From tasks graphs to asynchronous distributed checkpointing with local restart

Romain Lion, Samuel Thibault

From tasks graphs to asynchronous distributed checkpointing with local restart

1st Romain Lion
University of Bordeaux
Inria Bordeaux - Sud-Ouest
Bordeaux, France
ORCID 0000-0002-4768-7036

2nd Samuel Thibault
University of Bordeaux
Inria Bordeaux - Sud-Ouest
Bordeaux, France
ORCID 0000-0001-6411-809X

***Abstract*—The ever-increasing number of computation units assembled in current HPC platforms leads to a concerning increase in fault probability. Traditional checkpoint/restart strategies avoid wasting large amounts of computation time when such fault occurs. With the increasing amount of data dealt with by current applications, these strategies however suffer from their data transfer demand becoming unreasonable, or the entailed global synchronizations. Meanwhile, the current trend towards task-based programming is an opportunity to revisit the principles of the checkpoint/restart strategies. We here propose a checkpointing scheme which is closely tied to the execution of task graphs. We describe how it allows for completely asynchronous and distributed checkpointing, as well as localized node restart, thus opening up for very large scalability. We also show how a synergy between the application data transfers and the checkpointing transfers can lead to a reasonable additional network load, measured to be lower than +10% on a dense linear algebra example.**

***Index Terms*—Fault tolerance, task-based programming, checkpoint-restart, buddy in-memory**

I. Introduction

As the scale of supercomputers widens, the failure probability increases, and it is thus not uncommon for a large computation job to encounter the complete loss of one of its computation nodes. The default reaction of traditional stacks is to abort the whole job, thus losing all the benefits of the computations already performed. A spectrum of solutions has been proposed in the literature. At one end, Algorithm-Based Fault Tolerance (ABFT) rethinks the whole application algorithm so that it can cope by itself with the loss of the node and its data, and continue with the remaining nodes. At the other end, transparent checkpointing periodically saves the whole state of the job on external storage, so that it can be restarted from the latest checkpoint instead of from its very starting point, thus reducing the amount of repeated computation down to at most the computations of a checkpoint period.

While ABFT is a very effective solution, it requires intrusive application changes which can make it much less maintainable and lowers programmer productivity. Conversely, transparent checkpointing exempts the programmer from any application modification, but saving the whole job state to external storage can be prohibitive, making the checkpointing duration very long and thus strongly affecting the job completion time. Tracking application data modification during computation allows to save to a checkpoint only the data which changed since last checkpoint, but this also brings a significant tracking overhead. Some frameworks let the application itself express which data should be saved, but this requires significant application modification.

On the other hand, the task-based programming paradigm trend has emerged to make parallel programming more effective and to automatically address the management of the complex composition of current platforms. Task-based programming introduces new synergy opportunities between applications and the task-based runtime system they are leveraging. The task graph gives the runtime a lot of insight and flexibility in the computation management.

In this paper, we propose to refine the classical checkpoint methods by introducing checkpoints as cuts in the task graph rather than cuts in the execution time. This enables the runtime to perform them completely *asynchronously* along the application computation, without stalling the program execution, and in a completely *distributed* way without global synchronization, thus opening for scalability on extremely large platforms. Since the task graph gives the runtime an exact view of the computation, this even allows it to restart failed nodes *locally*, i.e. without even interrupting computations progressing on other nodes. In this paper, we focus on the checkpointing mechanism itself, to explain how task-based programming makes it strongly effective and scalable, and to show how reasonable its overhead can be.

We start with giving more details on the context and related work. We then introduce the proposed checkpointing principle for task graphs, how it avoids many classical concerns of checkpointing frameworks, and some additional optimizations. Some experimental results are then presented, opening up to a discussion over the the checkpointing policy possibilities.

II. Context and related work

One of the many challenges brought by the future extreme scale computing systems is reliability. As the number of computing cores grows, the global failure rate increases as well. The MTBF of the system will inevitably decrease with the increasing number of used cores and we can expect tens of failures during a few hours computation job [12]. The future

```
task_insert(&f1, 0, W, A);
task_insert(&f2, 1, W, B);
task_insert(&f3, 1, RW, B, R, A, W, C);
task_wait_for_all();
```

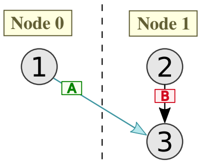

Fig. 1. Example of a simple distributed STF program over 2 nodes 0 and 1.

need for resilience is no more to demonstrate and poses a real challenge.

A first set of solutions comprises system-level checkpoint strategies that do not make assumptions about which data contains critical state for the distributed application. They are thus meant to make large-size checkpoints simply comprising all application data. While the transparency of these approaches is interesting for the programmers, the size of checkpoints is becoming more and more a no-go.

Some other solutions emphasize on incremental checkpointing [19], i.e. they avoid saving redundant data among checkpoints by computing differences between them or deducing it from memory pages access, but this comes with large runtime overhead. We will see that in our approach, it is possible to achieve this change detection for free, using mechanisms already implemented in task-based runtimes.

On the other hand, some application-level approaches such as FTI [4] or VeloC [14] are less affected by overhead issues, since they make the programmer specify the critical data in order to take more efficient checkpoints. Some of them offer to buffer application data prior to continuing computation in order to provide non-blocking checkpoints. The buffering however comes with significant time and memory overhead. While FTI and VeloC allow to save checkpoints among several storage media, we investigate an in-memory "buddy" checkpointing approach [10] since such an approach is the one which benefits the most from the task-based programming, as we will detail in this paper. Our proposal also does not preclude from future extension to on-disk checkpointing.

Buddy in-memory checkpointing can be a limiting aspect as it does not cover multiple-node failure scenarios, but it still provides efficient resilience as these scenarios are unlikely. This approach also makes it possible to adopt a local restart strategy such as [13], with computation kept running on the surviving nodes. This is made possible thanks to the ULFM MPI proposal implementation [5] and by using message logging [3], [7]. In this paper, we emphasize only on the checkpoint aspects, the rollback and the message logging are not evaluated.

III. Introducing checkpoints in task graphs

We here describe our proposal for introducing checkpoints in task graphs, and the immediate benefits in the checkpoint principles themselves.

A. Programming paradigm

In previous work [2], we have shown that task-based programming allows to achieve high performance over large computation platforms, with a high productivity thanks to the *Sequential Task Flow* programming model. Its principle is to express distributed applications with a sequential-looking source code, as shown on Figure 1. This example expresses that f1 is to be executed on node 0 [1], and will write its result to data A. Meanwhile, f2 is to be executed on node 1, and will write its result to data B. Eventually, f3 is to be executed on node 1, and will read from and write to data B, but also read from data A, and write to data C. The runtime will automatically infer, from the A and B data dependencies, the task dependency between the first two tasks and the third task, and thus the resulting task graph shown on the right. It will also infer that data A has to be transferred from node 0 to node 1, more precisely that node 0 has to send data A to node 1 (i.e. an `MPI_Isend` call), and node 1 has to receive data A from node 0 (i.e. an `MPI_Irecv` call).

```
task_insert(&f1, 0, W, A);
task_insert(&f2, 1, W, B);
checkpoint();
task_insert(&f3, 1, RW, B, R, A, W, C);
task_insert(&f4, 1, RW, B, R, A);
task_insert(&f5, 0, RW, A, R, B);
checkpoint();
task_insert(&f6, 1, RW, B, R, A, R, C);
task_insert(&f7, 0, RW, A, R, B);
task_wait_for_all();
```

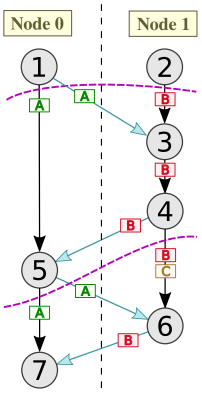

Fig. 2. Example of a distributed STF program with inserted checkpoints cuts as the violet dashed curves.

In the end, all nodes unroll the whole task graph [2], and each node determines by itself which tasks of the graph it will execute, but also which data it has to send or receive so as to fulfill data dependencies over the network. All of this is thus achieved deterministically in a completely distributed way without any global synchronization between nodes. Tasks can be affected to nodes as in this example, but they can also be bound dynamically as long it follows a deterministic policy in order to have the same task graph among all nodes, without requiring consensus between the nodes.

As shown on Figure 2, adding checkpoints to this programming model consists in introducing `checkpoint()` calls within the program. They effectively cut the task graph inferred by the STF model, between the tasks inserted before the call, and the tasks inserted after the call. Here, at the first checkpoint, node 0 will thus save data A produced by task 1, while node 1 will save data B produced by task 2. At the second checkpoint, node 0 will save the new value of data A produced by task 5, and node 1 will save data C produced by

[1] Here the placement on nodes is made explicit for simplicity, but STF allows it to be implicitly inferred from data placement for even more productivity.

[2] As explained in [2], pruning can be used to reduce the cost. This pruning can be adapted to the context of this work.

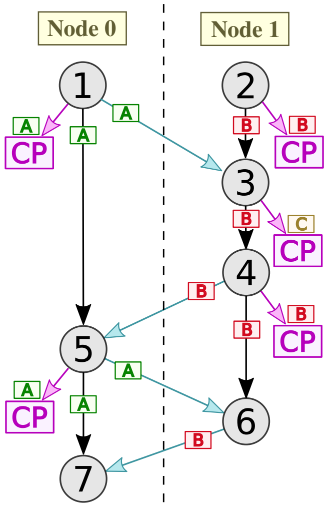

Fig. 3. Resulting asynchronous checkpoints.

task 3, and data B as produced by task 4. It is worth noticing that while the application didn't specify precisely which data should be saved, the runtime can determine that data C does not need to be saved in the first checkpoint, since no task has written any value in it yet at that "time" of the task graph.

Again, since this is done identically on every node, all nodes agree on exactly what will be saved in the checkpoints, without any need for synchronization at run time.

The placement of the `checkpoint()` calls is quite natural for e.g. iterative applications such as conjugate gradient methods or stencils: the calls can simply be placed at the end of each iteration, and the periodicity of checkpoints can be controlled by ignoring all calls except one every N. The checkpointing addition for existing task-based iterative applications thus reduces to a one-liner. For more convoluted applications, the placement is more questionable; we experimented with the Cholesky factorization, as discussed in Sections V and VI.

In this preliminary work, each node sends the content of its checkpoints through the network to another *buddy node*, where it is simply kept in memory. Saving the checkpoints to external storage will be addressed in future work. Saving checkpoints to the memory of other nodes actually brings interesting questions of synergy with the actual transfers already required by the application, discussed in Section IV-A. The choice of the buddy node (i.e. the checkpoint mapping) is discussed in Section VI-A. To give an initial idea, a simple choice is to save checkpoints of node n on node $(n+1)\%nb\_nodes$, thus conveniently distributing both the additional memory and network load.

B. Avoided pitfalls

Checkpointing techniques typically suffer from various drawbacks, as described by Elnozahy et al. [11]. Introducing checkpoints as cuts in the task graph however solves most of them.

Firstly, the actual data transfers for checkpoints can be achieved completely *asynchronously*, since they are mere additions to the task graph, as shown on Figure 3. There is indeed no requirement for nodes to start sending their respective checkpoints at the same time; and even for a given node, the actual transfers can be dispersed over time. For instance, on Figure 3, we can notice that since data C is the output of task 3, it can be sent to the buddy node as soon as task 3 terminates, without having to wait for task 4, whose output in data B will have to be transferred for the same checkpoint. This gives the runtime a lot of latitude to perform the data transfer, between the time when the data is produced, and the time when it is overwritten by another task (unless copy-on-write is used, as discussed in Section IV-B). The checkpoint thus synchronizes with the application progression with a very fine grain, unlike other approaches which require at least a complete synchronization with the application on each node. Different checkpoint instances may even *overlap*, as discussed in Section V. The checkpointing transfers can also be left as a background duty, to tend to avoid slowing down the communications required for the application progress. For a given node, the completion of a given checkpoint can be recorded (and thus the the previous checkpoint be discarded) when all the data transfers for the checkpoint are completed, completely independently from the completion of checkpoints for other nodes, i.e. the *output commit* is completely local.

Secondly, the obtained checkpoints are *non-coordinated*, or more precisely, they are only *implicitly coordinated*. Indeed, since each node unrolls the task graph, each node knows which buddy node it will save its checkpoints to, and each node knows which nodes it will buddy for, and thus the checkpoints it will save, without any need for synchronization before the actual checkpoint data transfers. The cuts in the task graph thus provide consistent restarting lines in the task graph which are actually noticed in a completely distributed way.

Eventually, with the addition of message logging[3] the rollback of a failed node can be performed *locally*, without involving other nodes beyond the buddy node and the nodes which have exchanged data with the failed node. Indeed, when the *replacement* node starts, it only has to receive the latest complete checkpoint from the buddy node, and receive data messages from the nodes which had exchanged data with the failed node since the point in the graph of the checkpoint. Conversely, the buddy node knows the complete checkpoint it can send to the replacement node, and the nodes which had exchanged data with the failed node can be notified by the buddy node from which point in their message log they should re-emit the previous data messages to the replacement node. In other words, the failed node rollback is as local as the application algorithm is. This also circumvents the domino effect: at worse the progression of the replacement node will be very late, which may delay the progression of other nodes through data dependencies, but that can be mitigated

[3] This is not covered in this paper, but it essentially follows classical message logging principles [3], [7]

through load rebalancing. Load rebalancing can be performed by choosing a new task-over-node distribution, as long as this distribution is agreed in consensus among all nodes to ensure a global consistency of the task graph [18]. The submission has to reach a barrier to ensure all nodes are synchronized, but this can be made while the previously-submitted tasks keep executing, thanks to the asynchronicity of the runtime.

It is noticeable that working with this kind of runtime, we are only saving application-related data into the checkpoints. The runtime system's state is only defined by these data, so there is no runtime data to save into the checkpoint. For example on performing a restart, the application declares the data into the runtime system, and the runtime initializes it according to the checkpoint content. The application then submits the tasks after the checkpoint line and the execution is resumed. The data contained in a checkpoint are therefore only the ones specified into the application code.

To summarize, in the proposed scheme, both the checkpoint data transfers and the recovery are completely distributed. A global consistency is insured to avoid domino effects, but this consistency is built implicitly from the task graph, the implicit coordination comes from all nodes unrolling the same graph in a deterministic way, thus providing Piece Wise Determinism [15]. Additionally, as discussed in Section VI-A, the choice of the buddy nodes can be arbitrary, so it can be easily tuned to follow the actual platform network links, for maximal localization of the checkpoint transfers.

C. Implementation

The checkpointing scheme proposed above was implemented in the StarPU task-based distributed runtime system. The support for distributed execution of StarPU resides in its StarPU-MPI layer, which is based on the MPI communication interface. The support for checkpointing was implemented on top of this layer, it is composed of about 1500 lines of C code, and initially did not require modifications of the core of the StarPU runtime, the existing data acquisition core primitives were sufficient to implement it. It should thus be easy to transpose this checkpoint proposal to other runtime systems. Some optimizations described below however required some extensions. Besides, this checkpointing support is essentially independent from the MPI standard, and would be easily ported on top of other communication standards, provided that they have support for error reporting and failure recovery.

IV. Optimization opportunities

The principles described above lead to scalability thanks to all operations being local and asynchronous. Some further optimizations are made possible thanks to the knowledge of the task graph.

A. Cache integration

In previous work [2], we have shown that caching data as implemented in StarPU-MPI is an essential part of distributed runtime systems. For instance on Figure 2, both tasks 3 and 4 read data A, but StarPU-MPI transfers data A only once from node 0 to node 1, thus saving network bandwidth. Our checkpointing implementation completely integrates with this cache support: if some data of a checkpoint happens to be also used by a task on the buddy node, only one transfer is required. For instance in the case of Figure 2, saving the checkpoints for node 0 (i.e. data A) on node 1 is actually costless, since node 1 already needs data A for its own computations, so the checkpoint does not introduce any additional network transfer. Conversely, saving the checkpoints for node 1 (i.e. data B and data C) on node 0 is not so costly, since data B is already required for tasks 5 and 7.

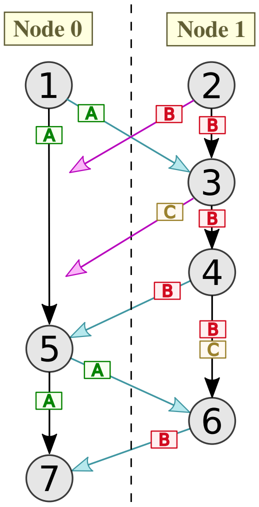

Fig. 4. Eventual data transfers between nodes 0 and 1.

Furthermore, integrating checkpoints with the existing cache support provides the factorization of data transfers across checkpoints. For instance in the case of Figure 2, if data C is not modified by tasks in the remainder of the program, further checkpoints will include it but not involve any new transfer over the network: the same cached copy will be carried over between checkpoints. The existing support for caching already notices the submission of tasks which modify the data to invalidate outdated copies. As a consequence, checkpoints naturally only trigger transfers for data which have been modified by tasks since the last checkpoint.

As a result, in the case of Figure 2, only two additional transfers are required for the checkpoints, as shown on Figure 4: data B as output by task 2, and data C as output (early) by task 3. This shows that there can actually be a synergy between the application algorithm (how its tasks and data are distributed), and the checkpointing (how it distributes the checkpoints data). We further discuss this in Section VI-A. This is also an argument, in some cases, in favor of saving checkpoints in the memory of another computation node rather than external storage, since for some application algorithms, some other nodes may already require the data for the computation itself anyway.

B. *Copy-on-write*

As explained in Section III-B, the asynchronicity of the proposed checkpoint principle can provide a lot of latitude to perform the checkpoint transfers, between the production by a task of the data to be checkpointed, and the overwrite of the data by another task. In our example, node 0 has ample time to send data A to node 1, before data A is overwritten by task 5. In other cases however, the checkpoint transfers may stall the execution of tasks. Node 1 indeed has to defer the execution of task 3 until after data B is transferred to node 0 for the first checkpoint, since task 3 overwrites the content of data B. A solution, to avoid such stalling of the execution of the application, is to create a duplicate of the output of task 1 in data B, so that this duplicate can be transferred to node 0 while task 3 can execute. Performing such a duplicate in the case of data A on node 0 would however be a waste of memory. We thus introduce a *copy-on-write* strategy, in the same sense as Operating Systems' memory management: when sending data for a checkpoint, we actually send an *alias* of data B, the latter being marked as copy-on-write. In the case of data B on node 1, when task 3 is ready to execute, a duplicate of data B is automatically created to separate the alias from the real data. In the case of data A on node 0, when task 5 is ready to execute, the transfer of data A to node 1 will already be largely over, and thus the alias dropped, so data A can be overwritten and no duplicate will have been created.

The support for copy-on-write is further helpful for optimizing the support of message logging (not detailed in this paper). When nodes save to the message log some data that they send over the network, they actually save an alias of the data, the latter being marked as copy-on-write. If no subsequent task modifies the data until the log gets cleared, no duplicate is actually performed.

More generally, the support for copy-on-write can be useful in task-based runtime systems. Some in-place algorithms such as QR [1] indeed need to make a duplicate of some matrix tiles to avoid an anti-dependency that would otherwise limit parallelism. But such copy can be useless depending on the scheduling order. Support for copy-on-write will nicely optimize such a case.

V. Results

To evaluate our approach we chose to perform a study on the Cholesky decomposition application, as it provides interesting graph properties, unlike iterative applications. These latter applications (such as stencils or conjugate gradient) indeed have a behaviour that is very regular, thus making it much simpler to tune the checkpointing placement and frequency to obtain an efficient result. The Cholesky decomposition, on the contrary, produces a convoluted task graph which thus raises checkpoint placement questions which we discuss in details in Section VI-B. We eventually made the Cholesky task graph submitted per column, with checkpoints being placed between the submission of each column.

For our experiments we use 25 Miriel nodes from the Plafrim experimentation platform. Each node has 2 Intel® Xeon® E5-2680 v3 Haswell 12cores @ 2.5 GHz and 128 GB of memory (5.3 Go/core @2 933 MHz), and the nodes are interconnected with an Infiniband 40 Gb/s network. We use MKL2019 with AVX2, and OpenMPI 4.0.3. The average GEMM performance with the chosen tile size (320x320) is 60.2 GFlop/s per core, which allows a theoretical performance bound of 36 120 GFlops for the available 600 cores. We performed the measures with the StarPU "lws" (local work stealing) scheduling policy. This policy distributes initial tasks evenly over available cores ; when a task terminates, its dependent tasks are attributed to the core where the dependency ran ; when a core runs out of tasks to perform, it steals tasks from other cores, starting from the cores that are the closest in the architecture topology. This is thus a policy that is very effective at avoiding scheduling contention even with a high number of cores.

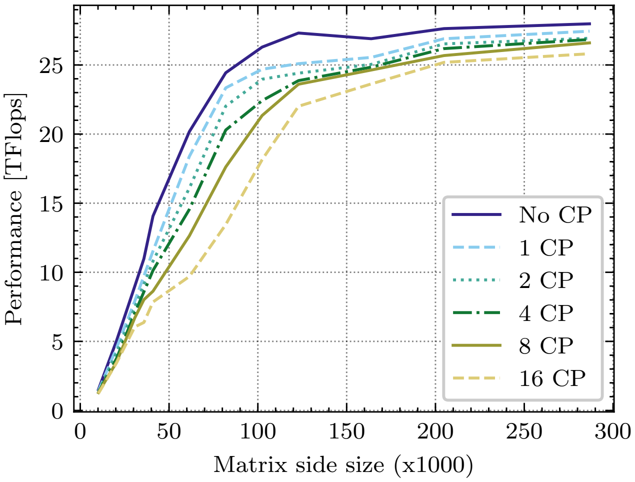

Fig. 5. Performance on 25 MPI nodes of the Cholesky column-wise task-based algorithm, with different numbers of checkpoints.

A. *Performance overhead*

Figure 5 shows the application performance according to the matrix side size, with different numbers of checkpoints taken over the execution. The interesting part is that even if as the matrix size grows the checkpoint size grows (since more tasks will have produced results), the checkpoints overhead actually reduces. This is majorly due to the fact that the computation time increases and thus the checkpoints time period decreases. For these measures, the checkpoint frequencies are actually far greater than what would usually be used, except for the greater matrix sizes. For instance for matrix side size 287 000, an execution without checkpoints lasts 280 s. For 16 checkpoints it lasts 304 s (+24 s i.e. +8.6%) for an average checkpoint period of 19 s. The overhead is however more important for smaller matrices; for the matrix side size 121 600, the 16 checkpoints execution lasts 5.7 s more than the normal execution, that lasts 22.5 s (this leads to an overhead of 25%, and a checkpoint frequency of 1.76 s). We thus get an idea how reasonable the overhead would be for much longer checkpointing periods.

#CP	Blind CP data (% additional)	No-redund. CP data (% additional)	Real CP data (% additional)
1	1.18 GB (+11.5%)	0.892 GB (+8.67%)	0.239 GB (+2.33%)
2	2.37 GB (+23.1%)	1.055 GB (+10.3%)	0.423 GB (+4.11%)
4	4.74 GB (+46.1%)	1.143 GB (+11.1%)	0.622 GB (+6.05%)
8	9.49 GB (+92.2%)	1.175 GB (+11.4%)	0.776 GB (+7.58%)
16	18.98 GB (+184%)	1.185 GB (+11.5%)	0.888 GB (+8.64%)

TABLE I
AMOUNT OF CHECKPOINTING DATA TRANSFERS AVERAGED PER NODE FOR MATRIX SIDE SIZE 121 600, FOR DIFFERENT NUMBERS OF CHECKPOINTS. THE CHOLESKY ALGORITHM ITSELF ALREADY INDUCES AN AVERAGE OF 10.28 GB DATA SENT PER NODE, THE PERCENTAGE IN PARENTHESES PROVIDES HOW MUCH CHECKPOINT DATA IS ADDITIONALLY SENT PER NODE RELATIVELY TO THAT.

B. Communication overhead

The amount of additional data transfers induced by these checkpoints is expressed in Table I. There we show the average quantity of Checkpoint Data exchanged per node, for matrix side size 121 600. The table exhibits the results of different approaches, to observe the benefit of the properties we discussed in the previous section. The “blind checkpoint data” (“Blind CP Data” column) is the quantity of data which would be transferred when the checkpoint mechanism is unaware of actual data changes. In this case each checkpoint saves the whole matrix triangle, which weights 29.57 GB, distributed over 25 nodes, thus 1.18 GB per node. The no-redundancy checkpoint (“No-redund. CP data” column) is the quantity of data sent when using an incremental checkpointing system, which does not transfer a data already present in previous checkpoints, i.e. a system that tracks the data changes. For instance when only one checkpoint is taken in the middle of the submission, that induces saving the left half part of the matrix triangle, which weights 3 quarters of the matrix triangle, distributed over 25 nodes, thus 0.892 GB per node on average. When two checkpoints are taken, at the first third and the second third of the submission, the first saves the left third of the matrix triangle, while the second saves the middle third of the matrix triangle, thus 1.055GB per node on average, and similarly for larger numbers of checkpoints. The obtained transfers savings are dramatic. Finally the measured amount of checkpointing data sent with our proposed checkpointing principle is shown in column “Real CP data”. While benefiting from the same reduction than the no-redundancy checkpoints, we reach even further reduction, thanks to the reuse of computational data. Some data are indeed needed for computation purpose, and are thus already on the buddy node; our proposal then avoids sending the data again, thus getting a significant communication saving.

These results are limited because the Cholesky factorization is only a small simple of the application spectrum that the StarPU runtime can execute. Even if extra experiments have not been conducted for this paper, we can anticipate some potential results. The proposed checkpoint solution should be efficient for iterative convergent application such as conjugate gradient: the application dynamic state is only the converging solution vector, so the checkpoint content is rather small. The performances for this kind of application should therefore be better than with the evaluated Cholesky. For a stencil based application however, the checkpoint communications may deteriorate the performances. Each checkpoint has to save the entire dataset, and this is not usually transferred by the application (it only transfers the borders), so there is no synergy with the application communications. Consequently this kind of application may have less good result than the ones presented here. The overhead can however still be tuned down trivially by lowering the checkpointing frequency.

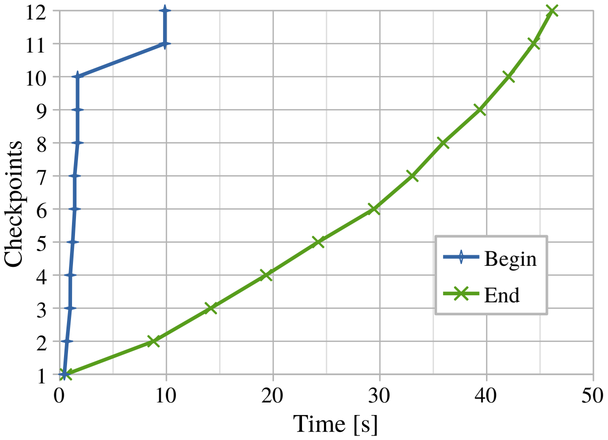

Fig. 6. Begin and End times of 12 checkpoints taken linearly in the iterations of the Cholesky column-wise algorithm.

C. Checkpoints progress

As discussed in Section III-B, the progression of checkpoints is completely asynchronous, and the data to be saved in a given checkpoint can be transferred as soon as it is produced by a task, thus spreading over time the additional communications of the checkpoint. This effect is shown on Figure 6 which shows for each of 12 checkpoints (numbered from 1 to 12) when the first data for a given checkpoint is transferred, and when the last data for the same checkpoint is transferred. We notice a very large overlap between the respective progressions of the checkpoints. Indeed some tasks are executed very early in the execution because of their high priority due to belonging to the critical path of the task graph, and despite having been submitted at the end of the task graph, and thus belonging to a later checkpoint. Since the corresponding data is made available for checkpointing immediately after the execution of these tasks, we observe that the first data transfer for all checkpoints occurs very early. This effect is however limited to the very few tasks which are close to the critical path, other tasks mostly follow column-wise completion order and thus column-wise checkpointing. Another expression of the submission code would have given another distribution; a submission that submits tasks by priority order would naturally see the checkpoints start later during execution, and the checkpoint termination would be different according to the task load evolution per checkpoint.

Another effect worth noticing on Figure 6 is that the period of the checkpoint completion time is not fixed: checkpointing progression becomes faster and faster. Indeed, the checkpoints are submitted linearly during the submission of the task graph, i.e. with a fixed iterative period; but in the Cholesky decomposition algorithm the submission iterations submit a decreasing number of tasks. This could be corrected by a compensated checkpoint iteration period, which would lead to a better checkpoint distribution over time, as discussed in Section VI-C.

VI. Trade-off discussion

The previous section provided a glimpse into the kind of overhead and network bandwidth cost that can be expected from the checkpointing principle proposed in this paper. We here further discuss the trade-offs that can be made when introducing checkpointing in a task-based application.

A. Checkpoint mapping over nodes

In previous work [2], we have shown that with the STF programming paradigm, the data mapping over nodes can be chosen at will, without having to modify the task submission part of the application source code; this allows to easily tune it to reduce data transfers. In the checkpointing principle proposed here, the situation is similar: the application specifies a checkpoint mapping, i.e. for each node specify which other node it will save its checkpoints on (the *buddy* node), ahead of the task submission loop. This makes it convenient and safe for trying different checkpoint mapping strategies: the execution will simply follow the provided mapping, and at worse the performance will be degraded because e.g. the mapping specifies to save the checkpoints of all nodes on just a few nodes whose network links will thus be jammed.

The best choice for the checkpoint mapping depends on the application and its data mapping. As mentioned in Section IV-A, some synergy emerges between the application algorithm and the checkpoint mapping. When for algorithmic reasons some data needs to be transferred from some node i to another node j, making node j the buddy node for node i comes at no extra network cost since the checkpoint will reuse the received data.

This choice also depends on the platform network: since the traffic between a node and its buddy node will be increased, the checkpoint mapping would rather be chosen such that this traffic will get routed e.g. through a direct link. For instance, with the typical habit of making MPI rank numbering increment by proximity, the simple choice of saving checkpoints of node n on node $(n{+}1)\%nb\_nodes$ will actually tend to generate such close routing. In general, the application data mapping will already be chosen so as to follow the platform network, thus reinforcing the synergy mentioned in the previous paragraph.

For instance, for dense linear algebra, the 2D block-cyclic tile mapping [16] maps tile (i,j) of a matrix on node $(i\%P) * Q + j\%Q$ where P and Q are chosen such as $P{*}Q = nb\_nodes$. This mapping is well-known for providing reasonably-optimized distributed execution, by limiting the amount of data transfers while keeping appropriate pipelining [9]. It indeed builds up an affinity between nodes that have tiles on the same rows, and between nodes that have tiles on the same columns, to benefit from the fact that linear algebra often propagates results along rows and columns. The checkpoint mapping can follow the same principle by e.g. saving checkpoints of node n on node $(n + 1)\%Q$. This leverages the affinity of nodes that have tiles on the same rows. In practice (and as shown in the previous section), the buddy node will indeed most often have already received the data to be saved in the checkpoint because the linear algebra algorithm will have broadcasted it to all nodes of the same row. And since there is affinity between several nodes within each panel, it should potentially be easy to save the checkpoints in several nodes in order to tolerate several failures. Another interesting choice could be $(n{+}Q)\%(nb\_nodes)$ which would leverage the affinity of nodes that have tiles on the same column. The former might however be preferable to better match with the proximity-based numbering of the nodes in the platform network.

For simplicity in this paper, we have assumed that the checkpoint for a given node is saved on a single buddy node. Our implementation however supports saving different pieces of the checkpoint on different buddy nodes, which provides even more flexibility to follow even more closely the application data transfers patterns. Each node can collect the acknowledgments that its buddy nodes send when they have received all of their pieces of a checkpoint. When a node fails, its buddy nodes can establish a consensus to determine which checkpoint version should be sent to the replacement node.

B. Checkpoint placement in the task graph

The placement of the checkpoints among tasks, i.e. the placement of the cuts in the task graph, is also up for discussion. Again, the STF programming paradigm allows to place it at will within the source code, independently from the choice of the checkpoint mapping, to try different strategies, with different resulting performance depending on the application. The only constraint is that all nodes must insert the checkpoints at the same place, for the checkpointing cuts to be consistent.

To reduce the amount of data to be saved in the checkpoints, they should rather be placed where the least amount of data is live. For iterative solvers for instance, they should be placed at the end of the iterations, to automatically avoid saving the temporary data used inside an iteration. It is usually quite natural to introduce checkpoints in such fork-join task-based applications, whose source code will usually clearly exhibit the join point at the end of the iteration, where the checkpoints can thus be trivially inserted.

In other cases such as the Cholesky factorization taken as example in Section V (and more generally linear algebra algorithms notably), the placement of checkpoints is more tricky. This is notably because iterations of the main loop of the usual source code are supposed to overlap. For instance, the usual task-based expression of the Cholesky factorization

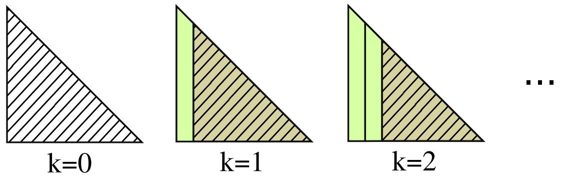

Fig. 7. Task submission of triangular-wise expression from Algorithm 1. White areas are initial state, hatched areas are submitted tasks during the current iteration (one per tile), green areas are final states, brown areas are intermediate states.

is triangle-wise, as shown on Algorithm 1 and Figure 7 because it is simple for numerical analysis experts to proofread. Benefiting from most of the parallelism of the task graph however requires overlapping the executions of the tasks of several iterations of the k loop; scheduling tasks strictly in the submission order would yield to low parallelism at the start of each loop iteration. This overlapping is commonly achieved by setting priorities on the tasks, so that e.g. GEMM tasks with large n and m indices are executed late even with low k indices, while POTRF and TRSM tasks are executed early to unleash parallelism as early as possible, since they are on the critical path.

Algorithm 1 STF tile Cholesky, triangle-wise

```
for (k = 0; k < NT; k++) do
    task_insert(&POTRF, RW, A[k][k]);
    for (m = k+1; m < NT; m++) do
        task_insert(&TRSM, R, A[k][k], RW, A[m][k]);
    for (n = k+1; n < NT; n++) do
        task_insert(&SYRK, R, A[n][k], RW, A[n][n]);
        for (m = n+1; m < NT; m++) do
            task_insert(&GEMM, R, A[m][k], R, A[n][k],
                RW, A[m][n]);
    checkpoint();
task_wait_for_all();
```

Submitting all the non-prioritized GEMM tasks is however a burden for the runtime to have to keep managing possibly millions of tasks that will actually be executed very late. A way to avoid such useless submissions is to use the polyhedral analysis of a source-to-source compiler such as PPCG [20] to rewrite the algorithm into a column-wise form as shown on Algorithm 2 and 8, which produces the same task graph, but column by column instead of triangle by triangle. This allows to make the progressive task submission of our previous work [2] behave much more efficiently. Another way would be to use a parametric representation of the task graph, such as used by PaRSEC [6], and unroll the task graph by columns, the principle remains the same.

Inserting a checkpoint in Algorithm 1 at the end of the k loop would be very problematic: after the first iteration ($k = 0$) the checkpoint would have to save the whole matrix triangle! After the second iteration ($k = 1$) it would have to save the whole matrix triangle but the first column of tiles, etc. Also, if priorities are used to defer the execution of GEMM tasks with high n and m indices, the corresponding data for the checkpoints would be available almost at the end of the execution only. Inserting a checkpoint in Algorithm 2 at the end of the n loop is however much more appropriate. After the first iteration ($n = 0$) the checkpoint has to save the first column of tiles of the matrix only; after the second iteration ($n = 1$) the checkpoint has to save the second column of tiles only; etc. This also much better follows the actual execution order of tasks as observed with the usually-used task priorities, so that the checkpoints complete progressively along the execution. This is why we used this form of expressing the Cholesky factorization for the results of Section V.

Algorithm 2 STF tile Cholesky, column-wise

```
for (n = 0; n < NT; n++) do
    for (k = 0; k < n; k++) do
        task_insert(&SYRK, R, A[n][k], RW, A[n][n]);
    task_insert(&POTRF, RW, A[n][n]);
    for (m = n + 1; m < NT; m++) do
        for (k = 0; k < n; k++) do
            task_insert(&GEMM, R, A[m][k], R, A[n][k],
                RW, A[m][n]);
        task_insert(&TRSM, R, A[n][n], RW, A[m][n]);
    checkpoint();
task_wait_for_all();
```

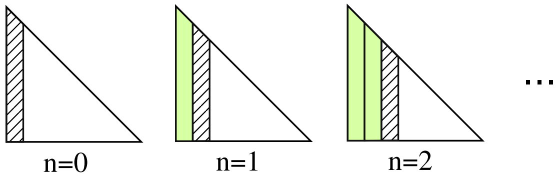

Fig. 8. Task submission of column-wise expression from Algorithm 2. White areas are initial states, hatched areas are submitted tasks during the current iteration (n+1 per tile), green areas are final states.

The behavior observed with this classical example yields to the general principle that the placement of checkpoints should be rather coherent with the actual task execution progression, so that checkpoints can complete at regular times. It must however be noted that the requirement is not strict. In the case of Algorithm 2, the task priorities are typically set so that the execution of POTRF and TRSM tasks do overlap between iterations of n, so as to release parallelism early, so we do have some amount of overlapping between the execution of the tasks before the checkpoint to be completed and the execution of the tasks after it. This overlapping is however quite limited, as the regular completion time of checkpoints shows in Section V.

C. *Frequency of checkpoints*

In the Subsection VI-B above, we typically introduced a checkpointing slot at each iteration of the most external *for* loop. Such a frequency may however be way too high compared to the checkpointing management cost; we have seen in Section V the cost of using a high checkpoint frequency. Such high frequencies are generally not useful anyway, compared to the cost of the checkpoint and of the restart. Optimized checkpointing frequency can be determined [8], [21] according to the platform and application caracteristics, and the checkpointing calls inserted at each iteration of the source code would be selectively enabled accordingly; in our experiments we selectively enabled them by hand. The models used for the optimization however generally assume that the checkpointing steps are completely separate from each other, while in our checkpointing principle, steps can overlap (as seen in the Cholesky example). The overhead can still be evaluated respectively for each checkpoint, and it is the termination time of checkpoints that is to be observed. The models also often assume that the sets of tasks separated by the checkpoints are identical, so as to reach a steady-case scenario. In the Cholesky example, this is not the case, the frequency of checkpoints should be regulated along the loop over n. Polyhedral analysis could however provide an estimation of the amount of computation of each loop, that models could take into account to provide a regulated frequency.

VII. Conclusion and future work

We have proposed a checkpointing principle which integrates closely with task-based programming. We have shown that this leads to a completely asynchronous and distributed checkpointing mechanism, and allows for a completely local restart of failing nodes. This work hence opens potential for very large-scale checkpointing support. We have also shown that, in some application cases, a synergy can be found between the data transfers for the application algorithm and the data transfers for the checkpoints, thus reducing the additional network pressure of the latter.

This work is only preliminary but very promising, and now deserves experimentation on larger scales and with different checkpoint mappings. This paper has not discussed the case of losing several nodes simultaneously. If none of them is the buddy node for another, the respective local recoveries can proceed concurrently, otherwise a secondary buddy node would be needed, with the same general principles as described in this paper. Instead of our proposed buddy in-memory checkpoints approach, saving checkpoints to external disk storage (and e.g. leverage libraries such as TAPIOCA [17]) would avoid the memory load on the nodes. The same principles as shown in this paper should be applicable so that integration with the task graph would also allow to achieve progressive asynchronous write-back to disk. Lastly, since a replacement node restarts earlier in the task graph than the current progression of the other (non-failed) nodes, dependencies in the task graph may make this delay impact the other nodes, leading to idle time there. Dynamic load rebalancing would be able to compensate such a delay. The progressive submission of the tasks allows for such a balancing to be done while respecting the programming paradigm of STF.

VIII. Acknowledgments

This work has received funding from the EU's Horizon 2020 research and innovation program, under grant agreement No. 801015 (EXA2PRO, http://www.exa2pro.eu).

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

References

[1] Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., Tomov, S.: "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators". In: IPDPS'11 - 25th IEEE International Parallel & Distributed Processing Symposium. Anchorage, Alaska, USA (May 2011)

[2] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.: "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model". TPDS - IEEE Transactions on Parallel and Distributed Systems (Dec 2017)

[3] Alvisi, L., Marzullo, K.: "Message logging: pessimistic, optimistic, causal, and optimal". IEEE Transactions on Software Engineering **24**(2) (Feb 1998), conference Name: IEEE Transactions on Software Engineering

[4] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., Matsuoka, S.: "Fti: High performance fault tolerance interface for hybrid systems". In: SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)

[5] Bland, W., Bouteiller, A., Herault, T., Bosilca, G., Dongarra, J.: "Post-failure recovery of MPI communication capability: Design and rationale". The International Journal of High Performance Computing Applications **27**(3) (Aug 2013)

[6] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., Dongarra, J.: "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability". Computing in Science and Engineering **15**(6) (Nov 2013)

[7] Bouteiller, A., Bosilca, G., Dongarra, J.: "Redesigning the message logging model for high performance". Concurrency and Computation: Practice and Experience **22**(16) (Nov 2010)

[8] Di, S., Robert, Y., Vivien, F., Cappello, F.: "Toward an optimal online checkpoint solution under a two-level hpc checkpoint model". IEEE Transactions on Parallel and Distributed Systems **28**(1), 244–259 (2017)

[9] Dongarra, J., Walker, D.: "Software libraries for linear algebra computations on high performance computers". SIAM Review **37**(2) (1995)

[10] Dongarra, J., Herault, T., Robert, Y.: "Revisiting the Double Checkpointing Algorithm". In: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Pchd Forum. IEEE, Cambridge, MA, USA (May 2013)

[11] Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: "A Survey of Rollback-recovery Protocols in Message-passing Systems". ACM Comput. Surv. **34**(3), 375–408 (Sep 2002)

[12] Gupta, S., Patel, T., Engelmann, C., Tiwari, D.: "Failures in large scale systems: long-term measurement, analysis, and implications". In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '17. Denver, Colorado (2017)

[13] Losada, N., Bosilca, G., Bouteiller, A., González, P., Martín, M.J.: "Local rollback for resilient MPI applications with application-level checkpointing and message logging". Future Generation Computer Systems **91**, 450–464 (Feb 2019)

[14] Nicolae, B., Moody, A., Gonsiorowski, E., Mohror, K., Cappello, F.: "VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale". In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (May 2019), iSSN: 1530-2075

[15] Strom, R., Yemini, S.: "Optimistic recovery in distributed systems". ACM Transactions on Computer Systems (TOCS) **3**(3), 204–226 (Aug 1985)

[16] Susan Blackford: “The Two-dimensional Block-Cyclic Distribution” (May 1997), http://www.netlib.org/scalapack/slug/node75.html

[17] Tessier, F., Vishwanath, V., Jeannot, E.: “TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers”. In: CLUSTER 2017 - IEEE International Conference on Cluster Computing. pp. 1–11. IEEE, Honolulu, United States (Sep 2017)

[18] Thibault, S.: On Runtime Systems for Task-based Programming on Heterogeneous Platforms. Habilitation à diriger des recherches, Université de Bordeaux (Dec 2018), https://hal.inria.fr/tel-01959127

[19] Vasavada, M., Mueller, F., Hargrove, P.H., Roman, E.: “Comparing different approaches for Incremental Checkpointing: The Showdown”. In: Ottawa Linux Symposium (2011)

[20] Verdoolaege, S., Juega, J.C., Cohen, A., Gómez, J.I., Tenllado, C., Catthoor, F.: “Polyhedral parallel code generation for cuda”. ACM Trans. Archit. Code Optim. **9**(4), 54:1–54:23 (Jan 2013)

[21] Young, J.W.: “A first order approximation to the optimum checkpoint interval”. Commun. ACM **17**(9), 530–531 (Sep 1974)