Task-Level Checkpointing System for Task-Based Parallel Workflows

Pere Vergés(✉), Francesc Lordan, Jorge Ejarque, and Rosa M. Badia

Department of Computer Sciences, Barcelona Supercomputing Center, Barcelona, Spain
{pere.verges,francesc.lordan,jorge.ejarque,rosa.m.badia}@bsc.es

Abstract. Scientific applications are large and complex; task-based programming models are a popular approach to developing these applications due to their ease of programming and ability to handle complex workflows and distribute their workload across large infrastructures. In these environments, either the hardware or the software may lead to failures from a myriad of origins: application logic, system software, memory, network, or disk. Re-executing a failed application can take hours, days, or even weeks, thus, dragging out the research. This article proposes a recovery system for dynamic task-based models to reduce the re-execution time of failed runs. The design encapsulates in a checkpointing manager the automatic checkpointing of the execution, leveraging different mechanisms that can be arbitrarily defined and tuned to fit the needs of each performance. Additionally, it offers an API call to establish snapshots of the execution from the application code. The experiments executed on a prototype implementation have reached a speedup of 1.9× after re-execution and shown no overhead on the execution time on successful first runs of specific applications.

Keywords: High-Performance Computing · Checkpointing · Task-based programming model · Recovery System · Fault Tolerance

1 Introduction

Supercomputers and cloud computing have become essential tools for researchers to work on their investigations. The amount of data used in scientific applications has dramatically escalated and the computation time required to execute them. Parallelizing applications using multiple networked computers shortens its execution time, making research more manageable. Furthermore, distributing the workload across large infrastructures enables higher levels of parallelism, unreachable when using one single machine.

Using shared distributed infrastructures such as clusters, supercomputers, or the Cloud, usually entails execution time limits and resource quotas – e.g., disk –, increasing the probability of unexpected issues that makes the application unable to complete. There are myriad reasons for that: network disruptions, inability to allocate memory, disk quota violations, issues with the shared file system, exceeding the allowed execution time, etc. Frequent solutions for these problems consist of retrying the failed computation on the same node or changing the host for that part upon failure detection. However, in most cases, the application fails due to the lack of resources. Therefore, either the crash affects the whole system or the queuing system ends the execution. A more complex solution to overcome these shortcomings consists of establishing checkpoints where the application saves its status and data values in persistent data space to avoid the re-computation of previous values on future re-executions of the application.

This article contributes to the current state of the art by proposing and evaluating a system that allows applications developed following a task-based programming model to recover from failures and reduce their re-execution time. Thus, application users will speed up significantly their research on their respective disciplines by avoiding computations that take hours, days, and even weeks while using extensive computing infrastructures.

The proposed system leverages the determinism of tasks to avoid re-executing non-failed tasks in case of breakdown by automatically copying their output as the execution goes on. Performing such copies entails a significant overhead on network and storage operations; the optimal balance for this trade-off between resilience and performance depends on each execution and the preferences of the end-user. To that end, the proposed system combines various mechanisms that systematically select which output values to checkpoint and envisages the customization of these decisions by incorporating mechanisms to define new policies. The user can define these policies by creating arbitrary checkpointing groups of tasks. Besides systematic copies, the system also provides application developers with a method to set up specific points in the application code to checkpoint the execution status.

The article continues by describing the baseline knowledge to understand the details of the presented work in Sect. 2. Section 3 discuss the design and implementation details of the solution. Section 4 evaluates and presents the performance measures that validate the solution's viability with a prototype, Sect. 5, casts a glance over the research already performed on the area. Finally, Sect. 6 concludes this work.

2 Checkpointing Task-Based Workflows

Task-based parallel programming models have become more popular and are a standard solution for creating parallel applications. This popularity is due to their higher development productivity due to their automatic exploitation of the inherent parallelism and, second, their ability to ease the implementation of scientific workflows by combining executions of different applications.

Such models build on the concept of task: a stateless logic executed asynchronously. It processes a specific set of input values to produce some output values. Applications are a combination of tasks where data establishes a dependency relation, defining a workflow. Often represented as a directed acyclic

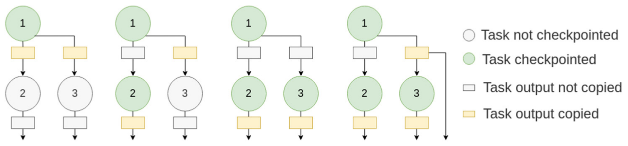


Fig. 1. Diagrams depicting four different situations where different output values are persisted to checkpoint Task 1.

graph, where nodes correspond to tasks, and edges illustrate data dependencies. Executing a task producing a data value will always precede the execution of a task consuming such value. A task will not start its execution until all its input data has been generated by its predecessor tasks.

Runtime systems supporting these programming models know which tasks are ready to execute and fully exploit the parallelism inherent to the application given the available infrastructure. Besides, they apply techniques to increase the application parallelism, such as data renaming to avoid false data dependencies. Instead of keeping a single value of the data, the runtime makes a new copy for each value computed for a datum, thus, enabling a task updating a datum to run ahead of others reading that value. Awareness of all the values that relate to the same data allows the system to consolidate a version and remove the preceding values that will no longer be used.

These runtime systems usually follow an architecture where one of the nodes hosts a process (the master) orchestrating the execution, and the other nodes run a middleware software (worker) that hosts the execution of the tasks. The worker can notice task failures when they terminate abruptly or an exception arises and notifies the failure to the master. When losing the connection with a worker, the master assumes that all the tasks offloaded to that node have failed. After unsuccessfully trying to recover from a task failure, the master terminates the execution, and the whole application fails. Errors on the master would end the execution abruptly. We aim to persist some data values so that the following re-executions recover them, avoiding a partial re-execution.

To that end, this work leverages the stateless, serverless, and determinism properties of deterministic tasks. A deterministic task always produces the same output values regardless of the node and moment it runs, given the same input values. Therefore, persisting all output values of the task beyond the run enables future executions of the application to skip the re-computation of the previous task. This technique is known in the bibliography as task-level checkpointing [10,12]; tasks that will not be re-executed in upcoming runs of the application are checkpointed tasks. Despite building on the task determinism, the presented solution is also valid for applications that exploit randomness in their computations – e.g., Monte Carlo simulations. Our solution design can execute stochastic algorithms. However, the seed of the pseudo-random generator must be treated as another application value to re-create the exact computation as it was in the previous execution and pass it in as another input value for each task.

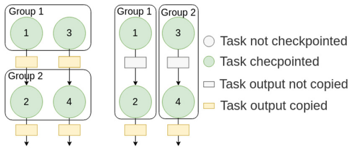


Fig. 2. Task diagram showing the output copies of a task workflow.

Being part of a workflow allows relaxing the conditions to consider a task as checkpointed from having all its output values persisted, to having each of the output values of the task, either persist or ensure that the value into consideration will no longer be used in the future – i.e., all tasks consuming the value have been checkpointed, and new tasks consuming the value cannot be created. Figure 1 depicts four different situations where Task 1, which produces two output values, is checkpointed by persisting different values. The first situation shows the original practice that persists all the output values to checkpoint the task. In the second case, it is unnecessary to persist one of the output values since checkpointing Task 2 ensures that the value will not be needed in the future. In this specific example, the second value persisted; however, as shown in the third situation, checkpointing Task 3 would have a similar effect and avoid persisting the value. The last case shows a slightly different situation, where the second output value is consumed by Task 3, but the output value could still create new tasks consuming the value. Therefore, the value needs to be saved despite Task 3 being checkpointed.

Whereas some task-based programming models define a static workflow before execution and, perhaps, scale the number of tasks according to the size of the processed data, some other models are more flexible and allow adapting the whole workflow depending on task results. In the latter case, the component spawning tasks require a mechanism to synchronize the results of some tasks to evaluate them and continue with the dynamic generation of tasks (e.g., to check convergence in a loop). These synchronization values need to be persisted and cannot be deleted even if future executions have no tasks consuming them to enable the re-creation of the same workflow.

Two essential aspects that automatically capture the progress of any application are the execution time and the amount of already finished tasks; hence, the proposed system implements two mechanisms building on them. The first mechanism, Periodic Checkpoint, registers the finished tasks and the produced data values and periodically triggers data operations to persist the output values computed until execution, avoiding unnecessary values as depicted in Fig. 1. The Finished Tasks mechanism behaves similarly, but the trigger is the completion of N tasks. The more frequent the checkpointing is, the fewer possibilities to avoid persisting values; however, the longer the checkpointing period is, the more tasks will need to be re-executed if the application fails.

Checkpointing can achieve an optimal balance between one execution performance and resilience by defining arbitrary checkpointing groups. The system persists only the final output values of each group and dismisses the values corresponding to intermediate versions or deleted data. Figure 2 illustrates an example of how arbitrarily grouping tasks impacts the amount of checkpointed values. The group distribution in the leftmost part of the figure depicts a case where groups are done according to the depth level in the graph. The group distribution shows that, for this specific workflow, creating groups according to data dependencies reduces the amount of persisted values from four to two by avoiding making persistent the intermediate value between them.

3 Solution Design and Implementation

Finding an optimal selection of values to checkpoint requires deep knowledge of the application workflow, the host infrastructure, the size of the problem solved with the application, and the current progress of each execution. The runtime system orchestrating the workflow execution is the only point where all this knowledge meets; therefore, there is the best place to select which values to persist. To that end, this article aims to provide the runtime system with a Checkpoint Manager (CM) component encapsulating the automatic management of the checkpointing for the execution. Figure 3 depicts an overview of the architecture of the proposed system.

The Runtime System (RS) notifies the CM of the different events required to checkpoint tasks. When the application generates a new task, the RS queries the CM whether the task was checkpointed in a previous execution. If it does, the task execution is skipped, and the checkpointed values are restored as if the task computed them for their later use in a non-checkpointed task or a synchronization point. Otherwise, if the task has not been checkpointed, the checkpoint manager registers its existence.

The RS also notifies the CM of other execution events such as finalizations of tasks, indicating the location of output values, accesses to synchronization values, and value deletions. With that information, the CM can know the values needed to recreate the workflow, request their persistence, or order their deletion when they will not be involved in future tasks or access synchronization points to minimize the I/O usage.

The CM component implements an engine supporting all the checkpointing mechanisms described in Sect. 2. Policymakers can combine them to create

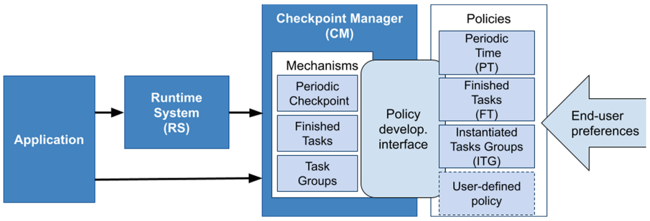


Fig. 3. Overview of the proposed checkpointing system

highly-efficient complex tactics to checkpoint applications fitting the specifics of each application. For that purpose, the CM offers an interface (Policy development interface) to customize each mechanism's behavior properly. The Periodic Time (PT) and Finished Tasks (FT) policies leverage the periodic checkpoint and finished tasks mechanisms while disabling the others. The application end-user can establish the period or the number of finished tasks to trigger them. Checkpointing some data values might have a cost higher than their re-computation; ignoring these values would improve the system's performance; similar to the period and the number of tasks, the system also allows indicating a set of tasks to be ignored by them.

As mentioned in the previous section, application-tailored policies rely on the task group mechanism. Upon task detection, the CM assigns the task to a group according to the selected checkpointing policy. To decide the group, the policy developer can use any information available at instantiation-time, e.g., the number of tasks, operation to perform, accessed data values, or preceding tasks. The Instantiated Tasks Groups (ITG) policy gathers tasks in N-sized groups according to their creation order. When the CM resolves a group closure – i.e., all tasks of the group have already been instantiated –, it determines the final output values of the group by analyzing the data accesses of all the tasks within the group. The RS requests the necessary operations to persist those values that have already been computed. For those that have not been generated, the RS monitors the task generation of each one of them. Upon its completion, requests the necessary operations to persist them.

Moreover, efficient checkpointing requires a deep understanding of the application. To ensure a certain quality of experience, application developers may not want to leave the end-user decisions about checkpointing in the hands of the end-user. To that end, the CM includes a mechanism to order snapshots of the current status of the execution from the application code. The runtime system will persist the useful output values of all the tasks until that point.

To affect the application execution minimally, the CM performs all the persistence operations asynchronously in a background thread with a lower priority and limits the maximum number of ongoing operations in parallel.

4 Evaluation

To validate the proposed design and evaluate its performance, we conducted several experiments aiming at (1) quantifying the overhead of the system when the application does not fail, (2) measuring the speedup when recovering from a failure, and assessing the impact of customizing the policies (3) skipping the checkpointing of some tasks and (4) developing application-tailored policies.

To that end, a prototype of the CM has been implemented and integrated into the COMPSs/PyCOMPSs runtime [4,5] and its performance has been evaluated when running four different applications: K-Means, PMXCV19, Principal Component Analysis (PCA) and Matrix Multiplication (Matmul).

Table 1. Execution time and relative overhead (baseline: NC) using different policies. The policy with the lowest overhead is highlighted with green background.

	NC	ITG	FT	PT
K-Means	220.12 s	222.56 s (1%)	229.34 s (4.5%)	228.44 s (4%)
PMXCV19	33 m	33.9 m (2.7%)	34.1 m (3.3%)	33.6 m (1.8%)
PCA	883.13 s	1075.13 s (21.7%)	1026.21 s (16.1%)	1284.07 s (45.4%)

K-Means[1] (2152 tasks) is a clustering algorithm that identifies K clusters within the input data. The algorithms start with K randomly generated centers. Iteratively, values are assigned to the closest center. These centers are recomputed using the data assigned to them. This process lasts until the centers converge, and their position is not updated. This application has two parts: data generation and center convergence.

PCA (see footnote 1) (685 tasks) is a dimensionality reduction algorithm that computes the principal components of a collection of points to use them to perform a change of data basis using only the first few principal components. It is often used to perform data analysis for predictive models.

PMXCV19[2] (2027 tasks) evaluates changes in the binding affinity between SARS-Cov-2 Spike protein and Human ACE2 (hACE2) receptor using the PMX algorithm [9]. It runs a large series of short Molecular Dynamic simulations executed using GROMACS.

Matmul (64 tasks) implements a blocked matrix multiplication. The resulting workflow consists of several chains of tasks corresponding to all tasks updating the same output block.

The presented results run using two nodes of the MareNostrum 4 supercomputer – each equipped with two 24-core Intel Xeon Platinum 8160 at 2.1 GHz. and 98 GB of main memory – interconnected with a Full-fat tree 100Gb Intel Omni-Path network.

4.1 Checkpointing Overhead

This experiment aims to measure the overhead induced by the checkpointing system when the application (K-Means, PMXCV19, and PCA) successfully finishes. To that end, a run with no checkpointing (NC) is compared to runs using different policies: PT (15-second interval), FT (every 10 finished tasks), and ITG (grouping every 10 instantiated tasks).

The results in Table 1 show the importance of adapting the checkpointing policy depending on the application being executed to minimize the time overhead. With the right policy, the checkpointing system overhead can be negligible depending on the application, with only a 1% of added time. However, picking the wrong policy may entail significant overheads, in the case of PCA, choosing PT over FT may add a 29.3% of overhead.

[1] Implementation with PyCOMPSs distributed within the dislib library [1].

[2] Implementation with PyCOMPSs offered as a BioExcel Building Blocks (BioBB) [3].

Regardless of the policy, picking the appropriate granularity for each policy has a significant impact. Table 2 shows the execution time and relative overhead of each application when running with the policy with a better result in Table 1, set up with different granularities: K-Means runs ITG with groups of 10, 50, and 100 tasks; PMXCV19, PT with 15, 30 and 60-second intervals; and PCA, FT triggering the checkpoint every 10, 50 and 100 completed tasks.

Table 2. Execution time and overhead (baseline: NC) using different granularities for the best policy in Table 1

	Fine-grain	Medium-grain	Coarse-grain
Kmeans (ITG)	222.56 s (1.1%)	244.25 s (11%)	264.36 s (20%)
PMXCV19 (PT)	33.6 m (1.8%)	33 m (0%)	33.1 m (0.3%)
PCA (FT)	1026.21 s (16.1%)	1016.20 s (15%)	1103.94 (24.9%)

Table 2 shows that balancing the checkpoint granularity is needed. Although coarser granularities reduce the number of copies, they can generate I/O-bandwidth peaks that may decrease performance.

4.2 Recovery Speedup

The second experiment aims to measure the speedup of an application when the application fails on the first execution and the checkpointing system recovers the state in a subsequent run. For that purpose, we forced an error when the application reached a certain point of the execution (For the Kmeans we chose the 8th iteration, PXMCV19 we make it fail at min 32 of the execution, finally at PCA we added an exception near the end of the fit function) and measured the duration of failed execution plus the time to finish the subsequent execution using different granularities – defined in Sect. 4.1 – for the best-performing policy for each application. Table 3 contains the obtained times and the speedup of the recovery compared to the same process when no checkpoint is enabled. The K-Means and PCA applications show that despite the overhead, more frequent checkpointing enables a faster recovery time due to the fewer tasks being recomputed on the recovery. The PMXCV19 application performs better with a medium granularity. However, the recovery difference with other granularities is insignificant.

Table 3. Failure, recovery execution time and speedup (baseline: No Checkpoint) using different granularities for the best policy in Table 1.

	No Checkpoint		Fine-grain			Medium-grain			Coarse-grain		
	1st Exec	2nd Exec	1st Exec	Recov.	SpeedUp	1st Exec	Recov.	SpeedUp	1st Exec	Recov.	SpeedUp
Kmeans (ITG) (s)	208.22	221.5	216.96	27.1	1.76x	232.38	25.83	1.39x	254.26	27.14	1.52x
PMXCV19 (PT) (m)	32	33	32	3.1	1.85x	32	2.3	1.89x	32	3.3	1.84x
PCA (FT) (s)	877.99	883.13	1026.21	187.30	1.45x	1016.20	861.26	0.93x	1103.94	855	0.89x

4.3 Avoid Checkpointing Tasks

This third experiment measures the impact of avoiding the persistence of the significant values computed by short tasks. The K-Means application has a pattern composed of two partially overlapped phases: the data set generation and the iterative center convergence. The experiment compares the behavior of a K-Means execution that fails on its 8th convergence iteration. Afterward, it is re-launched, disabling the checkpointing, enabling checkpointing with the FT policy (10-task granularity) for all the tasks and the same policy but disabling the checkpointing of those tasks corresponding to the data set generation phase.

Figure 4 depicts the traces of the failed (left) and recovery (right) executions for the no checkpointing (top), all-tasks checkpointing (middle) and generation-dismissed checkpointing (bottom) configurations. The blue tasks correspond to dataset-generating functions, and each batch of white tasks corresponds to a convergence iteration.

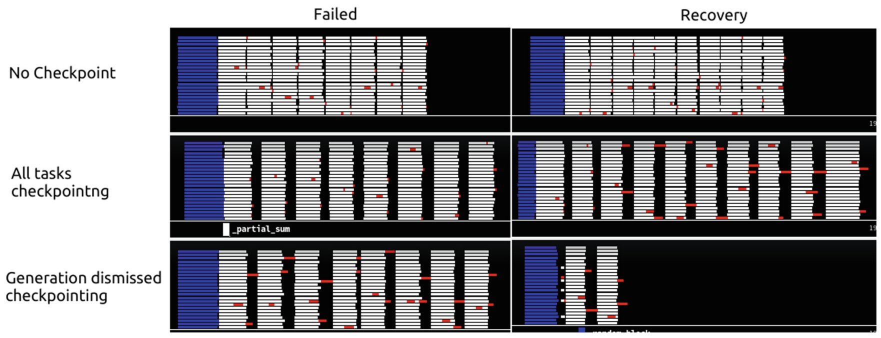


Fig. 4. K-Means execution traces without checkpointing (top), checkpointing all tasks (middle), and generation disabled checkpointing (bottom)

The traces of the first (failed) execution illustrate the effect of the I/O overhead due to the checkpointing. Limiting the number of concurrent checkpointing operations makes the overhead on both executions performing checkpointing similar regardless of the difference in the total number of checkpointed values. However, during the first part of the execution, the CM has no time to persist all the outputs of the generation phase. Thus, it must recompute part of them even if the checkpointing is enabled. The overall execution time grows from 222.99 s when checkpointing is disabled to 282.02 s (0.79× speedup) when the CM checkpoints all the tasks – 126.07 s on the first execution and 159.95 on the recovery. When disabling the checkpointing for the dataset generation tasks, the CM can keep up with the execution progress and avoid most of the tasks' re-execution in the recovery. In this case, the execution time shrinks to 175.75 s (1.27×) – 124.92 s on the initial run and 50.83 on the recovery.

4.4 Customized Policies

The last experiment aims to illustrate the impact of using customized policies leveraging the task groups mechanism on the number of persisted values. To that end, the experiment measures the number of persisted values when using the Matmul application to multiply two 4-by-4-block matrices, and the checkpointing system adopts two custom policies.

Each Matmul run generates a total of 64 tasks forming 16 chains – 1 per output block – of 4 tasks each. The algorithm iterates on all the blocks of the result matrix, instantiating all the tasks updating the block; from a graph point of view, the algorithm generates tasks in a depth-first manner. The custom policies used in the experiment create checkpointing groups of up to two tasks. The first policy (Same-depth policy) groups two tasks from the same depth level, and the second (Same-chain policy) groups two subsequent tasks from the same chain. Listings 1.1 and 1.2 respectively contain the implementation of the function assigning a task to a group for each policy.

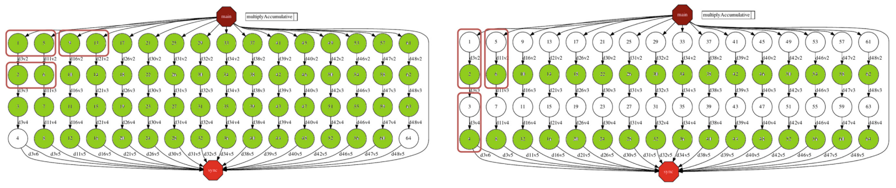
Listing 1.1. Same-Depth

```
void assignTaskToGroup(Task t){
  int id = t.getId();
  int mod = id % 4;
  mod = mod == 0 ? 4 : mod % 4;
  int gId = 1+id/8+((mod-1)*8);
  TaskGroup group=groups.get(gId);
  group.addTask(t);
  if (group.size()==2){
    group.close();
  }
}
```

Listing 1.2. Same-Chain

```
void assignTaskToGroup(Task t){
  int id = t.getId();
  int gId = ((int) id/2)+1;
  TaskGroup group=groups.get(gId);
  group.addTask(t);
  if (group.size()==2){
    group.close();
  }
}
```

Figure 5a illustrates the graph of a run using the Same-Depth policy. Green-colored tasks depict those tasks whose output values are persisted by the CM; the output of white-colored tasks is not persisted. It is appreciated how all output data is saved except for two tasks, for which the checkpoint did not have time to copy the results before the execution finished. Thus, the Same-Depth policy persists in 62 data values. Creating groups that take into account the data dependency allows the Same-Chain policy to avoid persisting intermediate



(a) Same-Depth policy

(b) Same-Chain policy

Fig. 5. Matmul's task graphs with both policies; tasks whose output is persisted by the CM are depicted in green. (Color figure online)

values (the output of tasks on the odd rows). Thus, the CM checkpoints 32 values, and there is time to persist as depicted in Fig. 5b.

Application-tailored policies persist fewer values and, thus, reduce the overhead. Avoiding the bottleneck of the concurrent operation allows checkpointing more advanced states of the execution and, therefore, faster recovery executions and lower disk usage.

5 Related Work

The most popular approach for facing failures using task-based parallel models, consists on re-executing the failed task several times, either in the failed node or in a different one. It does not have recovered in case the execution crashes. As instance we have Dask [11], when one node has network connection problems, it will reroute the computation to a different node. However, if the failed node is one with relevant results or the scheduling fails, all results previously executed will have to be re-computed again by other nodes. Additionally, there are PARSL [2], and COMPSs [8], which have some mechanism to retry tasks in case of failure, and even keep with the execution regardless of some tasks have failed. However, all these programming models do not have a recovery system that re-executes the application avoiding the computations performed in the failed execution.

Few workflow environments implement a recovery system that recovers a failed execution into a new one, avoiding re-computing the whole workflow. One of these systems is Pegasus [7]. In Pegasus, once one of the jobs surpasses the number of established failures, it will be marked as failed, and eventually, the whole application will crash. The recovery procedure is to mark nodes in the DAG that succeeded as finished. This allows the user to correct the problem by fixing the errors of ill-compiled nodes, incorrectly compiled codes, inaccessible clusters, etc. This way, the application can restart from the point of failure. Another environment with a recovery system is Legion [6], a data-centric task-based parallel programming system for distributed heterogeneous architectures. This system uses speculation, which allows them to discover non-predicated tasks while the system waits for predicates to finish. If there is a mis-speculation, the runtime must calculate the dependent operations that have been affected. Afterward will reset all operations impacted by it. This process is done recursively, so it could happen that when trying to recover from a failure, it would restart the whole execution from scratch. This checkpointing approach allows it to be performed independently on individual tasks without synchronization.

6 Conclusion

This article proposes a recovery system for task-based programming models. The introduced system copies task outputs to avoid re-executing the computed tasks in the previous execution run. The checkpointing system offers a checkpointing manager that, apart from encapsulating automatic checkpointing, has an interface that allows the end-user to create arbitrary tasks to checkpoint, enabling a specific checkpoint workflow formation to minimize the execution overhead. Moreover, the checkpointing implementation has been propounded to decrease the number of data copies by avoiding the copy of intermediate data values. The flexibility in creating different checkpointing workflows helps reduce the overhead, which can be as minimal as 0% of the execution time and allows for a faster recovery, achieving up to a 1.9× speedup. Additionally, the proposed solution offers an API call that establishes snapshots of the execution in the application code.

Acknowledgements. This work has been supported by the Spanish Government (PID2019-107255GB), by Generalitat de Catalunya (contract 2017-SGR-01414), and by the European Commission through the Horizon 2020 Research and Innovation program under Grant Agreement No. 955558 (eFlows4HPC-project). This work has partially been co-funded with 50% by the European Regional Development Fund under the framework of the ERFD Operative Programme for Catalunya 2014–2020.

References

1. Cid-Fuentes, J.Á., et al.: dislib: large scale high performance machine learning in python. In: 2019 15th International Conference on eScience (eScience) (2019)
2. Babuji, Y., et al.: Parsl: pervasive parallel programming in python. CoRR (2019)
3. Andrio, P., et al.: Bioexcel building blocks, a software library for interoperable biomolecular simulation workflows. Sci. Data **6**, 169 (2019)
4. Badia, R.M., et al.: Comp superscalar, an interoperable programming framework. SoftwareX **3**, 32–36 (2015)
5. Badia, R.M., et al.: Enabling python to execute efficiently in heterogeneous distributed infrastructures with pycompss. In: PyHPC 2017. Association for Computing Machinery, New York (2017)
6. Bauer, M., et al.: Legion: expressing locality and independence with logical regions. In: SC 2012: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2012)
7. Deelman, E., et al.: Pegasus, a workflow management system for science automation. Future Gener. Comput. Syst. **46**, 17–35 (2014)
8. Ejarque, J., Bertran, M., Cid-Fuentes, J.Á., Conejero, J., Badia, R.M.: Managing failures in task-based parallel workflows in distributed computing environments. In: Malawski, M., Rzadca, K. (eds.) Euro-Par 2020. LNCS, vol. 12247, pp. 411–425. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57675-2_26
9. Quan, O., Xu, H.: The study of comparisons of three crossover operators in genetic algorithm for solving single machine scheduling problem (2015)
10. Qureshi, K., Khan, F., Manuel, P., Nazir, B.: A hybrid fault tolerance technique in grid computing system. J. Supercomput. **56**, 106–128 (2011)
11. Rocklin, M.: Dask: parallel computation with blocked algorithms and task scheduling, pp. 126–132 (2015)
12. Vanderster, D., Dimopoulos, N., Sobie, R.: Intelligent selection of fault tolerance techniques on the grid, pp. 69–76 (2007)