Computer Science • 17 (3) 2016 http://dx.doi.org/10.7494/csci.2016.17.3.387

Eszter Kail
Péter Kacsuk
Miklós Kozlovszky

A NOVEL ADAPTIVE CHECKPOINTING METHOD BASED ON INFORMATION OBTAINED FROM WORKFLOW STRUCTURE

Abstract *Scientific workflows are data- and compute-intensive; thus, they may run for days or even weeks on parallel and distributed infrastructures such as grids, supercomputers, and clouds. In these high-performance computing infrastructures, 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 technique is taking checkpoints from time to time; when failure is detected, the last consistent state is restored. One of the most-critical factors that has great impact on the effectiveness of the checkpointing method is the checkpointing interval. In this work, we propose a Static (Wsb) and an Adaptive (AWsb) Workflow Structure Based checkpointing algorithm. Our results showed that, compared to the optimal checkpointing strategy, the static algorithm may decrease the checkpointing overhead by as much as 33% without affecting the total processing time of workflow execution. The adaptive algorithm may further decrease this overhead while keeping the overall processing time at its necessary minimum.*

Keywords scientific workflow, checkpoint, dynamic execution

Citation Computer Science 17 (3) 2016: 387–406

1. Introduction

Real-time users typically want to know an estimation regarding the execution time of their application before deciding to have it executed. In many cases, this estimation can be considered to be a soft deadline that shall be satisfied with some probability without serious consequences. Moreover, time-critical scientific workflows to be successfully terminated before hard deadlines imposes many challenges. Many fields research faces time constraints and soft or hard deadlines to task execution. A hard deadline means that the results are only meaningful before the hard deadline; if any of the results are late, then the whole computational workflow and its executions are a waste of time and energy.

Furthermore, scientific workflows are mainly enacted on distributed and parallel computing infrastructures such as grids, supercomputers, and clouds. As a result, a wide variety of failures can arise during execution. Scientific workflow management systems should deal with the failures and provide some kind of fault-tolerant behavior. There are a wide variety of existing fault-tolerant methods, but one of the most-frequently used proactive fault-tolerant method is checkpointing, where the system state is captured from time to time, and in case of a failure, the last-saved and consistent state is restored.

The drawback of the already-existing checkpointing methods is that they mostly use static checkpointing intervals. Using static intervals bypasses the opportunity to adapt the method to the new and actual status of the scientific workflow management system. During enactment, several conditions can change, ranging from network reachability issues to the checkpointing cost or even to the reliability of the computational architecture. From this perspective, they cannot be said to be optimal. Moreover, capturing checkpoints generates costs in both time and space. On one hand, the time overhead of checkpointing can have a great impact on the total processing time of the workflow execution; on the other hand, the needed disk size and network bandwidth usage can also be significant. By dynamically assigning the checkpointing frequency, we can eliminate unnecessary checkpoints. Where the danger of a failure is considered to be severe, we can introduce extra state savings.

Our Static (Wsb) Adaptive (AWsb) Workflow Structure-based checkpointing algorithm is based on a workflow model structure and failure statistics gathered about resources from historical executions. It extends related work on workflow-structure analysis, which focuses mainly on workflow similarity issues concerning the efficient storing and sharing of reproducible workflows [9], exception handling, and scheduling mechanisms, and also workflow execution-time estimation problems [8]. Our work also promotes research into fault-tolerant methods by including the information obtained from the workflow structure into the actual state analysis, and thus, into the checkpointing interval determination.

This paper contributes a novel Wsb checkpointing method for scientific workflows based on not communicating, but parallel-executable jobs. In our work, we also show a way that this method can be used adaptively in a dynamically changing

environment. Finally, with our adaptive algorithm, we create the possibility for the scientist to get feedback about the remaining execution time during enactment and the possibility of meeting a predefined soft or hard deadline.

Our paper is structured as follows: Section 2 gives a brief overview about the state of the art. In Section 3, we analyze the scientific workflow model and give some describing definitions. It also gives an algorithm to calculate the needed parameters and subsets of the workflow model. Section 4 introduces our Wsb algorithm for a static scenario, and Section 5 presents our AWsb algorithm for the online adaptive case. Section 6 demonstrates our simulation results. Finally, section 7 summarizes our work and further research directions.

2. State of the art

Concerning dynamic workflow execution, fault tolerance is a long-standing issue, and checkpointing is the most-widely-used method for achieving fault-tolerant behavior. Since grids, clusters, and clouds are highly dynamic in nature, they must overcome resource failure and check how changes in the topology and computational capability of the high-performance computing infrastructure resources affect the efficiency in terms of task completion.

Hwang et al. [3] divided workflow-failure-handling techniques into two different levels, namely task-level and workflow-level. Task-level techniques handle the execution failure of tasks like the task-independent scenario and the techniques similar to those earlier described, while workflow-level techniques may alter the sequence of execution in order to address the failures [2]. Hwang and Kesselman proposed three different techniques on the basis of assuming that there is more than one implementation possible for a certain computation with different execution characteristics. From this perspective, a hybrid failure-handling technique is used; namely, at the task and workflow levels to minimize the effects of the task-level fault-tolerant techniques on the whole workflow.

The efficiency of the used checkpointing mechanism is strongly dependent on the length of the checkpointing interval. Frequent checkpointing may increase the overhead, while rarely made checkpoints may lead to a loss of computation. Hence, the decision about the size of the checkpointing interval and checkpointing technique is a complicated task and should be based on knowledge specific to the application as well as the system. Therefore, various types of checkpointing optimization have been considered by the researchers.

According to the level where the checkpointing occurs (whether at the application, library, or system levels), the methods are differentiated. 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. The library-level solution requires a special library linked to the application that can perform the

checkpoint and restart procedures. The system-level solution can be implemented by a dedicated service layer that hides the implementation details from the application developers yet still gives the opportunity to specify and apply the desired level of fault tolerance [4].

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 and, thus, a consistent state for recovery. This may lead to the initial state due to the domino effect.

Meroufel and Belalem [6] proposed an adaptive time-based coordinated checkpointing technique without clock synchronization on the cloud infrastructure. Between the different VMs, jobs can communicate with each other through a message-passing interface. One VM (Virtual Machine) 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 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 an SLA (Service-level Agreement) violation. In Meroufels 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 a checkpoint. Logging the messages can also resolve the transit-message problem. The initiator selection is also investigated in Meroufel and Belalems other work [7]; they found that the impact of initiator choice is significant in terms of performance. They also proposed a simple and efficient strategy to select the best initiator.

The frequency of the checkpointing interval also imposes many opportunities in checkpointing algorithms. In [11], John W. Young defined his formula for the optimum periodic checkpoint interval in 1974, which is based on the checkpointing cost and the mean time between failures (MTBF) with the assumption that failure intervals follow an exponential distribution. In [1], Di et al also derived a formula to compute the optimal number of checkpoints for jobs executed in the cloud. Their 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, network issues arise, or when the failure distribution changes. Thus, static intervals may not lead to an optimal solution. By dynamically assigning checkpoint frequency, we can eliminate unnecessary checkpoints, or where the danger of a failure is considered to be severe, extra state savings can be introduced.

Di et al. also proposed another adaptive algorithm to optimize the impact of checkpointing and restarting cost [1]. In their work [10], Theresa et al 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 individual resources.

In our work, the determination of the checkpointing interval (besides some failure statistics) is primarily based on workflow characteristics, which is a key difference from existing solutions. We demonstrate that we can still get good insight into the number of checkpoints during job execution in order to achieve the desired level of performance with minimum overhead of the used fault-tolerant technique.

3. Model and workflow structure analyses

Given workflow model $G(V, \vec{E})$, where V is the set of nodes (tasks) and $\vec{E}$ is the set of edges representing data dependency, formally $V = \{T_i | 1 \leq i \leq |V|\}$ $\vec{E} = \{(T_i, T_j) \, | T_i, T_j \in V \text{ and } \exists T i \rightarrow T_j\}$. $|V| = n$ is the number of nodes (tasks in the workflow). Usually, scientific workflows are represented with Directed Acyclic Graphs (DAGs), where the numbers associated with tasks specify the time that is needed to execute each given task, and the numbers associated with the edges represent the time needed to start each subsequent task. This can involve data-transfer time from the previous tasks, resource starting time, or time spent in the queue. These values can be obtained from historical results, from a Provenance Database, or it can be estimated based on certain parameters; for example, on the number of instructions.

In one of our previous works [5], we defined the concepts for sensitivity and a sensitivity index, and we have also demonstrated calculations in simple workflow examples.

Definition 3.1 *A workflow model is said to be sensitive if failures occurring during the execution of a task in most cases causes the total workflow execution time (total processing time of the workflow execution) to increase.*

To formulate the sensitivity of a workflow model, we define the influenced zone of an individual task.

Definition 3.2 *The influenced zone of an individual task T_i is the set of tasks that, at submission time, is affected because a failure is occurred during the execution of a task T_i.*

In other words, if a failure does not have a global effect on workflow-execution time, then we can define the border of its effect or the set of tasks for which submission occurs at a later time. The influenced zone is always related to a certain delay parameter.

Based on this definition, the sensitivity index of graph $G(V, \vec{E})$ is defined as the ratio of the influenced zone to the remaining subgraph summarized by all tasks (which

is averaged over all tasks).

$$S = \frac{\sum_{i=1}^{|V|} \frac{|I_i|}{|G_{R,i}|}}{|V|}, \tag{1}$$

Where I_i is the influenced zone of vertex T_i, and $G_{R,i}$ is the remaining subgraph that is induced by vertex T_i as the entry point (or starting point) of the subgraph, and the original endpoint (T_e) serves as the endpoint of the subgraph. In other words, $G_{R,i}$ contains all of the paths that existed in the original graph between vertices T_i and T_e.

We also define the flexibility zone of a graph.

Definition 3.3 *The flexibility zone or zones of a workflow is a subworkflow of the original workflow, where changes in timing parameters may happen without affecting the total execution time of the workflow.*

This subgraph consists of multiple paths, which enables the time flexibility to the given task. The flexibility zone is always related to an influenced zone; thus, it is based on a certain delay interval.

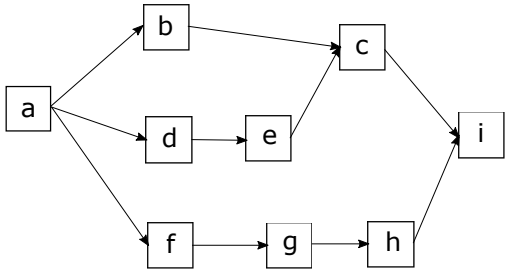

Figure 1. A simple workflow example.

3.1. Workflow structure analyses with complex graphs

As it is well demonstrated in Figure 1 (and according to one of our previous works [5]), the influenced zone, flexibility zone and sensitivity index of a simple workflow model can be easily determined; but, in complex workflow structures with a high number of vertices, it would need a very long time to carry out an exhaustive search to find the influenced and flexibility zones for all tasks and for the different delay parameters.

If we have n nodes, than we can have, at most, $|E| = \binom{n}{2} = \frac{(n\cdot(n-1))}{2}$ edges between the nodes, because duplicate edges and self-loops are not allowed in a model of scientific workflows. Duplicate edges are not allowed, because they are not needed (since an edge between two nodes can represent the dataflow of more than one of the datasets as well). Self-loops are also not tolerated, because DAGs cannot contain cycles. Deriving from that, if we ignore the orientation of the edges (in other

words, we form our graph into a non-directed one), the number of cycles may reach $\sum_{i=3}^{n} \binom{n}{3} \approx 2^n$.

The listing and analyzing of all of the cycles and from the results calculating the influenced zones cannot be performed in polynomial time. Instead, we have to invent another method to accomplish this.

The basic idea behind our algorithm is the fact that DAGs have topological orderings. DAGs are used to indicate a precedence relationship or relative ordering among the vertices. Given DAG $G(V, \vec{E})$, a topological ordering of G is a linear order of all vertices, which respects the precedence relation; i.e., if G contains edge (T_i, T_j), or with another notation $T_i \rightarrow T_j$, then T_i appears before T_j in the topological ordering. Concerning the graph demonstrated in Figure 1, a possible topological ordering would be $\{a, b, d, e, c, f, g, h, i\}$, but the series $\{a, d, e, b, c, f, g, h, i\}$ also gives a valid ordering. As can be seen from the example, many topological orders may exist for a given DAG.

Lemma 3.4 *A G graph is a DAG if and only if it has a topological ordering.*

As a consequence of lemma, we know that every DAG has topological orderings.

The topological order of a DAG can be computed in many ways, but maybe the most-frequently-used method is applying a Depth-First Search (DFS). DFS is a systematic way to find all vertices reachable from a source vertex, s.

Our algorithm to find the influenced zones and flexibility zones of a workflow model is based on DFS and consists of the following three steps:

1. Calculating the global flexibility concerning to the whole workflow model
2. Determining the influenced zones of each node
3. Calculating the flexibility zones of a workflow model

3.1.1. Calculating the global flexibility concerning the whole workflow model

The first step is to carry out a Depth-First Search (DFS) on the workflow model; during the search, the following values must be stored to each node T_i: T_i.start is the earliest possible start time and T_i.end represents the latest possible end time of a node (task) T_i without affecting the total execution time of the workflow.

By going through the workflow with DFS from entry task T_0 to end task T_e, we calculate and store values $T_i.start$ in each step by summarizing the values $T_j.start$ of predecessor task T_j, and the time that is needed to start task T_i (values assigned to edge (T_i, T_j) for all predecessors T_j, and we store the maximum of these values.

The T_i.end time for all i can be calculated in a similar manner, recursively backwards from the last or ending task T_e.

Definition 3.5 *Given DAG $G(V, \vec{E})$, the global flexibility of $T_i \in V$ is $gflex[T_i] = T_i.end - T_i.start$.*

In other words, global flexibility of task T_i gives the time flexibility of a task, in which the task execution can be freely managed.

If $gflex[T_i] = t(T_i)$ for vertex T_i, this means that this node does not have any flexibility in time, where $t(T_i)$ is the calculation time of node T_i.

Since we investigate workflows here with one entry node T_0 and one ending task T_e, these two nodes are surely part of the critical path in all cases; so, for their global flexibility, parameter $gflex[T_0] = t(T_0)$ and $gflex[T_e] = t(T_e)$ stand.

It can be also generally declared that, if task T_i's global flexibility is zero, then this task must be part of at least one of the critical paths.

Figure 1 shows a simple workflow model. For the sake of simplicity in this scenario, we assume that the data transfer time is 0 (values assigned to edges are all 0), and all of the tasks need one time unit to be executed.

Thus, there are two critical paths in the workflow: $a \to d \to e \to c \to i$ and $a \to f \to g \to h \to i$. From that follows that for all these tasks that are part of the critical paths $gflex[a] = gflex[d] = ... = gflex[i] = 1$. There is only one task, b for which $gflex[b] = 2$.

3.1.2. Determining the influenced zones of each node

If we have all of the global-flexibility values, we have to determine the influenced zones. In regards to to Definition 3.2: if a failure occurs during the execution of T_i, then the total execution time of T_i is increased; therefore, all of the tasks belonging to the influenced zone (or zones) of T_i can be started later than originally planned.

Formally, influenced zone I_i of vertex T_i for given delay d_f can be determined as follows: Starting from T_i, we carry out a search for all nodes $T_j \in SUCC(T_i)$ where $gflex[T_j] = T_j.end - T_j.start < d_f$.

According to Figure 1, the influenced zone of task b concerning a delay of one time unit consists of only task b; so, the failure has only a local significance. However, for all other nodes, the same one-time-unit delay has an influence zone consisting of the whole subworkflow originating from the actual node.

3.1.3. Calculating the flexibility zones of a workflow model

It can be realized that flexibility zones are connected to cycles in the workflow graph when ignoring the orientation of the edges (regarding DAGs, we can only talk about cycles when we omit the orientation of the edges). More precisely, this is the case with subgraphs that contain several cycles interconnected with each other. To calculate the flexibility zones of a workflow model, we use the base of the algorithms published by Li et al. in [**?**]. In this paper, the authors calculated the number of all topological orderings of a Directed Acyclic Graph. For this purpose, they introduced the following concepts (which we also need in our calculations): $PRED(T_j)$ and $SUCC(T_j)$ is the predecessor set and successor set of task T_j, respectively. Formally, $PRED(T_j) = \{T_i | T_i \to\to T_j\}$ and $SUCC(T_j) = \{T_k | T_j \to\to T_k\}$, where $T_j \to\to T_k$ indicates that a path exists from T_j to T_k.

Definition 3.6 *A static vertex is vertex T_i for which $|PRED(T_i)| + |SUCC(T_i)| = |V| - 1$ for given DAG $G(V, \vec{E})$.*

The placement of a static vertex is deterministic, so it is the same in all existing topological orders.

Definition 3.7 *Static vertex set $S \in V$ is a vertex set for which $|PRED(S)| + |SUCC(S)| = |V| - |S|$ for given DAG $G(V, \vec{E})$ and is minimal; that is, no proper subset of S has the same property.*

In Li's work, the authors proved that these static vertex sets are disjoint.

According to these static vertex sets, a graph can be partitioned into disjoint static vertices and vertex sets.

Since the static vertex set means that the nodes or subset of these nodes can be in arbitrary order to each other, we may divide the vertex set into disjoint parallel threads of tasks. Thus, if a subgraph resulting from the algorithm is not simple enough, we can further use these algorithms after dividing the subgraphs into disjoint parallel threads. So, our algorithm can be recursively adapted until the desired depth.

As a result, the minimal flexibility zones of a workflow will be those static vertex sets that cannot be further partitioned. Of course, upon applying the results of the workflow structure analyzes, we may conclude a lack of need for the minimal flexibility zones of the nodes, but a few sizes greater. Involving this method, we can determine the appropriate flexibility zones according to a given influenced zone.

Determining the static vertex set is based on the simple method used by Li et al. in [**?**].

4. Static Wsb algorithm

Given a workflow model $G(V, \vec{E})$, V is the set of nodes (tasks in the workflow) and $\vec{E}$ is the set of edges representing data dependency. There are $|V| = n$ tasks and m resources in the system. The execution time of a task without any failure tolerant behavior and without any failures (i.e., the calculation time of task T_i on resource j) is $t(T_i)_j$. This $t(T_i)_j$ value can be obtained from a provenance database or can be calculated based on the number of instructions that the code contains. Table 1 summarizes the notation for the variables of our system.

Table 1

Notation of the variables of the Wsb algorithm.

$t(T_i)_j$	Calculation time of task T_i on resource j
$t_{f,j}$	Fault detection time on resource j
$t_{s,j}$	Restart time on resource j
$C_j(t)$	Checkpointing cost on resource j
$T_{C,j}$	Checkpointing interval on resource j
$R_{i,j}$	Recomputation time of task T_i on resource j

For our first order model, let us assume that the checkpointing cost does not change during execution and does not depends on the type of resource, so we denote it with C. We also assume that the fault-detection time is negligible, so $t_{f,j} = 0$ for

all j, and we have only one type of resource. So, from now on, we omit notation $t(T_i)_j$, $t_{f,j}$, $t_{s,j}$, $T_{C,j}$, $R_{i,j}$; we only use $t(T_i)$, t_f, t_s, C_j, T_C, R_i respectively.

After a failure occurs during checkpointing interval T_C, the rework time that is needed to recalculate the lost values is, on average, $\frac{T_c}{2}$. From this, it follows that the expected rework time that is needed to successfully terminate given task T_i can be expressed by:

$$E(R_i) = \sum_{j=1}^{\infty} P(Y = j) \cdot j \cdot \left(\frac{T_c}{2} + t_s \right), \tag{2}$$

where $P(Y = j)$ denotes the probability of having j failures during the execution of task T_i. With these assumptions, we can calculate the expected wallclock (total processing) time of a task T_i as:

$$E(W_i) = t(T_i) + \left(\frac{t(T_i)}{T_C} - 1 \right) \cdot C + \sum_{j=1}^{\infty} P(Y = j) \cdot j \cdot \left(\frac{T_c}{2} + t_s \right) \tag{3}$$

Thus, if critical errors (failures that do not allow for the further execution of a job) and program failures do not occur during the execution, then the expected execution time can be calculated using the above equation. According to the definition of the expected value for a discrete random variable, we get $E(Y) = \sum_{j=1}^{\infty} P(Y = j) \cdot j$. From the above equation, Di et al [1] derived the optimal number of checkpointing intervals (X_{opt}) for a given task:

$$X_{opt} = \sqrt{\left(t(T_i) \cdot \frac{E(Y)}{2C} \right)} \tag{4}$$

If we assume that the failure events follow an exponential distribution, then we get that the optimal checkpointing interval during the execution of task T_i can be expressed by:

$$T_{copt} = \sqrt{(2CT_f)} \tag{5}$$

where T_f is the mean time between failures. This equation was derived by Young in 1974.

We will use equation (3) as a starting point to calculate the checkpointing interval in order to minimize the checkpointing overhead without affecting the total wallclock execution time of the whole workflow. In equation (3), the unknown parameter is the checkpointing interval; for W_i, we have an upper bound from the flexibility parameter of task T_i.

4.1. Large flexibility parameter

If flexibility parameter $gflex[T_i] >> t(T_i)$, then this means that we have ample time to successfully terminate the task. Maybe the task could be successfully executed even more times. In this case, it is not worth pausing the execution to take checkpoints,

but trying to execute it without any checkpoints. If failure occurs, we still have time to re-execute it. When there has already been more than one trial and no successful completion, then we should check the remaining time to execute the task without negatively affecting the total wallclock execution time. We would like to ensure that the task execution time does not affect the total execution time of the workflow (or only has an effect with probability p).

4.2. Adjusting the checkpointing interval

When the failure distribution is not known but we have a provenance database which contains the timestamps about the occurrences of failures for a given resource, then calculating the time that is needed to execute a task in the presence of failures with probability p is as follows:

If the mean time between failures is T_f, and we also have the deviance from provenance, then, with Chebyshev's inequality (6), we can determine the minimum size interval between the failures with probability p. This means that, with probability p, the failures do not happen within shorter time intervals

$$P\left(|\xi - T_f| \geq \epsilon\right) \leq \frac{D^2\xi}{\epsilon^2}. \tag{6}$$

We should find a valid ϵ for that $P\left(|\xi - T_f| \geq \epsilon\right) \leq 1-p$ stands. If we have this ϵ, then we can calculate $T_m = T_f - \epsilon$ as the minimum failure interval with a probability greater than p. From this follows that, with probability p, there will not be more than $k = \frac{t(T_i)}{Tm}$ failures during the execution time of $T_{i,j}$. If we substitute this k into equation (3), we get an upper bound for the total wallclock execution time of the given task with k failures:

$$W_i = t(T_i) + (\frac{t(T_i)}{T_c} - 1) \cdot C + k \cdot \left(\frac{T_c}{2} + t_s\right). \tag{7}$$

If we use the optimal checkpointing for given task T_i with T_f mean time between failures (MTBF) and the deviance from this MTBF is ξ, then T_p gives the upper bound of the wallclock execution time with probability p:

$$T_p = t(T_i) + \left(\frac{t(T_i)}{T_{copt}} - 1\right) \cdot C + k \cdot \left(\frac{T_{copt}}{2} + t_s\right). \tag{8}$$

We henceforth assume that the failures do not occur during checkpointing and recovery (restarting and restoring the last-saved state) time, only during calculations.

If the flexibility parameter still permits some flexibility (i.e., $gflex[T_i] > T_p$), then we can increase the checkpointing interval and so decrease the checkpointing overhead.

To calculate the checkpointing interval according to the flexibility parameter, we should substitute $gflex[T_i]$ into W_i:

$$gflex[T_i] > t(T_i) + (\frac{t(T_i)}{T_{cflex}} - 1) \cdot C + k \cdot \left(\frac{T_{cflex}}{2} + t_s\right). \tag{9}$$

We should fine T_{cflex} value for that (9) and $T_{cflex} > T_{copt}$ stands.

From these inequalities, the actual T_{cflex} can be calculated easily.

If $W_i - T_p = 0$, the flexibility only allows us to guarantee successful completion with probability p.

However, if the flexibility parameter does not permit any flexibility (moreover, if $W_i < T_p$), then maybe the soft deadline cannot be guaranteed with probability p.

4.3. Proof of the usability of our algorithm

According to (9), it is also numerically proven that the total execution time is a function of checkpointing interval T_c; or as it is indicated, a function of the number of checkpoints $n = \frac{t(T_i)}{T_c}$. As seen in Figure 2, the dependency is quadratic. Figure 2 shows five parabolas with a different number of failures (k values). All of the parabolas have minimum points, where the wallclock time of a task is minimal with an appropriate number of checkpoints. As k increases, the minimum points are shifted to the right. The dashed green line represents the curve with $k = 4$, where checkpointing cost $C = 2$ and calculation time $t(T_i) = 32$. This curve has its minimum points at four checkpoints $n = 4$. However, if we have time flexibility according to the curves in Figure 2, we have the possibility of decreasing the number of checkpoints. In the case of the dashed green line, if we have four checkpoints, then the wallclock time reaches its minimum, while having only two checkpoints increases the total wallclock time. According to the flexibility parameter, an appropriate number of checkpoints can be determined; thus, it is possible to minimize the checkpointing overhead without increasing the total wallclock execution time of the workflow.

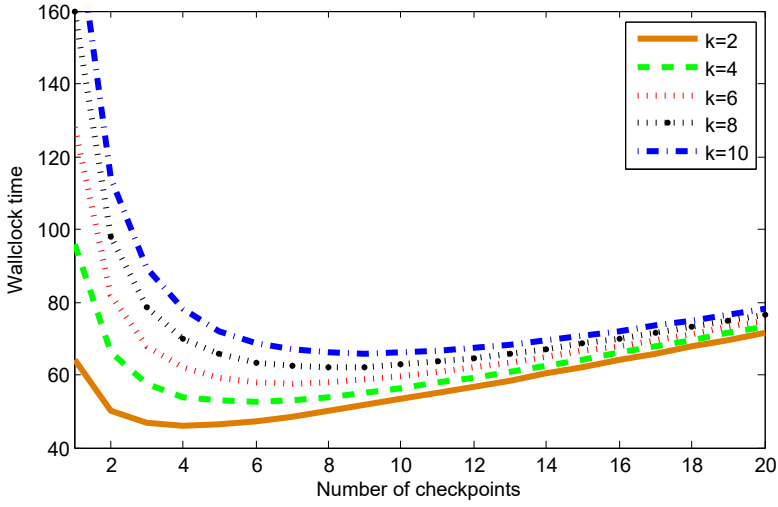

Figure 2. Total process time as a function of the number of checkpoints.

5. Adaptive Wsb algorithm

We talk about adaptive workflows and whether a workflow model can change during execution according to the dynamically changing conditions.

In previous chapters, we made calculations on the graphs that are based on prior knowledge obtained from previous enactments or estimations for runtime, communication, and data-transfer-time requirements. However, if the system supports provenance data storage and runtime provenance analysis, then we can base our calculations on realistic and up-to-date data. For example, if the precise timing of the task submissions that are under enactment and all of the tasks that are already terminated are known, then the accurate flexibility parameter of the running tasks can be calculated, and a more-precise estimation of the flexibility zones of the successor tasks can be made available. These calculations are always updated with newer and newer timing data but include fewer and fewer subgraphs with the advance of the execution steps. So, the remaining steps and calculations are getting simpler. Thus, if before workflow submission we calculate the global flexibility parameter vector of the whole workflow, and we also store the estimated starting time of the individual execution times relative to each other, then before executing a task, its starting time should be updated to the new situation caused by the failures. Of course, depending on the delay, the global flexibility parameters of all of the nodes belonging to the influenced zone of this task should be adjusted.

Based on these calculations, it is also possible to give a scientist more feedback about its workflow execution during enactment. For example, the researcher may get feedback on the probability of meeting soft or hard deadlines or whether the results will be outdated when the workflow execution terminates. So, it can be decided to stop the workflow, to modify the workflow, or to take other actions that are supported by the scientific workflow management system.

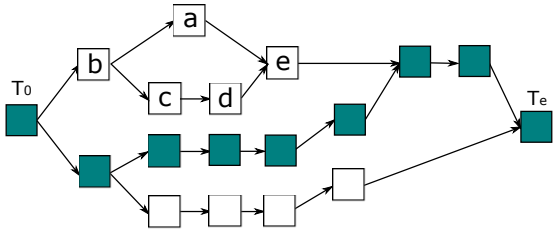

Figure 3. An example workflow with one critical path.

For the sake of simplicity, let us assume that data transfer time is negligibly small in our examples (there are not any values assigned to the edges) and task execution time is 1 time unit for all tasks in Figure 3. The critical path is built up from the blue tasks before submitting the workflow. As a result, the global flexibility parameters for all white tasks are two times the unit except for task a, where this value is three times the unit.

In Figure 4 during the execution of task a, a 1-time-unit failure has occurred. Since $gflex[a] = 2$ and $gflex[e] = 1$, this 1-unit delay has only a local significance. This means that this delay will not effect subsequent task e's submission time (it can also be determined from the alternative path through tasks c and d). So, the influenced zone of this failure consists only of task a.

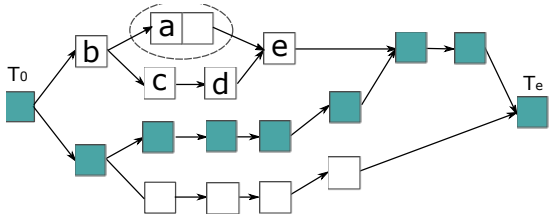

Figure 4. An example workflow with a one-time-unit delay during the execution of task a.

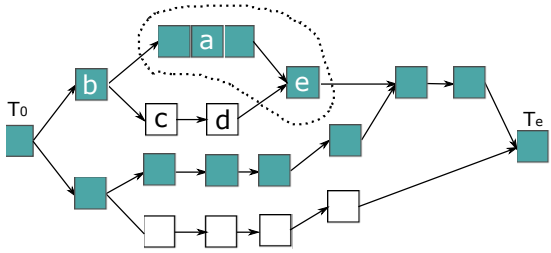

Figure 5. An example workflow with a two-time-unit delay during the execution of task a.

In Figure 5, the delay caused by the failure occurring during the execution of task a is two-times-the-unit long. In this case, the influenced zone is the set of tasks enclosed with the dotted line. This means that, due to this delay, task e should start later; but, the successor task of task e is not influenced, so the workflow-execution time can still remain the originally estimated time. Due to the postponed starting time of task e, the global flexibility parameters of tasks c and d need to be recalculated according to the new situation. In other words, the flexibility zone of task a consists of the subgraph induced by task b to task e. But, this delay has another effect as well; namely, the path driving through task a also became a critical path in addition to the original one. As a consequence, if any failure occurs during this path, the entire workflow execution lasts longer.

6. Results

For validation purposes, we have implemented both of our checkpointing algorithms in Matlab, a numerical computing environment by MathWorks.

To clarify the benefits of our static (Wsb) and adaptive (AWsb) algorithms, Figure 6 shows our sample workflow $G_{sample}\left(V, \vec{E}\right)$, where $V = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8\}$ and $\vec{E} = \{(T_1, T_2), (T_1, T_3), (T_3, T_4), (T_1, T_5), (T_5, T_6), (T_6, T_7), (T_2, T_8), (T_4, T_8), (T_7, T_8)\}$, running in a distributed environment, consisting of three resources: R_1, R_2, and R_3. For the sake of simplicity, the resources are identical and have identical failure distribution. We use $E(Y) = 2$ as the expected number of failures for an 18-time-unit-long task, and when changes occur during execution, this value is proportionally calculated to the changes. We also take advantage of the simplification that the data transfer times are negligibly small (they are all zeros) and the checkpointing cost has a constant value of $C = 2$. The workflow makespan (total wallclock time) is the longest path from T_0-T_e. We have simulated five scenarios with the same input parameters for our sample workflow:

1. optimal static case: Optimal checkpointing is used [1] (T_{copt} is the optimal checkpointing interval, X_{opt} is the number of checkpoints, W_{orig} is the total execution time).
2. static execution with our static Wsb algorithm: In this case, the Wsb algorithm is executed once before workflow submission, which calculates the number of checkpoints based on the workflow structure ($X_{stat-wsb}$ is the number of checkpoints, $W_{stat-wsb}$ is the total execution time).
3. dynamic execution with optimal checkpointing: In this case, the execution time of a task is changed, but the execution is based on the original optimal checkpointing interval. (Optimal checkpointing interval T_{copt} is used, $W_{dyn-opt}$ is the total execution time).
4. dynamic execution with our static (Wsb) algorithm: In this scenario, the execution time of a task is changed, but the execution is based on static Wsb algorithm that was carried out before workflow submission; thus, before the change (the checkpointing interval is the same as in the static execution with the Wsb algorithm, $W_{dyn-wsb}$ the total execution time).
5. dynamic execution with our adaptive (AWsb) algorithm: In this case, the execution time of a task is changed, and the adaptive AWsb algorithm recalculated the checkpointing intervals after the change ($X_{dyn-awsb}$ is the number of checkpoints, $W_{dyn-awsb}$ the total execution time).

In the above-defined dynamic scenarios, there is only one task during each individual execution of the workflow; namely, T_3, for which the execution time is changed compared to the predestined values.

The simulation was carried out with $t(T_i) = 18$ and based on this value $T_{copt} = 6$, $X_{opt} = 3$ and thus $W_{i-orig} = 28$ was calculated for all tasks T_i, where W_{i-orig} is the total processing time of T_i when optimal checkpointing interval (T_{copt}) is used.

Table 2 shows the actual parameters for all tasks of the workflow for the static and adaptive cases. Table 3 compares the number of checkpoints and the total wallclock time for the whole worklfow for the five scenarios.

As the results show, our static algorithm reduces the checkpointing overhead by 33%, as the number of checkpoints were decreased from 24 to 16 with our algorithm, and the total wallclock time of the workflow did not change. We can also notice that, in a dynamically changing environment where the execution time for the tasks can change unpredictably, our adaptive algorithm may further increase the number of checkpoints but decrease the total wallclock time compared to dynamic execution with the static-algorithm scenario.

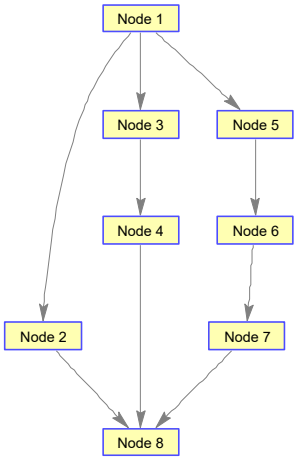

Figure 6. Sample workflow with 8 tasks.

Table 2
Simulation results for sample workflow (Fig. 6).

	$X_{stat-wsb}$	$t(T_i)_{dyn}$	$(W_i)_{dyn-awsb}$	$X_{dyn-awsb}$
$t(T_1)$	3	18	28	3
$t(T_2)$	0	18	36	0
$t(T_3)$	0	36	72	1
$t(T_4)$	1	18	28	3
$t(T_5)$	3	18	28	3
$t(T_6)$	3	18	36	0
$t(T_7)$	3	18	36	0
$t(T_8)$	3	18	28	3

Table 3
Comparison of number of checkpoints (X) and the total wallclock time (W) in the five scenarios.

X_{opt}	$X_{stat-wsb}$	$X_{dyn-awsb}$	W_{orig}	$W_{stat-wsb}$	$W_{dyn-stat}$	$W_{dyn-awsb}$	$W_{dyn-opt}$
24	16	13	140	140	164	156	140

We have also carried out simulations with randomly formed DAGs. In these cases, the number of tasks has moved between 10 and 60 nodes, and calculation time $t(T_i)$ was randomly generated within the interval of (10,100). The expected number of failures was increased during the simulations (Fig. 7 failure frequency), started for an average of a 55-time-unit-long task with $E(Y) = 2$ to $E(Y) = 10$, and it was proportionally adapted to the tasks according to their calculation time. Each point of the curve was averaged over 50 executions.

As Figure 7 shows, the results strongly vary, but they also show a significant improvement as a function of the failure frequency. It can also be declared that this significant improvement can be seen as a function of the checkpointing cost as well.

Our AWsb adaptive algorithm has also been tested with random graphs similar to the static case. As a consequence of the randomly generated workflows, the average difference between the total wallclock time of the dynamic execution with our Wsb algorithm case compared to dynamic execution with the AWsb scenario spread over a range of 0 % and 10 % improvement, and the number of checkpoints also shows a significant decrease in the latter case. So, we can conclude that the AWsb algorithm may decrease the checkpointing overhead to a further extent than the static Wsb algorithm while keeping the total processing time at its necessary minimum.

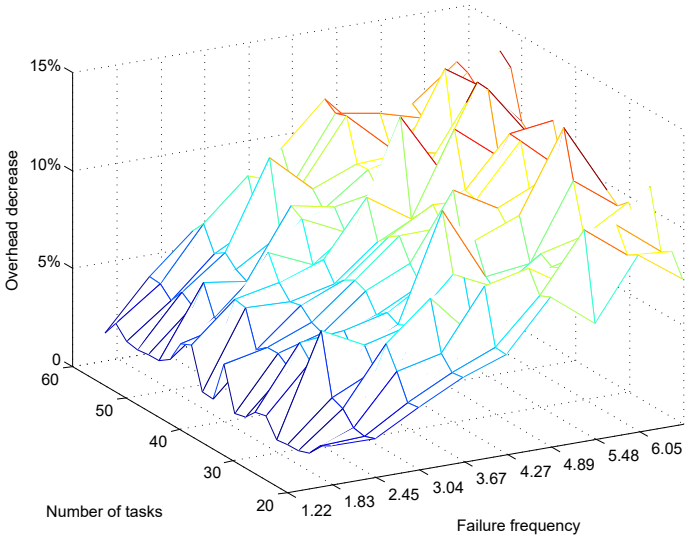

Figure 7. Results of our static algorithm.

6.1. Limitations of our work

In our simulations, we have simplified the calculations by using constant values as checkpointing cost C by neglecting the data-transfer and task-submission times during the executions (or by assuming identical resources). Nevertheless, these assumptions

can be easily resolved by substituting actual functions instead of using constant or simplified parameters.

The calculation time for complex graphs can be lengthy; but after a brief study at the myExperiment.org website, we have concluded that the mean size of the uploaded workflows moves between 30 and 50 nodes with manageable complexity. This revelation led us to develop the adaptive algorithm for which the recalculation time can be measured in hundreds of milliseconds.

Our algorithm cannot be used for an arbitrary type of failure or fault. It was intended to develop a mechanism against crash faults, or network outage. Of course, the proposed checkpointing method does not solve programming failures, byzantine failures, etc. in itself, as is the case with the optimal checkpointing strategy developed by Young [11] and Di [1].

A further limitation of the algorithms lies in the fact that they depend highly on historical execution data or on estimated data about execution time and failure distribution. Data about historical executions can be stored in a provenance database; but today, there are only limited capabilities for runtime provenance analysis, and of course the estimations lack precision.

7. Conclusion

We introduced Static (Wsb) and Adaptive (AWsb) Workflow Structure-based checkpointing methods that are based on failure statistics on resources and on information that can be obtained from the workflow structure. With the help of the introduced checkpointing method, the checkpointing overhead can be minimized by continually keeping the performance at a satisfactory level; namely, ensuring the successful completion of scientific workflows before soft or hard deadlines with a predefined probability of p. We also showed that this algorithm can be adapted to a dynamically changing environment by updating the results of the workflow structure analysis. Our simulation results showed that the checkpointing overhead can be decreased by as much as 33% with our static Wsb algorithm, and the adaptive AWsb algorithm may further decrease this overhead while keeping the total wallclock time at its necessary minimum.

Our future work is to use our algorithm to inform scientists to the extents of the probability with which hard and soft deadlines will be met during their workflow executions.

References

[1] Di S., Robert Y., Vivien F., Kondo D., Wang C.L., Cappello F.: Optimization of Cloud Task Processing with Checkpoint-restart Mechanism. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pp. 64:1–64:12, ACM, New York, NY, USA, 2013, `http://doi.acm.org/10.1145/2503210.2503217`.

[2] Garg R., Singh A.: Fault Tolerance in Grid Computing: State of the art and open issues. *International Journal of Computer Science and Engineering Survey (IJCSES)*, vol. 2, p. 8897, 2011.

[3] Hwang S., Kesselman C.: Grid workflow: a flexible failure handling framework for the grid. *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pp. 126–137, 2003.

[4] Jhawar R., Piuri V., Santambrogio M.: Fault Tolerance Management in Cloud Computing: A System-Level Perspective. *IEEE Systems Journal*, vol. 7(2), pp. 288–297, 2013.

[5] Kail E., Kacsuk P., Kozlovszky M.: New aspect of investigating fault sensitivity of scientific workflows. *Intelligent Engineering Systems (INES), 2015 IEEE 19th International Conference on*, pp. 185–188, 2015.

[6] Meroufel B., Belalem G.: Adaptive time-based coordinated checkpointing for cloud computing workflows. *Scalable Computing: Practice and Experience*, vol. 15, 2014.

[7] Meroufel B., Belalem G.: Policy Driven Initiator in Coordination Checkpointing Strategies. *Recent Advances in Telecommunications, Informatics And Educational Technologies, Proceeding of the 5th European Conference of Computer Science*, p. 146153, WSEAS, 2014.

[8] Pietri I., Juve G., Deelman E., Sakellariou R.: A Performance Model to Estimate Execution Time of Scientific Workflows on the Cloud. *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science*, WORKS '14, pp. 11–19, IEEE Press, Piscataway, NJ, USA, 2014, `http://dx.doi.org/10.1109/WORKS.2014.12`.

[9] Starlinger J., Cohen-Boulakia S., Khanna S., Davidson S., Leser U.: Layer Decomposition: An Effective Structure-based Approach for Scientific Workflow Similarity. *Proc. of the 10th IEEE International Conference in eScience*, 2014.

[10] Therasa.S A.L., Sumathi.G, Dalya.S A.: Article: Dynamic Adaptation of Checkpoints and Rescheduling in Grid Computing. *International Journal of Computer Applications*, vol. 2(3), pp. 95–99, 2010, published By Foundation of Computer Science.

[11] Young J.W.: A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM*, vol. 17(9), pp. 530–531, 1974, `http://doi.acm.org/10.1145/361147.361115`.

Affiliations

Eszter Kail
Obuda University, John von Neumann Faculty of Informatics, 1034 Bécsi str. 96/b., Budapest, Hungary, `kail.eszter@nik.uni-obuda.hu`

Péter Kacsuk
University of Westminster, 115 New Cavendish Street, London, United Kingdom; MTA SZTAKI, 1518 Budapest, Hungary, `peter.kacsuk@sztaki.mta.hu`

Miklós Kozlovszky
Obuda University, John von Neumann Faculty of Informatics, Biotech Lab, 1034 Bécsi str. 96/b., Budapest, Hungary; MTA SZTAKI, 1518 Budapest, Hungary,
kozlovszky.miklos@nik.uni-obuda.hu

Received: 24.11.2015
Revised: 7.05.2016
Accepted: 8.05.2016