The final authenticated version is
available online at https://doi.org/10.1007/978-3-030-57675-2_26

Managing Failures in Task-based Parallel Workflows in Distributed Computing Environments

Jorge Ejarque[0000−0003−4725−5097], Marta Bertran[0000−0001−9723−1291], Javier Álvarez Cid-Fuentes[0000−0001−7153−4649], Javier Conejero[0000−0001−6401−6229], and Rosa M. Badia[0000−0003−2941−5499]

Barcelona Supercomputing Center, Barcelona, Spain
{jorge.ejarque, javier.alvarez, francisco.conejero, rosa.m.badia}@bsc.es

Abstract. Current scientific workflows are large and complex. They normally perform thousands of simulations whose results combined with searching and data analytics algorithms, in order to infer new knowledge, generate a very large amount of data. To this end, workflows comprise many tasks and some of them may fail. Most of the work done about failure management in workflow managers and runtimes focuses on recovering from failures caused by resources (retrying or resubmitting the failed computation in other resources, etc.) However, some of these failures can be caused by the application itself (corrupted data, algorithms which are not converging for certain conditions, etc.), and these fault tolerance mechanisms are not sufficient to perform a successful workflow execution. In these cases, developers have to add some code in their applications to prevent and manage the possible failures. In this paper, we propose a simple interface and a set of transparent runtime mechanisms to simplify how scientists deal with application-based failures in task-based parallel workflows. We have validated our proposal with use-cases from e-science and machine learning to show the benefits of the proposed interface and mechanisms in terms of programming productivity and performance.

Keywords: Failure management · Scientific Workflows · Parallel Programming · Distributed computing

1 Introduction

E-science has evolved very fast during last few decades. At the beginning, small computations were performed in a single machine, while nowadays, large complex scientific workflows are executed in large distributed computing platforms. These workflows combine the execution of thousands of simulations with searching and data analytic algorithms to infer new knowledge from a large amount of data. Due to the nature of the infrastructure and the algorithms used on the workflow, some components of the computation can fail or become blocked. This can be due to resource failures, data corruption, or just because the initial conditions of

a simulation do not converge into a valid solution. These failures can make the whole workflow execution fail or hang without generating the expected results.

Most workflow managers, such as Galaxy [1] or Pegasus [5], have some fault tolerance mechanisms, but they are mainly focused on resource failures, and they do not deal with application failures or exceptions. In these cases, the responsible to deal with failures is the developer, who has to include some code in the application in order to implement custom mechanisms to make the whole workflow reliable. In a sequential application, this customized management inside the code requires additional software development efforts which can be managed with traditional error handling mechanisms provided by the programming languages, such as managing exceptions or inspecting the return values to decide how to adapt the code in case of failure. However, tasks in parallel and distributed workflows are asynchronously executed in remote resources, so implementing similar defensive codes for these workflows are more complex and can produce performance losses due to the unnecessary synchronizations and transfers to inspect task results.

This paper proposes a simple user interface to allow developers to easily indicate how to manage application failures. This interface extends the task definition in order to allow developers to provide hints about how the runtime has to react in case of a failure occurs during the task execution. Based on this developer hint, the runtime transparently implements a set of mechanisms to efficiently handle these failures, reducing the development efforts because developers do not need to add defensive code as explained above, and without affecting application performance, because the failure management is concurrently performed with the application execution.

A prototype of this proposal has been implemented in COMPSs [3], a task-based parallel programming model to easily implement parallel workflows for distributed computing environments, and it has been validated through two real applications from e-Science and Machine Learning areas. We have evaluated the productivity and performance of this solution compared to a user-developed alternative. The results of this evaluation demonstrate that the proposed solution reduces the code complexity and achieves better performance than a user-managed approach.

The rest of the paper is organized as follows: Section 2 presents the related work; Section 3 introduces the proposed mechanisms and Section 4 describes how they have been implemented in COMPSs. Then, Section 5 presents the evaluation; Finally, Section 6 draws the conclusions.

2 Related work

Failures in the execution of workflows are frequent, especially when executed in distributed computing platforms. For this reason, several workflow management systems provide some way of tolerating failures and its management.

For example, Galaxy [1] provides automatic job re-submission (e.g., on job failure due to a temporary cluster error). Also, in order to make Galaxy more ro-

bust in a production environment, technologies to enhance Galaxy's portability, security, reliability, and scalability have been adopted. Galaxy utilizes uWSGI[1] as its default web application server. It has several advantages, including improved fault tolerance, and the possibility of restarting Galaxy uninterruptedly. The mechanisms supported by Taverna [14] are similar, with retries at service and workflow level. Several retry types are supported, such as exponential back-off of retry times.

Kepler [11] proposes three complementary mechanisms for fault tolerance: a) a forward recovery mechanism that offers retries and alternative versions at the workflow level; b) a checkpointing mechanism, also at the workflow layer, that resumes the execution in case of a failure at the last saved good state; and c) an error-state and failure handling mechanism to address issues that occur outside the scope of the workflow layer.

Cylc [12] is a workflow management system proposed by the Earth Science community. It provides checkpointing, which keeps a list of completed tasks, and if the scheduler does not respond properly, the user can restart the experiment, from the last checkpoint. Users can also define retries for the different experiment jobs.

Pegasus [5] provides some failure management features as well. In case of transient infrastructure failures, such as a node being temporarily down in a cluster, Pegasus will automatically retry jobs. After a given number of retries (usually once), a hard failure occurs, because of which the workflow will eventually fail. In most of the cases, these errors are correctable (either the resource comes back online or application errors are fixed). Once the errors are fixed, the Pegasus workflow can be restarted from the point of failure. While executing a workflow, Pegasus creates the rescue workflow, which contains the description of the work that remains to be done.

Nextflow [6] provides several failure management mechanisms. First, it provides continuous checkpointing: all the intermediate results produced during the pipeline execution are automatically tracked. This allows to resume the execution from the last successfully executed step. Nextflow also provides a mechanism that allows tasks to be automatically re-executed when a command terminates with an error exit status. In Nextflow, it is also possible to define the *errorStrategy* directive in a dynamic manner for a given task. This is useful to re-execute failed jobs only if a certain condition is verified.

What is presented in this paper differs from previous approaches since what we propose is an individual and tailored management policy for each task type. The last approach described above (Nextflow) is the one closer to what it is presented in this paper, but it differs since it does not support all the possible policies for task failure management proposed in this paper. The proposed errorStrategy does not allow to indicate what to do with the non generated data or what to do with tasks which depend on the failed tasks. Moreover, Nextflow provides their own scripting language, and they do not offer the possibility of managing task exceptions as well as managing tasks which enter a hang state.

[1] ttp://projects.unbit.it/uwsgi

3 Application Failure Management

As raised in the introduction, developers are responsible to make applications reliable, predicting what could be the possible failures in each part of the application and implementing a code to recover the execution from these failures. In sequential programming, developers use return values, which are inspected in the main code to decide what to do in case the function returns a problem. However, in distributed parallel workflows, this management is not efficient because workflow tasks are executed in an asynchronous remote way, so waiting for a result to decide what to do next, requires unnecessary synchronization points and data movements to transfer results back and inspect them. The next subsections present common workflow task failures, their implications and the mechanisms that we propose to easily and efficiently manage them.

3.1 Common workflow task failures and implications

Workflow failures can be classified in the types enumerated below. Each of these types has different implications, which are described in the next paragraphs.

- **Tasks which stop their execution before completion.** They can be produced by an invalid input or errors returned by simulators. The main consequence of these failures is that task results are not completely generated and all the successor tasks could also fail or their results be invalid.
- **Task execution blocked or lasting more than expected.** These failures can be produced by tasks which are running algorithms that, depending on the input, can enter in a deadlock or never converge.
- **Tasks throwing exceptions.** These failures are similar to the first type but they can affect not only the dependent tasks but also others which are in the group or block.

3.2 Failure Management Mechanisms

To allow workflow developers to easily manage the different type of failures, we propose to extend the task definition and the runtime mechanism to implement the following features:

- **Failure reaction policy**: It allows developers to indicate to the runtime what to do when a task fails. This policy is described in the task definition interface and it is applied by the runtime to decide what to do with the successor tasks and what to do with the expected task results.
- **Automatic cancellation after timeout**: This feature is also activated by including the *timeout* property in the tasks definition. It allows users to define a maximum duration per task to avoid tasks running forever. Tasks cancelled because of exceeding the timeout are considered failures. Therefore, this feature can be combined with the failure reaction policy to decide what to do with the rest of the workflow.

```python
@task()
def word_count(block):
    ...
    return res

@task(f_res=INOUT)
def merge_count(f_res, p_res):
    ...
```

(a) Task annotation example

```python
for block in data:
    p_result = word_count(block)
    reduce_count(result, p_result)
result = compss_wait_on(result)
```

(b) Main code example

Fig. 1: PyCOMPSs application example.

– **Parallel distributed exception**: It allows developers to create "try/catch" blocks in task-based parallel workflows for distributed environments. With this functionality, the developer can implement a try code block using the programming model syntax, where tasks invoked inside this block will belong to the same task group. If one of these remote tasks throws an exception the runtime will catch it and the current and pending tasks of this group will be cancelled as it is done in try/catch blocks provided by some programming languages.

4 Implementation

This section provides more details about how the proposed mechanisms are implemented in COMPSs, by extending the COMPSs syntax to allow developers to specify a failure reaction policy, a timeout per task type and to define a try/catch block in parallel workflows, and by implementing the management of these extensions in the COMPSs runtime.

4.1 COMPSs overview

COMP Superscalar (COMPSs) is a task-based parallel programming for distributed computing. Based on sequential programming, application developers, by means of code annotations, select a set of methods whose invocations are considered tasks and indicate the direction of their parameters.

COMPSs runtime [9] orchestrates the execution of applications and its tasks on the underlying infrastructure. For this purpose, for each invocation to a task it analyses the data dependencies with previous ones according to the parameter annotations. With this information, COMPSs runtime builds a Directed Acyclic Graph (DAG) where nodes represent tasks and edges represent data dependencies between them. COMPSs runtime is able to infer the task-level parallelism from this graph, and schedules and submits tasks for execution. The runtime also takes care of all required data transfers. If a partial failure raises during a task execution, the master node handles it with job resubmission and reschedule techniques. However, after a maximum number of retries, the whole workflow is considered as failed and the whole execution is stopped.

```
1  @task(output_file={type:FILE, direction=OUT, default_value="EMPTY"}, on_failure="IGNORE")
2  def task_example(output_file):
3      ...
```

Fig. 2: Task definition with failure reaction policy and default value.

COMPSs provides Java as native programming language and it also provides bindings for Python (PyCOMPSs [2]) and C/C++ [7]. Figure 1 shows an example of a task annotation and COMPSs main program. The first line contains the task annotation in the form of a Python decorator, while the rest of the code is a regular Python method. The parameter `f_res` is of type INOUT (the data is read and written by the method), and the parameter `p_res` is set to the default type IN (the data is only read by the method). These directionality clauses are used at execution time to derive the data dependencies between tasks and are applied at object level, taking into account its references to identify when two tasks access the same object, and can also be applied at file level when parameters are files. A tiny synchronisation API completes the PyCOMPSs syntax. For instance, as shown in Listing 1.b, the `compss_wait_on` waits until all the tasks modifying the `result`'s value have finished and brings the value to the node which executes the main program (line 4). Once the value is retrieved, the execution of the main program code is resumed. Given that PyCOMPSs is mostly used in distributed environments, synchronising may imply a data transfer from remote storage or memory space to the node executing the main program.

4.2 Failure Reaction Policy

As introduced above, the failure reaction action policy provides a hint to the runtime about what to do if a task fails. This hint is provided in the task definition as indicated in Figure 2 and it will apply to all the instances of this type of task. It consist of adding the *on_failure* property to the task decorator, and the *default_value* property to the task parameter description. For the first case, the user can choose one of the following options:

- **FAIL**: If a task with this option fails, the whole application is stopped recovering the computed data until the moment of the failure.
- **RETRY** (Default): If a task with this option fails, the runtime re-executes it in the same node and, if the failure persists, resubmits it to a different one. If the task after these retries still fails, it applies the *FAIL* procedure.
- **IGNORE**: If a task with this option fails, the failure is ignored, the data not generated (return or with direction OUT) is set as indicated in the *default_value* property, and successor tasks are executed using these values.
- **CANCEL_SUCCESSORS**: If a task with this option fails, the runtime ignores the failure, recursively cancels its successors, and deletes all the data and versions which are not going to be generated by the failed task and its successors in order to keep the data coherence of the rest of the workflow.

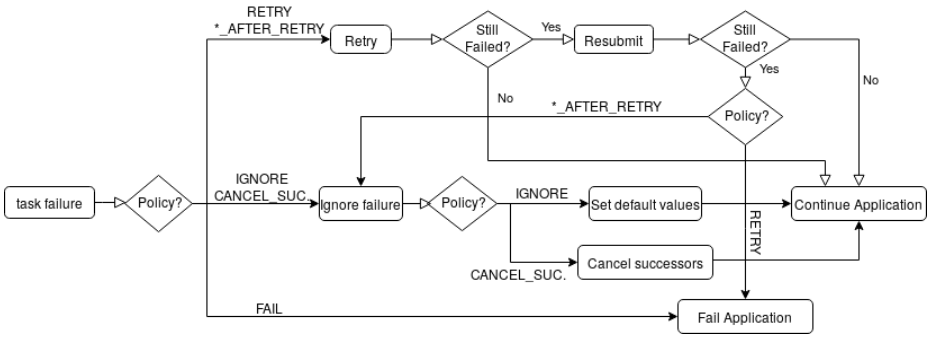

Fig. 3: Task failure management at runtime.

- **IGNORE_AFTER_RETRY**: If a task with this option fails, the runtime first applies the *RETRY* procedure to try to recover from temporary resource failures. If the failure persists, it applies the *IGNORE* procedure.
- **CANCEL_SUCCESSORS_AFTER_RETRY**: As in the previous option, if a task with this option fails, the runtime applies the *RETRY* procedure and, if the failure persists, it applies the *CANCEL_SUCCESSORS* procedure.

As we have seen before, an important issue when ignoring a failure is the value of data that has not been generated by the failed task. This value can be indicated by setting one of the following options in the *default_value* property:

- **EMPTY** (Default): The runtime will create an empty file or an empty object (an object created with the default constructor) depending on the parameter type.
- **NONE**: It will set the object or the file path as None (null in Java).
- [**Path/to/file**]: The parameter will be set as the content of a file indicated by a path (it can also be a serialized object).

The diagram depicted in Figure 3, summarises how the COMPSs runtime manages task failures. First, it captures task failures at worker processes. These failures are notified to the master, which applies the procedures defined in the policies, resubmitting or cancelling tasks, as well as doing the proper data management (e.g.setting default values, version rollback, data deletion) to keep the application execution consistency.

4.3 Timeout Task cancellation

Sometimes, the execution of a task may freeze due to a resource failure or it may never end (e.g., an optimization algorithm not converging to a solution). In these situations, workflow engines require a mechanism to avoid that the whole application gets blocked due to a single task. In our case, we propose to use a timeout mechanism combined with the failure reaction policies described

```
1 @task(time_out=50, on_failure="CANCEL_SUCCESSORS")
2 def task_timeout_example():
3     ...
```

Fig. 4: Task definition with timeout.

above. As in the previous case, this mechanism will be also indicated in the task definition with the *time_out* property as shown in Figure 4. The *time_out* property indicates the maximum duration (in seconds) for a task before being considered failed.

The defined timeout for a task is passed to the runtime worker when it is submitted. The worker sets up a timer according to the specified timeout duration which will send a signal when the timeout is reached. To manage the timeout, a custom signal handler is defined which will throw an exception interrupting the task execution and producing a failure in the task execution. The failure is managed according to the failure reaction policy as indicated above.

4.4 Exceptions in Parallel Distributed Workflows

Another mechanism to treat application failures is the exception. This is supported by most modern programming languages, however supporting this mechanism on parallel workflows executed in distributed environments is not trivial. In the next paragraphs, we will describe how the exception mechanism is implemented in COMPSs. Typically, exceptions are used in the following way: a user defines a *try* code block where some of the statements in the block can throw an exception; if an exception is thrown, the rest of the block execution is cancelled; and if a *catch* or *except* block (depending on the language) is defined, it is executed after catching the exception.

We propose to apply the same concept in parallel distributed workflows as shown in Figure 5a. In this case, we create a task group block (line 9) where some of the tasks invoked in this block can throw a *COMPSsException* during its remote asynchronous execution; this special exception type is defined to differentiate from other exceptions which just produce a task failure. At runtime, during the task group execution, the worker detects when a task throws a *COMPSsException* and sends it back to the master, which cancels the rest of the non-finished tasks of the group and continues the application execution by running the *except* block. Note that the task group definition has an implicit barrier at the end of the code block in order to wait until all the tasks of this block are finished. However, this implicit barrier could limit the maximum parallelism achieved by the application. For instance, if we want to run two independent task groups in a loop, the COMPSs runtime will execute the group of the first iteration and once all the tasks of this group are finished, it will execute the group of the second iteration.

To allow both blocks to run concurrently, developers can follow the approach described in Figure 5b. The implicit barrier can be disabled when defining the

```python
@task()
def task_exception():
  ...
  raise COMPSsException()
...
if __name__ == '__main__':
  ...
  try:
    with TaskGroup("group_name"):
      task_A()
      task_exception()
      task_B()
  except COMPSsException:
      task_C()
  ...
```

(a) Exception management with implicit synchronization

```python
@task()
def task_exception():
  ...
  raise COMPSsException()
...
if __name__ == '__main__':
  ...
  with TaskGroup("group_name", false):
    task_A()
    task_exception()
    task_B()
  ...
  try:
    compss_barrier_group("group_name")
  except COMPSsException:
    task_C()
  ...
```

(b) Exception management with explicit synchronization

Fig. 5: Code examples for remote exception management in parallel workflows.

task group block (line 8), and an explicit barrier can be set by adding a call of the *compss_barrier_group* (line 14). In both cases, tasks of the group will be canceled once the exception is thrown. However, the code area where the exception is thrown to the main code differs depending on the case. For the implicit synchronization case, the try/except block is set after the task group block, while in the explicit case, the exception is thrown in the *compss_barrier_group*, so the try/except block must be set at this point of the code.

5 Evaluation

To validate our proposal, we have applied the failure management mechanism in the following use cases where we have performed several experiments to evaluate the benefits of our approach in terms of productivity and performance.

5.1 BioExcel biobb: Model Protein Mutants workflow

BioExcel[2] is the European Centre of Excellence for provisioning support to academic and industrial researchers in the use of high-performance computing (HPC) and high-throughput computing (HTC) in biomolecular research. The BioExcel was established to provide the necessary solutions for long-term support of the biomolecular research communities: fast and scalable software, user-friendly automation workflows and a support base of expert core developers. In the framework of this project, BSC is developing together with the Institute for Research in Biomedicine (IRB) the *biobb*. The *biobb* is a library of Python wrappers offering a layer of compatibility and interoperability over the

[2] https://bioexcel.eu

BioExcel computational biomolecular tools, such as GROMACS [13]. The *biobb* is enabled by PyCOMPSs for its executions in large scale systems. The Model Protein Mutants workflow has been developed on top of the *biobb* (and runs on top of PyCOMPSs). This workflow can be described as an automated protocol to generate structures for protein variants detected from genomics data. The workflow combines multiple data transformations with invocations to GROMACS.

The first experiment to validate our failure management approach consists of making the protein mutants workflow reliable to application failures. In this experiment, we evaluate the productivity comparing the implementation using our proposed approach with the alternative of coding this feature directly in the application code. Figure 6 shows the main code of the protein mutants workflow and the task dependency graph generated when executing it with two mutations. As we can see, this application is composed of different independent chains of tasks. Therefore, a failure in one of the tasks of the chain invalidates the results of the whole chain. So, the most suitable mechanism for this application pattern is setting the *on_failure* property to *CANCEL_SUCCESSORS* in all tasks. In contrast to this, if we want to observe the same behaviour in this application when it is not supported by COMPSs, developers should modify the application in the way shown in Figure 7, where we have to capture the failure in the task code, and return this as well as modify the main workflow to continue the workflow execution depending on its result. This implementation required to add 87 lines of code and the cyclomatic complexity [10] of the code increased from 2 to 41 (measured with Radon[3]) due to the split of the main loop and the different *if* paths.

[3] https://pypi.org/project/radon/

```
@task(structure=FILE_OUT)
def fix_side_chain(structure):
  # task code
if __name__ == '__main__':
  fix_side_chain("init_struct.pdb")
  for n in range(num_mut):
    mutate(n, "init_struct.pdb", "mutate.pdb")
    pdb2gmx(n, "mutate.pdb", "pdb2gmx.gro", "pdb2gmx.zip")
    editconf(n, "pdb2gmx.gro", "editconf.gro")
    solvate(n, "editconf.gro","pdb2gmx.zip", "solvate.gro",
    ↪  "solvate.zip")
    grompp(n, "solvate.gro", "solvate.zip", "gppion.tpr")
    genion(n, "gppion.tpr", "pdb2gmx.zip", "genion.gro", "genion.zip")
    grompp(n, "genion.gro", "genion.zip", "gppmin.tpr")
    mdrun(n, "gppmin.tpr", "min.gro")
    grompp(n, "min.gro", "genion_top.zip", "gppnvt.tpr")
    mdrun_cpt(n, "gppnvt.tpr", "nvt.gro", "nvt.cpt")
    grompp_cpt(n, "nvt.gro", "nvt.cpt", "genion.zip", "gppnpt.tpr")
    mdrun_cpt(n, "gppnpt.tpr", "npt.gro", "npt.cpt")
    grompp_cpt(n, "npt.gro", "npt.cpt", "genion.zip", "gppmd.tpr")
    mdrun_cpt(n, "gppmd.tpr", "md"+str(n)+".gro", "md"+str(n)+".cpt")
```

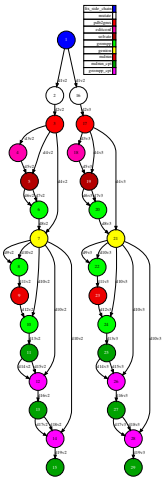

Fig. 6: Protein Mutants workflow code and generated DAG for 2 mutations

```python
#task code
@task(input_file=FILE_IN, output_file=FILE_OUT)
def mutate_pc(input_file, out_file):
    try:
       # original task_code where a failure generated an exception
       return 0
    except :
       return 1
#main workflow
if __name__ == '__main__':
    ...
    for n in range(num_mut):
        result[n] = mutate_pc(n, "init_struct.pdb", "mutate.pdb")
    #The folowing pattern is repeated for invoking next tasks
    for n in range(num_mut):
        result[n] = compss_wait_on(result[n])
        if (result[n] == 0 ):
            result[n] == pdb2gmx_pc(n, "mutate.pdb", "pdb2gmx.gro")
    ...
```

Fig. 7: Cancel successor code alternative.

Apart from the failure reaction policy, some of the GROMACS calls implement optimization algorithms which, depending on the input, might not converge. So, we have set the *time_out* property in *mdrun* and *mdrun_cpt* task definitions. Implementing the same feature in the task codes can be done as shown in Figure 8b and required adding 18 lines of code.

```python
@task
def task_example(out_file):
    try:
       #original task_code
    except :
        import os
        if not os.path.exists(out_file):
            with open('/tmp/test', 'w'):
                pass
```

(a) Failure Ignore code alternative

```python
#Timeout exception
class TimeOutError(BaseException):
    pass
#SIGALRM handler
def task_timed_out(signum, frame):
    raise TimeOutError
#task implementation
@task(...)
def task_example(..., time_out):
    import signal
    signal.signal(signal.SIGALRM,
    ↪  task_timed_out)
    signal.alarm(time_out)
    try:
        #original_code
        signal.alarm(0)
    except TimeOutError :
        ...
```

(b) Time out code alternative

Fig. 8: Failure detection code alternatives.

Another variant of this workflow is done by adding a final task which merges the results in a single graph. In this variant, the *CANCEL_SUCCESSORS* policy is not suitable since a failure will also cancel the merge task. To avoid this, we can change the *on_failure* policy to *IGNORE*, which by default will generate an

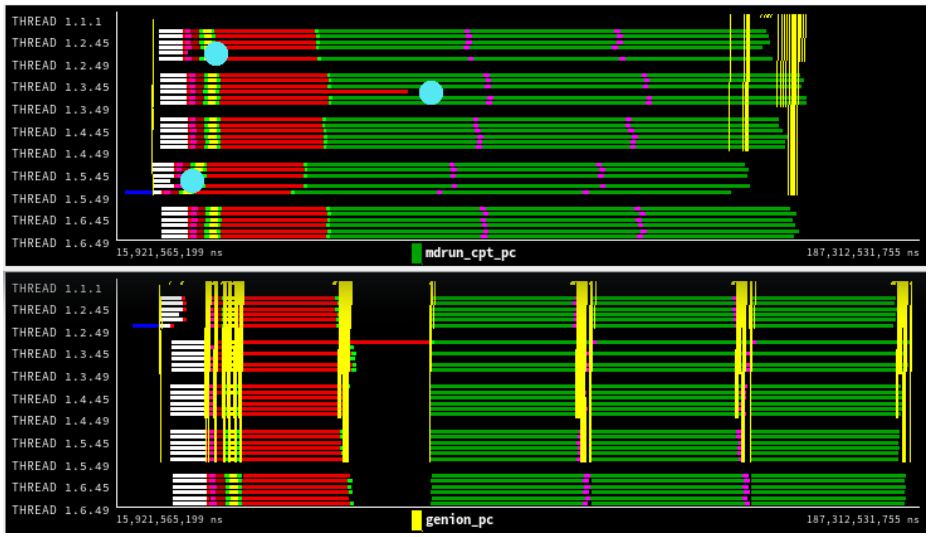

Fig. 9: Mutations workflow execution trace comparison. Each trace shows a timeline of task executions in the different computing resources. Horizontal color lines indicate the different tasks executions (Colors are the same as in Figure 6b). Vertical yellow lines indicate transfers between computing nodes.

empty file per failed result which can be ignored by the merge task. In case the developers have to code this feature in the application, they have to add the code of the figure for each output parameter. In the case of study, it required adding 108 lines of code.

Besides the productivity aspects, the proposed contribution also has a performance impact. Figure 9 shows the execution traces of the protein mutants workflow evaluating 30 mutations, where three of them produce failures. One failure is due to an incorrect mutation, another is due to an incorrect GROMACS configuration, and the last one produces a long execution time in the simulation. The upper trace shows the execution with the proposed failure management implemented in COMPSs and the lower traces show the execution with the coding alternative as explained above. The light blue dots show the points in time where the failures have occurred and we can observe that the successors of these tasks have not been executed. Yellow lines in the traces show data transfers. We can see that the execution with the new approach performs better since it does not require synchronizations and requires less data transfers.

5.2 Machine learning algorithms

Another area of application of the new features presented in this paper has been the dislib library[4] [4], a distributed computing machine learning library

[4] https://dislib.bsc.es

parallelized with PyCOMPSs. Some machine learning algorithms are iterative, where convergence is checked at every iteration step to decide whether the next iteration is necessary. Examples of this iterative behaviour are the K-means and Gaussian Mixture clustering algorithms, and Cascade Support Vector Machines (C-SVM) classification algorithm [8].

When implementing these algorithms in PyCOMPSs, it required a data synchronization in the main code to evaluate its convergence criterion and decide if a new iteration is required. Adding this data synchronization in the main loop implies a task barrier, where the main code waits for all tasks to finish. When running a single algorithm individually, this is not that critical, but when running several at the same time, this synchronization serializes all the executions. Cases where we would like to run multiple algorithms at a time occur in hyperparameter optimization algorithms, like Grid Search or Randomized Search.

In this paper, we have modified dislib's Cascade Support Vector Machine algorithm (C-SVM) in such a way that the evaluation of the convergence criterion is performed in a task. This task raises a *COMPSsException* whenever the convergence criterion is met. This has been combined with the Grid Search algorithm, that fits multiple models with multiple parameters. The objective is to be able to run the multiple models in parallel, which before was not possible due to the synchronization required to check the convergence criterion. Grid Search has been also modified to cancel non executed tasks when a *COMPSsException* is raised during the fitting process of one of the models.

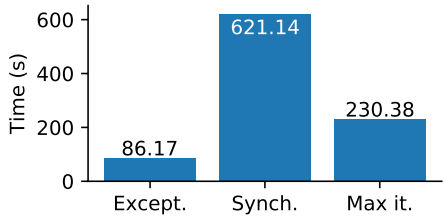

Fig. 10: Execution time of Grid Search with C-SVM using the exceptions mechanism (Except.), without the exceptions mechanism (Synch.), and without checking the convergence (Max it.).

Figure 10 shows the execution time of Grid Search in three scenarios. The first scenario (*Except.*) corresponds to using the exception mechanism presented in this paper to avoid synchronizations in the fitting of C-SVM. The second scenario (*Synch.*) corresponds to not using exceptions and synchronizing after every iteration. The third scenario (*Max it.*) corresponds to running C-SVM for a fixed number of iterations (10) instead of checking the convergence criteria. In all cases, the Grid Search algorithm fits 10 models in total.

We see that although the number of fitted models is low, Grid Search greatly benefits from avoiding convergence checks. Using the exception mechanism achieves

7x speedup over the scenario with synchronizations, and 2.7x speedup over running the models for a fixed number of iterations. In the first case, the improvement is because Grid Search can overlap the fitting of the different models. In the second case, the improvement is due to some models converging in less than 10 iterations. We expect this improvement in execution time to increase further if more than 10 models are trained simultaneously.

6 Conclusion

This paper presents a set of mechanisms to easily manage common application failures in task-based parallel workflows executed in distributed computing environments. We have proposed an extension to the task definition to enable developers to define how the runtime should react if a task of this type is failing or lasting a certain duration (timeout). We have also proposed different policies that are suitable for different types of failures and application patterns. Finally, we have also proposed mechanisms to support the exceptions management in parallel workflows where tasks are asynchronously executed in remote resources.

The proposed mechanisms have been validated with a bioinformatic workflow and a machine learning application where we have seen how the different policies are applied to real workflows and we have compared them with the alternative of coding these features inside the application code. We have observed that these features allow users to add failure management mechanisms without requiring to increase the amount of lines and complexity of the application codes. Moreover, as these mechanisms are automatically managed at runtime concurrently with the application execution, they avoid unnecessary synchronizations and transfers with their corresponding gain in performance.

Acknowledgment

This work has been supported by the Spanish Government (contracts SEV2015-0493 and TIN2015-65316-P), by the Generalitat de Catalunya (contract 2014-SGR-1051), and by the European Commission's Horizon 2020 Framework program through BioExcel Center of Excellence (contracts 823830, and 675728).The research leading to these results has received funding from the collaboration between Fujitsu and BSC (Script Language Platform).

References

1. Afgan, E., Baker, D., Batut, B., Van Den Beek, M., Bouvier, D., Čech, M., Chilton, J., Clements, D., Coraor, N., Grüning, B.A., et al.: The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2018 update. Nucleic acids research **46**(1), 537–544 (2018). https://doi.org/10.1093/nar/gky379
2. Amela, R., Ramon-Cortes, C., Ejarque, J., Conejero, J., Badia, R.M.: Enabling Python to Execute Efficiently in Heterogeneous Distributed Infrastructures

with PyCOMPSs. In: Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing. pp. 1–10. ACM, New York, NY, USA (2017). https://doi.org/10.1145/3149869.3149870
3. Badia, R.M., Conejero, J., Diaz, C., Ejarque, J., Lezzi, D., Lordan, F., Ramon-Cortes, C., Sirvent, R.: COMP superscalar, an interoperable programming framework. SoftwareX **3**, 32–36 (December 2015). https://doi.org/10.1016/j.softx.2015.10.004
4. Álvarez Cid-Fuentes, J., Solà, S., Álvarez, P., Castro-Ginard, A., Badia, R.M.: dislib: Large Scale High Performance Machine Learning in Python. In: Proceedings of the 15th International Conference on eScience. pp. 96–105 (2019). https://doi.org/10.1109/eScience.2019.00018
5. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., et al.: Pegasus, a workflow management system for science automation. Future Generation Computer Systems **46**, 17–35 (2015). https://doi.org/10.1016/j.future.2014.10.008
6. Di Tommaso, P., Chatzou, M., Floden, E.W., Barja, P.P., Palumbo, E., Notredame, C.: Nextflow enables reproducible computational workflows. Nature biotechnology **35**(4), 316–319 (2017). https://doi.org/10.1038/nbt.3820
7. Ejarque, J., Domínguez, M., Badia, R.M.: A hierarchic task-based programming model for distributed heterogeneous computing. The International Journal of High Performance Computing Applications **33**(5), 987–997 (2019). https://doi.org/10.1177/1094342019845438
8. Graf, H.P., Cosatto, E., Bottou, L., Durdanovic, I., Vapnik, V.: Parallel Support Vector Machines: The Cascade SVM. In: Proceedings of the 17th International Conference on Neural Information Processing Systems. pp. 521–528 (2004)
9. Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Álvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., Badia, R.M.: ServiceSs: an interoperable programming framework for the Cloud. Journal of Grid Computing **12**(1), 67–91 (March 2014). https://doi.org/10.1007/s10723-013-9272-5
10. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering **2**(4), 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837
11. Mouallem, P., Crawl, D., Altintas, I., Vouk, M., Yildiz, U.: A fault-tolerance architecture for kepler-based distributed scientific workflows. In: Int. Conference on Scientific and Statistical Database Management. pp. 452–460. Springer (2010). https://doi.org/10.1007/978-3-642-13818-8_31
12. Oliver, H.J.: Cylc (the cylc suite engine). Tech. rep. (2016), http://cylc.github.io/cylc/
13. Pronk, S., Páll, S., Schulz, R., Larsson, P., Bjelkmar, P., Apostolov, R., Shirts, M.R., Smith, J.C., Kasson, P.M., van der Spoel, D., et al.: Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics **29**(7), 845–854 (2013). https://doi.org/10.1093/bioinformatics/btt055
14. Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., Soiland-Reyes, S., Dunlop, I., Nenadic, A., Fisher, P., et al.: The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. Nucleic acids research **41**(W1), W557–W561 (2013). https://doi.org/10.1093/nar/gkt328