Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver

Journal Title
XX(X):2–30
©The Author(s) 2015
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/

Jan Hückelheim[1], Paul Hovland[2], Michelle Mills Strout[3] and Jens-Dominik Müller[1]

Prepared using sagej.cls

Abstract

Reverse-mode algorithmic differentiation is an established method for obtaining adjoint derivatives of computer simulation applications. In computational fluid dynamics (CFD), adjoint derivatives of a cost function output such as drag or lift with respect to design parameters such as surface coordinates or geometry control points are a key ingredient for shape optimisation, uncertainty quantification, and flow control. The computational cost of CFD applications and their derivatives makes it essential to use high-performance computing hardware efficiently, including multi- and many-core architectures. Nevertheless, OpenMP is not supported in most algorithmic differentiation tools, and previously shown methods achieve poor scalability of the derivative code.

We present the algorithmic differentiation of an OpenMP-parallelised finite volume compressible flow solver for unstructured meshes. Our approach enables us to reuse the parallelisation of the original code in the computation of adjoint derivatives. The method works by identifying code segments that can be differentiated in reverse-mode without changing their memory access pattern. The OpenMP parallelisation is integrated into the derivative code during the build process in a way that is robust to modifications of the original code and independent of the OpenMP support of the differentiation tool.

We show the scalability of our adjoint CFD solver on test cases ranging from thousands to millions of finite volume mesh cells on CPUs with up to 16 threads, as well as on an Intel XeonPhi card with 236 threads. We demonstrate that our approach is more practical to implement for production-size CFD codes, and produces more efficient adjoint derivative code than previously shown algorithmic differentiation methods.

Keywords

Algorithmic differentiation, OpenMP, finite volume, Unstructured mesh, Adjoint method

[1]Queen Mary University of London, London, UK
[2]Argonne National Laboratory, Lemont, IL, USA
[3]University of Arizona, Tucson, AZ, USA

Corresponding author:
Jan Hückelheim, Queen Mary University of London, Mile End Road, London, E1 4NS, UK.
Email: j.c.hueckelheim@qmul.ac.uk

1 Introduction

Computer simulations are used in industrial applications to augment or replace experiments that are expensive or time-consuming to conduct. A simulation can be used to compute an objective function $J \in \mathbb{R}^m$ such as fuel consumption or noise, based on the design of a system $\alpha \in \mathbb{R}^n$ consisting of e.g. shape or operating conditions. The design can be modified and the objective function recomputed in an iterative process to find a design that minimises the objective function output. J can often be expressed as a scalar or vector with few entries. In contrast, α often consists of millions of parameters.

For gradient-based optimisation methods or as a guidance for a human designer, it is desirable to obtain the Jacobian $\frac{\mathrm{d}J}{\mathrm{d}\alpha}$ that expresses the sensitivity of the objective function to design changes. The adjoint method provides a way to compute $\frac{\mathrm{d}J}{\mathrm{d}\alpha}$ at a relative cost that is independent of the number of design parameters, which makes the method feasible for industrial applications (1) where $m \ll n$. Given a computer program that implements $y := J(\alpha)$ with runtime T, the corresponding adjoint program $\bar{J}$ computes the product of the transpose Jacobian matrix with a vector $\bar{\mathbf{y}} \in \mathbb{R}^m$ as

$$\bar{\mathbf{x}} = \left(\frac{\mathrm{d}J}{\mathrm{d}\alpha}\right)^T \cdot \bar{\mathbf{y}},$$

which can be used to assemble $\frac{\mathrm{d}J}{\mathrm{d}\alpha}$ by calling $\bar{J}$ repeatedly, using each of the m unit vectors in $\mathbb{R}^m$ as inputs. The runtime complexity of evaluating J and $\bar{J}$ is the same, and so the time required to assemble the full Jacobian is $\mathcal{O}(m \cdot T)$.

If J is based on a physical model that is discretised in time and/or space, at least two separate paths exist for creating the adjoint program $\bar{J}$. This situation applies in the case of computational fluid dynamics (CFD).

One may mathematically derive an adjoint model based on the physical primal model. The adjoint model is then discretised and implemented in software. This is commonly referred to as the *continuous adjoint* approach (2), which has met with some success. A key advantage of this approach is that the same parallelisation strategy can generally be used for the primal and adjoint code (1). For a typical edge-based finite volume CFD solver on an unstructured mesh as covered extensively in the literature (3), a parallelisation strategy based on mesh colouring can be used for both the primal and adjoint solver, as illustrated in Figure 1.

However, the differentiation of the primal model, as well as the following discretisation of the adjoint model and its implementation, is a manual effort and hence error-prone. Aspects such as the correct handling of boundary conditions

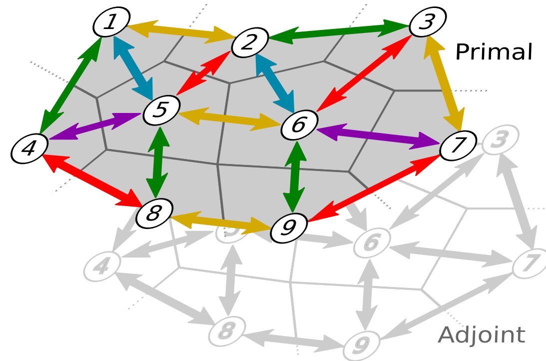

Figure 1. Control volumes and the dual graph that is used for an edge-based finite volume solver. The same spatial discretisation is typically used for both primal and adjoint computation. Any parallel schedule that is valid for the primal solver should therefore be applicable to the adjoint solver as well. We exploit this fact to automatically generate an OpenMP-parallel adjoint solver.

or turbulence models are subject to ongoing research. Since the adjoint system is discretised separately, the adjoint derivatives computed in the conventional continuous adjoint approach reflect the derivatives of the primal computation only in the limit of infinitely fine meshes. In addition, it is hard to keep the primal and adjoint codes consistent with each other in the presence of updates and bug fixes.

This situation makes discrete adjoint programs generated by algorithmic differentiation (AD) attractive (4). Based on the source code of a primal program, an AD tool can generate a program that accumulates the partial derivatives of the primal program's individual operations. In reverse-mode AD, the accumulation is performed in reverse order, starting from the program output, and tracing backwards through all operations that influence the output. The resulting *adjoint program* can be used to compute the derivative of the primal output with respect to all primal inputs, without requiring knowledge of the application domain or the underlying physical model.

The AD process can be incorporated into the automated build system of the program (5). Hence, updates to the primal implementation such as new turbulence models or boundary conditions and bug fixes are automatically reflected in the derivative computation. The runtime of the automatically generated code is typically within an order of magnitude of the primal runtime for serial cases and has been reported to be as low as 1.3 times the primal runtime (6).

Reverse-mode AD for programs using message passing parallelisation has been discussed previously (7; 8; 9; 10). While open problems remain, adjoint MPI is well enough understood to be used in practice.

In contrast, the differentiation of shared memory parallel code is not yet in widespread use. This situation is in spite of the ongoing trend to increasing numbers of CPU cores in each compute node (11). An established way to implement shared memory parallelisation is the OpenMP standard (12; 13), which recently also has been advancing into the domain of accelerators and many-core architectures (14).

In this work, we present the application of the source-transformation AD tool Tapenade (15) to the compressible finite volume flow solver `mgopt` (5). The approach, which we will refer to as SSMP for *self-adjoint shared memory parallelisation*, preserves the existing parallelisation strategy of the primal code when generating the adjoint code. The adjoint solver is equipped with OpenMP pragmas during the build process. The method does not rely on the OpenMP support of an AD tool and should therefore work independently of the tool used and is applicable to a whole class of parallel loops that are commonly found in finite volume solvers.

We make the following contributions in this paper:

1. We identify a class of primal programs for which source transformation adjoint OpenMP can be generated that mimics the parallelisation of the primal code.
2. We show an approach to instrument the adjoint code with OpenMP pragmas during the build process that works independently of the source-transformation AD tool being used, along with a correctness proof.
3. We present an adjoint CFD solver with OpenMP parallelism along with scalability measurements on CPU and Intel XeonPhi for a series of test cases.

The paper is organised as follows: Section 3 discusses the role of reverse-mode AD in adjoint CFD computations. We describe the parallelisation of the primal code in Section 4 and that of the adjoint code in Section 5. Test cases in Section 6 and timing results in Section 7 lead to a conclusion in Section 8.

2 Related work and background

Other approaches to developing parallel adjoint CFD solvers have been shown in the literature and are outlined in this section. None of them allow one to automatically transform a shared memory parallel scalable primal solver into an equivalently scalable adjoint solver. At the end of this section we review the conventional reverse-mode differentiation of OpenMP-parallel code shown in previous work.

Adjoint solvers have been hand-implemented following the continuous adjoint approach, for example as part of the solver framework OpenFoam (16). Continous adjoint solvers naturally inherit the parallelisation of the primal code, but the computed

derivatives suffer from the aforementioned consistency problems and significant manual implementation effort is required.

The authors of (17) show automatic generation of parallel primal and adjoint solvers. While their approach leads to derivatives that are consistent with the primal code, it is limited to finite element discretisations and does not work with existing legacy code.

Efforts to develop production-sized parallel discrete adjoint solvers have focused on MPI (18) and have not taken full advantage of the shared memory capabilities of modern hardware. Methods for MPI cannot be easily applied to OpenMP, since communication between threads with shared memory is not explicitly stated in the source code, and in some cases the presence of communication cannot be decided at compile time. This makes it hard to ensure the absence of data races in the adjoint code.

Previous work on differentiating OpenMP presents the parallelisation of vector-mode AD to compute multiple directional derivatives in parallel (19; 20; 21). Other work covering the differentiation of already parallel primal programs depends on the static analysis of the adjoint communication pattern (22; 23; 24), which rarely works for non-trivial programs or relies on critical or reduction pragmas (25) resulting in poor scalability. We summarise the solutions in the literature previous to this work:

Critical/atomic sections All write access to shared variables is safeguarded with critical sections or declared atomic. While this will guarantee a correct result, it comes with a considerable performance overhead. We observed this in our test cases shown in Section 7.

Reduction Some parallel loops can be expressed as a reduction into an array using the OpenMP `reduction` pragma. Due to the runtime and memory footprint of this approach, it is only an option for small arrays and scalar variables. In CFD applications, arrays are often large enough to occupy a significant fraction of the available memory.

Detection Detecting the presence of write conflicts at compile time is discussed in (23), while schemes to reschedule loops at runtime are studied e.g. in (26). Compile-time solutions are limited to simple cases and in particular, they cannot work for unstructured meshes where the schedule depends on the program input.

Automatic parallelisation Techniques for parallelising irregular computations exist (27; 28), but focus on cases where full parallelisation can be found. Multi-colouring parallelisations have not been automated as of yet.

In contrast to previous work, we do not attempt to prove the absence of write conflicts in shared memory or resort to critical sections to safeguard potential write conflicts.

Instead, we identify code segments for which the data dependencies of the primal and adjoint code have a one-to-one correspondence, and hence the same parallelisation strategy can be used for the adjoint code. With our approach, the correctness of the parallel adjoint code depends only on the correctness of the primal code.

3 Primal and adjoint CFD

Computational fluid dynamics (CFD) is commonly used in automotive, aerospace and many other industries to predict the behaviour of flow inside and around vehicles and other equipment. The governing equations that describe the behaviour of flow at a steady state can be discretised in space. We denote as α the vector of design parameters and as $\mathbf{u}$ the variables that describe the state of the flow, such as density and momentum. This system can be written as

$$F(\mathbf{u}, \alpha) = 0. \tag{1}$$

The residual function F is driven to zero by iteratively updating the discrete state $\mathbf{u}$

$$\mathbf{u}_{n+1} \leftarrow \mathbf{u}_n - P(\mathbf{u}_n, \alpha)^{-1} \cdot F(\mathbf{u}_n, \alpha),$$

where $P(\mathbf{u}_n, \alpha)$ is a preconditioning matrix that depends on the exact scheme being used. In each iteration, the function F that computes the residual, and the preconditioning matrix P need to be updated. If the scheme converges, we find that

$$\lim_{n\to\infty} \mathbf{u}_n = \mathbf{u}^*,$$

although in practice the iterative process is stopped after a finite number of steps as soon as the result is sufficiently accurate.

The adjoint method can help choose the design parameters α to minimise an objective function $J(\mathbf{u}, \alpha)$ that defines the quality of a given design based on a function of the design itself and the resulting flow conditions, for example pressure loss, noise, or average lift or drag over a certain time span. Given a program that computes both the flow field and the objective function, one can directly apply an AD tool to it and obtain $\frac{\mathrm{d}J}{\mathrm{d}\alpha}$. However, it is common practice and often more efficient to use AD only for certain parts of the program and to assemble the total derivative manually. To this end,

one can express the sensitivity of the objective function with respect to the design as

$$\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\partial J}{\partial \alpha} - \bar{\mathbf{u}}^T \frac{\partial F}{\partial \alpha},$$

where $\bar{\mathbf{u}}$ is the solution to the adjoint system

$$\left(\frac{\partial J}{\partial u}\right)^T - \left(\frac{\partial F}{\partial u}\right)^T \bar{\mathbf{u}} = 0.$$

This can be solved iteratively (29; 30) as

$$\bar{\mathbf{u}}_{n+1} \leftarrow \bar{\mathbf{u}}_n + P(\mathbf{u}^*, \alpha)^{-T} \cdot \left(\left(\frac{\partial J}{\partial u}\right)^T - \left(\frac{\partial F}{\partial u}\right)^T \cdot \bar{\mathbf{u}}_n \right),$$

using the temporal discretisation and transposed preconditioner of the primal program.

AD tools such as Tapenade (15), ADOL-C (31), Taf (25) or OpenAD (32) can generate code that computes the derivatives in these equations. The cost of repeatedly evaluating $F(u, \alpha)$ is a major contributor to the overall cost of the primal CFD solver. Likewise, the cost of evaluating $\left(\frac{\partial F}{\partial u}\right)^T \cdot \bar{\mathbf{u}}_n$ in each iteration is the main computational expense for an adjoint solver. This is enhanced by the fact that the adjoint system is linear and the preconditioning matrix P^T depends only on the primal solution and hence needs to be computed only once, while the adjoint residual is updated in each iteration. In the primal solver, P and F are recomputed in every iteration.

In addition, other parts of the code such as I/O, time stepping with Runge-Kutta or multigrid preconditioning (33) or linear system solvers can be reused in the adjoint solver. The evaluation of the objective function J typically does not require expensive operations such as linear system solves, and hence neither the evaluation of J nor $\left(\frac{\partial J}{\partial u}\right)^T$ contributes significantly to the overall runtime. Therefore, evaluating $\bar{F}$ often causes the only significant computational expense of algorithmically differentiated code in an adjoint solver. For `mgopt`, typical runtime contributions for F, P, and their adjoints are shown in Table 1.

4 Primal parallelisation strategy

The primal residual function F shown in Equation (1) is an example of an edge-based flux residual computation as can be found, for example, in (3; 34). It can be expressed as a loop over edges in a graph where information is exchanged between nodes connected by an edge, as expressed in Algorithm 1. The connectivity is stored in $edge2nodes(i)$, which is a mapping from the edge i to the nodes l, r that are connected by i.

Table 1. Percentage of runtime contributed by primal residual and preconditioning to overall primal runtime (**Column 3-5**), and percentage of runtime contributed by adjoint residual and preconditioning to overall adjoint runtime (**Column 6-8**). First-order cases run with inviscid flow, second-order cases run with the Spalart Allmaras turbulence model and Green-Gauss gradients, with explicit local timestepping or a Block Jacobi preconditioner.

Stepping	Order	F	P	other	$\left(\frac{\partial J}{\partial u}\right)^T$	P^T	other
Explicit	1st	83%	13%	4%	98%	0%	2%
Explicit	2nd	85%	11%	4%	98%	0%	2%
Block Jacobi	1st	25%	73%	2%	96%	1%	3%
Block Jacobi	2nd	27%	72%	1%	96%	1%	3%

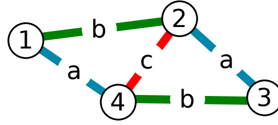

Figure 2. Example of a mesh with colouring (colours shown as letters on edges). Fluxes along all edges with the same colour can be computed in parallel. The nodes with numbers 2 and 4 have a vertex degree of 3 (i.e., three neighbours), meaning that at least 3 colours are needed for this mesh.

for $i \leftarrow 1 \dots n_{edges}$ **do**
  $l, r \leftarrow edge2nodes(i)$;
  $r_l \overset{+}{\leftarrow} f_l(u_l, u_r)$;
  $r_r \overset{+}{\leftarrow} f_r(u_l, u_r)$;
end

Algorithm 1: Interface-based residual computation 1D

This can be considered as a function of $\mathbf{u}$ with

$$\mathbf{r} = \mathbf{F}(\mathbf{u}), \tag{2}$$

where each entry in the output vector $\mathbf{r}$ is the result of a scalar sum-reduction of all contributions to that index.

It is tempting to use an OpenMP reduction pragma to parallelise the loop in Algorithm 1. Although the reduction pragma will prevent race conditions, it is not feasible for large cases because it will require a local copy of the residual vector for each thread. The consequence is not only an increased memory footprint proportional to the number of threads (which may be hundreds for many-core architectures) but also a slowdown of the execution, since each thread will only contribute to selected elements, but the entire vector needs to be initialised to zero on each thread and will be treated as a dense vector despite the number of nonzero elements per thread only being

$\mathcal{O}(\#nodes/\#threads)$. This could be solved using sparse accumulators (35), which have been described as a method to implement reduction operations with a runtime proportional to the number of nonzero entries. At the time of writing, no OpenMP implementation that we know of implements them.

For this reason, it is preferable to avoid write conflicts e.g. by using a colouring on the edges (36) or colouring with mini-partitions (37). In the form used in this work, the edges in the mesh are coloured such that for any node in the mesh, all edges connected to that node have a distinct colour. No two edges with the same colour are therefore connected to the same node, and the flux along all edges with a given colour can be computed in parallel without causing concurrent write access to any node. Similar colouring strategies have been described for other application areas as well, such as mesh smoothing (38).

The write access pattern of the primal loop over edges in the coloured graph in Figure 2 becomes apparent by expressing F as the sum of the contributions of individual iterations, and similarly for the adjoint code in Section 5.

$$\mathbf{F} = \begin{bmatrix} f_l(u_1,u_4) \\ 0 \\ 0 \\ f_r(u_1,u_4) \end{bmatrix} + \begin{bmatrix} 0 \\ f_l(u_2,u_3) \\ f_r(u_2,u_3) \\ 0 \end{bmatrix} \tag{a}$$

$$+ \begin{bmatrix} f_l(u_1,u_2) \\ f_r(u_1,u_2) \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ f_l(u_3,u_4) \\ f_r(u_3,u_4) \end{bmatrix} \tag{b}$$

$$+ \begin{bmatrix} 0 \\ f_l(u_2,u_4) \\ 0 \\ f_r(u_2,u_4) \end{bmatrix} \tag{c}$$

Given a mesh with a valid colouring, the following statements are all equivalent.

- Two edges $E_1 = \{i,j\}$ and $E_2 = \{k,l\}$ have the same colour.
- i,j,k,l are pairwise distinct.
- No index in the output array is written to by both the fluxes on E_1 and E_2.
- The flux update on E_1 and E_2 can be executed in parallel.

We can therefore implement the edge flux loop in Algorithm 1 using OpenMP as a serial loop over all colours in the mesh, within which we loop over all edges

with that colour in parallel without any further synchronisation except for the implicit synchronisation barrier at the end of each parallel `for` loop. The vector ia contains for each colour the index of the first edge with that colour. The parallelisation schedule is determined only after the colouring of the mesh is known, at runtime of the program.

```
for c ← 1 ... n_colours do
    !$OMP parallel for shared(u,r);
    for i ← ia(c) ... ia(c+1) − 1 do
        l, r ← edge2nodes(i);
        r_l +← f_l(u_l, u_r);
        r_r +← f_r(u_r, u_l);
    end
end
```

Algorithm 2: Interface-based residual computation 1D

5 Recycling the primal parallelisation for the adjoint solver

An AD tool without built-in support for OpenMP such as the most recent version 3.11 of Tapenade will simply omit the OpenMP pragmas when generating the adjoint code. In the absence of user intervention or sophisticated performance optimisation steps, a source-transformation AD tool will generate adjoint code as shown in Algorithm 3.

```
for c ← n_colours ... 1 do
    // possibly some OpenMP
    for i ← ia(c+1) − 1 ... ia(c) do
        l, r ← edge2nodes(i);
        [ū_l, ū_r] +← f̄_l(u_l, u_r) · r̄_l;
        [ū_l, ū_r] +← f̄_r(u_l, u_r) · r̄_r;
    end
end
```

Algorithm 3: Interface-based adjoint

With our new self-adjoint shared memory parallelisation approach (SSMP), we propose to take into account the observation that the loop nesting is preserved by the AD tool, so there is still an outer loop over colours and an inner loop over edges within each colour. We further observe in the above example that each iteration in the adjoint reads and writes to the same indices as the corresponding primal iteration. This is a consequence of the fact that potential write conflicts in the adjoint code are caused by nonexclusive read access in the primal as explained in Section 5.1.

By definition (39), the adjoint code of equation (2) has to be an implementation of

$$\bar{\mathbf{u}} = \bar{\mathbf{F}} \cdot \bar{\mathbf{r}} \tag{3}$$

with

$$\bar{\mathbf{F}} := \left(\frac{\partial \mathbf{F}}{\partial \mathbf{u}}\right)^T,$$

where $\bar{\mathbf{F}}$ for Figure 2 is given as

$$\bar{\mathbf{F}} = \begin{bmatrix} \frac{\partial f_l}{\partial u_1} & 0 & 0 & \frac{\partial f_r}{\partial u_1} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \frac{\partial f_l}{\partial u_4} & 0 & 0 & \frac{\partial f_r}{\partial u_4} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial f_l}{\partial u_2} & \frac{\partial f_r}{\partial u_2} & 0 \\ 0 & \frac{\partial f_l}{\partial u_3} & \frac{\partial f_r}{\partial u_3} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{(a)}$$

$$+ \begin{bmatrix} \frac{\partial f_l}{\partial u_1} & \frac{\partial f_r}{\partial u_1} & 0 & 0 \\ \frac{\partial f_l}{\partial u_2} & \frac{\partial f_r}{\partial u_2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial f_l}{\partial u_3} & \frac{\partial f_r}{\partial u_3} \\ 0 & 0 & \frac{\partial f_l}{\partial u_4} & \frac{\partial f_r}{\partial u_4} \end{bmatrix} \quad \text{(b)}$$

$$+ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & \frac{\partial f_l}{\partial u_2} & 0 & \frac{\partial f_r}{\partial u_2} \\ 0 & 0 & 0 & 0 \\ 0 & \frac{\partial f_l}{\partial u_4} & 0 & \frac{\partial f_r}{\partial u_4} \end{bmatrix}. \quad \text{(c)}$$

The implementation of (3) involving $\bar{\mathbf{F}}$ performs read operations in $\bar{\mathbf{r}}$ and write operations in $\bar{\mathbf{u}}$ at the same indices as the primal function $\mathbf{F}$ in $\mathbf{u}$ and $\mathbf{r}$. The coefficients of $\bar{\mathbf{F}}$ may depend on the primal variable u if f_l or f_r are nonlinear in $\mathbf{u}$, in which case the evaluation of $\bar{\mathbf{F}}$ further requires read access in $\mathbf{u}$.

5.1 Self-adjoint memory access

We assume that each memory location is either only read from or only written to during the entire parallel loop. The only exception is made for memory locations that are incremented using statements equivalent to a sum-reduction. In particular, we consider only loops without loop-carried dependencies where the result of one iteration does not depend on the result of previous iterations.

We state in Lemma 1 a basic property of the relationship between memory access in primal and adjoint codes. This is used to show in Proof 1 that the absence of concurrent read access in the primal code is a sufficient condition to ensure an adjoint code that is free of data races. We then define *self-adjoint memory access* as a property of the

memory access pattern of the primal code, and prove in the remainder of the section that this is a sufficient condition to ensure exclusive read access, allowing us to reverse-differentiate code with self-adjoint memory access while retaining its parallelisation.

Lemma 1. Let I be an iteration of a loop that reads data from the primal input vector $\mathbf{u}$ at the set of indices $M_{in}(\mathbf{u})$ and writes to the primal output vector $\mathbf{r}$ at the set of indices $M_{out}(\mathbf{r})$. Further, let $\bar{I}$ be an iteration of the corresponding adjoint loop that performs read access in the adjoint input vector $\bar{\mathbf{r}}$ at the set of indices $\bar{M}_{in}(\bar{\mathbf{r}})$ as well as the primal input vector $\mathbf{u}$ at the set of indices $\bar{M}_{in}(\mathbf{u})$, and performs write access to the adjoint output vector $\bar{\mathbf{u}}$ at the set of indices $\bar{M}_{out}(\bar{\mathbf{u}})$.

The following statements hold.

- $\bar{M}_{out}(\bar{\mathbf{u}}) \subseteq M_{in}(\mathbf{u})$
- $\bar{M}_{in}(\bar{\mathbf{r}}) \subseteq M_{out}(\mathbf{r})$
- $\bar{M}_{in}(\mathbf{u}) \subseteq M_{in}(\mathbf{u})$

The lemma becomes clear with the following proof.

Proof 1. Assume that I does not write to an index m in $\mathbf{r}$, that is, $m \notin M_{out}(\mathbf{r})$. In other words, the contribution of I to $\mathbf{r}_m$ is zero. Consequently, the m-th row of the Jacobian matrix is zero. Formally, $\frac{\partial I_m}{\partial \mathbf{u}_n} = 0$ for any n.
Likewise, if I does not read from an index n in $\mathbf{u}$, namely, $n \notin M_{in}(\mathbf{u})$, the Jacobian matrix of I will contain only zeroes in column n.
Obviously, if the n-th column is all zero in $\left(\frac{\partial I}{\partial \mathbf{u}}\right)$, then the n-th row in $\left(\frac{\partial I}{\partial \mathbf{u}}\right)^T$ is all zero. The adjoint of I is an implementation of

$$\bar{I} : \bar{\mathbf{u}} \leftarrow \left(\frac{\partial I}{\partial \mathbf{u}}\right)^T \cdot \bar{\mathbf{r}}.$$

Hence, the linear operator $\bar{I}$ will make no contribution to index n. Similarly, a zero row m in $\left(\frac{\partial I}{\partial \mathbf{u}}\right)$ will mean that $\bar{I}$ does not read from index m.

Förster previously observed (23) that because of this property, write conflicts at shared arrays in the adjoint code can happen only if there is concurrent read access to shared arrays in the primal code. He introduced the *exclusive read property* that, when held by a code I, guarantees that $\bar{I}$ does not produce write conflicts in shared arrays. In practice, this is hard to prove at compile time. We solve this problem by introducing the concept of *self-adjoint memory access*.

Definition 1. The memory access of a code is *self-adjoint* if the code has distinct read-only input and write-only output vectors, and the set of indices accessed in the primal

input vector is equal to the set of indices accessed in the primal output vector, that is, $M_{in}(\mathbf{u}) = M_{out}(\mathbf{r})$.

Lemma 2. If a primal parallel loop whose body has self-adjoint memory access writes a result into a shared output vector without data race, then its adjoint can be created with a shared adjoint output vector and executed in parallel without data race.

Proof 2. If the original loop has no data race, then it has a mapping from iteration numbers to non-overlapping output indices. By definition, it that loop has self-adjoint memory access, the mapping from iteration number to input indices is the same, and thus also non-overlapping. This means that the original loop has exclusive read access. The output indices of the adjoint loop are a subset of the input indices of the original loop, and therefore also non-overlapping. It follows that a code that is self-adjoint and free of data races also has an adjoint code that is free of data races.

5.2 Rules for adjoint OpenMP

In this section we summarise the correct treatment of parallel OpenMP loops during reverse-mode differentiation. The approach presented in this work affects only the treatment of shared arrays in loops with self-adjoint memory access. For all other types of variables and loops we summarise the existing approach in the literature (22; 23) as follows.

Primal variable scope	Scope of corresponding adjoint variable
private	private
firstprivate	reduction
lastprivate	shared (only last iteration writes)
reduction	firstprivate
shared write	shared read
shared read	**atomic write**

We recommend the following refined strategy for shared variables that are used as shared read variables in the primal code (last case in the above table).

- Following our method, for arrays $\bar{a}$ for which the primal variable a has self-adjoint memory access, use the *shared* work construct.
- Following the previous methods outlined in Section 2, for arrays with sparse memory access (large arrays, many threads), use the atomic construct, else use the reduction construct.*

*Reductions to array variables in C/C++ were introduced in the OpenMP 4.5 standard in November 2015

5.3 Automatic generation

To implement self-adjoint shared memory parallelisation (SSMP), a user has to identify loops with self-adjoint memory access and mark them in a preprocessing step of which the AD tool does not need to be aware. A postprocessing script is used to deploy the correct OpenMP pragmas in the generated adjoint code before compilation for all adjoint loops for which the corresponding primal loops were found to have self-adjoint memory access.

We use the technique known as the *subroutine technique* (13) or *function outlining* (40) for variable scoping. In this approach, all primal loop bodies of parallel loops with self-adjoint memory access are placed into dedicated subroutines. If the compiler uses function inlining during compilation, this will not change the performance. The names of these subroutines contain a particular tag.

All OpenMP shared variables are passed to the routine by using call-by-reference. OpenMP private variables are declared as local variables inside the routine. The generated adjoint code contains an adjoint argument for each active primal routine argument and an internal adjoint variable for each internal private variable. Because the memory access in the primal code is self-adjoint, all external subroutine arguments of the adjoint code are allowed to be shared variables, while all internal variables must be private.

The generated adjoint code can then be postprocessed in two simple steps. First, in front of every loop that contains a call to a subroutine that has a name that contains the aforementioned tag, the pragma

```
!$OMP PARALLEL DO DEFAULT SHARED
```

is placed. Second, most AD tools will place calls to a taping mechanism inside the generated adjoint code. This is used to store intermediate results during the original computation, which have an influence on the derivatives calculated during the adjoint computation. This is the case for nonlinear functions or branches. Therefore, inside all subroutines that are named with the tag, access to the taping mechanism provided by the AD tool are rerouted to a thread-safe stack implementation.

Arguably, the application of SSMP has limitations. It requires changes in the primal code, and the primal code inside parallel loops must not use global variables or common blocks (with the exception of constant parameters). If the primal code performs self-adjoint memory access to some array and conflicting access to another array, SSMP can be used for the former, but other approaches have to be used for the latter. A user must ensure that there is no concurrent read access from any memory address in the primal code even if it is accessible through multiple variable names (aliasing).

6 Test cases

To demonstrate the effectiveness of the SSMP approach, we apply it to the 3D unstructured edge-based fluid dynamics solver mgopt, which simulates compressible inviscid, laminar, or turbulent flow using the Spalart-Allmaras(41) or DES (42) model. The adjoint solver parallelised with SSMP is used to compute sensitivities for three different test cases that are derived from real-life applications: a U-shaped cooling channel, a turbine stator blade, and a fuel-efficient concept car. We present scalability and absolute speed of the adjoint solver with SSMP on a system with two CPUs, as well as on a XeonPhi accelerator card, and we compare them with the timings of an adjoint solver that has been parallelised using atomic pragmas without SSMP.

6.1 Methodology

The solver mgopt in the version considered in this work consists of $35,994$ lines of Fortran 90 code, of which $11,794$ lines are in core solver routines that are differentiated by using the AD tool Tapenade (15). The differentiated routines contain mainly the spatial discretisation of the flow model such as second-order Roe (43) flux, approximated spatial gradients of physical quantities and limiters, and turbulence models. The generated adjoint routines are used to build the adjoint CFD solver in an automated process that incorporates algorithmically differentiated routines with hand-coded adjoints for the BDF2, Runge-Kutta, geometric multigrid, and linear solver implementations using the fact that BDF2 and most Runge-Kutta schemes are self-adjoint and multigrid operators are self-adjoint up to a constant factor (33).

The routines that contribute most to the overall runtime are parallelised by using OpenMP based on an edge-colouring of the input mesh. The number of colours affects the number of synchronisation barriers inside the code and thus affects the parallel performance. We therefore report the number of colours required for each of the test cases. We compute the maximum vertex degree of each mesh as a lower bound for the minimum number of required colours (44). The mgopt solver uses a greedy edge colouring heuristic (45) without edge preordering, which sometimes results in a higher than optimum number of colours.

The tests are run on two Intel Xeon E5-2660 CPUs clocked at 2.2 GHz with 8 cores and 20 MB L3 cache each. We run Scientific Linux 6.2. The codes are compiled with gfortran 4.8.2 with the -O3 flag and scatter OpenMP thread affinity; AD code is generated with Tapenade 3.11 and then compiled with the same flags as the primal. We also test the performance of the adjoint solver on an Intel XeonPhi Coprocessor (MIC) 5110P Knights Corner clocked at 1.053 GHz. We use native execution to measure

timings on the MIC (i.e., the full code runs on the MIC, not just parallel sections). The solver is compiled for the MIC using ifort 14.0 with `-fast -mmic` flags and `balanced` OpenMP thread affinity. There are 60 cores of which we use up to 59 and leave the remaining core to the operating system. Each core supports up to 4 threads and has 512 kB of L2 cache. We measure performance for serial execution as well as with multiples of 59 threads up to 236 threads.

Timings are measured for the internal residual computation only. We exclude all I/O, linear solvers and temporal discretisation, because they are not differentiated and therefore not parallelised with SSMP. The baseline for the scalability figures of the primal solver on a CPU is the runtime of the serial primal solver on the CPU. For the scalability on the XeonPhi, the baseline timing is obtained by running the serial primal solver on the XeonPhi. Likewise, the baseline timing for the adjoint scalability is the runtime of the serial adjoint solver on the respective machine. The scalings for SSMP and atomic adjoint approaches use the same baseline, since the serial adjoint solver does not contain any OpenMP and can therefore not contain atomic pragmas.

We run three test cases to demonstrate the speedup of our OpenMP solver: flow inside a U-shaped cooling channel, flow around a turbine stator blade, and flow around a fuel-efficient concept vehicle. We chose cases with different sizes and mesh topologies, to test the parallelisation in a variety of conditions.

6.2 U-bend (BEND)

We simulate flow through a U-shaped cooling channel, which is used inside turbine fan blades to prevent damage from the combustion temperatures. The adjoint derivatives are useful to redesign the channel for lower pressure loss or better heat exchange, which can help increase the fuel efficiency of aircraft.

The mesh is generated as a fully structured cartesian mesh, after which a coordinate transformation is applied to create the bend shape. Although the mesh is structured, the solver treats it as an unstructured mesh. The simplicity of the geometry allows us to automatically generate a variety of different mesh sizes for this case to investigate the solver speed; see Figure 3 for an example with 200k nodes. The node numbering is randomised, making the test case more challenging as it causes the colouring heuristic to require more colours and also probably increases the number of cache misses. This is in contrast to the RR case in the following section where the difference in node numbers of any two points corresponds to their spatial distance, resulting in a better scalability of that test case.

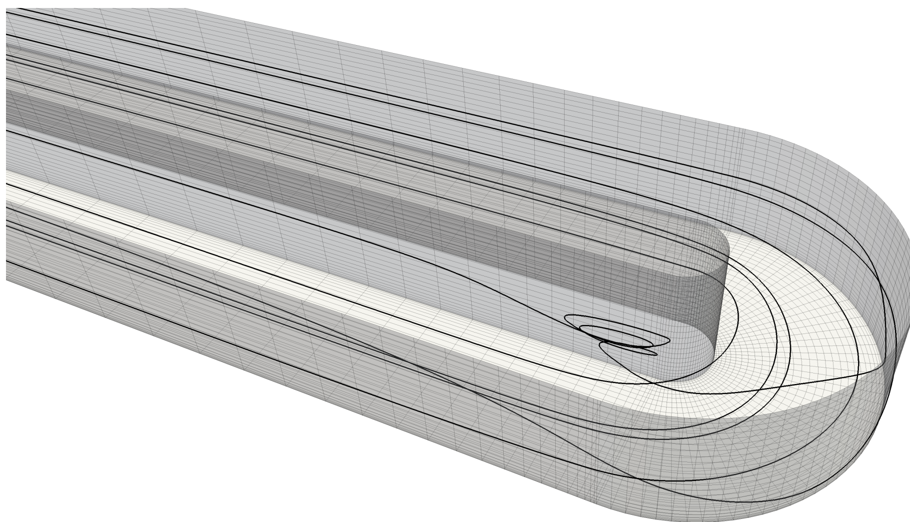

Figure 3. BEND: U-Bend used to model flow in an internal cooling channel of turbine wings. Structured hexahedral mesh. Only quadrilateral surface mesh and streamlines are shown.

The flow is set up with a fixed inlet velocity of Ma 0.02565 (the Mach number is defined as a fraction of the speed of sound; under the flow conditions inside the bend this is ca. 8 m/s) and a fixed pressure outlet. We use second-order-accurate Roe fluxes, the Spalart Allmaras turbulence model, and Green-Gauss spatial gradients, all of which are fully differentiated with Tapenade for the adjoint solver, and an explicit time-stepping scheme. The channel has viscous non-slip walls. The greedy mesh colouring algorithm in `mgopt` required 11 colours for this mesh, significantly more than the optimum number of colours of 6. All volume elements are hexahedral, and all boundary faces are quadrilateral.

6.3 Fuel-efficient vehicle (FEV)

We use the geometry of the fuel efficient vehicle "Droplet", designed as a design study for the Shell Eco-marathon at Warsaw University of Technology, Faculty of Power and Aeronautical Engineering, by the Student's Association of Vehicle Aerodynamics. The competition's goal is to build a vehicle that can drive as far as possible on 1 litre of fuel at an average speed of 8 mph. Adjoint-based aerodynamic shape optimisation of the car body is one way to reduce drag, which is crucial for the performance and fuel efficiency. We use second-order Roe fluxes, non-slip wall conditions for the car and the ground, and freestream boundary conditions for the remaining boundaries. The

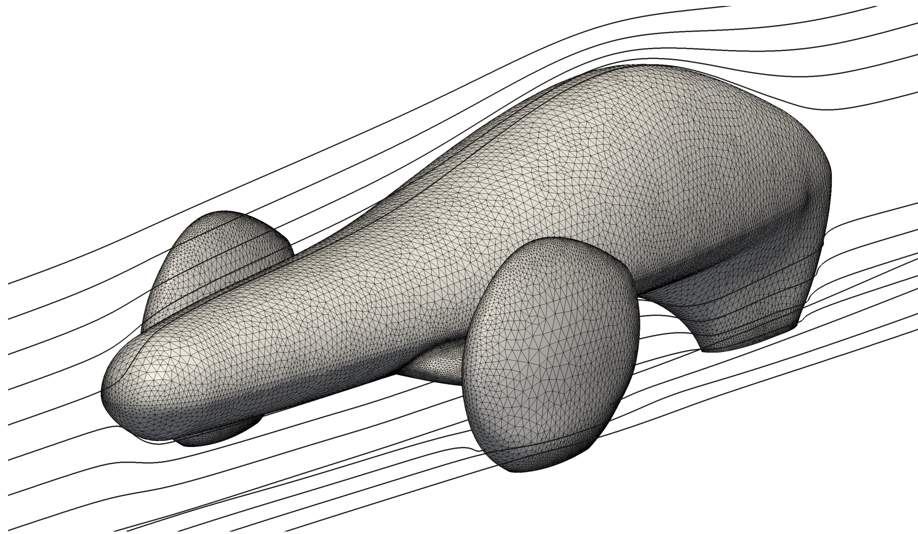

Figure 4. FEV: Fuel-efficient vehicle "Droplet". Tetrahedral mesh; only triangular surface mesh and flow streamlines are shown.

mesh consists of 1.08 million tetrahedral cells and 1.31 million edges; see Figure 4. The highest vertex degree is 23, and our colouring algorithm requires 25 colours. The number of colours is higher and the scalability is poorer than in all other test cases.

6.4 Turbine stator (RR)

The turbine stator used in this case is a geometry provided by Rolls-Royce Deutschland. Adjoint-based optimisation is used to improve the performance of jet engines. The mesh consists of 1.2M hexahedral elements with 3.7M edges. The colouring algorithm requires 7 colours for this mesh, which has a maximum vertex degree of 7; see Figure 5 for an illustration of the geometry and flow conditions. Results for this test case are presented only on the CPU, because the solver exceeded the memory limits of the MIC with this mesh.

7 Results

After the discussion of detection and implementation of self-adjoint shared memory parallelisation (SSMP), we show runtime and speedup with either SSMP or the approach using atomic constructs that was proposed by Förster (23).

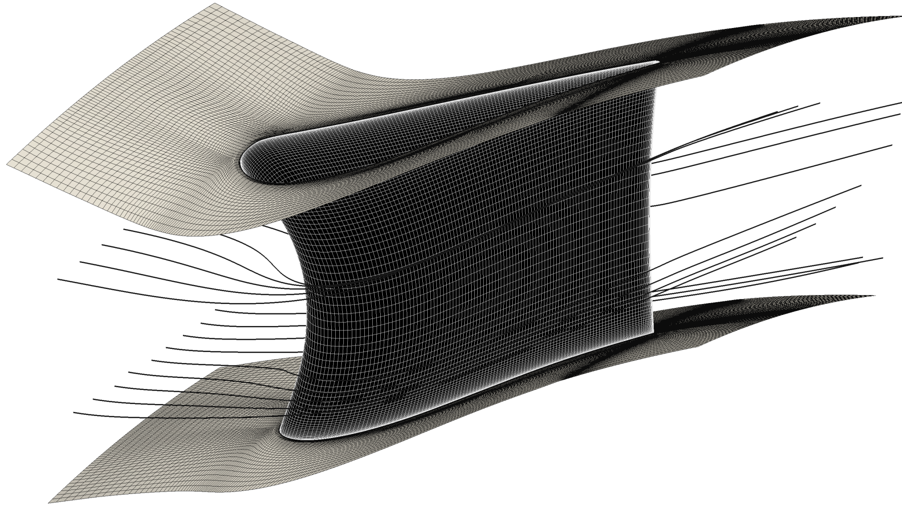

Figure 5. RR: Turbine stator test case. Hexahedral mesh, only quadrilateral surface mesh on hub and casing, and streamlines are shown.

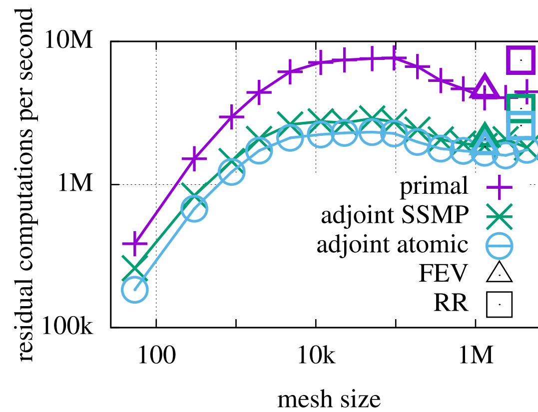

Figure 6. Edge residual computations per second for 16 CPU threads (higher is better). FEV and RR mesh with given size; BEND mesh with various mesh sizes from 27 to 1.5M cells. For MIC see Figure 7

7.1 Execution speed

The runtimes in this section are presented in a normalised fashion as the number of residual computations per second. This allows us to compare runtimes on different mesh sizes and with different numbers of iterations. To obtain the *execution speed*, we divide the measured program runtime with 16 CPU threads or 236 MIC threads by the number of iterations and the number of edges in the given mesh.

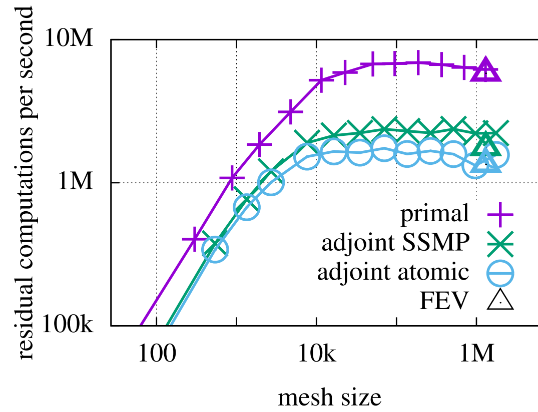

Figure 7. Edge residual computations per second for 236 MIC threads (higher is better). FEV mesh with given size; BEND mesh with various mesh sizes from 27 to 1.2M cells. For CPU see Figure 6

Table 2. Number of edge residual updates per second (computational speed) for FEV, RR, and BEND with 600k nodes. The SSMP approach increases speed by $12 - 28.1\%$, which is comparable to the speed gained by ordering mesh nodes

CPU	SSMP	Atomic	Difference
BEND	1.86M	1.61M	13.4%
FEV	2.09M	1.84M	12.0%
RR	3.40M	2.58M	24.1%
MIC MIC	SSMP	Atomic	Difference
BEND	1.99M	1.43M	28.1%
FEV	1.75M	1.34M	23.4%

In the absence of caching effects and synchronisation barriers, we would expect the execution speed to be the same for all test cases. In practice, however, we observe a vastly different execution speed for different meshes. On the CPU, the execution speed is significantly lower if the number of edges in the mesh is below 10k, since smaller meshes do not allow enough parallelism to compensate for the serial parts of the code such as the OpenMP barriers after each colour. The execution speed for a series of BEND meshes with increasing size is shown in Figure 6 for the CPU and Figure 7 for the MIC, where the speed difference for different mesh sizes is even more pronounced. We observe a drop in execution speed on the CPU when the mesh size exceeds 100k edges, which is probably caused by the fact that the state and residual vectors no longer fit into the $L3$ cache of the CPU. On the MIC we do not observe such an effect, probably because of the lack of shared $L3$ cache; see Figure 7.

We observe a roughly constant factor in speed difference between the adjoint solver with atomic constructs and that with SSMP for all mesh sizes. For the BEND case with 600k edges as well as the FEV and RR case, execution speed is listed in Table 2. The overhead of using atomic pragmas has a larger influence on the MIC than on the CPU, probably because of the lower serial speed of the MIC and the higher cost of synchronisation barriers for a large number of threads. The effect of atomic pragmas is significant, with at least 12% and up to 28.1% runtime difference in all test cases.

7.2 Scalability

The scaling for up to 16 CPU cores is shown in Figure 8 for the U-bend (BEND) with 600k nodes. The scaling is computed relative to the serial code with no OpenMP parallelism. The scaling results therefore indicate the runtime overhead introduced by the atomic pragmas. Similarly, the scalability of the code is shown for the Rolls-Royce stator (RR) and fuel-efficient vehicle (FEV) test case in Figure 9 and Figure 10. The speedups agree with previous studies on OpenMP in unstructured CFD solvers (46).

The adjoint code generated with SSMP consistently scales better than the adjoint solver with atomic pragmas as well as the primal solver. While it is not surprising that the adjoint code runs faster without atomic pragmas, some explanation is needed as to why the adjoint solver scales better than the primal solver. First, we note that the speedup of the adjoint solver is computed relative to the speed of the serial adjoint solver, whereas the speedup of the primal solver is relative to the speed of the serial primal solver. Hence, the better scaling hides the fact that the adjoint solver is still significantly slower than the primal solver. Similarly, a better scalability on one of the meshes does not indicate a better absolute runtime, and a better scaling on the MIC does not indicate a better absolute runtime than on the CPU (which is why absolute runtimes are discussed in Section 7.1).

The better scalability of the adjoint solver on the CPU can be explained by the fact that the adjoint solver performs more operations than does the primal solver but has the same number of colours and therefore the same number of synchronisation barriers. The higher computational cost therefore hides some of the parallelisation overhead. The fact that the RR case scales much better than the FEV case confirms this theory. The case has a large number of edges and a low number of colours, which also increased the amount of computational cost per synchronisation barrier.

On the MIC, the adjoint solver achieves a better scaling for the BEND case than does the primal solver, while the primal solver scales slightly better than the adjoint solver in the FEV case. The performance penalty of using atomic pragmas on the MIC

Table 3. Speedup factors for primal, adjoint with SSMP, and adjoint with atomic pragmas, compared with serial execution on the same machine with the same test cases. Data shown for BEND case with 600k vertices, and RR and FEV cases, 16 CPU and 236 MIC threads.

CPU	SSMP	Atomic	Primal	Figure
BEND	9.6	8.4	8.7	8
FEV	9.5	8.5	9.3	9
RR	13.4	13.0	12.3	10
MIC	SSMP	Atomic	Primal	Figure
BEND	138.2	99.4	124.2	11
FEV	114.4	87.7	127.6	12

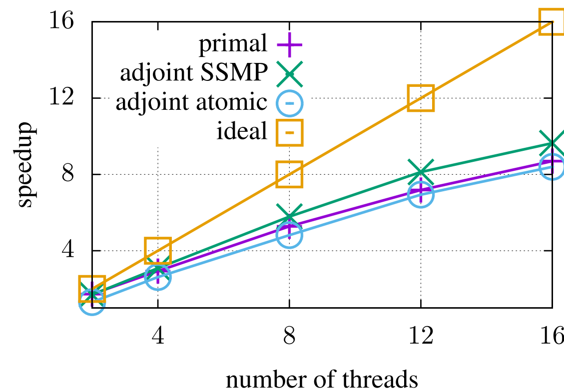

Figure 8. Scaling of primal and adjoint residual for BEND mesh with 600k vertices on CPU, with maximum speedup of 9.6 (SSMP), 8.4 (atomic), and 8.7 (primal) as in Table 3

is more pronounced than on the CPU, which is not surprising because the overhead of synchronisation constructs grows with the number of threads.

The MIC speedups shown in Figure 11 and Figure 12 might appear poor but are typical at least for first-generation XeonPhi coprocessors (47). The same is true for the absolute runtimes (48). The method that we present in this work affects only the AD process and should be orthogonal to future code optimisation steps such as vectorisation that are undertaken to speed up the primal code in the future.

8 Conclusions and future work

We presented SSMP, an approach to generate source-transformation adjoint code of structured or unstructured OpenMP-parallel stencil operations. The method was successfully applied to an adjoint CFD solver that supports shared memory parallelism on multi-core CPUs and many-core architectures such as the Intel XeonPhi.

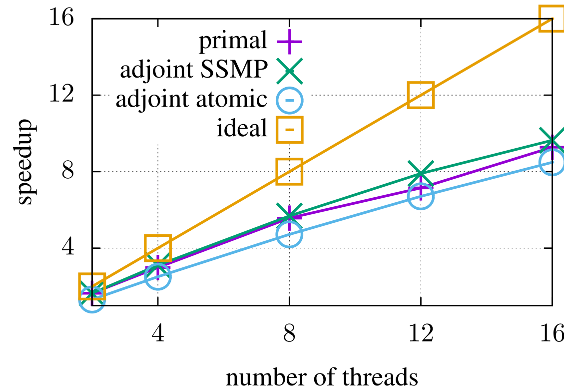

Figure 9. Scaling of primal and adjoint residual for FEV mesh on up to 16 CPU cores, with the maximum speedup of 9.5 (SSMP), 8.5 (atomic), and 9.3 (primal) as in Table 3

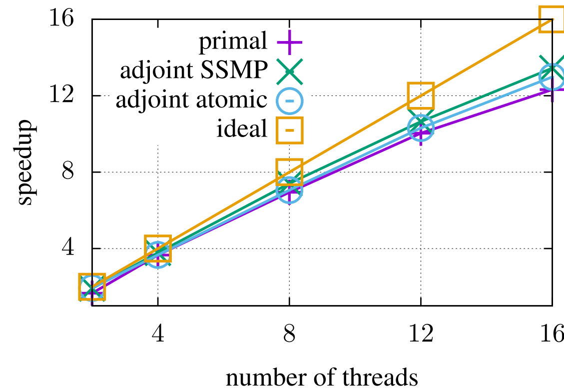

Figure 10. Scaling of primal and adjoint residual for RR mesh on up to 16 CPU cores, with the maximum speedup of 13.4 (SSMP), 13.0 (atomic), and 12.3 (primal) as in Table 3

The parallelisation of the adjoint code that can be automatically generated by using SSMP matches that of the primal solver. Since the computation is structured essentially in the same way as in the primal code, we had reason to believe that the generated adjoint code will be as scalable as the primal code. This fact was indeed confirmed in our experiments, where the scaling of the parallel adjoint solver even exceeds that of the primal solver in most cases. Furthermore, we demonstrated that SSMP performs significantly better than other approaches described in the literature that require OpenMP `atomic` pragmas or `critical` sections.

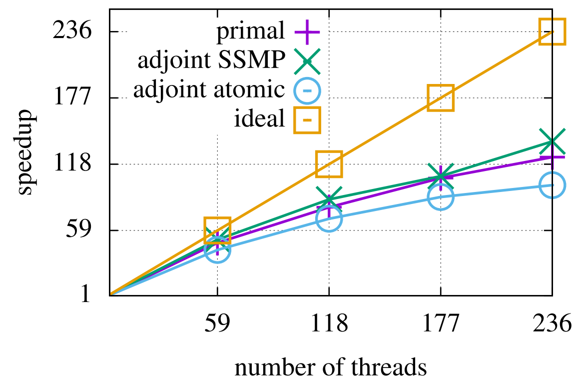

Figure 11. Scaling of primal and adjoint residual for BEND mesh with 600k vertices on up to 236 MIC cores, with the maximum speedup of 138.2 (SSMP), 99.4 (atomic), and 124.2 (primal) as in Table 3

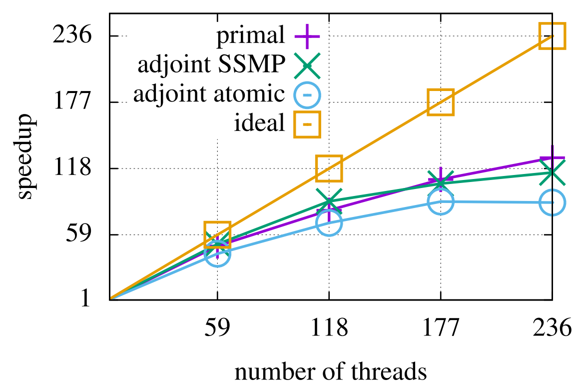

Figure 12. Scaling of primal and adjoint residual for FEV mesh on up to 236 MIC cores, with the maximum speedup of 114.4 (SSMP), 87.7 (atomic), and 127.6 (primal) as in Table 3

We defined the prerequisites that have to be met by a parallel loop for SSMP to be applicable, and gave a proof of correctness for the generated parallel adjoint code. This is an important step towards the integration of SSMP in algorithmic differentiation tools. Since the parallelisation of the primal code is usually the result of careful manual optimisation, we believe that the possibility to generate an equally scalable adjoint code makes our method attractive for a wide range of applications.

In our implementation, suitable code portions need to be identified by hand. The build process itself is automated so that changes in the primal code will correctly be

incorporated into the adjoint solver. It would be beneficial to integrate SSMP into an AD tool along with an automated detection of self-adjoint memory access, to make this approach available to a wider audience. The method currently works only for parallel loops where the indices used to access input and output arrays are identical. We plan to extend our approach to loops that implement a function with a Jacobian matrix with symmetric sparsity pattern.

Acknowledgements

This work has been conducted within the About Flow project on “Adjoint-based optimisation of industrial and unsteady flows”. The project has received funding from the European Union’s Seventh Framework Programme for research, technological development, and demonstration under grant agreement no 317006.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics and Computer Science programs under contract number DE-AC02-06CH11357.

This research utilised Queen Mary University of London’s MidPlus computational facilities, supported by QMUL Research-IT and funded by EPSRC grant EP/K000128/1.

We thank Rolls-Royce Deutschland for the permission to use the turbine stator test case, and Warsaw University of Technology for the permission to use the fuel efficient vehicle test case.

References

[1] Giles M and Pierce N. *Adjoint equations in CFD - Duality, boundary conditions and solution behaviour*. American Institute of Aeronautics and Astronautics, 1997. DOI: doi:10.2514/6.1997-1850.

[2] Jameson A, Shankaran S and Martinelli L. Continuous adjoint method for unstructured grids. *AIAA Journal* 2008; 46(5): 1226–1239. DOI:10.2514/1.25362.

[3] Blazek J. *Computational Fluid Dynamics: Principles and Applications: Principles and Applications*. Elsevier Science, 2001. ISBN 9780080545547.

[4] Giles MB and Pierce NA. An introduction to the adjoint approach to design. *Flow, Turbulence and Combustion* 2000; 65(3): 393–415. DOI:10.1023/A:1011430410075.

[5] Christakopoulos F, Jones D and Müller JD. Pseudo-timestepping and verification for automatic differentiation derived CFD codes. *Computers & Fluids* 2011; 46(1): 174 – 179. DOI:10.1016/j.compfluid.2011.01.039.

[6] Müller JD and Cusdin P. On the performance of discrete adjoint CFD codes using automatic differentiation. *International journal for numerical methods in fluids* 2005; 47(8-9): 939–945.

[7] Hovland PD. *Automatic differentiation of parallel programs*. PhD Thesis, University of Illinois at Urbana-Champaign, 1997.

[8] Schanen M, Naumann U, Hascoët L et al. Interpretative adjoints for numerical simulation codes using MPI. *Procedia Computer Science* 2010; 1(1): 1825–1833.

[9] Utke J, Hascoët L, Heimbach P et al. Toward adjoinable MPI. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009. pp. 1–8.

[10] Naumann U, Hascoët L, Hill C et al. A framework for proving correctness of adjoint message-passing programs. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag. ISBN 978-3-540-87474-4, 2008. pp. 316–321. DOI: 10.1007/978-3-540-87475-1_44.

[11] Borkar S. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*. DAC 2007, New York, NY, USA: ACM. ISBN 978-1-59593-627-1, 2007. pp. 746–749. DOI:10.1145/1278480.1278667.

[12] Dagum L and Enon R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 1998; 5(1): 46–55.

[13] Löhner R and Baum JD. Handling tens of thousands of cores with industrial/legacy codes: Approaches, implementation and timings. *Computers & Fluids* 2013; 85(0): 53–62. DOI: http://dx.doi.org/10.1016/j.compfluid.2012.09.030.

[14] Schmidl D, Cramer T, Wienke S et al. Assessing the performance of OpenMP programs on the Intel Xeon Phi. In *Euro-Par 2013 Parallel Processing*. Springer, 2013. pp. 547–558.

[15] Hascoet L and Pascual V. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans Math Softw* 2013; 39(3): 20:1–20:43. DOI:10.1145/2450153.2450158.

[16] Othmer C. A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows. *International Journal for Numerical Methods in Fluids* 2008; 58(8): 861–877. DOI:10.1002/fld.1770.

[17] Farrell PE, Ham DA, Funke SW et al. Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing* 2013; 35(4): C369–C393.

[18] Towara M, Schanen M and Naumann U. MPI-parallel discrete adjoint OpenFOAM. *Procedia Computer Science* 2015; 51: 19–28.

[19] Bücker HM, Rasch A and Wolf A. A class of OpenMP applications involving nested parallelism. In *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004. pp. 220–224.

[20] Bücker HM, Lang B, an Mey D et al. Bringing together automatic differentiation and OpenMP. In *Proceedings of the 15th International Conference on Supercomputing*. ICS 2001, New York, NY, USA: ACM. ISBN 1-58113-410-X, 2001. pp. 246–251. DOI: 10.1145/377792.377842.

[21] Bücker HM, Lang B, Rasch A et al. Explicit loop scheduling in OpenMP for parallel automatic differentiation. In *Annual International Symposium on High Performance Computing Systems and Applications*, volume 16. 2002. pp. 121–126. DOI:10.1109/HPCSA.2002.1019144.

[22] Förster M, Naumann U and Utke J. Toward adjoint OpenMP. Technical Report AIB-2011-13, RWTH Aachen, Software and Tools for Computational Engineering, 2011.

[23] Förster M. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. PhD Thesis, RWTH Aachen, 2014.

[24] Schanen M, Förster M, Lotz J et al. Adjoining hybrid parallel code. In Topping BHV (ed.) *Proceedings of the Eighth International Conference on Engineering Computational Technology*, volume 100. Kippen, Stirlingshire. ISBN 978-1-905088-55-3, 2012. p. 18.

[25] Giering R, Kaminski T, Todling R et al. Tangent linear and adjoint versions of NASA/GMAO's Fortran 90 global weather forecast model. In *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006. pp. 275–284.

[26] Saltz J, Mirchandaney R and Crowley R. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers* 1991; 40(5): 603–612. DOI:http://doi.ieeecomputersociety.org/10.1109/12.88484.

[27] Gutiérrez E, Asenjo R, Plata O et al. Automatic parallelization of irregular applications. *Parallel Computing* 2000; 26(13–14): 1709–1738.

[28] Ravishankar M, Eisenlohr J, Pouchet LN et al. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 2012.

[29] Giles MB. On the iterative solution of adjoint equations. In *Automatic differentiation of algorithms*. Springer, 2002. pp. 145–151.

[30] Krakos JA. *Unsteady adjoint analysis for output sensitivity and mesh adaptation*. PhD Thesis, Massachusetts Institute of Technology, 2012.

[31] Walther A, Griewank A and Vogel O. ADOL-C: Automatic differentiation using operator overloading in C++. *PAMM* 2003; 2(1): 41–44.

[32] Utke J, Naumann U, Fagan M et al. OpenAD/F: A modular open-source tool for automatic differentiation of Fortran codes. *ACM Trans Math Softw* 2008; 34(4): 18:1–18:36. DOI: 10.1145/1377596.1377598.

[33] Giles MB. On the use of Runge-Kutta time-marching and multigrid for the solution of steady adjoint equations. Technical Report NA00/10, Oxford University Computing Laboratory, 2000.

[34] Diskin B, Thomas JL, Nielsen EJ et al. Comparison of node-centered and cell-centered unstructured finite-volume discretizations: Viscous fluxes. *AIAA journal* 2010; 48(7): 1326–1338.

[35] Gilbert JR, Moler C and Schreiber R. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications* 1992; 13(1): 333–356.

[36] Guo X, Lange M, Gorman G et al. Developing a scalable hybrid mpi/openmp unstructured finite element model. *COMPUTERS FLUIDS* 2015; 110: 227–234. DOI:10.1016/j.compfluid.2014.09.007.

[37] Giles MB, Mudalige GR, Sharif Z et al. Performance analysis of the op2 framework on many-core architectures. *SIGMETRICS Perform Eval Rev* 2011; 38(4): 9–15. DOI:10.1145/1964218.1964221.

[38] Gorman GJ, Southern J, Farrell PE et al. Hybrid OpenMP/MPI anisotropic mesh smoothing. *Procedia Computer Science* 2012; 9: 1513–1522.

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

[40] Liao C, Hernandez O, Chapman B et al. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience* 2007; 19(18): 2317–2332.

[41] Spalart P and Allmaras S. *A one-equation turbulence model for aerodynamic flows*. American Institute of Aeronautics and Astronautics, 1992. DOI:doi:10.2514/6.1992-439.

[42] Spalart PR. Detached-Eddy simulation. *Annual Review of Fluid Mechanics* 2009; 41: 181–202.

[43] Roe PL. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of computational physics* 1981; 43(2): 357–372.

[44] Erdős P and Wilson RJ. On the chromatic index of almost all graphs. *Journal of combinatorial theory, series B* 1977; 23(2-3): 255–257.

[45] Lotfi V and Sarin S. A graph coloring algorithm for large scale scheduling problems. *Computers Operations Research* 1986; 13(1): 27 – 32. DOI:http://dx.doi.org/10.1016/0305-0548(86)90061-4.

[46] Camelli F, Löhner R and Mestreau EL. Timings of an unstructured-grid CFD code on common hardware platforms and compilers. In *46th AIAA Aerospace Sciences Meeting and Exhibit, American Institute of Aeronautics and Astronautics Inc., Reno, NV, USA*. 2007.

[47] Economon TD, Palacios F, Alonso JJ et al. Towards high-performance optimizations of the unstructured open-source SU2 suite. *AIAA Paper* 2015; Vol. 1949.

[48] Che Y, Xu C, Fang J et al. Realistic performance characterization of CFD applications on Intel many integrated core architecture. *The Computer Journal* 2015; DOI:10.1093/comjnl/bxv022.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.