Experiments on Checkpointing Adjoint MPI Programs

Ala Taftaf, Laurent Hascoët

Experiments on Checkpointing Adjoint MPI Programs

A.Taftaf and L. Hascoët

INRIA, 2004 Route des Lucioles, Sophia-Antipolis, France, {Elaa.Teftef, Laurent.Hascoet}@inria.fr

Checkpointing is a classical strategy to reduce the peak memory consumption of the adjoint. Checkpointing is vital for long run-time codes, which is the case of most MPI parallel applications. However, for MPI codes this question has always been addressed by ad-hoc hand manipulations of the differentiated code, and with no formal assurance of correctness. In a previous work, we investigated the assumptions implicitly made during past experiments, to clarify and generalize them. On one hand we proposed an adaptation of checkpointing to the case of MPI parallel programs with point-to-point communications, so that the semantics of an adjoint program is preserved for any choice of the checkpointed part. On the other hand, we proposed an alternative adaptation of checkpointing, more efficient but that requires a number of restrictions on the choice of the checkpointed part. In this work we see checkpointing MPI parallel programs from a practical point of view. We propose an implementation of the adapted techniques inside the AMPI library. We discuss practical questions about the choice of technique to be applied within a checkpointed part and the choice of the checkpointed part itself. Finally, we validate our theoretical results on representative CFD codes.

I. Introduction

Checkpointing is a classical technique to mitigate the overhead of adjoint Algorithmic Differentiation (AD). In the context of source transformation AD with the Store-All approach, checkpointing[1] reduces the peak memory consumption of the adjoint, at the cost of duplicate runs of selected pieces of the code. Checkpointing is best described as a transformation applied with respect to a piece of the original code (a

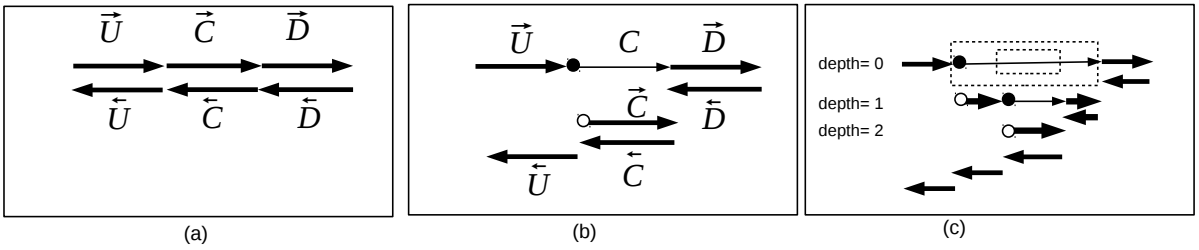

Figure 1. (a) A sequential adjoint program without checkpointing. The two thick arrows in the top represent the forward sweep, propagating the values in the same order as the original program, and the two thick arrows in the bottom represent the backward sweep, propagating the gradients in the reverse order of the computation of the original values. (b) The same adjoint program with checkpointing applied to the part of code C. The thin arrow reflects that the first execution of the checkpointed code C does not store the intermediate values in the stack. (c) Application of the checkpointing mechanism on two nested checkpointed parts. The checkpointed parts are represented by dashed rectangles.

"checkpointed part"). For instance figure 1 (a) and (b) illustrate checkpointing applied to the piece C of a code, consequently written as $U;C;D$. On the adjoint code of $U;C;D$ (see figure 1 (a)), checkpointing C means in the forward sweep **not** storing the intermediate values during the execution of C. As a consequence, the backward sweep can execute $\overleftarrow{D}$ but lacks the intermediate values necessary to execute $\overleftarrow{C}$. To cope with that, the code after checkpointing (see figure 1 (b)) runs the checkpointed piece again, this time storing the intermediate values. The backward sweep can then resume, with $\overleftarrow{C}$ then $\overleftarrow{U}$. In order to execute C twice (actually C and later $\overrightarrow{C}$), one must store (a sufficient part of) the memory state before C and restore it before $\overleftarrow{C}$. This storage is called a *snapshot*, which we represent on figures as a $\bullet$ for taking a snapshot and as a $\circ$ for restoring it. Taking a snapshot "$\bullet$" and restoring it "$\circ$" have the effect of resetting a part of the machine state after "$\circ$" to what it was immediately before "$\bullet$".

Checkpointing is vital for long run-time codes, which is the case for most MPI parallel applications. However, for MPI codes this question has always been addressed by ad-hoc hand manipulations of the differentiated code, and with no formal assurance of correctness. In a previous work[2], we proposed an

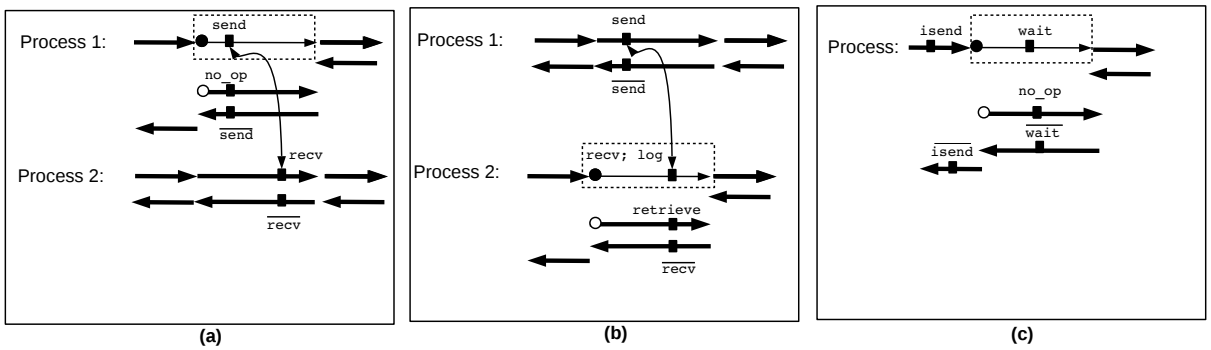

Figure 2. Three examples in which we apply checkpointing coupled with receive-logging. For clarity, we separated processes: process 1 on top and process 2 at the bottom. In (a), an adjoint program after checkpointing a piece of code containing only the *send* part of point-to-point communication. In (b), an adjoint program after checkpointing a piece of code containing only the *recv* part of point-to-point communication. In (c), an adjoint program after checkpointing a piece of code containing a `wait` without its corresponding non blocking routine `isend`.

adaptation of checkpointing to the case of MPI parallel programs with point-to-point communications, so that the semantics of an adjoint program is preserved for any choice of the checkpointed part. This adapted technique, called "receive-logging", is sketched in figure 2. For simplicity, we omit the `mpi_` prefix from subroutine names and omit parameters that are not essential in our context. The receive-logging technique relies on logging every message at the time when it is received.

- During the first execution of the checkpointed part, every communication call is executed normally. However, every *receive* call (in fact its *wait* in the case of non-blocking communication) stores the value it receives into some location local to the process. Calls to *send* are not modified.
- During the duplicated execution of the checkpointed part, every *send* operation does nothing. Every *receive* operation, instead of calling any communication primitive, reads the previously received value from where it has been stored during the first execution. We say that these operations are "de-activated".

Although this technique does not impose restrictions on the choice of the checkpointed part, message-logging makes it memory-costly. In the previous work,[2] we proposed a refinement to our general technique. It consists in duplicating the communications, "message-resending", whenever it is possible. The principle is to identify `send-recv` pairs whose ends belong to the same checkpointed part, and to re-execute these communication pairs identically during the duplicated part, thus, performing the actual communication twice. Meanwhile, communications with one end not belonging to the checkpointed part are still treated by receive-logging. Figure 3 (b) shows the application of checkpointing coupled with receive-logging

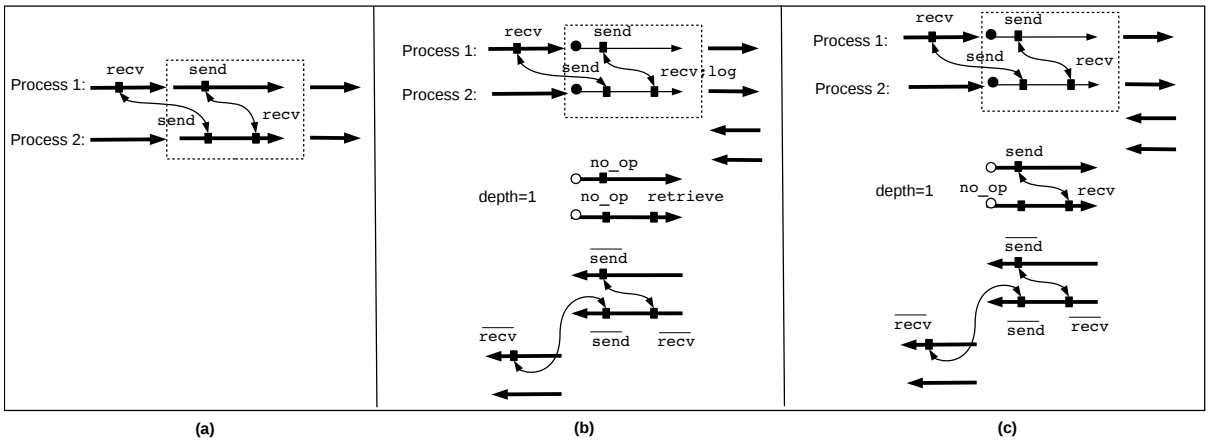

Figure 3. In (a), an MPI parallel program running in two processes. In (b), the adjoint corresponding to this program after checkpointing a piece of code by applying the receive-logging. In (c), the adjoint corresponding after checkpointing a piece of code by applying receive-logging coupled the with message-resending.

technique to some piece of code. In this piece of code, we select a `send-recv` pair and we apply the

message-resending to it. As result, see figure 3 (c), this pair is re-executed during the duplication of the checkpointed part and the received value is no more logged during the first instance of this checkpointed part. We say, here, that the *send* and *receive* operations are "activated".
We call an end of communication `orphan` with respect to a checkpointed part, if it belongs to this checkpointed part while its partner is not, e.g. `send` that belongs to the checkpointed part while its `recv` is not. In the case where one end of communication is paired with more than one end, e.g. `recv` with wild-card MPI_ANY_SOURCE value for source, this end is considered as `orphan` if one of its partners does not belong to the same checkpointed part as it.

However, to apply message-resending, the checkpointed part must obey an extra constraint which we will call "right-tight". A checkpointed part is "right-tight" if no communication dependency goes from downstream the checkpointed part back to the checkpointed part. For instance, there must be no `wait` in the checkpointed part that corresponds with a communication call in an other process which is downstream (i.e. after) the checkpointed part e.g. the checkpointed part in Figure 3 is right-tight.

In the general case, we may have a nested structure of checkpointed parts, in which some of the checkpointed parts are right-tight, and the others are not. Also, even when all the checkpointed parts are right-tight, an end of communication may be `orphan` with respect to some checkpointed parts and `non-orphan` with respect to other ones. This means that, for memory reasons, an end of communication may be activated during some depths of the checkpointed adjoint, i.e. we apply the message-resending to this end, and not activated during the other depths, i.e. we apply receive-logging to this end. In the case of *send* operations, combining the receive-logging and message-resending techniques is easy to implement, however, in the case of *receive* operations, this requires a specific behavior. More precisely:

- Every *receive* operation that is activated at depth d calls `recv`. If this operation is de-activated at depth $d+1$, it has to log the received value.
- Every *receive* operation that is de-activated at depth d reads the previously received value from where it has been stored. If this *receive* is activated at depth $d+1$, it has to free the logged value.

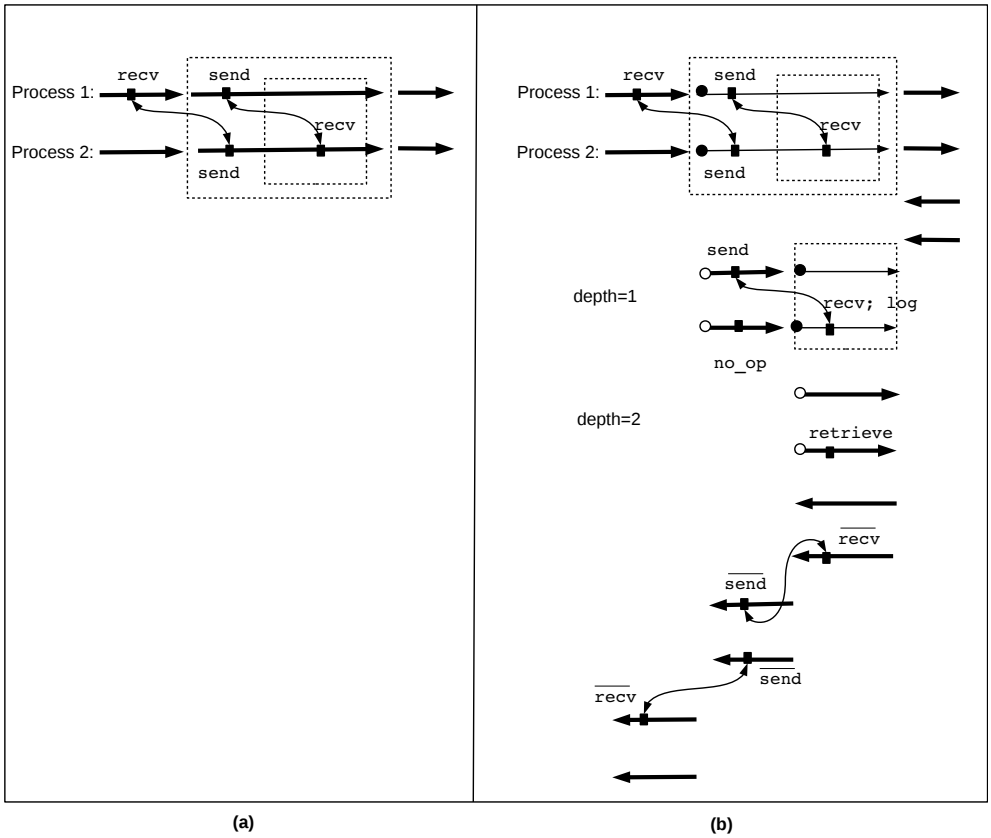

Figure 4. In (a), an MPI parallel program run on two processes. In (b), the adjoint corresponding after checkpointing two nested checkpointed parts, both of them right-tight. The receive-logging is applied to the orphan ends of communications and the message-resending is applied to the non-orphan ones

Figure 4 (a) shows an example, in which we selected two nested checkpointed parts. In figure 4 (a), we see that the `recv` of process 2 is `non-orphan` with respect to the outer checkpointed part and `orphan` with respect to the inner one, i.e. its corresponding `send` belongs only to the outer checkpointed part.

Since the outer checkpointed part is right-tight, we chose to apply message re-sending to the `recv` of process 2 together with its `send`. As result of checkpointing, see figure 4 (b), the *receive* call of process 2 is activated when the depth of checkpointing is equal to 1. Since this *receive* will be de-activated during the depth just after, i.e. during depth=2, its received value has been logged during the current depth and retrieved during the depth just after.

In this paper, we see checkpointing MPI parallel programs from a practical point of view. In section II, we propose an implementation of receive-logging coupled with message-resending inside the AMPI library.[3] In section III, we propose a further refinement to the receive-logging technique. In sections IV and V, we discuss practical questions about the choice of technique to be applied within a checkpointed part and the choice of the checkpointed part itself. Finally, in section VI, we validate our theoretical results on representative CFD codes.

II. Implementation Proposal

We propose an implementation of receive-logging coupled with message re-sending inside the AMPI library. This proposal allows for each end of communication to be activated during some depths of the checkpointed adjoint, i.e. we apply the message-resending to it, and de-activated during some others, i.e. we apply the receive-logging to it.

II.A. General view

The AMPI library is a library that wraps the calls to MPI subroutines in order to make the automatic generation of the adjoint possible in the case of MPI parallel programs. An interface for this library has already been developed in the operator overloading AD tool dco,[4,5] and under development in our AD tool Tapenade.[6] This library provides two types of wrappers:

- The "forward wrappers", called during the forward sweep of the adjoint code. Besides calling the MPI subroutines of the original MPI program, these wrappers store in memory the needed information to determine for every MPI subroutine, its corresponding adjoint, we call this "adjoint needed information". For instance, the forward wrapper that corresponds to a `wait`, FWD_AMPI_wait calls `wait` and stores in memory the type of non blocking routine with whom the `wait` is paired.
- The "backward wrappers" called during the backward sweep of the adjoint code. These wrappers retrieve the information stored in the forward wrappers and use it to determine the adjoint. For instance, the backward wrapper that corresponds to a `wait`, BWD_AMPI_wait calls `irecv` when the original `wait` is paired with an `isend`.

A possible implementation of receive-logging coupled with message-resending inside the AMPI library will either add new wrappers to this library, or change the existing forward wrappers. Let us assume that the future implementation will rather change the existing forward wrappers. In this case, these wrappers will be called more than once during the checkpointed adjoint, i.e. these wrappers will be called every time the checkpointed part is duplicated. An important question to be asked, thus, when the adjoint needed information has to be saved? Is it better to save this information during the first execution of the checkpointed part or is it better to save this information each time the message-resending is applied, or is it better to save this information the last time the message-resending is applied?
Since this information is used only to determine the adjoint, we think that the third option is the best in terms of memory consumption. We notice, however, that if no message-resending is applied to the forward wrapper, then, we have to save this information during the first execution of the checkpointed part. Also, if the stack is the mechanism we use to save and retrieve the adjoint needed information, then, this information has to be retrieved and re-saved each time we do not apply the message-resending.

II.B. Interface proposal

It is quite difficult to detect statically if a checkpointed part is right-tight or if an mpi routine is orphan or not with respect to a given checkpointed part. This could be checked dynamically but it would require performing additional communications to the ones that already exist, i.e. each `send` has to tell its corresponding `recv` in which checkpointed part it belongs and vice versa. We believe that a possible implementation of receive-logging coupled with message-resending will require the help of the user to specify when applying the message-resending, for instance through an additional parameter to the AMPI_send and AMPI_recv subroutines. We call this parameter "resending". To deal with the case of nested structure of checkpointed parts, the resending parameter may for instance, specify for each depth

of the nested structure, whether or not message-resending will be applied e.g. an array of booleans, in which the value 1 at index i reflects that message-resending will be applied at depth=i and the value 0 at index j reflects that message-resending will not be applied at depth=j, i.e. we will apply rather receive-logging.

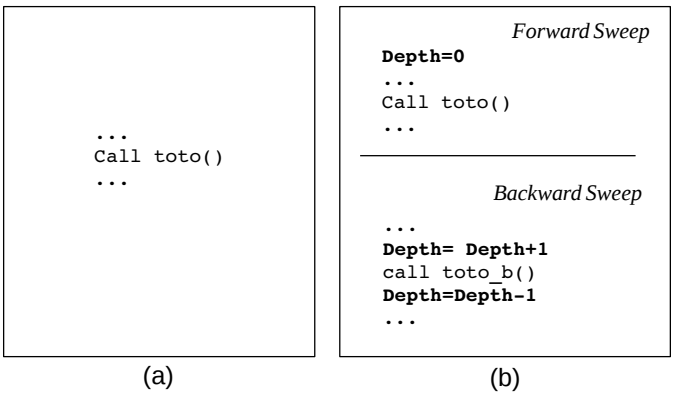

Figure 5. (a) a program that contains a call to a subroutine "toto". (b) the adjoint program after checkpointing the call to "toto". In the adjoint code we placed instructions that allow a dynamic detection of the depth

From the other side, we may detect dynamically the depth of each end of communication belonging to a nested structure of checkpointed parts. The main idea is to:

- create a new global variable, that we call "Depth", and initiate it to zero at the beginning of the adjoint program.
- increment the variable Depth, before each forward sweep of a checkpointed part.
- decrement the variable Depth, after each backward sweep of a checkpointed part.

At run time, the depth of an end of communication is the value of Depth. The instructions that allow initiating, incrementing and decrementing Depth may be easily placed by an AD tool inside the adjoint program. For instance, our AD tool Tapenade checkpoints every call to a subroutine. This means that if we have a call to a subroutine "toto" in the original code, we will have a call to "toto" in the forward sweep of the adjoint code and a call to "toto_b" in the the backward sweep of this code, in which "toto_b" contains the forward sweep and the backward sweep of the subroutine "toto", see figure 5. To detect the depth of each end of communication that belongs to "toto" at run time, it suffices to increment Depth before the call to "toto_b" and decrement Depth after the call to "toto_b", see figure 5.

Let us assume that Depth will be set as an AMPI global variable. i.e. AMPI_Depth. Figure 6 shows the various modifications we suggest for the wrappers AMPI_FWD_send and AMPI_FWD_recv. We see in figure 6 that we added resending as an additional parameter to our AMPI wrappers. For each end of communication, we check if the message-resending is applied at the current depth through a call to a function called "isApplied". This function takes the value of AMPI_depth and resending as inputs and returns true if the message-resending is applied at the value of AMPI_Depth and false in the opposite case. We check also if the message-resending will ever be applied in the following depths, through a call to a function called "willEverBeApplied". This function takes the values of AMPI_Depth and resending as inputs and returns true if the message-resending will ever be applied in the following depths and false in the opposite case. The algorithm sketched in figure 6 may be summarized as:

- When message-resending is applied at the depth d, we call the `recv` and `send` subroutines. If message-resending is not applied at depth $d+1$, then we log in addition the received value. If message-resending will never be applied after depth, then we have to save the adjoint needed information in both send and receive operations.
- When message-resending is not applied at depth, we retrieve the logged value in the receive side. If message-resending is applied at depth+1, than, it is better in terms of memory to free the logged value. As we already mentioned, if the stack is the mechanism we use to save and retrieve the adjoint needed information, then this information has to be retrieved and re-saved in both send and receive operations.

We note that in our implementation proposal, if the user decides to apply the message-resending to one static mpi call, then this decision will be applied to all the run-time mpi calls that match this static call.

```
AMPI_FWD_recv(V,resending)
{
If (AMPI_Depth==0)||(isApplied(resending,AMPI_Depth)== true) then
   call MPI_recv(V)
   If (isApplied(resending, AMPI_Depth+1)==false) then
     log(V)
   endif
   If (willEverBeApplied(resending, AMPI_Depth)==false) then
     store the needed information for the adjoint
   Endif
Else
   retrieve(V)
   If (isApplied(resending, AMPI_Depth+1)==true) then
     free(V)
   endif
   restore the needed information for the adjoint
   store the needed information for the adjoint
}
```

```
AMPI_FWD_send(V,resending)
{
If (Depth==0)||(isApplied(resending,AMPI_Depth)== true) then
  call MPI_send(V)
  If (willEverBeApplied(resending, AMPI_Depth)==false) then
     store the needed information for the adjoint
  Endif
Else
   restore the needed information for the adjoint
   store the needed information for the adjoint
}
```

Figure 6. the modifications we suggest for some AMPI wrappers

III. Further refinement: logging only the overwritten receives

We propose a further refinement to our receive-logging technique. This refinement consists in not logging every received value that is not used inside the checkpointed part, or, it is used but it is never modified since it has been received until the next use by the duplicated instance of the checkpointed part, e.g. see figure 7. Formally, given `Recv` the set of variables that hold the received values inside the checkpointed part, `Use` the set of variables that are read inside the checkpointed part and `Out` the set of variables that are modified inside the checkpointed part (only the variables that are modified by more than one *receive* operation are included in the `Out` set of variables) and in the sequel of the checkpointed part, we will log in memory the values of variables `OverwrittenRecvs` with:

$$\text{OverwrittenRecvs} = \text{Recv} \cap \text{Use} \cap \text{Out}$$

The values of `OverwrittenRecv` are called "`overwritten recvs`". Clearly, this is a small refinement as in the real codes, the number of `overwritten recvs` is much more important than the number of `non-overwritten` ones.

IV. Choice of techniques to be applied

We saw previously various techniques to reduce the memory cost of the receive-logging technique. Some of them duplicate the call to mpi communications, which may add extra cost in terms of time execution and some of them propose not logging all the received values, but only those that are used and will be probably overwritten by the rest of the program. One important question to be asked, then, is for a given checkpointed piece, what is the best combination of techniques to be applied, i.e. what is the combination that allows a reduction of the peak memory consumption without consuming too much in terms of time execution?
In the case where the checkpointed part is not right-tight, we can only apply receive-logging to all the

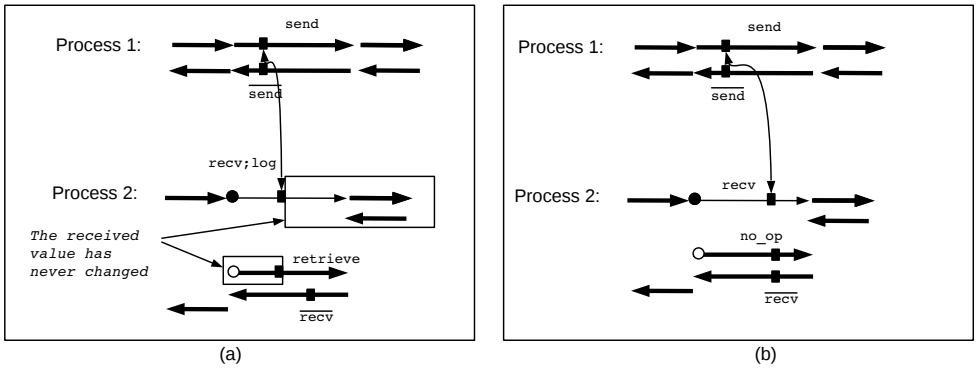

Figure 7. (a) An adjoint code after checkpointing a piece of code containing only the *receive* part of point-to-point communication. Checkpointing is applied together with the receive-logging technique, i.e. the *receive* call logs its received value during the first execution of the checkpointed part and retrieves it during the re-execution of the checkpointed part. In this example, the received value is never modified since it has been received until the next use by the duplicated instance of the checkpointed part, i.e. in the part of code surrounded by rectangles. (b) The same adjoint after refinement. In this code the received value is not saved anymore.

ends of communications inside this checkpointed part.
In the opposite case, i.e. the checkpointed part is right-tight:

- for all `orphan` ends of communications, we can only apply receive-logging.
- for the `non-orphan` ends of communications, we have the choice between applying the receive-logging and the message-resending techniques. When the `non-orphan` ends are `overwritten recvs`, then, it is more efficient in terms of memory to apply message-resending to these `overwritten recvs` together with their `sends`. Actually, applying receive-logging to these `recvs` will require extra storage. From the other hand, when the `non-orphan` ends are basically `non-overwritten recvs`, then, applying receive-logging to these `recvs` and theirs `sends` has the same cost in terms of memory as applying message-resending to these pairs `sends-recvs`. Thus, in this case we prefer applying receive-logging to these `recvs` and their `sends` as it requires less number of communications than in the case where message-resending is applied.

V. Choice of checkpointed part

So far, we have discussed the strategies to be applied to communication calls, given the placement of checkpointed portions. We note, however, that this placement is also some thing that can be chosen differently by the user, with the objective of improving the efficiency of the adjoint code. This issue is discussed by this section.

In real codes, the user may want to checkpoint some processes P independently from the others, either because checkpointing the other processes is not worth the effort, i.e. checkpointing the other processes does not reduce significantly the peak memory consumption, or checkpointing them will instead increase the peak memory consumption. In this case, is it more efficient in terms of memory to :

1. checkpoint only P, in which case we will have many `orphan` ends of communications which means applying the receive-logging to the majority of mpi calls inside the checkpointed part,
2. or, checkpoint the set of processes P together with the other processes with whom P communicate, in which case we will apply the message-resending to all the mpi calls inside the checkpointed part ?

As the receive-logging technique is in general memory costly, one may prefer the option 2. However, in real codes, the option 2 may sometimes not be the best choice. Actually, choosing the best checkpointed part depends on many factors such as: the cost of `overwritten recvs`, the cost of snapshot of other processes, etc.. It depends also on the needs of the end-user, i.e. efficiency in terms of time or memory? The diagram of figure 8 summarizes the various decisions to be made regarding an example case with 2 processes and 2 alternative checkpointing choices “A” and “B”.

- In “A”, only process 1 is checkpointed. In this case, we can only apply receive-logging.
- In “B”, the two processes 0 and 1 are checkpointed. In this case, we will apply the message-resending.

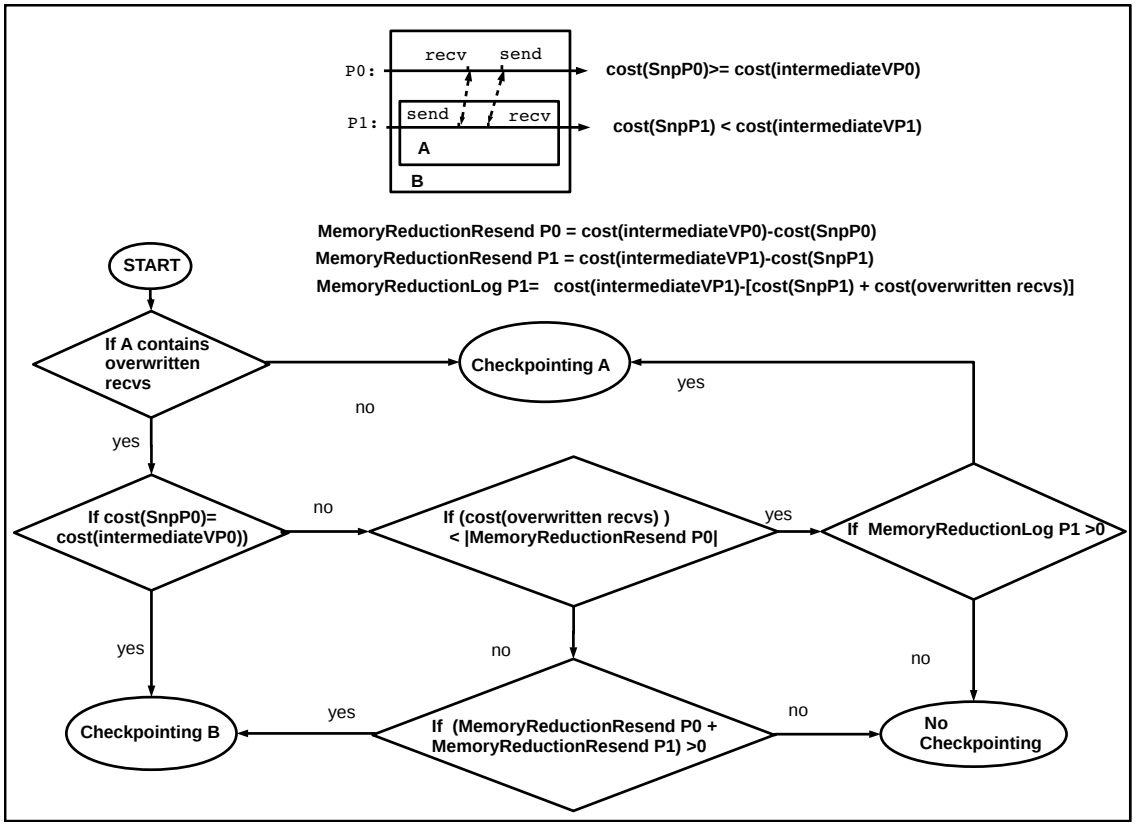

Figure 8. Top: a program run on two processes. In this program, we selected two different checkpointed parts. Down: a diagram that summarizes the best checkpointed part to be chosen in each case.

In diagram 8, we assume that the efficiency in terms of memory is our priority. We see, thus, that in some cases, e.g. when the memory cost of snapshot of process 0, cost(SnpP0), is almost equal to the memory cost of logging the intermediate values of the same process, cost(intermediateVP0), checkpointing "B" is the most efficient in terms of memory. Since checkpointing "A" is always the most efficient in terms of number of communications, i.e. the receive-logging does not duplicate the communications, the choice of the best checkpointed part depends on the needs of the end-user.
From the other side, the diagram shows that in some other cases, e.g. when we have no `overwritten recvs`, checkpointing "A" is the most efficient not only in terms of number of communications, but also in terms of memory consumption.

VI. Experiments

To validate our theoretical works, we selected two representative CFD codes in which we performed various choices of checkpointed parts. Both codes resolve the wave equation by using an iterative loop that at each iterations resolves:

$$U(x,t+dt) = 2U(x,t) - U(x,t-dt) + [c*dt/dx]^2 * [U(x-dx,t) - 2U(x,t) + U(x+dx,t)]$$

In which U models the displacement of the wave and c is a fixed constant. To apply checkpointing, we used the checkpointing directives of Tapenade, i.e. we placed \$AD CHECKPOINT-START and \$AD CHECKPOINT-END around each checkpointed part. By default, the checkpointed code applies the message-resending technique, i.e. by default the resulting adjoint duplicates the calls to MPI communications. To apply the receive-logging, we de-activated by hand the duplication of MPI calls. In addition, for each `recv` call, we added the needed primitives that handle the storage of the received value during the first call of this `recv` and the recovery of this value when it is a duplicated instance of the `recv`.

VI.A. First experiment

The first test is run on 4 processes. Figure 9 shows the various communications performed by these processes at each iteration of the global loop. We see in this figure, that at the end of each iteration, the process 0 collects the computed values from the other processes. In this code, we selected two

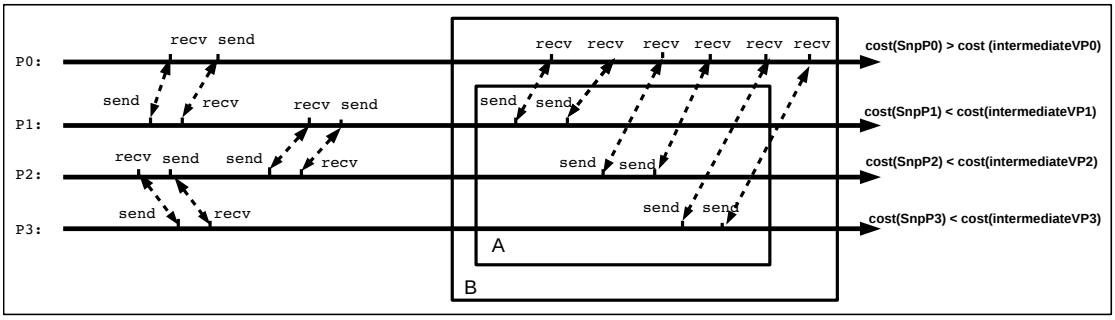

Figure 9. Representative code in which we selected two checkpointed parts

alternative checkpointed parts: "A", in which we checkpoint the processes 1,2 and 3 and "B", in which we checkpoint all the processes. We see in figure 9, that checkpointing the process 0 increases the peak memory consumption of this process, i.e. the memory cost of snapshot of process 0, cost(SnpP0), is greater than the memory cost of logging its intermediate values, cost(intermediateVP0). We applied the receive logging to all MPI calls of the part of code "A" and the message-resending to all the MPI calls of the part "B".

The results of checkpointing "A" and "B" are shown in table 1. We see that the code resulting from checkpointing "A" is more efficient than the code resulting from checkpointing "B" not only in terms of number of communications, i.e. it performs less than 24000 communications, but also in terms of memory consumption, i.e. it consumes less than 1.3 MB. The efficiency in terms of number of communications was expected since the receive-logging de-activates the duplication of MPI communications and thus does not add extra communications to the adjoint code as it is the case of the message-resending. From the other side, the efficiency in terms of memory consumption can be explained by the fact that the checkpointed part "A" does not contain any `overwritten recvs`, i.e. it contains only `sends`, and thus does not require any extra storage. These results match the analysis of subsection V.

	without CKP	CKP "B"	CKP "A"
Memory cost of P0 (MB)	8	9.3	8
Memory cost of P1,2,3 (MB)	12.6	9.4	9.4
Total Memory cost (MB)	45.8	37.5	36.2
Number of communications	48000	72000	48000

Table 1. Results of the first experiment

VI.B. Second experiment

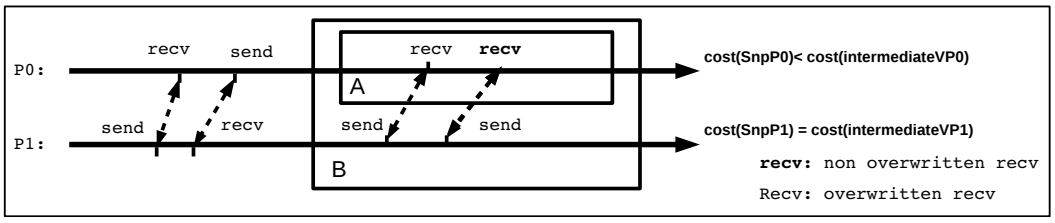

Figure 10. Representative code in which we selected two checkpointed parts

The second test is run on two processes. The communications performed by these two processes are shown in figure 10. In this test we study two alternative checkpointed parts as well. The first part "A" is run on only one process, i.e. process 0 and the second part "B" is run on the two processes. Here, checkpointing the process 1 does neither increase, nor decrease the peak memory consumption, i.e. the cost of snapshot of process 0 is almost equal to the cost of logging the intermediate values of process 0.

The results of checkpointing "A" and "B" are shown in the table 2. Unlike the first experiment, checkpointing "B" here is more efficient in terms of memory. In fact, the resulting adjoint consumes less than 41 KB than the adjoint resulting from checkpointing "A". This can be explained by two facts: the first one is that "A" contains `overwritten recvs` and the second one is that checkpointing process P1

does not decrease the memory consumption. These results also match the analysis of subsection V. We notice here, that checkpointing "A" is always more efficient in terms of number of communications than checkpointing "B". Clearly, the choice of the best checkpointed part depends here on the needs of the user.

	without CKP	CKP "B"	CKP "A"
Memory cost of P0 (MB)	15.58	12.36	12.39
Memory cost of P1 (MB)	12.45	12.42	12.43
Total Memory cost (MB)	28.03	24.78	24.82
Number of communications	16000	24000	16000

Table 2. Results of the second experiment

VII. Discussion And Further Work

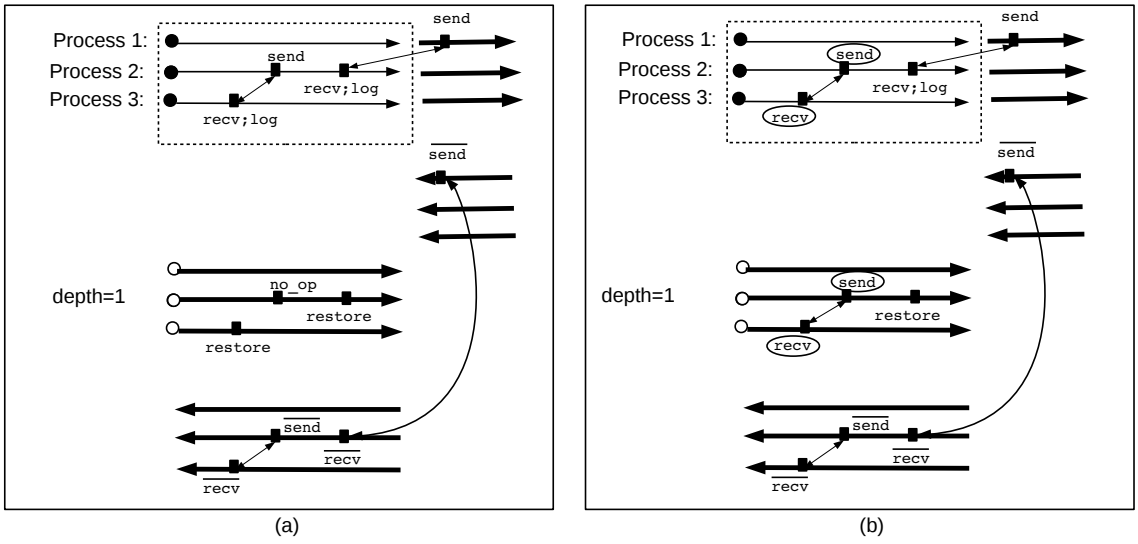

Figure 11. (a) The receive-logging applied to a parallel adjoint program. (b) Application of the message re-sending to a `send-recv` pair with respect to a non-right-tight checkpointed code

We considered the question of checkpointing in the case of MPI-parallel codes. Checkpointing is a memory/run-time trade-off which is essential for adjoint of large codes, in particular parallel codes. However, for MPI codes this question has always been addressed by ad-hoc hand manipulations of the differentiated code. In a previous work, we introduced, a general checkpointing technique that can be applied for any choice of the checkpointed part. This technique is based on logging the received messages, so that the duplicated communications need not take place. On the other hand, We proposed a refinement that reduces the memory consumption of this general technique by duplicating the communications whenever possible. In this work, we proposed an implementation of these techniques inside the AMPI library. We discussed practical questions about the choice of strategy to be applied within a checkpointed part and the choice of the checkpointed part itself. At the end, we validated our theoretical results on representative CFD codes.

There are a number of questions that should be studied further:

We imposed a number of restrictions on the checkpointed part in order to apply the message-resending. These are sufficient conditions, but it seems they are not completely necessary. Figure 11 shows a checkpointed code which is not right-tight. Still, the application of the message re-sending to a `send-recv` pair (whose ends are surrounded by circles) in this checkpointed part, does not introduce deadlocks in the resulting checkpointed code.

The implementation proposal we suggest in section II allows an application of receive-logging coupled with message-resending that may be considered as "semi-automatic". Actually, this proposal requires the help of user to specify for each end of communication, the set of depths in which it will be activated, i.e. in which depths message-resending will be applied to this end. An interesting further research is, thus, how to automatically detect this information, for instance by detecting if a checkpointed part is

right-tight and also if an end of communication is orphan or not with respect to a given checkpointed part.

Acknowledgments

This research is supported by the project "About Flow", funded by the European Commission under FP7-PEOPLE-2012-ITN-317006. See "http://aboutflow.sems.qmul.ac.uk".

References

[1]Dauvergne, B. and Hascoët, L. The Data-Flow Equations of Checkpointing in reverse Automatic Differentiation. *International Conference on Computational Science, ICCS 2006, Reading, UK*, 2006.

[2]Taftaf, A and Hascoët, L. On The Correct Application Of AD Checkpointing To Adjoint MPI-Parallel programs *European Congress on Computational Methods in Applied Sciences and Engineering*, 2016

[3]J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, U. Naumann, Toward Adjoinable MPI. *Proceedings of the 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering, PDSEC'09*, 2009.

[4]M. Schanen, U. Naumann L. Hascoët, J. Utke, Interpretative Adjoints for Numerical Simulation Codes using MPI. *International Conference on Computational Science*, ICCS 2010.

[5]M. Towara, M. Schanen, U. Naumann, MPI-Parallel Discrete Adjoint OpenFOAM. *International Conference on Computational Science*, ICCS 2015.

[6]Hascoët, L. and Pascual, V. The Tapenade Automatic Differentiation tool: Principles, Model, and Specification. *ACM Transactions On Mathematical Software*, 39,3, "http://dx.doi.org/10.1145/2450153.2450158", 2013.