Orchestration of Software Packages in Data Science Workflows

Cristián Ramón-Cortés
Javier Conejero
Jorge Ejarque
Rosa M. Badia

May 2019
Outline

▶ Introduction
  • Motivation
  • COMPSs / PyCOMPSs
  • Current annotations (Python only)

▶ Integration
  • New annotations
  • Exit value, prefix and I/O Stream annotations

▶ Use Case: NMGB-MONARCH
  • NMGB-MONARCH
  • Parallelization design
  • Evaluation

▶ Conclusions and Future Work
Introduction
Motivation

- Data Science applications:
  - Complex pipelines developed by field experts
  - Widely used state of the art software packages for specific actions
  - Heterogeneous requirements

Cumbersome handmade pipelines

Specialized Frameworks

- Kepler
- DASK
- FireWorks
- COMPSS
- Galaxy
- Parsl
- slurm
- nextflow
- autosubmit
- swift

Programmability
Performance
Scalability
COMPSs Motivation

THE GOAL:
Any field expert can scale up an application to thousands of cores

Ease the development of distributed applications

Parallel Issues
- Identifying parallel regions
- Concurrency management
- Execution orchestration

Distributed Issues
- Remote execution
- Data transfers
COMPSs

- Based on sequential programming
- Minimal impact on user code
  - General purpose programming language + annotations
- Aimed at exploiting the inherent parallelism of sequential applications on distributed environments.
- Sequential execution starts in the master node, and tasks are offloaded to worker nodes.
COMPSs

- Task-based programming model
  - Task is the unit of work
  - Implicit Workflow: Builds a task graph at runtime that expresses potential concurrency

- Infrastructure agnostic
  - Same application runs on clusters, grids, clouds, and containers
PyCOMPSs Annotations

- Python decorators for task selection + synchronization API
  - Instance and class methods
  - Task data directions

```python
@task(a=IN, b=IN, c=INOUT)
def multiply_acum(a, b, c):
    c += a * b

@task(returns=dict)
def wordcount(block):
    ...

@task(result=INOUT)
def reduce(result, pres):
    ...

def main(a, b, c):
    for block in data:
        pres = wordcount(block)
        reduce(result, pres)
    result = compss_wait_on(result)

    # f = compss_open(fn)
    # compss_delete_file(f)
    # compss_delete_object(o)
    # compss_barrier()
```
Integration
New Programming Model annotations (1)

- **Binaries**: Execution for regular binaries (i.e., BASH, fortran, C)
  - Binary
  - Working Directory *(opt)*

```python
@binary(binary = "path_to_bin")
@task()
def myBinaryTask():
    pass
```

- **OmpSs**: Execution of OmpSs binaries
  - Binary
  - Working Directory *(opt)*

```python
@ompss(binary = "path_to_bin")
@task()
def myOmpSsTask():
    pass
```

- **MPI**: Execution of MPI binaries
  - Binary
  - MPI Runner
  - Computing Nodes
  - Working Directory *(opt)*

```python
@mpi(mpi_runner = "mpirun",
     binary = "path_to_bin",
     computing_nodes = "N")
@task()
def myMPITask():
    pass
```
New Programming Model annotations (2)

- **COMPSs**: Nested COMPSs applications
  - Application name
  - Runcompss command
  - Runcompss extra flags *(opt)*
  - Computing Nodes
  - Working Directory *(opt)*

```python
@compss(runcompss = "runcompss", app_name = "mpirun", computing_nodes = "N")
@task()
def myNestedCOMPSsTask():
    pass
```

- **MultiNode**: Native Java/Python multi-node tasks
  - Computing Nodes

```python
@multinode(computing_nodes = "N")
@task()
def myMultiNodeTask():
    # Python code
```
New Programming Model annotations (3)

- **Exit value**

```python
@binary(binary = "binary")
@task(returns=int)
def task_ev():
    pass

./binary; ev=？
```

- **I/O Stream Parameters**

```python
@binary(binary = "binary")
@task(file_in=FILE_IN_STDIN,
     file_out=FILE_OUT_STDOUT | FILE_INOUT_STDOUT,
     file_err=FILE_OUT_STDERR | FILE_INOUT_STDERR)
def task_io(param, file_in, file_out, file_err):
    pass

./binary < file_in > file_out >&2 file_err
```

- **Parameters Prefix**

```python
@binary(binary = "binary")
@task(file1=FILE_IN,
     file2={Type: FILE_INOUT, Prefix: "-q="},
     k_value={Prefix: "k"})
def task_prefix(p="-p", file1=None, file2=None, k_value=10):
    pass

./binary -p file1.in -q=file2.inout k10
```
Use Case: NMMB-MONARCH
NMNM-Monarch

- **Multiscale Online Nonhydrostatic Atmosphere Chemistry**
  - Multiscale: Global to regional scales allowed (up to 1km)
  - Fully on-line coupling: weather-chemistry feedback processes allowed
  - Enhancement with data assimilation system

**Objective:**

Predict the atmospheric life cycle

- The model couples online the NMMB with the gas-phase and aerosol continuity equations to solve the atmospheric chemistry processes in detail

- Designed to account for the feedback among gases, aerosol particles, and meteorology
NMNB-Monarch

- Originally:
  - BASH workflow
  - Fortran 77 binaries
  - Fortran 90 binaries
Parallelization with COMPSs/PyCOMPSs

- Migrate the workflow code to sequential Java / Python code keeping the same structure
- Determine the potential tasks
- Include the creation of images and animations
@binary(binary="deeptemperature.x")
@task(returns=int,
    seamask=FILE_IN,
    deep_temperature=FILE_OUT)
def deeptemperature(seamask, deep_temperature):
    pass

@constraint(computingUnits="16")
@mpi(mpi_runner="mpirun",
    binary="/path/to/NEMS.x",
    computing_nodes="$NEMS_NODES",
    working_dir="/path/to/nems/out")
@task(returns=int,
    stdout_file=FILE_OUT_STDOUT,
    stderr_file=FILE_OUT_STDERR)
def nems(stdout_file, stderr_file):
    pass

@task(fname=FILE_IN,
    i1=FILE_OUT, i2=FILE_OUT, i3=FILE_OUT, i4=FILE_OUT, i5=FILE_OUT,
    i6=FILE_OUT, i7=FILE_OUT, i8=FILE_OUT, i9=FILE_OUT)
def generate_figures(date, fname, vname, i1, i2, i3, i4, i5, i6, i7, i8, i9):
    # Python code

@task(gif_name=FILE_OUT, varargsType=FILE_IN)
def generate_figures(fig_name, skip_frames, *args):
    # Python code
Task graph
Performance

- Strong Scaling. 3 Days simulation

![Graph showing execution time and speed-up for BASH, Java, and Python across different steps]

- Per step analysis. 1 Day simulation @ 4 workers (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 Sim.</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>598</td>
<td>412</td>
</tr>
</tbody>
</table>
Simulation Results
Conclusions and Future Work
Conclusions and Future Work

▶ Enabling the orchestration of Software Packages in Data Science workflows
  • Complex workflows in a single language (Java or Python) with an homogeneous annotation for many software packages
  • Transparent orchestration, data management, and execution of binaries, OmpSs, MPI, nested COMPSs, and native multi-node tasks

▶ NMMB-MONARCH has been parallelized with COMPSs and PyCOMPSs (Java and Python workflows)
  • Task level parallelization with Binaries, MPI, and native functions
  • Programmability and performance improvements

▶ Next steps
  • Extend the annotation for more software packages
  • Pre/post actions when spawning non-native tasks
Thank you

cristianrcv/pycompss-autoparallel

http://compss.bsc.es/

cristian.ramon-cortes@bsc.es
Backup
Application Behaviour

Fixed Step 1st Iteration 2nd Iteration 3rd Iteration Results

NEMS task
NEMS communications
## Programmability

- Better configuration management
- Better object-oriented structure
- Improves maintenance, extension, and debugging

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Blank</th>
<th>Comment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original NMMB-MONARCH Workflow</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran 90</td>
<td>23</td>
<td>394</td>
<td>2806</td>
<td>7581</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>8</td>
<td>182</td>
<td>3568</td>
<td>6518</td>
</tr>
<tr>
<td>BASH</td>
<td>16</td>
<td>185</td>
<td>134</td>
<td>776</td>
</tr>
<tr>
<td><strong>New NMMB-MONARCH Workflow with COMPSs/PyCOMPSs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran 90</td>
<td>23</td>
<td>394</td>
<td>2806</td>
<td>7581</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>8</td>
<td>182</td>
<td>3568</td>
<td>6518</td>
</tr>
<tr>
<td>Java</td>
<td>18</td>
<td>560</td>
<td>890</td>
<td>2721</td>
</tr>
<tr>
<td>Python</td>
<td>18</td>
<td>515</td>
<td>710</td>
<td>2399</td>
</tr>
</tbody>
</table>