OATAO
Open Archive Toulouse Archive Ouverte

Open Archive Toulouse Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is a Publisher's version published in: http://oatao.univ-toulouse.fr/25696

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr

computing derivatives with AD: parallel communications and unsteady computations. The tangent mode of AD can deal with these features quite easily, while the adjoint mode of AD is affected by these features as it is based on reversal of the data-flow of the primal code [12]. Unfortunately, many real-life applications require computing derivatives of relatively few outputs (cost functions, constraints…) with respect to many inputs (state variables, design parameters, mesh coordinates…). Tangent AD is out of question in such cases since $p \ll q$. As long as the adjoint mode provided by AD tools did not address these serious limitations, most studies [1-3] circumvented them by using the following strategies:

- Applying AD on selected parts of the code without MPI calls and manually assembling the differentiated routines to obtain a correct adjoint code
- Restricting the use of AD to code solving problems that are either stationary of forced to become stationary. As an exception, earth sciences have long used adjoints of unsteady simulations [4-5], pioneering the so-called checkpointing schemes that we advocate here.

In this talk, we report on the outcome of exploiting the recently acquired maturity of AD tools [6] at differentiating parallel code in adjoint mode by automatic inversion of the MPI communication layer [7] while handling the cost of an unsteady computation.

2. SOFTWARE USED

We now provide details related to the flow solver and the AD tools used in our study.

2.1. Flow solver

The code under consideration in the present study – hereafter the *primal* code – is a computational fluid dynamics (CFD) solver that integrates the governing equations for compressible fluid flow. It belongs to a recent trend of CFD codes that use a high-order spatial discretization adapted to

---

KEYWORDS: computational fluid dynamics, automatic differentiation, sensitivity analysis, flow control, optimization.

ABSTRACT:

We compute flow sensitivities by differentiating a high-order computational fluid dynamics code. Our fully discrete approach relies on automatic differentiation (AD) of the original source code. We obtain two transformed codes by using the AD tool Tapenade (INRIA), one for each differentiation mode: tangent and adjoint. Both differentiated codes are tested against each other by computing sensitivities in an unsteady test case. The results from both codes agree to within machine accuracy, and compare well with those approximated by finite differences. We compare execution times and discuss the encountered technical difficulties due to 1) the code parallelism and 2) the memory overhead caused by unsteady problems.

1. INTRODUCTION

Numerical codes in engineering and physical sciences are most often used to approximate the solution of governing equations on discretized domains. A natural step beyond obtaining solutions in specific conditions is to seek those conditions that modify the solution towards a specific goal, either for optimization or control purposes. In this context, gradient-based methods play an important role. They require invariably the computation of derivatives, a task that can be automated by Automatic Differentiation (AD) tools. In short, AD augments outputs Y_i from inputs X_j into a "differentiated" code that additionally computes some derivatives dY_i/dX_j requested by the user. AD provides two main modes, the tangent/direct/forward mode and the adjoint/reverse/backward mode. If $1 \le i \le p$ and $1 \le j \le q$ bound the size of the output and input spaces, respectively, then the tangent mode is most efficient when $q \ll p$ while the adjoint mode is the only realistic option for $p \ll q$.

Two features of high-performance codes used in industry and academia complicate the process of

AerospaceEurope
Conference2020
BORDEAUX, FRANCE | 25 - 28 FEBRUARY 2020

CEAS
Council of European
Aerospace Societies
HOSTED BY :
3AF
Association Aéronautique
et Astronautique de France

compressible/incompressible flow computations on complex geometries modeled by unstructured meshes. This makes them suitable candidates to become industrial tools in the near future [8]. Our application code is Jaguar [9-10], a solver for aerodynamics applications developed to suit the future needs of the aerospace industry. The code is written following features from the Fortran 90 standard onward, with a zero-halo partitioning scheme and MPI-based parallelization.

2.2. AD tool

AD can be based on two working principles: operator overloading (OO) or source transformation (ST). The OO approach barely modifies the primal code: the data-type of numeric variables is simply modified to contain their derivative in addition to their primal value, while arithmetic operations are overloaded to act on both components of the variables. While the debate is still active, it is generally agreed that ST AD tools require a much heavier development, which is in general paid back by a better efficiency of the differentiated code mostly in terms of memory consumption. Benchmark tests have pointed out a tendency for OO-differentiated codes to be more memory demanding and somewhat slower than their ST counterparts [2]. On the other hand, the higher flexibility of the OO model makes it almost readily applicable to languages with sophisticated constructs, such as C++ or Python, for which no ST tool exists to date. For a given application, the choice between AD tools based on ST or OO is dictated by these constraints: with Jaguar being written in Fortran, ST appears to be the natural choice. Moreover, for the size and number of time steps of our targeted applications, it is essential to master the memory footprint of the final adjoint code. For this study, we have selected the ST-based AD tool Tapenade [11], developed by INRIA.

2.3. AD of very long time-stepping sequences

The adjoint mode of AD leads to a code which executes the differentiated instructions in the reverse order of the primal code. However, these differentiated instructions (the "backward sweep") use partial derivatives based on the values of the variables from the primal code. The primal code, or something close to it, must therefore be executed beforehand, forming the "forward sweep". As codes generally overwrite variables, a mechanism is needed to recover values overwritten during the forward sweep, as they are needed during the backward sweep. Recovering intermediate values can be done either by recomputing them at need, from some stored state, or by storing them on a stack during the forward sweep and retrieving them during the backward sweep. Neither option scales well on large codes, either with a memory use that

grows linearly with the primal code run time, or with an execution time that grows quadratically with the primal code run time. We envision applications with 10^5 to 10^6 time steps to integrate the fluid flow equations. The classical answer to this problem is a memory-recomputation trade off known as "checkpointing" [12]. A well-chosen checkpointing strategy can lead to execution time and memory consumption of the adjoint code that grow only logarithmically with the primal code run time.

A checkpointing strategy is constrained by the structure of the primal code. Checkpointing amounts to designating (nested) portions of the code, for which we are ready to pay duplicate execution to gain memory storage of its intermediate computations. These portions must have a single-entry point and a single exit point, for instance procedure calls or code parts that could be written as procedures. For this reason, one cannot in practice implement the theoretical optimal checkpointing scheme, which is defined only on a fixed-length linear sequence of elementary operations of similar cost and nature. A checkpointing scheme on a real code can still achieve a logarithmic behavior, but in general below the theoretical optimal. Moreover, since checkpointed portions are supposed to be executed twice or more, they must be "reentrant": it must be possible to re-create the exact machine state at their entry point, and running them twice must not alter the rest of the execution. As a consequence, a checkpointed portion of code must always contain both ends of an MPI communication, and similarly both halves of a non-blocking MPI communication [13].

Time-stepping simulations are more fortunate: at the granularity of time steps, the code is indeed a fixed-length sequence of elementary operations of similar cost and nature. The binomial checkpointing scheme [14] exactly implements the optimal strategy in that case, and Tapenade applies it when requested. A checkpointing strategy is also constrained by the characteristics of the storage system. The binomial strategy assumes a uniform and negligible cost for storing and retrieving the memory state before checkpoints ("snapshots"). This is in reality never the case. New research [15] looks for checkpointing strategies that take this memory cost into account, as well as different access times for different memory levels.

In general, few studies confront unsteady problems directly, and most works reported in the literature focus on problems around a fixed-point solution. Convergence towards that fixed steady state is often enforced by means of implicit iterative schemes with preconditioning. Few iterations are necessary, and only the final converged state

requires storage before computing the inverted set of instructions. Consequently, memory and computational overhead are kept low. It is unfortunate, however, that many problems of industrial relevance are inherently unsteady. In acoustics and combustion, for instance, unsteadiness simply cannot be ignored, which is what motivates the present study.

2.4. Parallel communications

An additional challenge arises when the code to be inverted by the AD tool contains message-passing instructions, which also need to be invoked in inverted order. Much conceptual work has been devoted to AD of MPI code [16-20]. However, it is with the recent advent of the Adjoinable MPI library [7] that several AD tools (Adol-C, Rapsodia, dco, OpenAD, Tapenade) support AD of code containing MPI calls. Related projects include the Adjoint MPI library [18,20,21] compatible with the dco suite of AD tools, and CoDiPack [22] for C++ code - both based on OO. The automatic inversion of MPI calls necessary to derive a parallel adjoint code can thus be performed by three tools. In [21], the CFD code OpenFOAM was adjointed with the combination of dco/c++ and Adjoint MPI. CoDiPack was used to adjoint the CFD code SU2 [23]. To our knowledge, these are the most similar studies to ours in terms of letting the AD tool handle the parallel communication layer automatically - yet without solving a time-dependent problem.

Avoiding MPI idioms in code fed to an AD tool is an attractive choice. Individually differentiated routines can be manually assembled into an adjoint code that preserves the often heavily optimized parallel communications layer of the primal code. A disadvantage of this approach is the increased workload incurred every time a different problem is tackled, where the optimization concerns different quantities from those previously considered. A certain degree of freedom in choosing the cost function and automation in assembling differentiated procedures has been achieved in [1]. Alternatively, [24] has presented the transposed forward-mode algorithmic differentiation to take advantage of those code portions featuring symmetric properties in order to obtain adjointed code using the forward-mode AD. Either way, handling MPI calls differently from the rest of the code contradicts the ultimate goal of AD, which is full automation of the differentiation process regardless of the programming features actually used in the primal code. It is true, however, that each specific library that involves side-effects raises new issues, limitations, and challenges that cannot be readily solved by AD tools. Given the efforts that have been devoted towards making MPI calls compatible with AD tools [16-20], we aim to test and document the outcome of letting the AD tool handle them alone. In this respect, our work intends to provide a proof of concept illustrating that the route followed, which on the whole has been avoided in the literature, is in fact practicable.

3. TEST CASE

We consider a viscous, two-dimensional incompressible flow in a square periodic domain spanning $L = 1$ in the streamwise (x) and vertical (y) directions. The velocity field at the initial instant t_0 is given by

$$u = U \tanh[r(y - 1/4)], \qquad y \leq 1/2 \qquad (1)$$
$$u = U \tanh[r(3/4 - y)], \qquad y > 1/2 \qquad (2)$$
$$v = U\delta \sin[2\pi(x + 1/4)], \qquad\qquad (3)$$

where all quantities are made non-dimensional with L and the streamwise reference velocity $U_0 = 1$ as follows:

$$t = \tilde{t}U_0/L, \; y = \tilde{y}/L, \; x = \tilde{x}/L, \; U = \tilde{U}/U_0, \; r = \tilde{r}L. \; (4)$$

The parameters of the problem are U, r and δ. These are the streamwise velocity amplitude, the shear parameter and the ratio of vertical to streamwise velocity amplitudes, respectively. We set $\delta = 0.05$ for the remainder of the study so that is is no longer a free parameter. We analyse the evolution of the overall enstrophy Ω, defined as

$$\Omega = \int_0^1 \int_0^1 \frac{1}{2} \omega_z^2 \, dx \, dy, \qquad\qquad (5)$$

where $\omega_z = \partial_x v - \partial_y u$ is the vorticity. It can be readily shown from Eqs. 1-3 and Eq. 5 that at $t = t_0$,

$$\Omega = U^2 \left[6r \tanh\left(\frac{r}{4}\right) - 2r \tanh^3\left(\frac{r}{4}\right) + 3\delta^2\pi^2 \right]/3 \quad (6)$$

and we choose $r = 40$ with $U = 1$ to yield an initial enstrophy level $\Omega_{ref} = 53.36$ which we set as a constraint for all r. This implies U is a function of r only, determined by re-arranging Eq. 6 as follows:

$$U(r) = \left[3\Omega_{ref}/(6r \tanh(r/4) - 2r \tanh^3(r/4) + 3\delta^2\pi^2) \right]^{1/2}. \qquad (7)$$

The Reynolds number is $Re_0 = U_0 L/\nu = 1.176 \times 10^4$. We show the spatial distribution of ω_z at four different instants obtained with $r = 160$ on Fig. 1. The dependence of Ω on time and r is clear from Fig. 2.
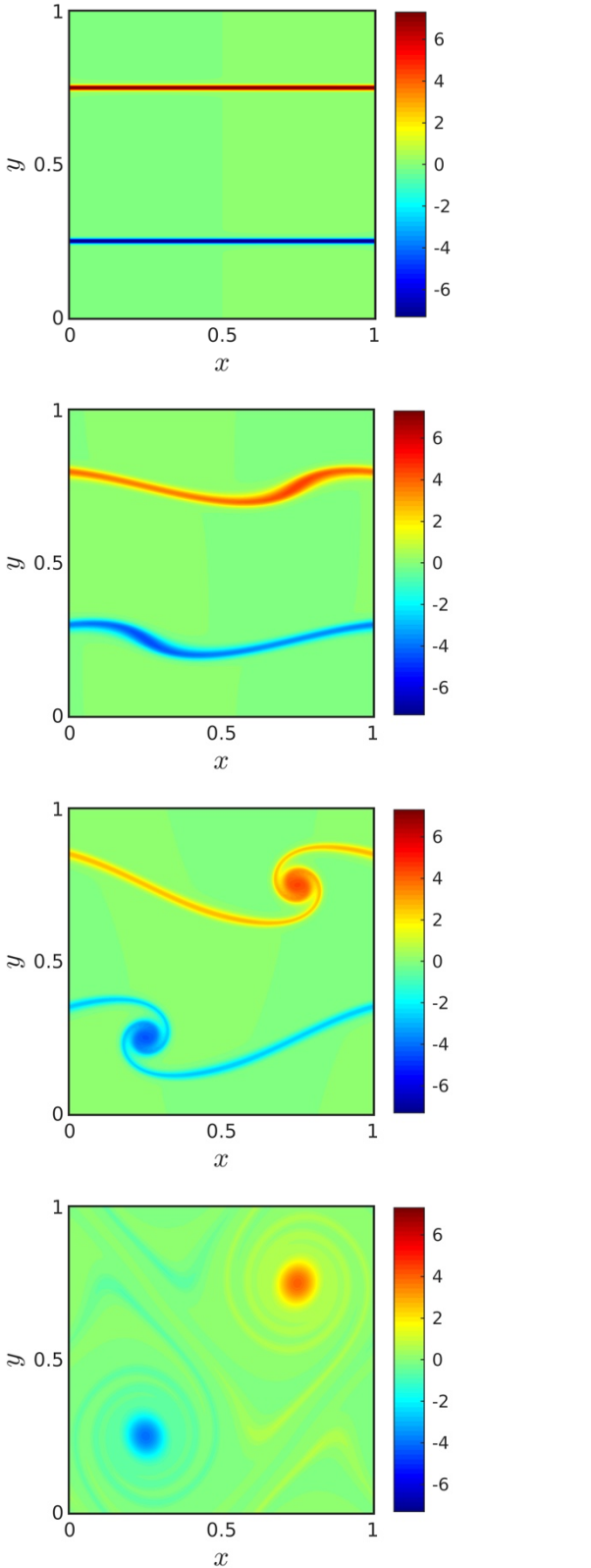
Figure 1 Spatial distribution of out-of-plane vorticity $\omega_z/\sqrt{\Omega_{ref}}$ at instants $t\sqrt{\Omega_{ref}} = \{0,7,10,23\}$ from top to bottom, with $r = 160$.
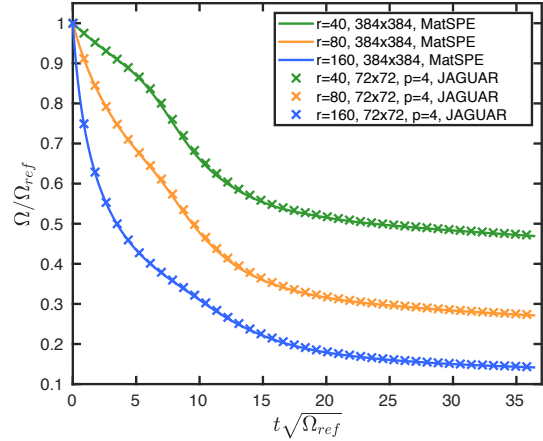

Figure 2 Time dependence of Ω at three different values of shear parameter r. Results obtained with codes Jaguar and MatSPE. The legend includes the number of Fourier modes used in each direction for the MatSPE simulation (of which 1/3 are zero-padded for dealiasing), while the JAGUAR data includes parameter p which is the selected order of the spatial discretization of the spectral difference scheme. The grid used in the JAGUAR simulation is a 72×72 structured mesh which, together with the setting $p = 4$, yields 360 degrees of freedom (DoF) per spatial direction. So we are effectively comparing 256^2 DoF with MatSPE against 360^2 DoF with JAGUAR.

The incompressible 2D Navier-Stokes equations are solved with the initial and boundary conditions outlined above using JAGUAR on a structured mesh with 72×72 square elements. The Mach number is set to zero to approach, as much as possible, incompressibility. The solution and flux points are located following Gauss-Lobatto-Chebyshev and Legendre collocation, respectively, and the CFL is kept constant at 0.5. The fluxes at the cell faces are computed with the Roe scheme. The temporal integration is done with a six-stage, fourth-order low-dissipation low-dispersion Runge-Kutta scheme optimized for the spectral difference code using the procedure in [25]. We use an in-house, fully spectral code (MatSPE) designed for periodic incompressible viscous flows to solve the same test case and validate Jaguar. The output of the two codes is compared on Fig. 2, showing that the agreement between the codes is extremely good.

We define the following cost function

$$J = \int_0^T \Omega(t)\, dt. \qquad (8)$$

Its derivative with respect to r will be the target sensitivity we compute by means of AD. From a physical point of view, $\Omega(t)$ is directly proportional to the rate of kinetic energy dissipation in the flow due

4

AerospaceEurope Conference 2020
BORDEAUX, FRANCE | 25 - 28 FEBRUARY 2020
CEAS
Council of European Aerospace Societies
HOSTED BY:
3AF
Association Aéronautique et Astronautique de France

to the action of viscosity. Hence, the area under a curve of $\Omega(t)$ on Fig. 2 for a given time interval is a proxy for the kinetic energy dissipated by the flow during that time. It may be argued that since our numerical experiments only target dJ/dr, it is pointless to use adjoint-mode AD because r is a scalar. One must bear in mind, however, that the final application of our study is to compute sensitivities of some J with respect to many inputs. Restricting the number of inputs to one in the sequel is only a convenient way to validate our proposed work flow and the AD derivatives, as well as to compare and study performance.

The computation of J in the primal code is carried out by adding a contribution to the time integral at each new time step in a running sum fashion, using a simple trapezoidal rule. Such a framework is convenient to illustrate the fundamental issue of adjoint-mode AD that we already described in section 2.3. The tangent-differentiated code accumulates contributions of each time step to dJ/dr, along with the primal time-stepping sequence, i.e. in the same order. This is conceptually simple: once we reach the iteration corresponding to $t = T$ (which we will call iteration number N), both J and dJ/dr are known and the program can end. In contrast, the adjoint-differentiated code will first run an initial forward sweep that will integrate the Navier-Stokes equations from $t = 0$ to the iteration N of the time-stepping loop, chiefly to generate the final state of the program variables. Only then can the backward sweep of the adjoint code start to accumulate derivatives, computing the sensitivity of J with respect to the state variables at iteration $N - 1$, and carry on stepping back in time to finally obtain dJ/dr when $t = 0$ is reached. In order to provide intermediate values from the forward sweep to the backward sweep in the correct order (i.e. reversed), a combination of stack storage and additional forward recomputation is needed, making a good checkpointing scheme essential. The recomputations and stack use will inevitably imply that one run of the adjoint code requires significantly more memory and execution time than the tangent code. We thus expect the tangent code to still outperform the adjoint code when $p = q$ or when q is only a few times larger than p, but the adjoint code will definitely outperform the tangent code when $p \ll q$, which is the case in many applications.

4. SENSITIVITY VALIDATIONS WITH FINITE DIFFERENCES

The estimates of a given sensitivity computed with tangent- or adjoint-mode AD codes should agree almost to machine precision, as they result from the same computation modulo associativity-commutativity. In contrast, the reference value against which to validate the sensitivity is obviously a finite-difference (FD) estimate, which is subject to errors due to the contribution of higher derivatives. For the viscous test case, we compute our FD estimates with two independent realizations at r and $r + dr$, where $dr/r = 10^{-5}$. We thus expect an agreement between FD and AD derivatives up to more or less half of the decimals, whereas we expect a much better agreement between tangent AD and adjoint AD derivatives.

5. DIFFERENTIATION WORK FLOW

We adopted the following work flow with regards to the Jaguar flow solver, the working principles of Tapenade and the Adjoinable MPI library:

A. Identify the part of the code that computes the function to differentiate, exactly from the differentiation input variables to the differentiation output, and make it appear as a procedure (the "head" procedure). This may require a bit of code refactoring. The differentiation tool must be given (at least) this root procedure and the call tree below it.

B. Migrate all MPI calls to Adjoinable MPI, whether AD will be applied in tangent or in adjoint mode. This involves two steps. First, as Adjoinable MPI does not support all MPI communication styles (e.g. one-sided), the code must be transformed to only use the supported styles, which is a reasonably large subset. Second, effectively translate the MPI constructs into their Adjoinable MPI equivalent, which occasionally requires minor modifications to the call arguments. As Adjoinable MPI is just a wrapper around MPI, the resulting code should still compile and run, and it is wise to test that.

C. Provide the AD tool with the source of the head procedure and of all the procedures that it may recursively call, together with the specification of the differentiation input and output variables. Then differentiate, after which two steps follow. First, fix all issues signaled by the AD tool, e.g. unknown external procedures or additional info needed, and validate the differentiated code. Second, address performance issues and in particular optimize the checkpointing strategy by adding AD-related directives to the source. This may also involve special treatment of linear algebra procedures such as solvers.

AerospaceEurope Conference2020
BORDEAUX, FRANCE | 25 - 28 FEBRUARY 2020
CEAS
Council of European Aerospace Societies
HOSTED BY:
3AF
Association Aéronautique et Astronautique de France

6. RESULTS

The temporal integration of the equations of motion is carried out from $t = 0$ to the $n - th$ iteration of the time integration loop in Jaguar, with $n = 6.8 \times 10^5$. With the CFL setting outlined in section 3, this number of time steps corresponds to $t\sqrt{\Omega_{ref}} = 19.8$, which from Fig. 1 can be seen as the time when the decay of Ω becomes slow for all three values of r. The value of dJ/dr is given in Tab. 1, where the results from FD, tangent-mode AD and adjoint-mode AD are all gathered. The agreement between FD-based sensitivities and AD validates the differentiation procedure.

Table 1 dJ/dr computed with three different methods, for the time integration interval between $t = 0$ and $t = T$ (6.8×10^5 time steps). Viscous test case with $r = 160$, run with 16 parallel processes.

Differentiation method	Sensitivity dJ/dr
FD (MPI)	-0.22002**254**
Tangent AD (AMPI)	-0.22002394265**381**
Adjoint AD (AMPI)	-0.22002394265**861**

The agreement between the two AD-based sensitivities is excellent, within round-off error of double precision arithmetic. It was expected in case of correct differentiation by Tapenade, but nevertheless it is puzzling when comparing the drastic differences between the two differentiated codes. The fact that the adjoint-differentiated code outputs the correct answer after carrying out the time-stepping loop backwards confirms the absence of any stability or convergence issues related to inverting the instructions of a code that integrates in time a reversible and dissipative system. More specifically, there is no issue of numerical instability caused by a term with negative diffusivity.

The time required for the various computations is shown on Tab. 2. A factor of two is indicated for the FD computation, given that two runs of the primal code are required. The computations are run on 16 Intel(R) Xeon(R) Gold 6140 processors at 2.30GHz. The Intel Fortran compiler version 18.0.2 is used, with identical optimization flags for all codes: -xCORE-AVX2 -ipo -O3 -qopt-malloc-options=3. It appears that the tangent-mode AD can be faster than two runs of the primal code, requiring 1.7 times the execution time of the primal code. The higher accuracy of sensitivity computations from tangent-mode AD thus comes with the added benefit of a faster computation than that of FD. Furthermore, each additional cost function differentiated with respect to r will require an additional FD computation, whereas the cost of each new sensitivity with respect to r will keep the cost of the tangent-mode execution constant. We note in passing that the FD computation is based on the

MPI code before the modifications of step B. outlined in section 5, so that the tangent-mode AD is faster than the FD computations despite the move from MPI to Adjoinable MPI. This indicates that the cost of this additional wrapper on top of the MPI library is negligible.

Table 2 Same as Tab. 1, but showing execution times normalized by the primal code execution time.

Differentiation method	Normalized run time
FD (MPI)	1 (x2)
Tangent AD (AMPI)	1.7
Adjoint AD (AMPI)	15.4

Tab. 2 also shows the slowdown factor of the adjoint code, which is 15.4 and deserves some discussion. An initial experiment without any specific checkpointing scheme simply ran out of memory after only less than a hundred timestepping iterations. Therefore, binomial checkpointing is unavoidable. It accounts for a significant part of this adjoint slowdown: since we chose to allow for 80 snapshots for binomial reversal of 6.8×10^5 time steps, the binomial model tells us this costs an average 3.9 extra recomputations per time step. This leaves us with roughly an 11-fold slowdown to account for, which is still higher than expected. There is certainly still room for improvement of our current checkpointing scheme. We still have to investigate the impact of the very deep call tree inside each time step, or the possibility to improve the time to write and read memory snapshots. Both the tangent and the adjoint codes produced by Tapenade received no further optimization other than compilation options. Nevertheless, with their present performance, the adjoint code is already preferable to the tangent code as soon as the number of input variables, with respect to which we request sensitivities, goes over 15.

7. CONCLUSIONS AND FUTURE WORK

A CFD code with a high-order spatial discretization based on spectral differences and an optimized parallel communications layer has been automatically differentiated by letting the AD tools (Tapenade and Adjoinable MPI) handle the communications layer in an automated way. In adjoint mode, the inversion of the communications during the backward sweep was found to produce correct code which could execute in 15 times the primal code execution time. The computational overhead is to a large extent the consequence of having to resort to binomial checkpointing to trade storage for computational time in order to invert a temporal integration loop with a number of iterations of the order of 10^6. We have presented a detailed outline of the code modifications required to achieve the correct differentiation of the parallel code.

AerospaceEurope
Conference2020
BORDEAUX, FRANCE | 25 - 28 FEBRUARY 2020

CEAS
Council of European
Aerospace Societies

HOSTED BY :
3AF
Association Aéronautique
et Astronautique de France

A viscous flow test case was treated. Even though it is a physically dissipative system, it has been computed using the backward mode of AD without running into stability issues, and yielding the correct derivative at the end of the computation. The run time for the tangent-differentiated code was 1.7 times slower than a single run of the primal. It is therefore readily superior to finite difference approximations even for single derivative computations. A natural further step would involve embedding the derivative solver into an optimal control loop. The strength of a code such as Jaguar lies in its ability to handle acoustics problems, such as the noise radiated by the wake of an object during a given time. Optimization in this type of problems constitutes an interesting perspective to the present work.

8. ACKNOWLEDGEMENTS

REFERENCES

1. Müller, J.D., Hückelheim, J., Mykhaskiv, O. (2018). STAMPS: a finite-volume solver framework for adjoint codes derived with source-transformation AD. In: 2018 Multidisciplinary Analysis and Optimization Conference, p2928.

2. Kenway, G., Mader, C., He, P., Martins, J. (2019). Effective adjoint approaches for computational fluid dynamics. *Prog. Aerosp. Sci.* **110**, p100542.

3. Xu, S., Radford, D., Meyer, M., Müller, J.D. (2015) Stabilisation of discrete steady adjoint solvers. *J. Comput. Phys.* **299**, pp75-195.

4. Charpentier, I. (2001). Checkpointing schemes for adjoint codes: application to the meteorological model Meso-NH. *SIAM J. Sci. Comput.* **22**(6), pp2135-2151.

5. Heimbach, P., Hill, C., Giering, R. (2005). An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Gener. Comput. Syst.* **21**(8), pp1356-1371.

6. Hückelheim, J., Hascoët, L., Müller, J.D. (2017). Algorithmic differentiation of code with multiple context-specific activities. *ACM Trans. Math. Softw.* **43**(4).

7. Utke, J., Hascoët, L., Heimbach, P., Hill, C., Hovland, P., Naumann, U. (2009). Toward Adjoinable MPI. In: 2009 IEEE International Symposium on Parallel Distributed Processing, pp1-8.

8. Witherden, F., Jameson, A. (2017). Future directions in computational fluid dynamics. In: 23rd AIAA Computational Fluid Dynamics Conference, p3791.

9. Cassagne, A., Boussuge, J.F., Villedieu, N., Puigt, G., D'ast, I., Genot, A. (2015). Jaguar: a new CFD code dedicated to massively parallel high-order LES computations on complex geometry. In: The 50th 3AF International Conference on Applied Aerodynamics (AERO 2015).

10. Brunet, V., Croner, E., Minot, A., De Laborderie, J., Lippinois, E., Richard, S., Boussuge, J.F., Dombard, J., Duchaine, F., Gicquel, L., Poinsot, T., Puigt, G., Staffelbach, G., Segui, L., Vermorel, O., Villedieu, N., Cagnone, J.S., Hillewaert, K., Rasquin, M., Lartigue, G., Moureau, V., Couaillier, V., Martin, E., de la Llave Plata, M., Le Gouez, J.M., Renac, F. (2018). Comparison of various CFD codes for LES simulations of turbomachinery: from inviscid vortex convection to multi-stage compressor. In: ASME Turbo Expo 2018: Turbomachinery Technical Conference and Exposition. Vol. 2C: Turbomachinery.

11. Hascoët, L., Pascual, V. (2013). The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Trans. Math. Softw.* **39**(3).

12. Griewank, A., Walther, A. (2008). Evaluating derivatives: principles and techniques of algorithmic differentiation, *Siam*.

13. Hascoët, L., Utke, J. (2016). Programming language features, usage patterns, and the efficiency of generated adjoint code. *Optim. Methods Softw.* **31**(5), pp885-903.

14. Walther, A., Griewank, A. (2004). Advantages of binomial checkpointing for memory-reduced adjoint calculations. In: Numerical mathematics and advanced applications, Springer, pp834-843.

15. Aupy, G., Herrmann, J., Hovland, P., Robert, Y. (2016). Optimal multistage algorithm for adjoint computation. *SIAM J. Sci. Comput.* **38**(3), pp232-255.

16. Hovland, P., Bischof, C. (1998). Automatic differentiation for message-passing parallel programs. In: Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, pp98-104.

17. Heimbach, P., Hill, C., Giering, R. (2002). Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver. In: Computational Science ICCS 2002, Springer, pp1019-1028.

18. Schanen, M., Naumann, U., Hascoët, L., Utke, J. (2010). Interpretative adjoints for numerical simulation codes using MPI. *Procedia Comput. Sci.* **1**(1), pp1825-1833.

19. Schanen, M., Naumann, U. (2012). A wish list for efficient adjoints of one-sided MPI communication. In: European MPI Users' Group Meeting, Springer, pp248-257.

20. Schanen, M., Förster, M., Naumann, U. (2010). Second-order algorithmic differentiation by source transformation of MPI code. In: European MPI Users' Group Meeting, Springer, pp257-264.

21. Towara, M., Schanen, M., Naumann, U. (2015). MPI-parallel discrete adjoint OpenFOAM. *Procedia Comput. Sci.* **51**, pp19-28.

22. Sagebaum, M., Albring, T., Gauger, N. (2017). High-performance derivative computations using CoDiPack. arXiv preprint arXiv:1709.07229.

23. Albring, T., Sagebaum, M., Gauger, N. (2016). Efficient aerodynamic design using the discrete adjoint method in SU2. In: 17th AIAA/ISSMO multidisciplinary analysis and optimization conference, p3518.

24. Hückelheim, J., Hovland, P., Strout, M., Müller, J.D. (2018). Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optim. Methods Softw.* **33**(4-6), pp672-693.

25. Berland, J., Bogey, C., Bailly, C. (2006). Low-dissipation and low-dispersion fourth-order Runge-Kutta algorithm. *Comput. & Fluids.* **35**(10), pp1459-1463.