Experiments on Checkpointing Adjoint MPI Programs

A. Taftaf
INRIA, Sophia-Antipolis, France, Elaa.Taftaf@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 for most MPI parallel applications. However, the presence of MPI communications seriously restricts application of checkpointing. In the most popular approach, a number of restrictions apply on the form of communications that occur in the checkpointed piece of code. In previous work, we proposed a general technique that lifts this restriction. This technique (called “receive-logging”) is based on logging the values received, so that no duplicated communication is needed. However, the message logging makes it more costly than the popular approach. We proposed a refinement to our technique to duplicate communications whenever it is possible, so that the refined receive-logging now encompasses the popular approach. In this work we see checkpointing MPI parallel programs from a practical point of view. We discuss an important question about the choice of the checkpointed pieces. We validate our theoretical results on a representative code in which we perform various choices of checkpointed pieces. We apply the refined receive-logging to these checkpointed pieces and we quantify the expenses in terms of memory and computation time for each resulting checkpointed adjoint.

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 for 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 particular, both ends of each communication must belong to the same checkpointed piece and the non blocking routines and their waits must be checkpointed together. If only one end is contained in the checkpointed piece of code, the resulting adjoint fails.

In previous work, we proposed a technique to apply checkpointing to adjoint MPI codes. This technique (called “refined receive-logging”) is more general than the popular approach, i.e. it imposes less restrictions. The main idea of this technique (see figure 1) is to duplicate every communication whose ends belong to the checkpointed piece and to apply “receive-logging” to all the remaining communications, which are actually ends of communications whose other ends are outside the checkpointed piece (we call them “orphan communications”). Applying the receive-logging to one end of a communication means that:

- During the first execution of the checkpointed piece, this end of communication is executed normally. However, if it is a receive operation, then it stores its received value into some location local to the process.
- During the duplicated execution of the checkpointed piece, this end of communication is not executed anymore. However, if it is a receive operation, then it reads the previously received value from where it has been stored during the first execution of the checkpointed piece.
In this work we study checkpointing of MPI programs from a practical point of view. Before actually experimenting the refined receive-logging on a representative example, we want to study the question of the choice of checkpointed pieces. More precisely, we want to ask: if we have the choice between two checkpointed pieces, both of them reducing in the same way the peak memory consumption, is it better to choose the one that does not contain any orphan communication, in which case the application of the refined receive-logging implies the application of the standard approach? Or is it better to choose the one that contains a blend of orphan and non-orphan communications? One may think that the first choice is better as it does not require any message logging. However, in practice, this may not be right. In fact, applying checkpointing on MPI parallel programs has not only the cost of logging orphan receives, but also the cost of snapshots. In general the memory cost of the refined receive-logging may be written as:

\[
\text{CheckpointingCost} = \sum_{i=1}^{n} (\text{SnapshotCost}_{P_i}) + \sum_{i=1}^{m} (\text{OrphanReceiveCost}_i)
\]

where \(\text{SnapshotCost}_{P_i}\) is the snapshot cost at each process \(P_i\), \(n\) is the number of processes involved in the checkpointed piece, \(\text{OrphanReceiveCost}_i\) is the memory cost of one orphan receive and \(m\) is the number of orphan receives inside the checkpointed piece. We note here that when the checkpointed piece includes many communications, it contains consequently few orphan receives and thus the cost of \(\sum_{i=1}^{n} (\text{orphanReceiveCost}_i)\) is small. However, including many communications, means also that sometimes we have many processes involved in the checkpointed piece and thus the cost of \(\sum_{i=1}^{m} (\text{SnapshotCost}_{P_i})\) is high. Consider the example of figure 2 with two alternative checkpointed pieces \(C_1\) and \(C_2\). Piece \(C_1\) contains only non-orphan communications and piece \(C_2\) contains only orphan communications, actually only sends. We apply the refined receive-logging to both alternatives.

The cost of checkpointing piece \(C_1\) is \(\sum_{i=1}^{3} (\text{SnapshotCost}_{P_i})\) and the cost of checkpointing piece \(C_2\) is \(\sum_{i=1}^{3} (\text{SnapshotCost}_{P_i})\). We observe that although piece \(C_2\) contains many orphan communications, the memory cost of checkpointing \(C_2\) is lower than that of checkpointing \(C_1\). This can be explained by the fact that these orphan communications are sends and then they do not require any message logging. However, if the orphan communications were receives operations, the result might have been different. Actually, in this case we would have to compare the snapshot cost of the process 0 with the message logging cost of the orphan receives.

We validate our theoretical results on a representative code, with various choices of checkpointed pieces in which we apply the refined receive-logging. We quantify the cost in term of memory and computation time for every resulting checkpointed adjoint.

References