Automatic parallel implementations of adjoint codes for structured mesh applications

Gábor Dániel Balogh
3in Research Group,
Faculty of Information Technology and Bionics
Pazmany Peter Catholic University
Esztergom, Hungary
Email: balogh.gabor.daniel@itk.ppke.hu

István Z. Reguly
Faculty of Information Technology and Bionics
Pazmany Peter Catholic University
Budapest, Hungary
Email: reguly.istvan@itk.ppke.hu

***Abstract*—Algorithmic Differentiation (AD) shown to be an essential tool to get sensitivity information in multiple areas of science such as Computational Fluid Dynamics (CFD) applications or finance. Yet there is no sufficient tool to ease the cost of providing performance portable AD codes, especially for modern hardware like GPU clusters. This paper sketches our plans and progress so far to extend the OPS framework with an adjoint tape (storage for descriptors of intermediate steps and intermediate states of variables) and shows preliminary performance results on CPU nodes. The OPS (Oxford Parallel library for Structured mesh solvers) has shown good performance and scaling on a wide range of HPC architectures. Our work aims to exploit the benefits of OPS to provide performance portable adjoint implementations for future structured mesh stencil applications using OPS with minimal modifications.**

***Keywords*-automatic differentiation; domain specific language; adjoint methods;**

I. Introduction

Algorithmic Differentiation is used to evaluate derivatives of the function which defined by a computer program.

Sensitivity or derivative information have a wide range of use such as design optimisation for CFD applications[1] or real time risk management[2]. Algorithmic Differentiation (AD) has shown to be an essential tool to get sensitivity information in such applications due to the high computational cost of calculating finite differences.

With AD we can get derivatives of a whole computer program with respect to some input variables. These derivatives than can be used for various optimisation tasks. However writing an AD code for a reasonably big application is a tiresome and error prone task which makes it infeasible to hand write for real life problems, and especially to maintain an efficient parallel implementation and update it for fast-changing new hardware. It has been known for a long time that AD is a powerful tool to use with CFD applications[3], [4], [5], [6], and there are multiple studies on their performance with automatic transformations [7]. Over the years there have been a number of tools developed to ease the cost of writing AD codes using high level descriptions of the application[8] or in case of C++ with templates and operator overloads [9]. The latter gives the flexibility of C++, but may lead to infeasible memory requirements due to huge "tapes". These overloading based methods lack the knowledge of the structure of the application, therefore cannot recognise big computational steps. However they can perform well with heterogeneous systems [9], [10] as well. The key method behind these tools is using the chain rule in a operation level. Basically the idea is that computers perform simple operations like addition and multiplication at the low level, and for these operations we can easily calculate the derivative. At the end, the overload based tools create a tape recording of each operation performed on all data. However the application itself have a coarse grained structure with the computational steps of the numerical method used.
With a tool like OPS (Oxford Parallel library for Structured mesh solvers) we define these computational steps as parallel loops and OPS is responsible for generating parallel implementation for the loops, from the generated loops we can naturally build a directed acyclic graph of computational steps for the application. This raises the idea of a coarse grained tape. If one can provide the derivative function for each loop body then we only need to traverse this coarse grained DAG to perform the necessary steps of the chain rule. Moreover OPS main goal is to ensure performance portable and future proof implementations with a code generation step on a range of hardware.

II. Algorithmic Differentiation

Let's assume we have an application with some input variables $x \in \mathbb{R}^n$ which perform a series of steps than compute the final result $y \in \mathbb{R}$.

$$x \xrightarrow{f_1(x)} x_1 \xrightarrow{f_2(x_1)} x_2 \cdots x_m \xrightarrow{f_{m+1}(x_m)} y \qquad (1)$$

We want to get $\frac{\partial y}{\partial x}$. Since $y = f_n(f_m(\cdots f_1(x)))$ we can apply the chain rule to get the formula:

$$\frac{\partial y}{\partial x} = \frac{\partial x_1}{\partial x}\frac{\partial x_2}{\partial x_1}\cdots\frac{\partial y}{\partial x_m} \tag{2}$$

There are two main modes of algorithmic differentiation based on from which direction we start to evaluate 2. The forward (tangent) mode requires more computation but does not require to actually require to create a DAG. The reverse (adjoint) mode on the other hand requires computation proportional of computing f itself, but requires to traverse the DAG in reverse order which significantly increase the memory usage, since to perform the steps we need to restore data to the correct state, so wee need to store the states in a so called tape. The main challenge is to efficiently save all intermediate state and the computational steps (and their derivative functions) while computing the result of the simulation and then managing the states of all data while propagating the derivatives from $\frac{\partial y}{\partial x_m}$ to the desired $\frac{\partial y}{\partial x}$. For more detailed information about AD the reader is referred to [11].

III. OPS

OPS is a domain specific language embedded in C++. It is used to describe a stencil application with a high level abstraction while delivering high performance for the applications. For the application developer it looks like simple traditional library, but OPS uses code generation to create parallel implementations for the applications. OPS supports OpenMP, OpenCL and CUDA and their combination with MPI.

There are four main parts of the abstraction behind OPS. Blocks: a collection of structured grid blocks. These have a dimensionality but no size. Datasets: data defined on blocks, with explicit size. Halos: description of the interface between datasets defined on different blocks. Computations: description of an elemental operation applied to grid points, accessing datasets on a given block. The principal assumption of the OPS abstraction is that the order in which elemental operations are applied to individual grid points during a computation may not change the results, within machine precision (OPS does not enforce bitwise reproducibility). This assumption lets OPS to parallelise the computations arbitrarily.

In OPS an algorithm is a sequence of computational loops (kernels) which are operating on a specified range on predefined grid while accessing datasets on the grid and grid invariant scalar values. This means in OPS the developer can define grids, global constants, datasets on a specified range on the grid and computations on these data.

The access of the datasets on the grids happens through stencils and for each loop the developer provides a descriptor for each dataset that describes the access of the loop to that dataset. This tells the backend which datasets will be written and which datasets are read only. Moreover from the stencils we can tell for each grid point (or iteration in the loop) which data from other grid points are accessed. This is an important information for the adjoint computations since in the reverse sweep the read data will be written which introduce race conditions.

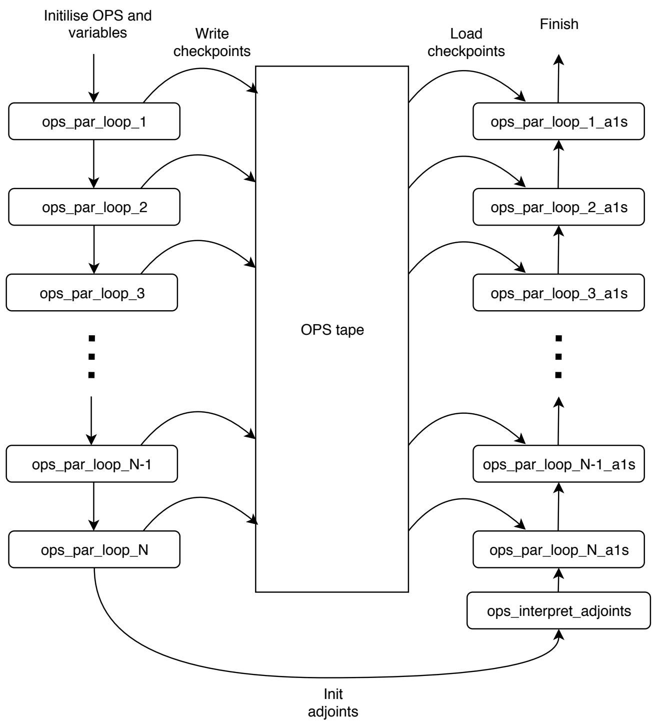

Figure 1. OPS checkpoint stores and loads during execution. After initialisation each ops_loop registers itself to the DAG and during the kernel execution it will write the checkpoints in OPS_tape. During the reverse run the kernels will load the checkpoints from the tape.

IV. Adjoint tape from computational steps

Our first goal was to extend OPS with a directed acyclic graph (DAG) of the application's execution with all the information about the computational loops required for performing the reverse run at runtime. To achieve this we used the kernel descriptors that are used to perform lazy execution in OPS [12] with the difference that we need to perform the adjoint version of the kernels. The scheme of the whole process is shown in Figure 1, after a setup phase each loop will automatically register itself to the DAG and save all written data to checkpoints, then the developer can initialise adjoints for the result and interpret adjoints, during which the adjoint for each loop will be executed in reverse order loading all the checkpoints stored during the forward run.

The DAG is built during the computation, each loop creates its own descriptor with a pointer to the kernel function and its own adjoint function and add it to the end

Grid Size	Mem. usage	With checkpointing		
iteration count	all	10	100	500
256 × 256	3.15 MB	256.9 MB	256.9 MB	771.5 MB
512 × 512	12.23 MB	268.2 MB	780.3 MB	2.76 GB
1024 × 1024	48.42 MB	304.4 MB	1.8 GB	8.8 GB

Table I
MEMORY USAGE OF THE POISSON APPLICATION. THE SECOND COLUMN CONTAINS THE MEMORY REQUIREMENTS OF THE PRIMAL COMPUTATIONS AND IN THE LAST THREE COLUMNS ARE SHOW THE INCREASED PEAK MEMORY USAGE WITH CHECKPOINTING.

of the DAG. During the code generation phase OPS tries to find an adjoint version of the loop body and generate a parallel implementation for the adjoint kernel as well. The big difference between the kernel generation and the adjoint versions is that the access patterns are reversed. This introduces race conditions that are not present in the original loop. To avoid race conditions we compute the maximum width of the stencils for the kernel and then divide the iteration range along the highest dimension to stripes in a way that stripes that are not adjacent to each other are independent. Using these stripes we can perform every second stripe parallel to each other and than have one synchronisation point and than perform all other stripes in a second run.

The other important aspect is that to perform these adjoint kernels we need to have the data in the original state from the forward run. There are two simple approaches; the first is to recompute all of the loops leading to the current loop restoring the state of all datasets or we can store copies of the datasets before we overwrite them. In our work we created checkpoint for each overwritten dataset which leads to significant memory overhead, but much faster than recomputing all data. The repetitive allocation of small data chunks has significant overhead, therefore we created a memory pool to the checkpoints. This checkpointing system allocates increasingly big chunks of memory when the current dataset doesn't fit in the previous pool.

V. RESULTS

We measured the memory requirements of the pools in case of one of OPS's own Poisson example application[13]. Since both the iteration count, the size of the grid and the application itself is quite small, the results int Table I illustrating the memory required to save all overwritten data, but the trend holds for bigger applications as well. Note that the measured memory overhead of checkpointing increasing in big steps due to our current memory pooling strategy.

Regarding runtime performance we measured the performance of a stochastic local volatility (SLV) model ported to ops (see [14]). SLV constitute state-of-the-art models to describe asset price processes, notably foreign exchange rates. We performed measurements on a single system, both

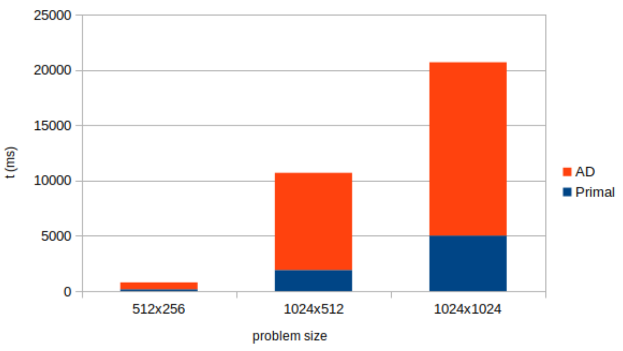

Figure 2. Runtime of the SLV code for different grid sizes. The forward run of the code noted with blue and the red parts stand for the backward run with the adjoint computations.

features will be implemented in the future. Batching will increase performance since the independent systems can be trivially parallelised during the backward run as well. In Figure 2 the runtime of this SLV implementation with OpenMP is shown. The overhead of the backward sweep is around 3-4 times of the time spent on the forward computation. This result means the the total cost of computing the derivatives of this SLV code is five times bigger than computing only the original function, which is a close to the state of the art techniques. However the current implementation has a significant memory overhead.

VI. CONCLUSION

The computation of derivative informations for CFD applications in finance is a well studied area. The gained sensitivity information has a wide range of use and the fast and efficient computation is necessary for applications. Although efficient tooling for supporting GPU clusters is lacking. The OPS framework is designed to automatically generate parallel implementations for different hardware for an application from its' high level code. Our aim is to use this design goal and extend the OPS framework wit an API and backend to support automatic differentiation for both CPU and GPU clusters. In this early stage we showed that OPS is flexible enough to support modifications that allow the us to calculate derivative informations for structured grid applications. We extended OPS with a method to create the DAG of the applications and perform checkpointing of overwritten data automatically. Compared to traditional OPS applications need to change to support adjoints but we found these modifications negligible compared to the cost of implementing an adjoint version for an application. We showed reasonable performance on a single node with OpenMP parallelisation of adjoint computations. However the current prototype suffers from high memory requirement due to checkpointing all overwritten data. Hence now we are looking into improve the checkpointing strategy in OPS with recomputing the state of the datasets from fewer checkpoints. Our experience with creating this prototype is promising regarding the future MPI and CUDA extensions.

Acknowledgment

This research has been carried out partly in the project Thematic Research Cooperation Establishing Innovative Informatic and Info-communication Solutions, which has been supported by the European Union and co-financed by the European Social Fund under grant number EFOP-3.6.2-16-2017-00013.
Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

References

[1] C. Othmer, "A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows," *International Journal for Numerical Methods in Fluids*, vol. 58, no. 8, pp. 861–877, 2008.

[2] L. Capriotti and J. Lee, "Case studies of real-time risk management via adjoint algorithmic differentiation (aad)," in *High-Performance Computing in Finance*. Chapman and Hall/CRC, 2018, pp. 339–370.

[3] C. Bischof, G. Corliss, L. Green, A. Griewank, K. Haigler, and P. Newman, "Automatic differentiation of advanced cfd codes for multidisciplinary design," *Computing Systems in Engineering*, vol. 3, no. 6, pp. 625–637, 1992.

[4] A. Carle, G. LL, B. CH, and N. PA, "Applications of automatic differentiation in cfd," 1994.

[5] M. Giles, D. Ghate, and M. C. Duta, "Using automatic differentiation for adjoint cfd code development," 2005.

[6] R. Sanchez, T. Albring, R. Palacios, N. Gauger, T. Economon, and J. Alonso, "Coupled adjoint-based sensitivities in large-displacement fluid-structure interaction using algorithmic differentiation," *International Journal for Numerical Methods in Engineering*, vol. 113, no. 7, pp. 1081–1107, 2018.

[7] J.-D. Müller and P. Cusdin, "On the performance of discrete adjoint cfd codes using automatic differentiation," *International journal for numerical methods in fluids*, vol. 47, no. 8-9, pp. 939–945, 2005.

[8] D. A. Fournier, H. J. Skaug, J. Ancheta, J. Ianelli, A. Magnusson, M. N. Maunder, A. Nielsen, and J. Sibert, "Ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models," *Optimization Methods and Software*, vol. 27, no. 2, pp. 233–249, 2012.

[9] U. Naumann and J. Toit, "Adjoint algorithmic differentiation tool support for typical numerical patterns in computational finance," *Journal of Computational Finance*, vol. 21, no. 4, 2018.

[10] J. Du Toit, J. Lotz, and U. Naumann, "Adjoint algorithmic differentiation of a gpu accelerated application," 2013.

[11] U. Naumann, *The art of differentiating computer programs: an introduction to algorithmic differentiation*. Siam, 2012, vol. 24.

[12] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop tiling in large-scale stencil codes at run-time with ops," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, 2017.

[13] "OPS github repository," https://github.com/OP-DSL/OPS.

[14] I. Z. Reguly, B. Moore, T. Schmielau, J. du Toit, and G. R. Mudalige, "Batch solution of small pdes with the ops dsl," in *International Conference on High Performance Computing*. Springer, 2019, pp. 124–141.