ECCOMAS Congress 2016
VII European Congress on Computational Methods in Applied Sciences and Engineering
M. Papadrakakis, V. Papadopoulos, G. Stefanou, V. Plevris (eds.)
Crete Island, Greece, 5–10 June 2016

ON THE CORRECT APPLICATION OF AD CHECKPOINTING TO ADJOINT MPI-PARALLEL PROGRAMS

A.Taftaf, L. Hascoët

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

Keywords: Algorithmic Differentiation, Adjoint, Checkpointing, Message Passing, MPI

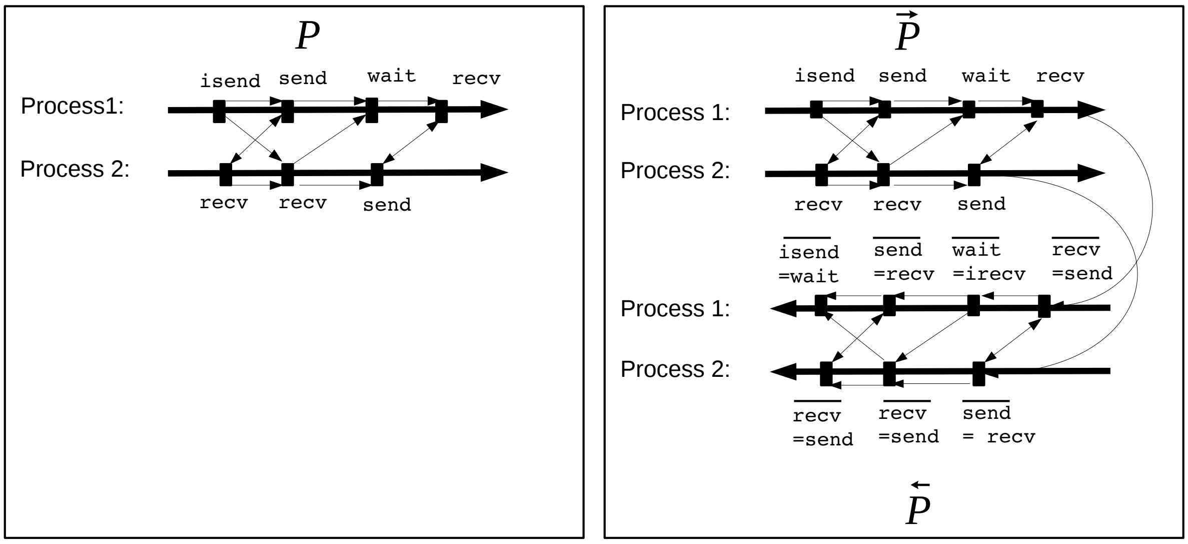
Abstract. *Checkpointing is a classical technique to mitigate the overhead of adjoint Algorithmic Differentiation (AD). In the context of source transformation AD with the Store-All approach, checkpointing reduces the peak memory consumption of the adjoint, at the cost of duplicate runs of selected pieces of the code. Checkpointing is vital for long run-time codes, which is the case of most MPI parallel applications. However, the presence of MPI communications seriously restricts application of checkpointing.*
In most attempts to apply checkpointing to adjoint MPI codes (the "popular" approach), a number of restrictions apply on the form of communications that occur in the checkpointed piece of code. In many works, these restrictions are not explicit, and an application that does not respect these restrictions may produce erroneous code.
We propose techniques to apply checkpointing to adjoint MPI codes, that either do not suppose these restrictions, or explicit them so that the end users can verify their applicability. These techniques rely on both adapting the snapshot mechanism of checkpointing and on modifying the behavior of communication calls.
One technique is based on logging the values received, so that the duplicated communications need not take place. Although this technique completely lifts restrictions on checkpointing MPI codes, message logging makes it more costly than the popular approach. However, we can refine this technique to blend message logging and communications duplication whenever it is possible, so that the refined technique now encompasses the popular approach. We provide elements of proof of correction of our refined technique, i.e. that it preserves the semantics of the adjoint code and that it doesn't introduce deadlocks.

1 INTRODUCTION

Adjoint algorithms, and in particular those obtained through the adjoint mode of Automatic Differentiation (AD) [1], are probably the most efficient way to obtain the gradient of a numerical simulation. Given a piece of code P, adjoint AD (with the "Store-All" approach) consists of two successive pieces of code. The first one, the "forward sweep" $\overrightarrow{P}$ computes the original values and stores in memory the overwritten variables needed to compute the gradients. The second one, the "backward sweep" $\overleftarrow{P}$, computes the gradients, using the intermediate values stored as needed. Primarily, i.e. before any form of program optimisation, the adjoint program is simply $\overrightarrow{P}$ followed by a $\overleftarrow{P}$.

Many large-scale computational science applications are parallel programs based on Message-Passing, implemented for instance by using the MPI message passing library. We will call them "MPI programs". MPI programs consist of one or more threads (called "MPI processes") that communicate through message exchanges.

In most attempts to apply checkpointing to adjoint MPI codes (the "popular" approach), a number of restrictions apply on the form of communications that occur in the checkpointed piece of code. In many works, these restrictions are not explicit, and an application that does not respect these restrictions may produce erroneous code.
We propose techniques to apply checkpointing to adjoint MPI codes, that either do not suppose these restrictions, or explicit them so that the end users can verify their applicability. These techniques rely on both adapting the snapshot mechanism of checkpointing and on modifying the behavior of communication calls.


P
isend
send
wait
recv
Process1:
Process 2:
recv
recv
send
$\overrightarrow{P}$
isend
send
wait
recv
Process 1:
Process 2:
recv
recv
send
$\overline{\text{isend}}$ =wait
$\overline{\text{send}}$ =recv
$\overline{\text{wait}}$ =irecv
$\overline{\text{recv}}$ =send
Process 1:
Process 2:
$\overline{\text{recv}}$ =send
$\overline{\text{recv}}$ =send
$\overline{\text{send}}$ = recv
$\overleftarrow{P}$

(a) (b)

Figure 1: (a) Communications graph of an MPI parallel program with two processes. Thin arrows represent the edges of the communications graph and thick arrows represent the propagation of the original values by the processes. (b) Communications graph of the corresponding adjoint MPI parallel program. 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.

1.1 Communications graph of adjoint MPI programs

One commonly used model to study message-passing is the communications graph [[2], pp. 399403], which is a directed graph (see figure 1 (a)) in which the nodes are the MPI communication calls and the arrows are the dependencies between these calls. For simplicity, we omit the `mpi_` prefix from subroutine names and omit parameters that are not essential in our context. Calls may be dependent because they have to be executed in sequence by a same process, or because they are matching `send` and `recv` calls in different processes.

- The arrow from each `send` to the matching `recv` (or to the `wait` of the matching `isend`) reflects that the `recv` (or the `wait`) cannot complete until the `send` is done. Similarly, the arrow from each `recv` to the matching `send` (or to the `wait` of the matching `irecv`) reflects that the `send` will block until the `recv` is done.
- The arrows between two successive MPI calls within the same process reflect the dependency due to the program execution order, i.e. instructions are executed sequentially. In the sequel, we will not show these arrows.

A central issue for correct MPI programs is to be deadlock free. Deadlocks are cycles in the communications graph.

There have been several works on the adjoint of MPI parallel programs [3],[4], [5], [6], [7]. When the original code performs an MPI communication call, the adjoint code must perform another MPI call, which we will call an "adjoint MPI call".

- For instance the adjoint for a receiving call `recv`(b) is a *send* of the corresponding adjoint value $\bar{b}$. In practice, this will write as `send`$(\bar{b})$; $\bar{b} = 0$.
- Symmetrically the adjoint for a sending call `send`(a) performs a *receive* of the corresponding adjoint value. In practice this will write as `recv`(tmp) ; $\bar{a} + = temp$.

This way, the adjoint code will perform a communication of the adjoint value (called "adjoint communication") in the opposite direction of the communication of the primal value, which is what should be done according to the AD model. This creates in $\overleftarrow{P}$ a new graph of communications (see figure 1 (b)), that has the same shape as the communications graph of the original program, except the inversion of the direction of arrows. This implies that if the communications graph of the original program is acyclic, then the communications graph of $\overleftarrow{P}$ is also acyclic. Since $\overrightarrow{P}$ is essentially a copy of P with the same communications structure, the communications graphs of $\overrightarrow{P}$ and $\overleftarrow{P}$ are acyclic if the communications graph of P is acyclic. Since we observe in addition that there is no communication from $\overrightarrow{P}$ to $\overleftarrow{P}$, we conclude that if P is deadlock free, then $\overline{P} = \overrightarrow{P}; \overleftarrow{P}$ is also deadlock free.

1.2 Checkpointing

Storing all intermediate values in $\overrightarrow{P}$ consumes a lot of memory space. In the case of serial programs, the most popular solution is the "checkpointing" mechanism [8] (see figure 2). Checkpointing is best described as a transformation applied with respect to a piece of the original code (a "checkpointed part"). For instance figure 2 (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 2 (a)), checkpointing C means in the forward sweep **not** storing the intermediate values during the execution of C. As a consequence, the backward

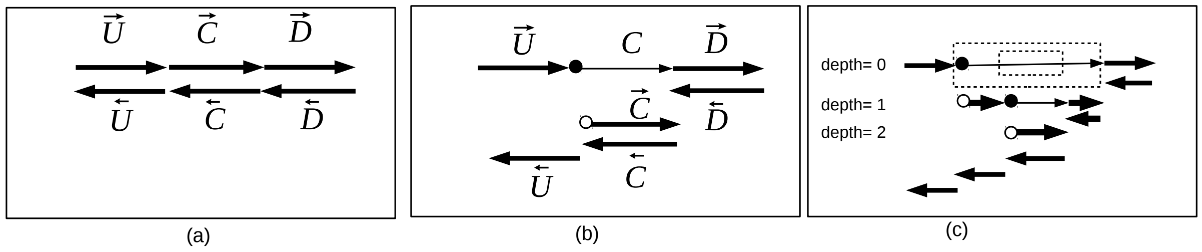

$\overrightarrow{U}$ $\overrightarrow{C}$ $\overrightarrow{D}$
$\overleftarrow{U}$ $\overleftarrow{C}$ $\overleftarrow{D}$
(a)
$\overrightarrow{U}$ C $\overrightarrow{D}$
$\overrightarrow{C}$ $\overleftarrow{D}$
$\overleftarrow{U}$ $\overleftarrow{C}$
(b)
depth= 0
depth= 1
depth= 2
(c)

Figure 2: (a) A sequential adjoint program without checkpointing. (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.

sweep can execute $\overleftarrow{D}$ but lacks the stored values necessary to execute $\overleftarrow{C}$. To cope with that, the code after checkpointing (see figure 2 (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 • for taking a snapshot and as a ◦ for restoring it. Taking a snapshot "•" and restoring it "◦" have the effect of resetting a part of the machine state after "◦" to what it was immediately before "•". We will formalize and use this property in the demonstrations that follow. To summarize, for original code $U; C; D$, whose adjoint is $\overrightarrow{U}; \overrightarrow{C}; \overrightarrow{D}; \overleftarrow{D}; \overleftarrow{C}; \overleftarrow{U}$, checkpointing C transforms the adjoint into $\overrightarrow{U}; \bullet; C; \overrightarrow{D}; \overleftarrow{D}; \circ; \overrightarrow{C}; \overleftarrow{C}; \overleftarrow{U}$.
The benefit of checkpointing is to reduce the peak size of the stack in which intermediate values are stored: without checkpointing, this peak size is attained at the end of the forward sweep, where the stack contains $k_U \oplus k_C \oplus k_D$, where k_X is the values stored by code X. In contrast, the checkpointed code reaches two maximums $k_U \oplus k_D$ after $\overrightarrow{D}$ and $k_U \oplus k_C$ after $\overrightarrow{C}$. The cost of checkpointing is twofold: the snapshot must be stored, generally on the same stack, but its size is in general much smaller than k_C. The othor part of the cost is that C is executed twice, thus increasing run time.

1.3 Checkpointing on MPI adjoints

Checkpointing MPI parallel programs is restricted due to MPI communications. In previous works, the "popular" checkpointing approach has been applied in such a way that a checkpointed piece of code always contains both ends of each communication it performs. In other words, no MPI call inside the checkpointed part may communicate with an MPI call which is outside. Furthermore, non-blocking communication calls and their corresponding waits must be both inside or both outside of the checkpointed part. This restriction is often not explicitly mentioned. However, if only one end of a point to point communication is in the checkpointed part, then the above method will produce erroneous code. Consider the example of figure 3 (a), in which only the `send` is contained in the checkpointed part. The checkpointing mechanism duplicates the checkpointed part and thus duplicates the `send`. As the matching `recv` is not duplicated, the second `send` is blocked. The same problem arises if only the `recv` is contained in the checkpointed part (see figure 3 (b)). The duplicated `recv` is blocked. Figure 3 (c) shows the case of a non-blocking communication followed by its `wait`, and only the `wait` is contained in the checkpointed part. This code fails because the repeated `wait` does not correspond to any pending communication.

We propose techniques that adapt checkpointing to MPI programs with point-to-point com-

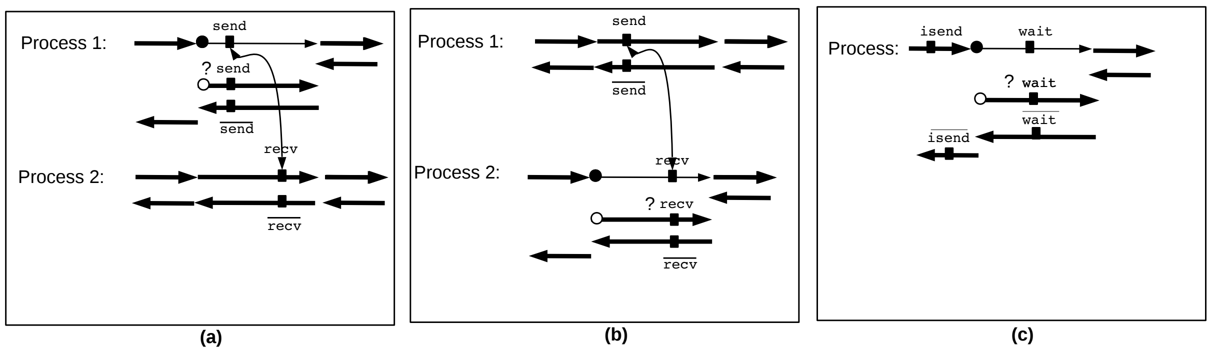


Figure 3: Three examples of careless application of checkpointing to MPI programs, leading to wrong code. For clarity, we separated processes: process 1 on top and process 2 at the bottom. In (a), an adjoint program after checkpointing a piece of code containing only the *send* part of point-to-point 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`.

munications. These techniques either do not suppose restrictions on the form of communications that occur in the checkpointed code, or explicit them so that the end user can verify their applicability. One technique is based on logging the values received, so that the duplicated communications need not take place. Although this technique completely lifts restrictions on checkpointing MPI codes, message logging makes it more costly than the popular approach. However, we can refine this technique to replace message logging with communications duplication whenever it is possible, so that the refined technique now encompasses the popular approach. In section 2, we give a proof framework for correction of checkpointed MPI codes, that will give some sufficient conditions on the MPI adapted checkpointing technique so that the checkpointed code is correct. In section 3 , we introduce our MPI adapted checkpointing technique based on message logging. We prove that this technique respects the assumptions of section 2 and thus that it preserves the semantics of the adjoint code. In section 3, we show how this technique may be refined in order to reduce the number of values stored in memory. We prove that the refinement we propose respects the assumptions of section 2 and thus that it preserves the semantics of the adjoint code as well.

2 ELEMENTS OF PROOF

We propose adaptations of the checkpointing method to MPI adjoint codes, so that it provably preserves the semantics of the resulting adjoint code for any choice of the checkpointed part. To this end, we will first give a proof framework of correction of checkpointed MPI codes, that relies on some sufficient conditions on the MPI adapted checkpointing method so that the checkpointed code is correct.

On large codes, checkpointed codes are nested (see figure 2 (c)) , with a nesting level often as deep as the depth of the call tree. Still, nested checkpointed parts are obtained by repeated application of the simple pattern described in figure 2 (b). Specifically, checkpointing applies to any sequence of forward, then backward code (e.g. $\overrightarrow{C}$; $\overleftarrow{C}$ on figure 2 (b)) independently of the surrounding code. Therefore, it suffices to prove correctness of one elementary application of checkpointing to obtain correctness for every pattern of nested checkpointed parts.

To compare the semantics of the adjoint codes without and with checkpointing, we define the effect $\mathcal{E}$ of a program P as a function that, given an initial machine state σ, produces a

new machine state $\sigma_{new} = \mathcal{E}(P, \sigma)$. The function $\mathcal{E}$ describes the semantics of P. It describes the dependency of the program execution upon all of its inputs and specifies all the program execution results. The function $\mathcal{E}$ is naturally defined on the composition of programs by :
$\mathcal{E}((P_1; P_2), \sigma) = \mathcal{E}(P_2, \mathcal{E}(P_1, \sigma))$.

When P is in fact a parallel program, it consists of several processes p_i run in parallel. Each p_i may execute point-to-point communication calls. We will define the effect $\mathcal{E}$ of one process p. To this end, we need to specify more precisely the contents of the execution state σ for a given process, to represent the messages being sent and received by p. We will call "R" the (partly ordered) collection of messages that will be received (i.e. are expected) during the execution of p. Therefore R is a part of the state σ which is input to the execution of p, and it will be consumed by p. It may well be the case that R is in fact not available at the beginning of p. In real execution, messages will accumulate as they are being sent by other processes. However, we consider R as a part of the input state σ as it represents the communications that are expected by p. Symmetrically, we will call "S" the collection of messages that will be sent during the execution of p. Therefore, S is a part of the state σ_{new} which is output by execution of p and it is produced by p.
We must adapt the definition of $\mathcal{E}$ for the composition of programs accordingly. We explicit the components of σ as follows. The state σ contains:

- W, the values of variables
- R, the collection of messages expected, or "to be received" by p
- S, the collection of messages emitted by p

With this shape of σ, the form of the semantic function $\mathcal{E}$ and the rule of the composition of programs become more complex. Definition of $\mathcal{E}$ on one process p imposes the prefix R_p of R (the messages to be received) that is required by p and that will be consumed by p. Therefore, the function $\mathcal{E}$ applies pattern matching on its R argument to isolate this "expected" part. Whatever remains in R is propagated to the output R. Similarly, S_P denotes the suffix set of messages emitted by p, to be added to S. Formally, we will write this as:
$\mathcal{E}(p, \langle W, R_P \oplus R, S \rangle) = \langle W', R, S \oplus S_P \rangle$
To explicit the rule of code sequence, suppose that p runs pieces of code C and D in sequence, with C expecting incoming received messages R_C and D expecting incoming received messages R_D. Assuming that the effect of C on the state is:
$\mathcal{E}(C, \langle W, R_C \oplus R, S \rangle) = \langle W', R, S \oplus S_C \rangle$
and the effect of D on the state is:
$\mathcal{E}(D, \langle W', R_D \oplus R, S \rangle) = \langle W'', R, S \oplus S_D \rangle$,
then $C; D$ expects received messages $R_C \oplus R_D$ (for the appropriate concatenation operator $\oplus$) and its effect on the state is:
$\mathcal{E}(C; D, \langle W, R_C \oplus R_D \oplus R, S \rangle) = \langle W'', R, S \oplus S_C \oplus S_D \rangle$.

Adjoint programs operate on two kinds of variables. On one hand, the variables of the original primal code are copied in the adjoint code. In the state σ, we will note their values "V". On the other hand, the adjoint code introduces new adjoint variables to hold the derivatives. In the state σ, we will denote their values "$\overline{V}$".
Moreover, adjoint computations with the store-all approach use a stack to hold the intermediate values that are computed and pushed during the forward sweep $\overrightarrow{P}$ and that are popped and used during the backward sweep $\overleftarrow{P}$. We will denote the stack as "k". In the sequel, we will use a

fundamental property of the stack mechanism of AD adjoints, which is that when a piece of code has the shape $\overrightarrow{P}; \overleftarrow{P}$, then the stack is the same before and after this piece of code. To be complete, the state should also describe the sent and received messages corresponding to adjoint values (see section 1.1). As these parts of the state play a very minor role in the proofs, we will omit them. Therefore, we will finally split states σ of a given process as: $\sigma = \langle V, \overline{V}, k, R, S\rangle$.
For our needs, we formalize some classical semantic properties of adjoint programs. These properties can be proved in general, but this is beyond the scope of this paper. We will consider these properties as axioms.

- Any "copied" piece of code X (for instance C) that occurs in the adjoint code operates only on the primal values V and on the R and S communication sets, but not on $\overline{V}$ nor on the stack. Formally, we will write:
  $\mathcal{E}(X, \langle V, \overline{V}, k, R_X \oplus R, S\rangle) = \langle V_{new}, \overline{V}, k, R, S \oplus S_X\rangle$, with the output V_{new} and S_X depending only on V and on R_X.

- Any "forward sweep" piece of code $\overrightarrow{X}$ (for instance $\overrightarrow{U}$, $\overrightarrow{C}$ or $\overrightarrow{D}$) works in the same manner as the original or copied piece X, except that it also pushes on the stack new values noted δk_X, which only depend on V and R_X. Formally, we will write:
  $\mathcal{E}(\overrightarrow{X}, \langle V, \overline{V}, k, R_X \oplus R, S\rangle) = \langle V_{new}, \overline{V}, k \oplus \delta k_X, R, S \oplus S_X\rangle$

- Any "backward sweep" piece of code $\overleftarrow{X}$ (for instance $\overleftarrow{U}$, $\overleftarrow{C}$ or $\overleftarrow{D}$), on one hand operates on the adjoint variables $\overline{V}$ and, on the other hand, uses exactly the top part of the stack δk_X that was pushed by $\overrightarrow{X}$. In the simplest AD model, δk_X is used to restore the values V that were held by the primal variables immediately before the corresponding forward sweep $\overrightarrow{X}$. There exists a popular improvement in the AD model in which this restoration is only partial, restoring only a subset of V to their values before $\overrightarrow{X}$. This improvement (called TBR) guarantees that the non-restored variables have no influence on the following adjoint computations and therefore need not be stored. The advantage of TBR is to reduce the size of the stack. Without loss of generality, we will assume in the sequel that the full restoration is used, i.e. no TBR is used. With the TBR mechanism, the semantics of the checkpointed program are preserved at least for the output $\overline{V}$ so that this proof is still valid. Formally, we will write:
  $\mathcal{E}(\overleftarrow{X}, \langle V, \overline{V}, k \oplus \delta k_X, R, S\rangle) = \langle V_{new}, \overline{V}_{new}, k, R, S\rangle$, where V_{new} is equal to the value V before running $\overrightarrow{X}$ (which is achieved by using δk_X and V) and $\overline{V}_{new}$ depends only on V, $\overline{V}$ and δk_X.

- A "take snapshot" operation "$\bullet$" for a checkpointed piece C does not modify V nor $\overline{V}$, expects no received messages, and produces no sent messages. It adds into the stack enough values Snp_C to permit a later re-execution of the checkpointed part. Formally, we will write :
  $\mathcal{E}(\bullet, \langle V, \overline{V}, k, R, S\rangle) = \langle V, \overline{V}, k \oplus Snp_C, R, S\rangle$, where Snp_C is a subset of the values in V, thus depending on only V.

- A "restore snapshot" operation "$\circ$" of a checkpointed piece C does not modify $\overline{V}$, expects no received messages and produces no sent messages. It pops from the stack the same set of values Snp_C that the "take snapshot" operation pushed "onto" the stack. This modifies V so that it holds the same values as before the "take snapshot" operation.
  We introduce here the additional assumption that restoring the snapshot may (at least conceptually) add some messages to the output value of R. In particular:

Assumption 1. *The duplicated $recvs$ in the checkpointed part will produce the same values as their original calls.*

Formally, we will write:
$\mathcal{E}(\circ, \langle V, \overline{V}, k \oplus Snp_C, R, S\rangle) = \langle V_{new}, \overline{V}, k, R_C \oplus R, S\rangle$ where V_{new} is the same as V from the state input to the take snapshot.

Our goal is to demonstrate that the checkpointing mechanism preserves the semantics i.e.:

Theorem 1. *For any individual process p, for any checkpointed part C of p, (so that $p = \{U;C:D\}$), for any state σ and for any checkpointing method that respects the* **Assumption 1***:*

$$\mathcal{E}(\{\overrightarrow{U};\overrightarrow{C};\overrightarrow{D};\overleftarrow{D};\overleftarrow{C};\overleftarrow{U}\},\sigma) = \mathcal{E}(\{\overrightarrow{U},\bullet,C,\overrightarrow{D},\overleftarrow{D},\circ,\overrightarrow{C},\overleftarrow{C},\overleftarrow{U}\},\sigma)$$

Proof. We observe that the non-checkpointed adjoint and the checkpointed adjoint share a common prefix $\overrightarrow{U}$ and also share a common suffix $\overleftarrow{C};\overleftarrow{U}$. Therefore, as far as semantics equivalence is concerned, it suffices to compare $\overrightarrow{C};\overrightarrow{D};\overleftarrow{D}$ with $\bullet, C, \overrightarrow{D}, \overleftarrow{D}, \circ, \overrightarrow{C}$.
Therefore, we want to show that for any initial state σ_0 :

$$\mathcal{E}(\{\overrightarrow{C};\overrightarrow{D};\overleftarrow{D}\},\sigma_0) = \mathcal{E}(\{\bullet,C,\overrightarrow{D},\overleftarrow{D},\circ,\overrightarrow{C}\},\sigma_0)$$

Since the semantic function $\mathcal{E}$ performs pattern matching on the R_0 part of its σ_0 argument, and the non-checkpointed code has the shape $\{\overrightarrow{C};\overrightarrow{D};\overleftarrow{D}\}$, R_0 matches the pattern $R_C \oplus R_D \oplus R$. Therefore, what we need to show writes as:

$$\mathcal{E}(\{\overrightarrow{C};\overrightarrow{D};\overleftarrow{D}\},\langle V_0,\overline{V}_0,k_0,R_C\oplus R_D\oplus R,S_0\rangle) =$$
$$\mathcal{E}(\{\bullet,C,\overrightarrow{D},\overleftarrow{D},\circ,\overrightarrow{C}\},\langle V_0,\overline{V}_0,k_0,R_C\oplus R_D\oplus R,S_0\rangle)$$

We will call σ_2, σ_3 and σ_6 the intermediate states produced by the non-checkpointed code (see

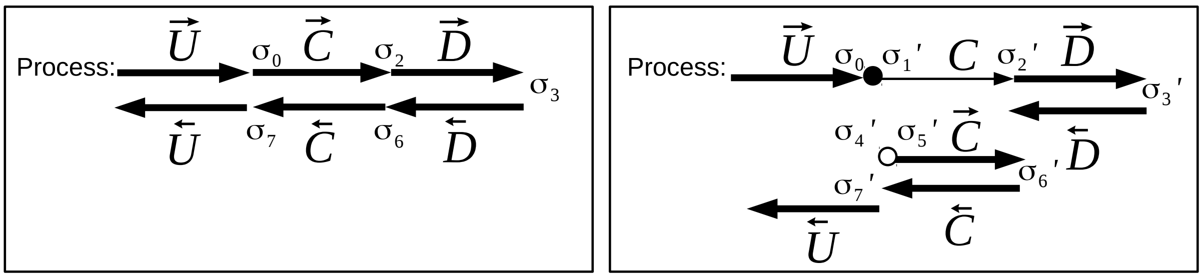


Figure 4: (a) An adjoint program run by one process. (b) The same adjoint after applying checkpointing to C. The figures show the locations (times) in the execution for the successive states σ_i and σ'_i.

figure 4 (a)). Similarly, we call σ'_1, σ'_2, σ'_3, σ'_4, σ'_5, σ'_6 the intermediate states of the checkpointed code (see figure 4 (b)). In other words: $\sigma_2 = \mathcal{E}(\overrightarrow{C},\sigma_0)$; $\sigma_3 = \mathcal{E}(\overrightarrow{D},\sigma_2)$; $\sigma_6 = \mathcal{E}(\overleftarrow{D},\sigma_3)$ and similarly $\sigma'_1 = \mathcal{E}(\bullet,\sigma_0)$; $\sigma'_2 = \mathcal{E}(C,\sigma'_1)$; $\sigma'_3 = \mathcal{E}(\overrightarrow{D},\sigma'_2)$; $\sigma'_4 = \mathcal{E}(\overleftarrow{D},\sigma'_3)$; $\sigma'_5 = \mathcal{E}(\circ,\sigma'_4)$;
$\sigma'_6 = \mathcal{E}(\overrightarrow{C},\sigma'_5)$.

Our goal is to show that $\sigma'_6 = \sigma_6$. Considering first the non-checkpointed code, we propagate the state σ by using the axioms already introduced:

$$\begin{aligned} \sigma_2 \doteqdot \mathcal{E}(\overrightarrow{C}, \sigma_0) &= \mathcal{E}(\overrightarrow{C}, \langle V_0, \overline{V_0}, k_0, R_C \oplus R_D \oplus R, S_0 \rangle) \\ &= \langle V_2, \overline{V_0}, k_0 \oplus \delta k_C, R_D \oplus R, S_0 \oplus S_C \rangle \end{aligned}$$

with V_2, S_C and δk_C depending only on V_0 and R_C

$$\begin{aligned} \sigma_3 \doteqdot \mathcal{E}(\overrightarrow{D}, \sigma_2) &= \mathcal{E}(\overrightarrow{D}, \langle V_2, \overline{V_0}, k_0 \oplus \delta k_C, R_D \oplus R, S_0 \oplus S_C \rangle) \\ &= \langle V_3, \overline{V_0}, k_0 \oplus \delta k_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle \end{aligned}$$

with V_3, S_D and δk_D depending only on V_2 and R_D

$$\begin{aligned} \sigma_6 \doteqdot \mathcal{E}(\overleftarrow{D}, \sigma_3) &= \mathcal{E}(\overleftarrow{D}, \langle V_3, \overline{V_0}, k_0 \oplus \delta k_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle) \\ &= \langle V_2, \overline{V_6}, k_0 \oplus \delta k_C, R, S_0 \oplus S_C \oplus S_D \rangle \end{aligned}$$

with V_2 and $\overline{V}_6$ depending only on V_3, $\overline{V_0}$ and δk_D

Considering now the checkpointed code, we propagate the state σ', starting from $\sigma'_0 = \sigma_0$ by using the axioms already introduced:

$$\sigma'_1 \doteqdot \mathcal{E}(\bullet, \sigma_0) = \mathcal{E}(\bullet, \langle V_0, \overline{V_0}, k_0, R_C \oplus R_D \oplus R, S_0 \rangle)$$

The snapshot-taking operation $\bullet$ stores a subset of the original values V_0 in the stack "Snp_C".

$$\sigma'_1 = \langle V_0, \overline{V_0}, k_0 \oplus Snp_C, R_C \oplus R_D \oplus R, S_0 \rangle$$

$$\sigma'_2 \doteqdot \mathcal{E}(C, \sigma'_1) = \mathcal{E}(C, \langle V_0, \overline{V_0}, k_0 \oplus Snp_C, R_C \oplus R_D \oplus R, S_0 \rangle)$$

The forward sweep of the checkpointed code $\overrightarrow{C}$ is essentially a copy of the checkpointed code C. As the only difference between the two states σ'_1 and σ_0 is the stack k and both C and $\overrightarrow{C}$ don't need the stack during run time ($\overrightarrow{C}$ stores values in the stack, but doesn't use it), the effect of C on the state σ'_1 produces exactly the same output values V_2 and the same collection of sent values S_C as the effect of $\overrightarrow{C}$ on the state σ_0 .

$$\sigma'_2 = \langle V_2, \overline{V_0}, k_0 \oplus Snp_C, R_D \oplus R, S_0 \oplus S_C \rangle$$

The next step is to run $\overrightarrow{D}$:

$$\sigma'_3 \doteqdot \mathcal{E}(\overrightarrow{D}, \sigma'_2) = \mathcal{E}(\overrightarrow{D}, \langle V_2, \overline{V_0}, k_0 \oplus Snp_C, R_D \oplus R, S_0 \oplus S_C \rangle$$

The output state of $\overrightarrow{D}$ uses only the input state's original values V and received values R. As V and R are the same in both σ'_2 and σ_2, the effect of $\overrightarrow{D}$ on the state σ'_2 produces the same variables values V_3, the same collection of messages sent through MPI communications S_D and the same set of values stored in the stack δk_D as the effect of of $\overrightarrow{D}$ on the state σ_2.

$$\sigma'_3 = \langle V_3, \overline{V_0}, k_0 \oplus Snp_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D \rangle$$

Then, the backward sweep starts with the backward sweep of D.

$$\sigma'_4 \doteqdot \mathcal{E}(\overleftarrow{D}, \sigma'_3) \;=\; \mathcal{E}(\overleftarrow{D}, \langle V_3, \overline{V_0}, k_0 \oplus Snp_C \oplus \delta k_D, R, S_0 \oplus S_C \oplus S_D\rangle$$

The output state of $\overleftarrow{D}$ uses only its input state's original values V, the values of the adjoint variables $\overline{V}$ and the values stored in the top of the stack δk_D. As V, $\overline{V}$ and δk_D are the same in both σ'_3 and σ_3, the effect of $\overleftarrow{D}$ on the state σ'_3 produces exactly the same variables values V_2 and the same values of adjoint variables $\overline{V}_6$ as the effect of $\overleftarrow{D}$ on the state σ_3.

$$\sigma'_4 \;=\; \langle V_2, \overline{V_6}, k_0 \oplus Snp_C, R, S_0 \oplus S_C \oplus S_D\rangle$$

$$\sigma'_5 \doteqdot \mathcal{E}(\circ, \sigma'_4) \;=\; \mathcal{E}(\circ, \langle V_2, \overline{V_6}, k_0 \oplus Snp_C, R, S_0 \oplus S_C \oplus S_D\rangle$$

The snapshot-reading operation $\circ$ overwrites V_2 by restoring the original values V_0. According to **Assumption 1**, the snapshot-reading $\circ$ conceptually also restores the collection of values that have been received during the first execution of the checkpointed part R_C.

$$\sigma'_5 \;=\; \langle V_0, \overline{V_6}, k_0, R_C \oplus R, S_0 \oplus S_C \oplus S_D\rangle$$

$$\sigma'_6 \doteqdot \mathcal{E}(\overrightarrow{C}, \sigma'_5) \;=\; \mathcal{E}(\overrightarrow{C}, \langle V_0, \overline{V_6}, k_0, R_C \oplus R, S_0 \oplus S_C \oplus S_D\rangle$$

The output state after $\overrightarrow{C}$ uses only on the input state's values V and the received values R. As V and R are the same in both σ'_5 and σ_0, the effect of $\overrightarrow{C}$ on the state σ'_5 produces the same original values V_2 and the same set of values stored in the stack δk_C as the effect of $\overrightarrow{C}$ on the state σ_0.

$$\sigma'_6 \;=\; \langle V_2, \overline{V_6}, k_0 \oplus \delta k_C, R, S_0 \oplus S_C \oplus S_D\rangle$$

Finally we have $\sigma'_6 = \sigma_6$. □

We have shown the preservation of the semantics at the level of one particular process p_i. The semantics preservation at the level of the complete parallel program P requires to show in addition that the collection of messages sent by all individual processes p_i matches the collection of messages expected by all the p_i. At the level of the complete parallel code, the messages expected by one process will originate from other processes and therefore will be in the messages emitted by other processes.

This matching of emitted and received messages depends on the particular parallel communication library used (e.g. MPI) and is driven by specifying communications, tags, etc. Observing the non-checkpointed code first, we have identified the expected *receives* and produced *sends* $S_U \oplus S_C \oplus S_D$ of each process. Since the non-checkpointed code is assumed correct, the collection of $S_U \oplus S_C \oplus S_D$ for all processes p_i matches the collection of $R_U \oplus R_C \oplus R_D$ for all process p_i.

The study of the checkpointed code for process p_i has shown that it can run with the same expected *receives* $R_U \oplus R_C \oplus R_D$ and produces at the end the same sent values $S_U \oplus S_C \oplus S_D$. This shows that the collected *sends* of the checkpointed version of P matches its collected expected *receives*.

However, matching *sends* with expected *receives* is a necessary but not sufficient condition for correctness. Consider the example of figure 5, in which we have two communications between two processes ("comm A" and "comm B"):

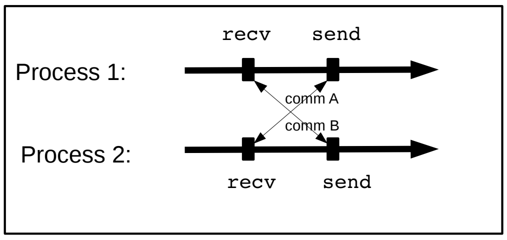


Figure 5: Example illustrating the risk of deadlock if *send* and *receive* sets are only tested for equality.

- The set of messages that process 1 expects to receive R= $\{$comm B$\}$. The set of messages that it will send is S= $\{$comm A$\}$.
- The set of messages that process 2 expects to receive R= $\{$comm A$\}$. The set of messages that it will send is S= $\{$comm B$\}$.

The above required property that the collection of *sends* $\{$comm A, comm B$\}$ matches the collection of *receives* $\{$comm A, comm B$\}$ is verified. However, this code will fall into a deadlock.
Semantic equivalence between two parallel programs requires not only that collected *sends* match collected *receives* but also that there is no deadlock. If, conversely:

Assumption 2. *the resulting checkpointed code is deadlock free,*

then, the semantics of the checkpointed code is the same as that of its non-checkpointed version.

To sum up, a checkpointing adjoint method adapted to MPI programs is correct if it respects these two assumptions:

Assumption 1. *The duplicated $recvs$ in the checkpointed part will collect the same values as their original calls.*

Assumption 2. *The checkpointed code is deadlock free.*

For instance, the "popular" checkpointing approach that we find in most previous works is correct because the checkpointed part which is duplicated is self-contained regarding communications. Therefore, it has always been assumed that the *receive* operations in that duplicated part receive the same value as their original instances. In addition, the duplicated part, being a complete copy of a part of the original code that does not communicate with the rest, is clearly deadlock free.
We believe, however, that this constraint of a self-contained checkpointed part can be alleviated. We will propose a checkpointing approach that respects our two assumptions for any checkpointed piece of code. We will then study a frequent special case where the cost of our proposed checkpointing approach can be reduced.

3 A GENERAL MPI-ADJOINT CHECKPOINTING METHOD

We introduce here a general technique that adapts the checkpointing to the case of MPI parallel programs and that can be applied to any checkpointed piece of code. This technique is

basically inspired by the works that have been done in the context of resilience [9]. Therefore, before detailing this general technique, we will start with a small analogy between the checkpointing in the context of resilience "Resilience-checkpointing" and the checkpointing in the context of AD-Adjoints. In both mechanisms, processes take snapshots of the values they are computing to be able to restart from these snapshots when it is needed. The difference is the reason why taking these snapshots. In the case of "Resilience-checkpointing", the reason is to recover the system from failure, whereas in the case of AD-adjoint, the reason is mostly the reduction of the peak of memory used. Also, the snapshots are called "checkpoints" in the case of "Resilience-checkpointing". Clearly the checkpoints in the context of Adjoint-AD are different from the checkpoints in the context of resilience. We recall that the checkpoints (or also the checkpointed parts) in the case Adjoint-AD are rather intervals of computation that are re-executed when it is needed.
There are two types of checkpointing for resilience: the non-coordinated checkpointing, in which every process takes its own checkpoint independently from the other processes and the coordinated checkpointing in which every process has to coordinate with other process before taking its own checkpoint. We are interested rather by the non-coordinated checkpointing, more precisely by the non-coordinated checkpointing coupled with Message logging. To cope with failure, every process saves in a remote storage checkpoints , i.e. complete images of the process memory. Also, every process saves the messages it receives and every `send` or `recv` event that it performs. In case of failure, only the failed process restarts from its last checkpoint. The other non-failed processes continue their executions normally. The restarted process runs exactly in the same way as before the failure, except that it does not perform any `send` call already done before the failure. The restarted process does not perform either any `recv` call already done, but retrieves instead the value that has been received and stored by the `recv` before the failure.

By analogy, we propose an adaptation of the checkpointing technique to MPI adjoint codes. This adapted technique (we call it "receive-logging") 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 (it is "deactivated"). Every *receive* operation, instead of calling any communication primitive, reads the previously received value from where it has been stored during the first execution.

- The type of storage used to store the received values is First-In-First-Out. This is different from the stack used by the adjoint to store the trajectory.

In the case of nested checkpointed parts, this strategy can either reuse the storage prepared for enclosing checkpointed parts, or free it at the level of the enclosing checkpointed part and re-allocate it at the time of the enclosed checkpoint. This can be managed using the knowledge of the nesting depth of the current checkpointed part.

Notice that this management of storage and retrieval of received values, triggered at the time of the `recv`'s or the `wait`'s, together with nesting depth management, can be implemented by a specialized wrapper around MPI calls, for instance inside the AMPI library [7].

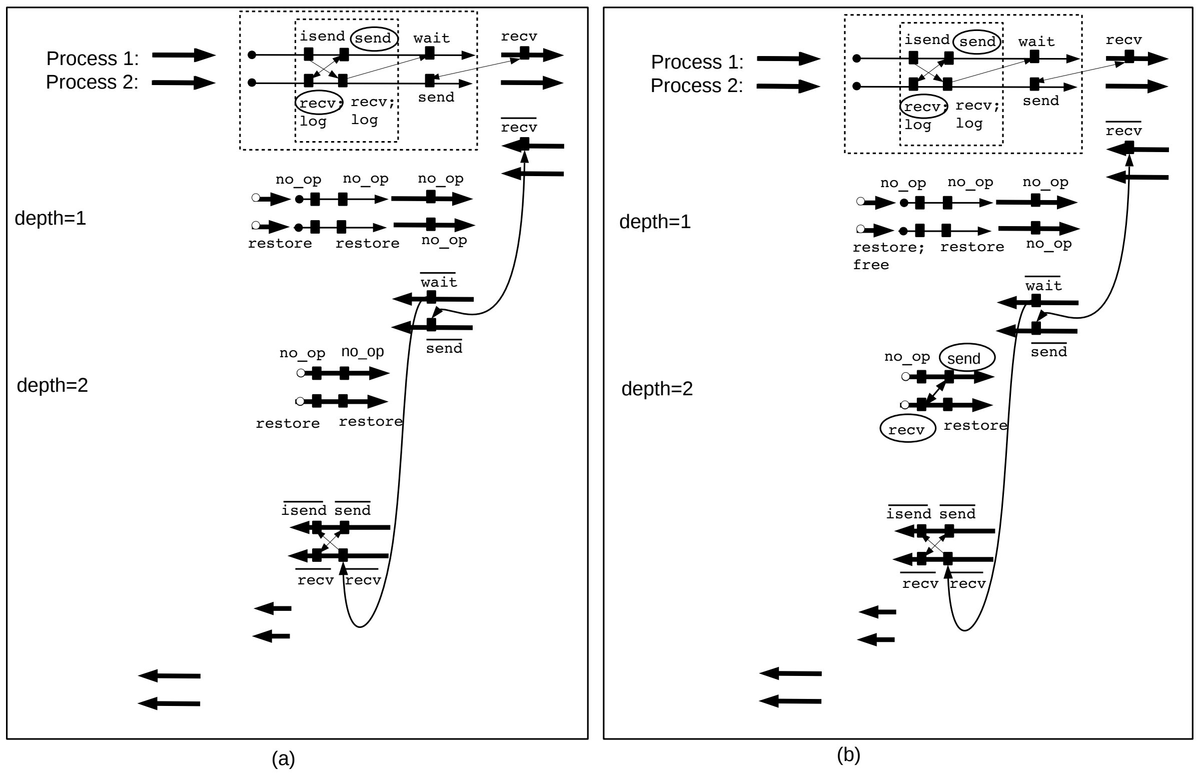


Figure 6: (a) Checkpointing a parallel adjoint program on two nested checkpointed parts by using the receive-logging method. (b) Refinement of the checkpointed code by applying the message re-sending to a `send-recv` pair with respect to the inner checkpointed code which is right-tight

Figure 6 (a) shows the example of two nested checkpointed parts together with an arbitrary communication pattern that straddles across the boundaries of the checkpointed parts.
During execution of the duplicated checkpointed parts, no communication call is made and *receive* operations read from the local storage instead. We can see that the communication pattern of the forward sweep is preserved by checkpointing, the communication pattern of the backward sweep is also preserved, and no communication takes place during duplicated parts.

To show that this strategy is correct, we will check that it verifies the two assumptions of section 2.

3.1 Correctness

By construction, this strategy respects **Assumption 1** because the duplicated *receives* read what the initial *receives* have received and stored.
To verify **Assumption 2** about the absence of deadlocks, it suffices to consider one elementary application of checkpointing, shown in the top part of figure 7. Communications in the checkpointed code occur only in $\overrightarrow{U}$, $\overrightarrow{C}$, $\overrightarrow{D}$ (about primal values) on one hand, and in $\overleftarrow{D}$, $\overleftarrow{C}$, $\overleftarrow{U}$ (about derivatives) on the other hand. The bottom part of the figure 7 shows the communications graph of the checkpointed code, identifying the sub-graphs of each piece of code. Dotted arrows express execution order, and solid arrows express communication dependency. Communications may be arbitrary between $G_{\overrightarrow{U}}$, G_C and $G_{\overrightarrow{D}}$ but the union of these 3 graphs is the same as for the forward sweep of the non-checkpointed code, so it is acyclic by hypothesis. Similarly, communications may be arbitrary between $G_{\overleftarrow{D}}$, $G_{\overleftarrow{C}}$ and $G_{\overleftarrow{U}}$ but (as $G_{\overrightarrow{C}}$ is by def-

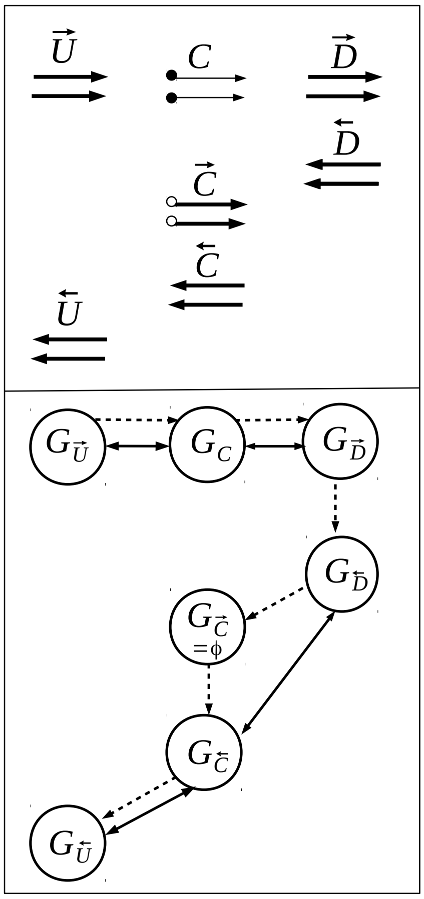


Figure 7: Communications graph of a checkpointed program with pure receive-logging method

inition empty) these graphs are the same as for the non-checkpointed backward sweep. Since we assume that the non-checkpointed code is deadlock free, it follows that the checkpointed code is also deadlock free.

3.2 Discussion

The receive-logging strategy applies for any choice of the checkpointed piece(s). However, it may have a large overhead in memory. At the end of the general forward sweep of the complete program, for every checkpointed part (of level zero) encountered, we have stored all received values, and none of these values has been used and released yet. This is clearly impractical for large codes.
On the other hand, for checkpointed parts deeply nested, the receive-logging has an acceptable cost as stored values are used quickly and their storage space may be released and used by checkpointed parts to come. We need to come up with a strategy that combines the generality of receive-logging with the memory efficiency of an approach based on re-sending.

4 USING MESSAGE RE-SENDING WHENEVER POSSIBLE

We may refine the receive-logging by re-executing communications when 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.

However, 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, i.e. there is no communication dependency arrow going from D to C in the communications graph of the checkpointed code. For instance, there must no `wait` in the checkpointed part that corresponds with communication call in other process which is downstream (i.e. after) the checkpointed part.

Figure 6 (a) shows an example of two nested checkpointed parts in which the outer checkpointed part is not right-tight, whereas the inner checkpointed part is right-tight since the dependency from the second `recv` of process 2 to the `wait` of the `isend` of process 1 only goes from the checkpoint inside to its outside. In the figure 6 (a), we identify a `send-recv` pair (whose ends are surrounded by circles) that belongs to both nested checkpointed parts. As the outer checkpointed part is not right-tight, we can apply the message re-sending to the `send-recv` pair only with respect to the inner checkpointed part. We see on figure 6 (b) that the `send-recv` pair is re-executed during the execution of the duplicated instance of the inner checkpointed part. As the duplication of the pair `send-recv` is placed between the $\overline{wait}$ of process 1 and the first $\overline{recv}$ of process 2 and since $\overline{wait}$ is a non blocking routine, the duplication of this `send-recv` pair does not create deadlock in the resulting adjoint.

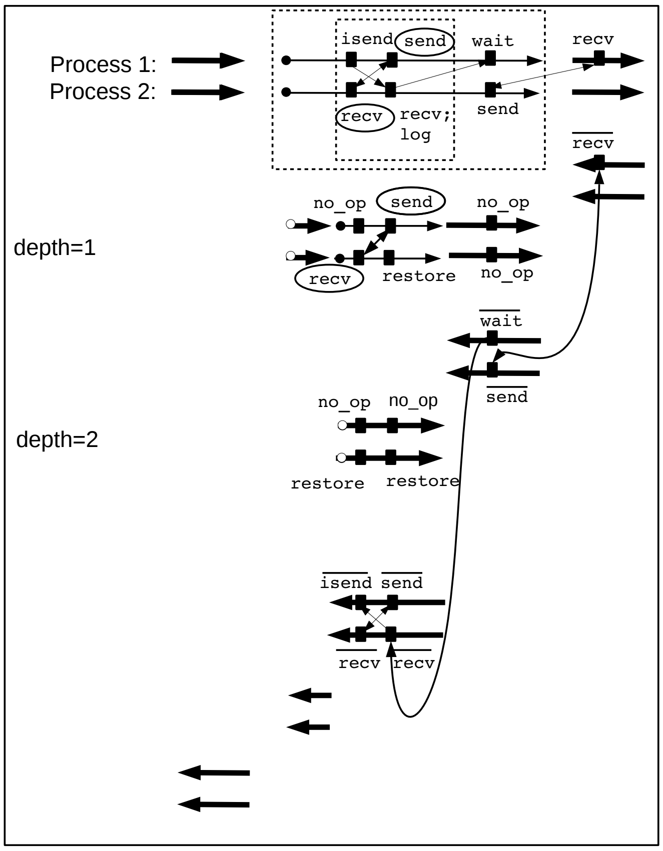


Figure 8: Application of the message re-sending to a `send-recv` pair with respect to the outer checkpointed part which is not right-tight

Figure 8 shows a counterexample, illustrating the danger of applying message re-sending to a checkpointed part which is not right-tight. We reuse the example of figure 6 (a). Instead of applying the message re-sending to the pair `send-recv` (whose ends are surrounded by circles) with respect to the inner checkpointed code as it is the case in figure 6 (b), we applied the message re-sending to the pair `send-recv` with respect to the outer checkpointed code which is not right-tight. Figure 8 shows the cycle in the communications graph of the resulting adjoint. We see on the figure that, between the $\overline{recv}$ of process 1 and the $\overline{send}$ of process 2 takes place the duplicated run of the outer checkpointed part. In this duplicated run, we find a duplicated `send-recv` pair that causes a synchronization. Execution thus reaches a deadlock, with process 1 blocked on the $\overline{recv}$, and process 2 blocked on the duplicated `recv`.

Only when the checkpointed part is right-tight can we mix message re-sending of communications pairs that are contained in the checkpointed part with receive-logging of the others. The interest is that memory consumption is limited to the (possibly few) logged *receives*. The cost of extra communications is tolerable compared to the gain in memory.

4.1 Correctness

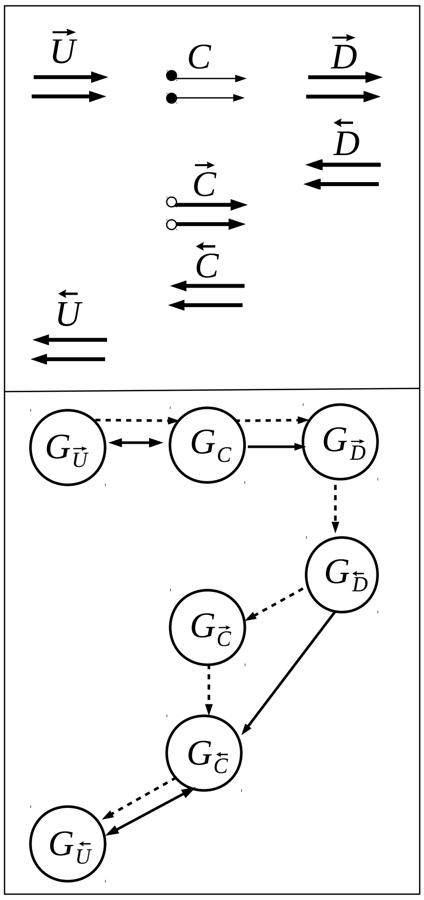


Figure 9: Communications graph of a checkpointed program by using the receive-logging coupled with the message re-sending

The subset of the duplicated *receives* that are treated by receive-logging still receive the same value by construction. Concerning the duplicated `send-recv` pair, the duplicated checkpointed part computes the same values as its original execution (see step from σ'_5 to σ'_6 in section 2). Therefore the duplicated `send` and the duplicated `recv` transfer the same value.

The proof about the absence of deadlocks is illustrated in figure 9. In contrast with the pure receive-logging case, $G_{\overrightarrow{C}}$ is not empty any more because of re-sent communications. $G_{\overrightarrow{C}}$ is a subgraph of G_C and is therefore acyclic. Since the checkpointed part is right-tight, the dependency from G_C to $G_{\overrightarrow{D}}$ and from $G_{\overleftarrow{D}}$ to $G_{\overleftarrow{C}}$ are unidirectional. There is no communication dependency between $G_{\overrightarrow{C}}$ and $G_{\overleftarrow{D}}$ and $G_{\overleftarrow{C}}$ because $G_{\overrightarrow{C}}$ communicates only primal values and $G_{\overleftarrow{D}}$ an $G_{\overleftarrow{C}}$ communicate only derivative values.
Assuming that the communications graph of the non-checkpointed code is acyclic, it follows that:

- Each of $G_{\overrightarrow{U}}$, $G_{\overrightarrow{C}}$, $G_{\overrightarrow{D}}$, $G_{\overleftarrow{D}}$, $G_{\overleftarrow{C}}$ and $G_{\overleftarrow{U}}$ is acyclic.
- Communications may be arbitrary between $G_{\overrightarrow{U}}$ and G_C but since these pieces of code occur in the same order in the non-checkpointed code, and it is acyclic, there is no cycle involved in ($G_{\overrightarrow{U}}$; G_C). The same argument applies to ($G_{\overleftarrow{C}}$; $G_{\overleftarrow{U}}$).

Therefore, the complete graph on the bottom of figure 9 is acyclic.

5 DISCUSSION AND FURTHER WORK

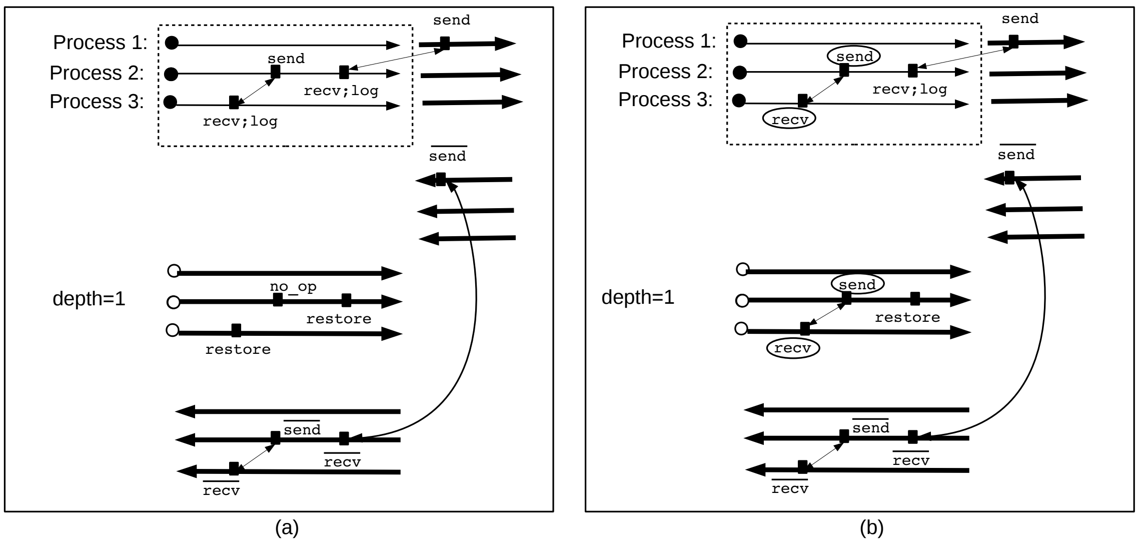


Figure 10: (a) The receive-logging applied to a parallel adjoint program. (b) Application of the message re-sending to a `send-recv` pair with respect to a non-right-tight checkpointed code

We studied checkpointing in the case of MPI parallel programs with point-to-point communications. We proved that any technique that adapts the checkpointing mechanism to MPI parallel programs and that respects some sufficient conditions, is a correct MPI checkpointing technique, in the sense that, the checkpointed code resulting from the application of this MPI checkpointing technique preserves the semantics of the non-checkpointed adjoint code. We introduced, a general MPI checkpointing technique that respects the sufficient conditions 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. We proposed a refinement that reduces the memory consumption of this general technique by duplicating the communications whenever possible. 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 refinement. These are sufficient conditions, but it seems they are not completely necessary. Figure 10 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.

In real codes we may have nested structure of checkpointed parts in which each checkpointed part may be or not right-tight. Applying the message re-sending to only checkpointed parts that are right-tight means that some communication calls will be activated in some levels and deactivated in the other levels. Thus, to implement the refined receive-logging, we need to think about the way we could automatically alternate between these two situations for each communication call. For instance, a *receive* that is deactivated at a level and activated at the level just after has to release its stored value.

Finally, these checkpointing techniques need to be experimented in real codes. It would be interesting to measure the memory consumption of the general checkpointing technique before and after the application of message re-sending.

6 ACKNOWLEDGEMENT

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] A. Griewank, A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Other Titles in Applied Mathematics, #105. SIAM, 2008.

[2] M. Snir, S. Otto, *The Complete Reference: The MPI Core*. Cambridge, MA, USA: MIT Press, 1998.

[3] P. Hovland, Automatic differentiation of parallel programs *Ph.D. dissertation, University of Illinois at Urbana-Champaign*, 1997.

[4] A. Carle, M. Fagan, Automatically differentiating MPI-1 datatypes: The complete story, in *Automatic Differentiation of Algorithms: From Simulation to Optimization*, ser. Computer and Information Science, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds. New York: Springer, 2002, ch. 25, pp. 215-222.

[5] P. Heimbach, C. Hill, R. Giering, *An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation Future Generation Computer Systems 21*. 2005. p. 1356-1371.

[6] U. Naumann, L. Hascoët, C. Hill, P. Hovland, J. Riehme, J. Utke, A Framework for Proving Correctness of Adjoint Message Passing Programs. *In Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer. 2008. p. 316-321.

[7] 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.

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

[9] F. Capello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward Exascale Resilience: A 2014 Update. *Supercomput.Front. Innovations 1(1) (2014)*.