Achieving dynamic workflow management system by applying provenance based checkpointing method

E. Kail[1], P. Kacsuk[2,3] and M. Kozlovszky[1,2]
[1] Óbuda University, John von Neumann Faculty of Informatics, Biotech Lab
Bécsi str. 96/b., H-1034, Budapest, Hungary
[2] MTA SZTAKI, LPDS, Kende str. 13-17, H-1111, Budapest, Hungary
[3] University of Westminster, 115 New Cavendish Street, London W1W 6UW
{kail.eszter, kozlovszky.miklos}@nik.uni-obuda.hu,
kacsuk@sztaki.mta.hu

Scientific workflows are data and compute intensive thus may run for days or even for weeks on parallel and distributed infrastructures such as HPC systems and cloud. In HPC environment the number of failures that can arise during scientific workflow enactment can be high so the use of fault tolerance techniques is unavoidable. The most frequently used fault tolerance techniques are job replication and checkpointing. While job replication is based on the assumption that the probability of single failures is much higher than of simultaneous failures, the checkpointing saves certain states and the execution can be restarted from that point later on. The effectiveness of the checkpointing method depends on the checkpointing interval. Common technique is to dynamically adapt the checkpointing interval. In this work we give a brief overview of the different checkpointing techniques and propose a new provenance based dynamic checkpointing method.

I. INTRODUCTION

Scientific workflows being data and compute intensive may require long execution time, which can even last for weeks. During such long intervals it is inevitable to adapt to the dynamically changing environment which can be caused by unwanted input data, crash faults or network problems. In of our earlier works [4] we defined the main requirements of dynamic workflow execution systems as: the ability to react to or to handle unforeseen scenarios raised during the workflow enactment phase, to adapt to new situations, to change the abstract or concrete workflow model or to give faster execution and higher level performance according to the actual environmental conditions and intermediary results. In our other work [2] we have defined the three main areas of dynamism which are optimization of the workflow execution according some criteria, user-steering (user or administrator interaction during execution) [3] and fault tolerance behavior. In this work we investigated fault tolerance behavior.

Fault tolerance is the ability of a system to perform its functions even in the presence of a failure. There are two main groups of failures that could arise during enactment. The first group includes the crash faults or fail-stop faults which may come with faulty system components that result in complete data loss. The other group consists of byzantine faults which result the system components to behave unpredictably and maliciously. Byzantine failures can occur, e.g., due to software bugs, (transitional or permanent) hardware malfunction, or malicious attack. In our work we consider only crash faults where the complete state of the actual task and environment must be restored.

Fault tolerance policy can be reactive and proactive. While the aim of proactive techniques is to avoid situations caused by failures by predicting them and taking the necessary actions, reactive fault tolerance policies reduce the effect of failures on application execution when the failure effectively occurs. There are several solutions in the literature for fault tolerant behavior and other complementary methods in its connected fields [1].

To achieve fault tolerant behavior the most widely adopted methods are:

- Checking and monitoring which is a key factor in failure detection.
- Checkpointing and resubmission where the system state is captured and saved based on predefined parameters (i.e.: time interval, number of instructions) and when the system undergoes some kind of failure the last consistent state is restored and computation is restarted from that point on.
- Replication where critical system components are duplicated using additional hardware or with scientific workflows critical tasks are replicated and executed on more than one processor. We can differentiate active and passive replication. Passive replication means that only one primary processor is invoked in the execution of a task and in the case of a failure the backup ones take over the task processing. In the active form all the replicas are executed at the same time and in the case of a failure the replica can continue the execution without intervention. The idea behind task replication is that replication size r can tolerate r-1 faults while keeping the impact on the execution time minimal. We call r the replication size. While this technique is useful for time-

critical tasks its downsides lies in the large resource consumption, so our attention is focused on mainly checkpointing methods in this work.

We propose a new checkpointing algorithm, that monitors the resources and dynamically adjust the checkpointing interval based on the task's dependency Factor and the already occurred failures.

In the proposed algorithm there is no need to take global checkpoints of the workflow, and therefore there is no need of synchronization of any kind (based on time or based on communication channels between the processors). The parallel threads of the workflow may run on different type of computing infrastructures (for example on virtual machines of different cloud providers) therefore it would be a complex challenge to solve the synchronization between them. The new proposal is based on provenance support. Provenance carries information about the source, origin and processes that are involved in producing data. The main target of collecting provenance support is to provide reusability and reproducibility among a scientist's community but provenance support can provide users, scientists, workflow developers and administrators with wide range of services. For example provenance can also support fault tolerant behavior by providing statistics about historical executions, such as failure rates or distribution and by storing the intermediary results generated by each tasks of the workflow.

Our paper is organized as follows. After the introduction we give a brief overview about existing checkpointing methods, and in chapter III we introduce our checkpointing method. After a brief conclusion the bibliography closes our work.

II. RELATED WORK

Concerning dynamic workflow execution fault tolerance is a very important issue and checkpointing is the most widely used methods to achieve fault tolerant behavior. We investigated the different algorithms in order to give a brief overview of them.

According to the level where the checkpointing occurs we differentiate application level checkpointing, library level checkpointing and system level checkpointing methods. Application level checkpointing means that the application itself contains the checkpointing code. The main advantage of this solution lies in the fact, that it does not depend on auxiliary components however it requires a significant programming effort to be implemented while library level checkpointing is transparent for the programmer. Library level solution requires a special library linked to the application that can perform the checkpoint and restart procedure. System level solution can be implemented by a dedicated service layer that hides the implementation details from the application developers but still give the opportunity to specify and apply the desired level of fault tolerance [5].

From another perspective we can differentiate coordinated and uncoordinated methods. With Coordinated checkpointing (synchronous) the processes will synchronize to take checkpoints in a manner to ensure that the resulting global state is consistent. This solution is considered to be domino-effect free. With uncoordinated checkpointing (independent) the checkpoints at each process are taken independently without any synchronization among the processes. Because of the absence of synchronization there is no guarantee that a set of local checkpoints result in having a consistent set of checkpoints. It may lead to the initial state due to domino-effect.

The frequency of the checkpointing interval also imposes many opportunities in checkpointing algorithms. Young in [6] has already in 1974 defined his formula for the optimum periodic checkpoint interval which is based on the checkpointing cost and the mean time between failures (MTBF) with the assumption that failure intervals follow an exponential distribution.
Sheng et al in [7] has also derived a formula to compute the optimal number of checkpoints for jobs executed in the cloud. His formula is generic in a sense that it does not use any assumption on the failure probability distribution.

The drawback of these solutions lies in the fact that the checkpointing cost can change during the execution if the memory footprint of the job changes, or depending on network reachability issues or when the failure distribution changes. Thus static intervals may not lead to the optimal solution. By dynamically assigning checkpoint frequency we can eliminate unnecessary checkpoints or where the danger of a failure is considered to be severe it can introduce extra state savings.

Meroufel and Belalem [8] proposed an adaptive time-based coordinated checkpointing technique without clock synchronization on cloud infrastructure. Between the different VMs jobs can communicate with each other through a message passing interface. One VM is selected as initiator and based on timing it estimates the possible time interval where orphan and transit messages can be created. There are several solutions to deal with orphan and transit messages, but most of them solve the problem by blocking the communication between the jobs during this time interval. However blocking the communication increases the response time and thus the total execution time of the workflow which can lead to SLA violation. In Meroufel's work they avoid blocking the communication by piggybacking the messages with some extra data so during the estimated time intervals it can be decided when to take checkpoint or logging the messages can resolve the transit messages problem.

The initiator selection is also investigated in Meroufel and Belalem's another work [9] and they found that the impact of initiator choice is significant in term of performance. They also propose a simple and efficient strategy to select the best initiator.

Sheng at al also propose a new adaptive algorithm to optimize the impact of checkpointing regarding the checkpointing or restarting costs in [7].
Theresa et al in their work [10] propose two dynamic checkpoint strategies: Last Failure time based Checkpoint Adaptation (LFCA) and Mean Failure time based Checkpoint Adaptation (MFCA) which takes into account the stability of the system and the probability of failure concerning the individual resources.

To the best of our knowledge there does not exist an adaptive algorithm that takes into account the effect of the failure occurring on a task on the execution time of the whole workflow.

III. PROPOSED MODEL

A. Environmental Conditions

- the system resources are monitored and failures can be detected as soon as possible, therefore the fault detection time (t_f) does not add high latency to the overall makespan of the workflow execution. (t_f=0)
- Task A_j cannot be started before it has received the output from all its predecessors and the results of Task A_i can only be sent to its successor tasks after the task has been finished.
- There is an ideal case so that tasks can be executed as soon as the results from the predecessor tasks are ready and available. The system resources are inexhaustible in number, so the system can allocate the required number of resources to execute all the tasks parallel that are independent from each other.
- The system supports the collection of provenance data, therefore the intermediary results generated by the individual tasks are saved and in case of failure they can be easily retrieved. Thus there is no need to take checkpoints at the end of the tasks, and there is no need to take global checkpoints, since in the case of failure only the effected task should be rolled back.
- The system also support provenance data about failure statistics, so the probability of failures for a certain period of time is available for each resource component taking into account the aging factor as well.

B. General notation

Workflows in general and also scientific workflows are represented as directed acyclic graphs (DAG) $W = (N, E)$, where the nodes (N) represent the computational tasks or jobs and the directed edges (E) represent the dependency between them. The dependency can be data dependency, and control dependency. In the former case the output of a Task A_i gives the input of a Task A_j if there exists an $A_iA_j \in E$ directed edge in the workflow. The control dependency describes the precedence of the tasks: If an A_iA_j directed edge exists in the workflow, then the execution of task A_i must precede the execution of task A_j in time. Here follows a list of the most frequently used symbols in this paper:

T_c the optimal checkpointing interval,
C is the checkpointing cost
X is optimal number of checkpoints during the execution of a task
$T(A_i)$ is the execution time of task A_i
T_f is the mean time between failures (MTBF)
$E(Y)$ is the expected number of failures during the execution of a task
t_l is the loading time, to restore the last saved checkpoint state,
t_f is the fault detection time, the time to detect the failure
A_0 is the first or entry task of the workflow.

C. Algorithm

The primary goal of this algorithm is to minimize the effect of the checkpointing overhead (time, resource) while still keeping to the soft-deadline of the workflow and the performance level at a satisfactory level.

Young [6] and Sheng [7] have already proved that the optimal checkpointing interval can be computed by (1) and (2). In both cases the fault detection time is considered t_f=0. Equation (2) is a more general form of Young's formula, because it does not depend on any probability distribution, unlike Young's (1) formula which needs to assume that failure intervals follow an exponential distribution.

$$T_c = \sqrt{2CT_f} \tag{1}$$

$$X = \sqrt{\frac{T(A_i) \cdot E(Y)}{2 \cdot C}} \tag{2}$$

In our proposed algorithm we use (2) as a starting point to compute the checkpointing intervals. The main idea is that there is a dependency factor between the tasks. Namely if a failure occurs during the execution of a task A_i then it not only has a local effect on the task itself, but has a global effect also on the whole workflow concerning the execution time. Since if a failure occurs during the execution of task A_i then it has to be re-executed from the last checkpoint. It means the execution of the task ends later, so it may cause all of the successor tasks of task A_i to wait for the results. This can result the whole workflow execution to last longer.

We define local cost (3) of a failure on task A_i which is the execution time overhead of a task when during execution one failure occurs.

$$C_{local} = \frac{\overline{ET(A_i)} + \frac{T_c}{2} + t_l}{ET(A_i)} \quad (3)$$

We define global failure cost (4) of a task A_i: which is the execution time overhead of the whole workflow, when one failure occurs during Task A_i

$$C_{global} = \frac{\frac{T_c}{2} + t_l + rank(A_i) + brank(A_i)}{rank(A_0)} \quad (4)$$

where (5) and (6) are classic formulas that are used in tasks scheduling [12] [11]. Basically the rank() function is the critical path from task A_i to the last task, and can be computed recursively backward from the last task. The brank() value is the backward rank value from task A_i backward to the entry task A_0. It is the longest distance from the entry task to task A_i excluding the computation cost of the task itself.

$$rank(A_i) = ET(A_i) + \max_{A_j \in succ(A_j)} rank(A_j) \quad (5)$$

$$brank(A_i) = \max_{A_j \in pred(A_i)} \{brank(A_j) + ET(A_j)\} \quad (6)$$

It can also be calculated recursively downward from task A_0.

Before executing a task A_i, C_{global} can be evaluated. If $C_{global} < 1$ then one failure occurring during execution of task A_i does not add extra latency to the total execution time of the workflow so in that case T_c can be increased until $C_{global} = 1$. From that we get:

$$T_c = 2 \cdot (rank(A_0) - t_l - rank(A_i) - brank(A_i)) \quad (7)$$

Based on the global failure cost and using the assumption that when faults occur during a checkpoint interval (between two consecutive checkpoints) the expected average time loss is half of the checkpointing interval. This is the average time to re-execute the task from the latest checkpoint.

If there occurred already failures during the actual task, or during the predecessors of the actual task, then it may be possible, that default checkpointing intervals cannot be increased, because the cumulative overhead of the occurred faults can negatively affect the whole workflow execution time.

To take earlier faults into account we need information about the realistic execution time of the tasks. With provenance support the real execution time can be obtained and it can be substituted in (7) in place of the brank value. The $C_{global} < 1$ inequality gives the answer whether the checkpointing interval can be increased without lengthening the whole workflow execution time.

IV. CONLUSION

In this paper we investigated the different checkpointing techniques, which are the most widely used proactive fault tolerant methods. We gave a brief overview of the different checkpointing perspectives with special attention on those solutions where the checkpointing intervals are periodic or it changes adaptively during execution. We proposed a provenance based dynamic algorithm that takes into account the global cost of a failure occurring during the execution of a task, and depending on this value can adjust the checkpointing interval in order to eliminate blind checkpoints while still maintaining soft deadlines. In our future work we would like to deeper investigate the various fault tolerant methods for crash faults or even for byzantine faults.

REFERENCES

[1] Bala, Anju, and Inderveer Chana. „Fault tolerance-challenges, techniques and implementation in cloud computing". IJCSI International Journal of Computer Science Issues vol. 9, 2012

[2] Kail, E., Bánáti, A., Kacsuk, P., Kozlovszky, M.: Provenance based adaptive and dynamic workflows. In: 15th IEEE International Symposium on Computational Intelligence and Informatics, pp 215-219, IEEE Press, Budapest, 2014

[3] E. Kail, P., Kacsuk, M. Kozlovszky.: "A Novel Aproach to User-steering in Scientific Workflows" In Proceedings of CGW'14, 2014

[4] E. Kail, A. Bánáti, K. Karóczkai, P. Kacsuk, M. Kozlovszky, Dynamic workflow support in gUSE, MIPRO, 2014 Proceedings of the 37th International Convention

[5] Jhawar, Ravi, Vincenzo Piuri, and Marco Santambrogio. „Fault Tolerance Management in Cloud Computing: A System-Level Perspective". IEEE Systems Journal 7, vol 2, 2013 doi:10.1109/JSYST.2012.2221934

[6] J.W. Young. "A first order approximation to the optimum checkpoint interval" in Comminications ACM, 1974

[7] S. Di, Y. Robert, F. Vivien, D. Kondo, Cho-Li Wang, and F. Cappello. „Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism", 1–12. ACM Press, 2013. doi:10.1145/2503210.2503217.

[8] B. Meroufel, G. Belalem: "Adaptive time-based coordinated checkpointing for cloud computing workfl ows". in Scalable Computing: Practice and Experience, Vol 15, No 2, 2014

[9] B. Meroufel, B. Ghalem: „Policy Driven Initiator in Coordination Checkpointing Strategies". http://www.wseas.us/e-library/conferences/2014/Istanbul/TELEDU/TELEDU-20.pdf.

[10] Antony Lidya Therasa.S, Sumathi.G, Antony Dalya.S.: "Dynamic Adaptation of Checkpoints and Rescheduling in Grid Computing" in International Journal of Computer Applications vol. 3, 2010

[11] Laiping Zhao; Yizhi Ren; Yang Xiang; Sakurai, K., "Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems," High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on , vol., no., pp.434,441, 1-3 Sept. 2010, doi: 10.1109/HPCC.2010.72.

[12] Topcuoglu, H.; Hariri, S.; Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," Parallel and Distributed Systems, IEEE Transactions on , vol.13, no.3, pp.260,274, Mar 2002,