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 (stdio) 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 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 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 deep_temperature binary in Python. We do not show the equivalent Java version due to space constraints.

@binary(binary='/path/to/deeptemperature.x')
@task(returns=int,
    workingDir='/path/to/nems/output',
    deep_temperature=FILE_OUT)
def deep_temperature():
    pass

Fig. 1. Annotation of the deep_temperature 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.

@constraint(computingUnits='$NEMS_CUS_PER_NODE')
@mpi(runner='mpirun',
    binary='/path/to/nems.x',
    workingDir='/path/to/nems/output',
    computingNodes='$NEMS_NODES')
def nems():
    pass

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

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_112 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).

<table>
<thead>
<tr>
<th>Step</th>
<th>Execution Time (s)</th>
<th>Speed-up (u)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BASH</td>
<td>Java</td>
</tr>
<tr>
<td>Fixed</td>
<td>290</td>
<td>117</td>
</tr>
<tr>
<td>Variable</td>
<td>26</td>
<td>19</td>
</tr>
<tr>
<td>Model Simulation</td>
<td>244</td>
<td>242</td>
</tr>
<tr>
<td>Post process</td>
<td>38</td>
<td>34</td>
</tr>
<tr>
<td>Total</td>
<td>601</td>
<td>413</td>
</tr>
</tbody>
</table>

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.
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


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.