Parallel derivative computation using ADOL-C

Andreas Kowarz, Andrea Walther

{Andreas.Kowarz, Andrea.Walther}@tu-dresden.de
Institute of Scientific Computing, Technische Universität Dresden
01062 Dresden, Germany

Abstract: Derivative computation using Automatic Differentiation (AD) is often considered to operate purely serial. Performing the differentiation task in parallel may require the applied AD-tool to extract parallelization information from the user function, transform it, and apply this new strategy in the differentiation process. Furthermore, when using the reverse mode of AD, it must be ensured that no data races are introduced due to the reversed data access scheme. Considering an operator overloading based AD-tool, an additional challenge is to be met: Parallelization statements are typically not recognized. In this paper, we present and discuss the parallelization approach that we have integrated into ADOL-C, an operator overloading based AD-tool for the differentiation of C/C++ programs. The advantages of the approach are clarified by means of the parallel differentiation of a function that handles the time evolution of a 1D-quantum plasma.

1 Introduction

Automatic differentiation (AD) is a technique that has been developed and improved in the last decades. It allows to compute numerical derivative values within machine accuracy for a given function of basically unlimited complexity. Thus, unlike finite differences, no truncation errors must be taken into account. When calculating first order derivatives using the *forward* mode of AD, i.e., determining Jacobian-vector products, the computational effort is comparable to that of finite differences. This way, e.g., columns of the Jacobian can be computed efficiently. However, if a vector-Jacobian product is to be computed, e.g., a specific row of the Jacobian, the computational effort is proportional to the number of entries in the row when applying finite differences. The same task can be performed much more efficiently by use of the *reverse* mode of automatic differentiation. In particular, the computational effort is then independent of the rows dimension. A comprehensive introduction to AD can be found in [Gri00].

To provide reverse mode differentiation, AD tools based on operator overloading need to create an internal representation of the function $F : I\!R^n \rightarrow I\!R^m,\ y = F(x)$, to be differentiated, where $x = XI$ denotes the set of independent variables and $y = YD$ the set of dependent variables. The internal representation can be based on graphs [BS96] or sequential tapes. ADOL-C is an operator overloading based tool that provides automatic differentiation for functions given as C/C++ source code [GJU96]. The internal repre-

sentation of the considered function is created using a taping mechanism and a so-called *augmented* data type `adouble` that replaces `double` variables. Such a mechanism can be found in many AD-tools offering reverse mode differentiation based on operator overloading. Essentially, only the function value is computed within the overloaded operator or intrinsic function, respectively. In addition, information exactly describing the operation is recorded onto a tape. This information comprises the type of the operation/function, e.g. MULT, SIN, etc., as well as representations of the involved result and arguments, represented by so-called *locations*. After the evaluation of the function, the created tape represents the computational graph of the function as the sequence of operations that have been processed, in execution order. Based on this information, the program flow sequence can be easily inverted by interpreting the tape in reverse order.

Taking into account the steadily increasing demand for parallel program execution, an approach has to be found that allows to utilize the parallelization strategy of the provided user function for parallelizing the derivative computation. When ADOL-C is to be applied in a parallel environment that is created using OpenMP, several challenges have to be met. Firstly, the created tapes do not contain any information describing the parallel evaluation of the considered function. This is due to the fact that OpenMP statements cannot be overloaded. Furthermore, if all threads of the parallel environment would write onto the same tape, the serialization of the program flow would be inevitable, and, moreover, write conflicts would be very likely. Secondly, when computing derivative information applying the reverse mode of AD, the data access behavior is also reversed, i.e., read accessed function variables turn into write accessed derivative variables and vice versa. This potentially results in data access races.

In this paper, we present parallelization strategies that have been incorporated into the AD-tool ADOL-C. We mainly concentrate on the parallel reverse mode of AD but also give information on the new features that allow a parallel tape based forward mode. In the following section, a short overview of the facilities for generating a parallel AD-environment inside ADOL-C is given. Special issues for parallel reverse mode differentiation are discussed in Section 3 whereas Section 4 is dedicated to a numerical example which demonstrates the runtime advantages that can be achieved by applying the parallel reverse mode. A short summary and an outlook complete this paper.

2 Extensions to the ADOL-C

ADOL-C has been developed over a long period of time under strict sequential aspects. Although the generated tapes have been used for a more detailed analysis and the construction of parallel derivative code, e.g., [Bis91], ADOL-C could hardly be applied out of the box within a parallel environment, so far. The most convenient chance in this context is given by the use of ADOL-C in a message passing system based on distinct processes for all cooperators. This requirement is fulfilled by, e.g., MPI [HW99]. Due to separated address space and the realization of cooperators as processes of the operating system, the ADOL-C environment is multiplied. In particular, all control variables used in ADOL-C are available within each process exclusively. From the users point of view, only two

conditions must be met to allow a successful application of ADOL-C. First, the uniqueness of the created tape name, also called tag, must be ensured. This can be achieved by carefully choosing the tag, e.g., in dependence of the process' rank. Second, it must be considered that data transfer between the working processes is not reflected by the internal representation created by ADOL-C. Then, ADOL-C may be applied as usual allowing the computation of derivatives for functions that implement data partitioning techniques, especially when only a limited degree of communication is necessary.

Many parallel applications rely on a high amount of synchronization to communicate computed information at given points among involved cooperators. In a message passing environment this would also mean to invoke more or less expensive transfer routines. Therefore, such applications are typically parallelized for a shared memory environment using OpenMP [DM98]. Extensive enhancements have been added to ADOL-C to allow the application in such cases. Originally, the location of an augmented variable is assigned during its construction utilizing a specific counter. Creating several variables in parallel results in the possibility to loose the correctness of the computed results due to a data race in this counter. Initial tests based on the protection of the creation process by use of critical sections showed unambiguous behavior. Even when using only two threads in the parallel program, the runtime increased by a factor of roughly two rather than being decreased. For this reason, a separate copy of the complete ADOL-C environment is provided for every worker thread. The copy mechanism can be implemented using two different ways, either based on the OpenMP `threadprivate` clause, or by utilizing the thread number. Possible effects of the chosen strategy are discussed in Section 4.

Besides this decision, another issue had to be answered. As already identified by G. M. Amdahl [Amd67], every parallel program possesses a certain fraction that can only be handled serially. In many situations not only the parallel part of the function is object to the derivation efforts but also the serial parts. This entails the question of how to transfer information between the serial and parallel program segments and vice versa.

From serial to parallel
: Data transfer is this direction can be performed quit easily. For all variables alive at the moment when the parallel region starts, a copy may be created for each thread.

From parallel to serial
: This is the more difficult direction as it requires to decide which values from which thread should be copied to the serial part. Furthermore, the handling of variables created within the parallel part must be solved.

For the current implementation, the following decisions have been made.

- The handling of parallel regions by ADOL-C comprises only augmented variables but not user variables of standard data type.
- Control structures utilized by ADOL-C are duplicated for each thread, and are default initialized during the first creation of a parallel region. The values of these control variables are then handed on from parallel region to parallel region.

- For performance reasons, two possibilities of handling the tape information have been implemented. In the first case, control information including the values of augmented variables are transferred from the serial to the parallel variables every time a parallel region is created. Otherwise, this process is invoked only during the creation of the first parallel region. In either case, the master thread creates a parallel copy of the variables for its own use.
- No variables are copied back from parallel to serial variables after completion of a parallel region. This means, results to be preserved must be transferred using variables of standard data type.
- The creation or destruction of a parallel region is not represented within the initiating tape. Coupling of serial and parallel tapes must therefore be arranged explicitly by using the construct of external differentiated functions, see [Kow08].
- Different tapes are used within serial and parallel regions. Tapes begun within a specific region, no matter if serial or parallel, may be continued within the following region of the same type.
- Tapes created during a serial region can only be evaluated within a serial region. Accordingly, tapes written during a parallel region must be evaluated there.
- Nested parallel regions are not supported and remain object to later enhancements.

All in all, the described facts result in augmented variables with a special property that depends on the specific handling of the tape information. With the start of a new parallel region either a *threadprivate* or a *firstprivate* behavior, respectively, is simulated, [DM98]. This means that the value of the augmented variable is taken either from the previous parallel region or from the serial region, respectively. In either case, the value used within the parallel region is invisible from within the serial region.

Initializing the OpenMP-parallel regions for ADOL-C is only a matter of adding a macro to the outermost OpenMP statement. Two versions of the macro are available, which are only different in the way the tape information is handled. Using `ADOLC_OPENMP`, this information including the values of the augmented variables is always transferred from the serial to the parallel region. In the other case, i.e., using `ADOLC_OPENMP_NC`, this transfer is performed only the first time a parallel region is entered. This reduces the copy overhead for iterative processes. Due to the inserted macro, the OpenMP statement has the following structure:

```
#pragma omp ...  ADOLC_OPENMP
                 or
#pragma omp ...  ADOLC_OPENMP_NC
```

Within a parallel region, different tapes are created by the threads. Succeeding the taping phase, derivatives are computed using the various tapes. This can be done either by complete user steering, or semi-automatic by applying the concept of extern differentiated functions.

While forward mode differentiation is, in a way, straightforward, the computation of derivatives utilizing the reverse mode of AD needs special attention.

3 Reverse mode for data parallelism

For parallel reverse mode differentiation, we focus on functions that feature a special type of data parallelism. Basically, data parallelism describes the subdivision of the data domain of a given problem into several regions. These regions are then assigned to a given number of processing elements, which apply the same tasks to each of them. Data parallelism is commonly exploited in many scientific and industrial applications and exhibits a "natural" form of scalability. Since the problem size for such applications is normally expressed by the size of the input data to be processed, an upscaled problem can typically be solved using a correspondingly higher number of processing elements at only a modestly higher runtime [DFF+03].

For our purpose, data parallelism is extended beyond the pure nature described above. Accordingly, it shall be allowed that the complete set of independent variables XI of the given function F may be used by all p processing elements PE_i, $i = 1, \ldots, p$, for reading. Denoting by $XI_r^{(i)}$ and $XI_w^{(i)}$ the read and write accessed subsets of XI, respectively, for the various processing elements, one has that

$$\forall PE_i : \quad XI_r^{(i)} \subseteq XI, \qquad XI_w^{(i)} = \emptyset. \tag{1}$$

This allows for example parallel functions that handle the evolution of systems consisting of many components. There, computations for the individual components can be performed independently, provided that the interaction among them can be determined using XI. Derivative information for such applications can be provided using the scheme depicted in Figure 1. In contrast to the general read access in terms of the independents,

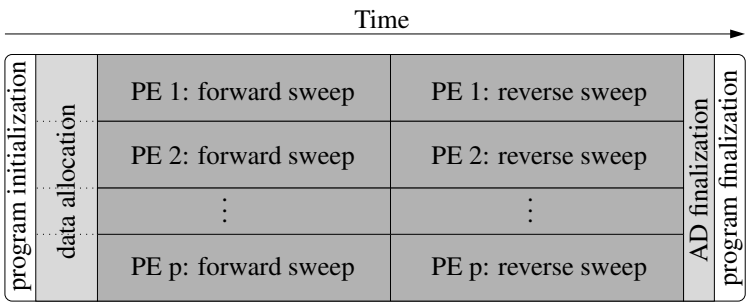

Figure 1: Basic layout of data-parallel calculations in an AD-environment

write access may only be allowed for distinct subsets $YD^{(i)}$ of the dependent variables. For a given number p of processing elements it is required that

$$YD = \bigcup_{i=1}^{p} YD^{(i)} \qquad \text{and} \qquad YD^{(i)} \cap YD^{(j)} = \emptyset \qquad \text{with} \quad i, j \in [0, p],\ i \neq j.$$

The set of intermediate variables $IV^{(s)}$ associated with the serial function evaluation is considered to be reproduced for all processing elements yielding p sets of variables $IV^{(i)}$, $i = 1, \ldots, p$. In this way, intermediate values can be used at the certain processing element without the potential of memory conflicts. Overall, considering the subset $YD^{(i)}$ and the set of intermediate variables $IV^{(i)}$, which are used exclusively by the processing elements PE_i, it holds that

$$\forall PE_i \, \forall PE_j : \begin{cases} YD^{(i)} \cap YD^{(j)} = \emptyset & IV^{(i)} \cap IV^{(j)} = \emptyset \quad \text{for} \quad i \neq j \\ YD^{(i)} = YD^{(j)} & IV^{(i)} = IV^{(j)} \quad \text{for} \quad i = j \end{cases} . \tag{2}$$

Any function exhibiting the properties (1) and (2) is considered correctly parallelized in the sense that data races are debarred.

Obviously, equation (2) allows to compute derivatives for the given function as long as only the sets $IV^{(i)}$ and $YD^{(i)}$ are involved. Due to the distinct nature of the sets defined for the processing elements, forward and reverse mode of AD can be applied safely. A less obvious situation is given as soon as independent variables are involved in the computation. As known from the theory of the reverse mode, read accessed function variables result in write accessed derivative variables. More precisely, the following relation holds

$$\begin{array}{ll} \underline{\text{Function}} & \underline{\text{Derivative (reverse mode)}} \\ v_i = \varphi_i(v_j)_{j \prec i} & \bar{v}_j \mathrel{+}= \bar{v}_i * \frac{\partial \varphi_i(v_j)_{j \prec i}}{\partial v_j} \qquad \forall j \in \{j : j \prec i\}. \end{array}$$

Therein, the term $\varphi_i(v_j)_{j \prec i}$ denotes an elemental operation or intrinsic function to compute an intermediate value v_i, and $\prec$ refers to the dependency relation as defined in [Gri00]. As can be seen, due to the required instructions in the reverse mode, the data access layout of the function variables is reversed for the adjoint variables $\bar{v}_i$ and $\bar{v}_j$ associated with each intermediate variable v_i and v_j, respectively. Hence, read accesses on the independent variables x_k, $k = 1, \ldots, n$, induce the potential of data races in the adjoint computations.

However, similar to the handling of intermediate variables, different sets of adjoint variables $\overline{XI}^{(i)}$ can be provided for each processing element corresponding to the set XI. Adjoint values may then be updated locally by each processing element independently and thus globally in parallel. Due to the additive nature of the derivative computations, global adjoint values may later be assembled using the local information produced by the various processing elements. As this assembling results in significant computational effort, it should be executed for all relevant adjoints only once and in a single step. Thus, the assembling step must be performed after the last update of an adjoint variable $\bar{x}_k^{(i)} \in \overline{XI}^{(i)}$. However, updates of adjoints $\bar{x}_k^{(i)}$ are principally possible at any point of the reverse computations. This results from the property of the function that independent variables may be accessed at any given time of the function evaluation. Thus, a single adjoint assembling step is only possible if no $\bar{x}_k^{(i)}$ is used as an argument of a derivative instruction before all updates on the set $\overline{XI}^{(i)}$ have been performed. For the considered type of applications that feature the properties (1) and (2), this potential conflict can never occur. Since the individual independent variables are accessed for reading only, they cannot appear on the

right-hand side of an derivative instruction. Hence the assembling phase that computes the global derivative values can be safely moved to the end of the derivation process.

4 Numerical Example

The example that is used to demonstrate the parallel derivation of a given function is taken from physics. Due to its complex structure it is also necessary to apply in addition to the parallelization other techniques derived in [Kow08], in particular, nested taping, external differentiated functions, and checkpointing facilities. Only the combination of these techniques allows the derivation based on the reverse mode of AD for this example. The implementation of the function was performed by N. Gürtler [Gür06], and the differentiation was realized in a cooperation between the RWTH Aachen and the TU Dresden. To our knowledge, the parallel derivation of the given function including the coping with the high internal complexity and inherent challenges currently features uniqueness and establishes a new level of the application of operator overloading based AD.

The example describes the time propagation of a 1D-quantum plasma. Therein, the plasma particles can be represented by a N-particle wave function $\Psi(1,\ldots,N)$, and the system is modeled by multi-particle Schroedinger equations. For reduction of the complexity, spin effects are neglected. However, a direct solution is numerically highly expensive. Since an approximation is often sufficient to describe the physical behavior, the simulation is based on Quantum-Vlasov equations. Interchange and correlation effects are neglected. As an entry point into quantum plasma simulations and for reasons of complexity only the one-dimensional case is modeled. However, due to the necessary discretization in the order of N dimensions, the direct solution is still very expensive. For the analysis of expected values for many distributions, calculations based on a representative ensemble of quantum states are sufficient. Prior to the numerical simulation, a discretization of the resulting equation system is performed. Applying cyclic boundary conditions yields the sparse cyclic tridiagonal system (3).

$$U_i^{+,n+1}\Psi_i^{n+1} = U_i^{-,n}\Psi_i^n \tag{3}$$

For details on the discretization and the definition of the operator U, the reader is referred to [Gür06, Gür07]. With a system like (3), a complete description of the time propagation of the discretized wave function Ψ is given. Therein, the term Ψ_i^n denotes the wave function of the ith particle at time n.

The final step, which succeeds the time propagation of the plasma is used to compute the expected value $<\eta>$ of the particle density. The discrete version of this target functions is given by

$$<\eta> = \sum_{i=1}^{N}\sum_{j=1}^{K} z(j)\Delta z|\Psi_{i,j}|^2 \ . \tag{4}$$

The reduction of the high amount of output information resulting from the time propagation to a single value allows an easier evaluation of the entire system.

A code for simulating the time propagation of a one-dimensional ideal quantum plasma has been developed [Gür06]. It creates the source of the differentiation efforts and features the program layout that is depicted in Figure 2. There, all parallelization statements are al-

```
...
startup_calculation(..);
for (n = 0; n < T; ++n) {
   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < N; ++i)
         part1(..);
      #pragma omp for
      for (i = 0; i < N; ++i)
         part2(..);
   }
}
target_function(..);
...
```

Figure 2: Basic layout of the plasma code including parallelization statements

ready included. The representation is based on C++ notation, and the OpenMP statements are adjusted accordingly. Due to the layout of the overall function, the reverse mode of AD is to be preferred for the differentiation of the code.

For the proof of concept of the code as well as its derivation, we used the following parameters for all runtime measurements that are discussed in this section.

- number of wave functions $N = 24$, simulation time $t = 30$, $T = 40000$ time steps
- length of the simulation interval $L = 200$, discretized with $K = 10000$ steps
- plasma frequency $\omega_P = 1.23$

All units are transformed to the atomic scale, see [Gür06]. The most important runtime reduction in the simulation results from computing only the first 50 of the 40000 time steps. After this period, the correctness of the derivatives can already be validated and the characteristics of the runtime behavior that are of special interest are already fully visible. To preserve the numerical stability of the code, the time discretization is based on $T = 40000$ steps nevertheless. All runtime measurements have been performed using the *SGI ALTIX 4700* system installed at the TU Dresden.

Two versions of parallel derivative computations are discussed in the remainder of this section. They apply the same parallelization approach, i.e., every participating thread of

the parallel environment performs all calculations for a specific number of the considered N wave functions. The main difference is to be found in the way the data access in the parallel environment is performed. Figure 3 depicts the speedups measured for the derivation of 24 wave functions. As can be seen, the code version that is based on

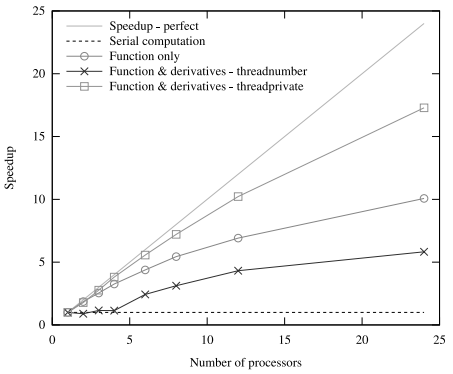

Figure 3: Speedups for the parallel differentiation of the plasma code for $N = 24$ wave functions

a `threadprivate` parallelized AD-environment achieves the highest speedups. Furthermore, it performs much better than thread number based reverse version, and it even outperforms the speedup of the original functions. This allows the conclusion that, carefully implemented, the derivative calculations can decrease the distracting effect of the necessary synchronization within the parallel environment.

The results attained through the parallelization of the differentiation of the plasma code clarify that operator overloading based AD is prepared to meet the challenges that are brought up by the most complex codes applied in science and engineering. Thus, for the parallelization of derivative calculations using operator overloading AD, an answer has been found to a question that is still open for many other applications.

5 Summary & outlook

Automatic differentiation based on operator overloading features a long history and has shown to be highly valuable for most derivation tasks. Especially for programming languages for which AD-enabled compilers are not available or miss a critical feature, the operator overloading based approach often presents the only reasonable technique. The increasing complexity of the investigated functions more and more requires the application of parallelization techniques. It is obvious that automatic differentiation must face this fact and provide adequate differentiation strategies. In this paper, we presented new parallelization approaches that have been incorporated into the tool ADOL-C. By means of the time propagation of a 1D quantum plasma we could show that parallel reverse mode differentiation can be performed efficiently using operator overloading based AD.

The main limitation of AD utilizing the overloading facilities of programming languages is always to be found in the size of created internal function representation. Within this context, the unrolling of loops presents a major drawback that may result in an unfavorable runtime behavior. Hence, one of the main challenges to be answered in the future is the development of techniques that allow a much more compact representation of loops. This not only avoids the expensive storing of every loop iteration, but also presents a major step towards an automatic parallel differentiation of parallelized user functions.

References

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. volume 30 of *AFIPS conference proceedings*, pages 483–485, National Press Building, Washington, D.C. 20004, USA, 1967. Thompson Book Co. Spring joint computer conference, Atlantic City.

[Bis91] C. H. Bischof. Issues in Parallel Automatic Differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 100–113. SIAM, Philadelphia, PA, 1991.

[BS96] C. Bendtsen and O. Stauning. FADBAD, a Flexible C++ Package for Automatic Differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

[DFF+03] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *Sourcebook of Parallel Computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 05(1):46–55, 1998.

[GJU96] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.

[Gri00] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Applied Mathematics. SIAM, Philadelphia, PA, 2000.

[Gür06] N. Gürtler. Simulation eines eindimensionalen idealen Quantenplasmas auf Parallelrechnern, 2006. Diploma thesis in physics, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany.

[Gür07] N. Gürtler. Parallel Automatic Differentiation of a Quantum Plasma Code, 2007. Diploma thesis in computer science, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, Germany.

[HW99] R. Hempel and D. W. Walker. The Emergence of the MPI Message Passing Standard for Parallel Computing. *Computer Standards & Interfaces*, 21:51–62, 1999.

[Kow08] A. Kowarz. *Advanced Concepts of Automatic Differentiation based on Operator Overloading*. PhD thesis, TU Dresden, 2008. to appear.