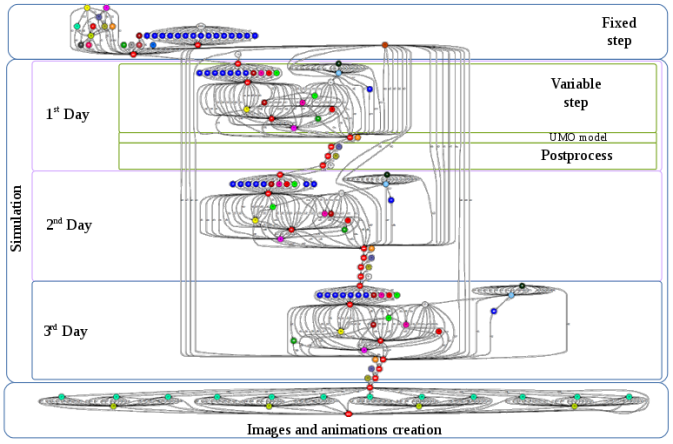

Fig. 3. Dependency graph of three days simulation

3) Computing infrastructure: The infrastructure used in this comparison is the Nord 3 cluster (a subset of the MareNostrum III Supercomputer [8]), located at the Barcelona Supercomputing Center (BSC). This supercomputer is composed of 84 nodes, each with two Intel SandyBridge-EP E5-2670 (8 cores at 2.6 GHz with 20 MB cache each), a main memory of 128 GB, FDR-10 Infiniband and Gigabit Ethernet network interconnections, and 1.9 PB of disk storage.

The Java version used is 1.8.0_u112 64 bits, the Python version used is 2.7.13, and the MPI used is OpenMPI 1.8.1.

4) Performance: Table I compares each step of the previous version (BASH) against the Java and Python ones using 4 worker processes (64 cores).

Step	Execution Time (s)			Speed-up (u)	
	BASH	Java	Python	Java	Python
Fixed	290	117	119	2.48	2.43
Variable	26	19	22	1.37	1.18
Model Simulation	244	242	233	1.01	1.04
Post process	38	34	33	1.12	1.15
Total	**601**	**413**	**415**	**1.45**	**1.45**

TABLE I. PERFORMANCE PER STEP WITH 4 WORKERS (64 CORES)

Notice that both Java and Python versions improve the performance in the *Fixed* and *Variable* steps due to the possibility of performing multiple tasks at the same time during these steps. In opposition, the *Model Simulation* and *Post Process* do not improve because they are composed, respectively, by a single task, and two tasks with a sequential dependency. However, the performance of these steps does not degrade either.

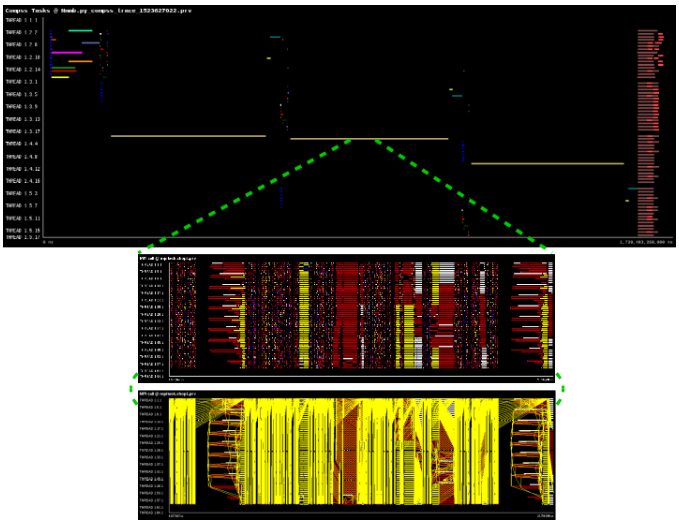

Fig. 4. Paraver trace of the Python version using 4 workers (64 cores)

Figure 4 shows Paraver [9] traces of a three-day simulation with 4 workers (64 cores) of the Python workflow. There are three timelines where each row corresponds to a thread in the worker. The top view shows a task view, where different coloured segments represent tasks, and each colour identifies a different task type. The middle and bottom views are the internals of the MPI NEMS task, where the yellow lines represent communications between MPI ranks.

The trace detail in the middle and bottom views show, respectively, the MPI events and the MPI communications of the *Model simulation* task. Notice that, during this phase, both frameworks, COMPSs and MPI, are working together and sharing the computing nodes. Although in this case MPI is using the four nodes, other applications may reserve some nodes for MPI and use the others to compute remaining sequential PyCOMPSs tasks.

C. Conclusions and Future Work

Our prototype enables non-expert users to develop complex Data Science workflows where some steps require a highly optimised state of the art software package. It provides a single task-based framework with a homogeneous annotation to orchestrate the execution of native Java and Python tasks along with binaries, MPI simulations, multi-threaded applications, and nested applications. From the users' point of view, the annotation is transparent and does not require to modify nor the behaviour, nor the syntax of the underlying software packages.

Moreover, the evaluation demonstrates that our prototype eases the development of complex workflows such as NMMB-MONARCH. We have developed two new implementations in Java and Python that provide better configuration management and object-oriented structure and improve the debugging, maintenance, and extension of the code. Also, the performance analysis demonstrates that our prototype is capable of extracting the parallelism of the NMMB-MONARCH application achieving 1.45 overall speed-up and sharing resources with multi-node frameworks such as MPI.

II. Acknowledgment

This work has been published in proceedings of the 2018 IEEE 14th International Conference on e-Science (e-Science) [10]. Cristian Ramon-Cortes predoctoral contract is financed by the Ministry of Economy and Competitiveness under the contract BES-2016-076791.

References

[1] R. M. Badia and et al., "COMP superscalar, an interoperable programming framework," *SoftwareX*, vol. 3, pp. 32–36, 12 2015. [Online]. Available: http://dx.doi.org/10.1016/j.softx.2015.10.004

[2] Workflows and Distributed Computing - Barcelona Supercomputing Center (BSC). (2018) COMP Superscalar. [Online]. Available: http://compss.bsc.es

[3] F. Lordan *et al.*, "ServiceSs: an interoperable programming framework for the Cloud," *Journal of Grid Computing*, vol. 12, no. 1, pp. 67–91, 3 2014. [Online]. Available: https://digital.csic.es/handle/10261/132141

[4] E. Tejedor *et al.*, "PyCOMPSs: Parallel computational workflows in Python," *The International Journal of High Performance Computing Applications (IJHPCA)*, vol. 31, pp. 66–82, 2017. [Online]. Available: http://dx.doi.org/10.1177/1094342015594678

[5] A. Duran *et al.*, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[6] Z. Janjic and R. Gall, "Scientific Documentation of the NCEP Nonhydrostatic Multiscale Model on the B Grid (NMMB). Part 1 Dynamics, Technical Report," NCEP, BOULDER, COLORADO, Tech. Rep. 80307-3000, 4 2012.

[7] C. Hill *et al.*, "The architecture of the Earth System Modeling Framework," *Computing in Science Engineering*, vol. 6, no. 1, pp. 18–28, 1 2004.

[8] Barcelona Supercomputing Center (BSC). (2017) MareNostrum 3. [Online]. Available: https://www.bsc.es/marenostrum/marenostrum/mn3

[9] ——. (2017) Paraver Tool. [Online]. Available: https://tools.bsc.es/paraver

[10] J. Conejero *et al.*, "Boosting Atmospheric Dust Forecast with PyCOMPSs," in *2018 IEEE 14th International Conference on e-Science (e-Science)*, Oct 2018, pp. 464–474.

Cristian Ramon-Cortes is a PhD Student for the Computer Architecture Department (DAC - UPC) working in collaboration with the Workflows and Distributed Computing group (WDC) at the Barcelona Supercomputing Center (BSC). He holds a MSc. on Research and Innovation in Informatics - High Performance Computing (MIRI - HPC, 2017), an Engineering Degree on Computer Science (FIB, 2014), an Engineering Degree on Industrial Engineering (ETSEIB, 2014), and a Dual BSc. Diploma (CFIS, 2014); all of them from the Technical University of Catalonia (UPC). During his career at the BSC he has contributed in the design and development of COMPSs, PyCOMPSs, and PMES. His areas of interest include Programming Models for Distributed Platform, Workflow Managers, Task Flows, Data Flows, and Streaming Technologies.