Procedia Computer Science

Volume 51, 2015, Pages 19–28

ICCS 2015 International Conference On Computational Science

# MPI-Parallel Discrete Adjoint OpenFOAM

Markus Towara, Michel Schanen, and Uwe Naumann

Software and Tools for Computational Engineering,
RWTH Aachen University, Aachen, Germany
[towara|schanen|naumann]@stce.rwth-aachen.de

**Abstract**

OpenFOAM is a powerful Open-Source (GPLv3) Computational Fluid Dynamics tool box with a rising adoption in both academia and industry due to its continuously growing set of features and the lack of license costs. Our previously developed discrete adjoint version of OpenFOAM allows us to calculate derivatives of arbitrary objectives with respect to a potentially very large number of input parameters at a relative (to a single *primal* flow simulation) computational cost which is independent of that number. Discrete adjoint OpenFOAM enables us to run gradient-based methods such as topology optimization efficiently. Up until recently only a serial version was available limiting both the computing performance and the amount of memory available for the solution of the problem. In this paper we describe a first parallel version of discrete adjoint OpenFOAM based on our adjoint MPI library.

*Keywords:* OpenFOAM, Adjoints, MPI, Algorithmic Differentiation

## 1 Introduction

This paper is about coupling discrete adjoint OpenFOAM [27] with the adjoint MPI (AMPI[1]) library [26] to achieve robust scaling and a better usage of distributed memory. Following some motivation in Section 2 we briefly introduce in Section 3 discrete adjoint OpenFOAM and AMPI. For the original contribution of this paper both are combined in Section 4 to obtain MPI-parallel discrete adjoint OpenFOAM. Feasibility of the proposed approach is documented in section 5 by a case study including numerical results, scaling behavior, and memory requirements. In section 6 we close with a conclusion and a summary of other ongoing work.

## 2 Motivation

Computational Fluid Dynamics (CFD) simulations have over the past decades become a viable extension of or alternative to experiments (e.g. conducted in wind-tunnels). Simulations are often less expensive, easily reproducible, and offer a wider range of boundary conditions. They

[1] https://www.stce.rwth-aachen.de/software/ampi.html

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015


doi:10.1016/j.procs.2015.05.181

enable the inspection of flow phenomena in places which are not easily accessible / measurable (for example, blood flow [1], nano particles [29], combustion inside an engine [23], flow in zero or low gravity [14]). OpenFOAM features a wide range of applications and a continuously growing user base in both academia and industry. The source code is available in C++.

Adjoint methods are crucial ingredients of state of the art gradient-based solvers for high dimensional optimization problems. Their advantage over finite difference approximation is twofold. The computational cost of calculating the gradient of some cost function $J = J(\boldsymbol{\alpha}) \in \mathbb{R}$ (see section 5 for an example) with respect to parameters $\boldsymbol{\alpha} \in \mathbb{R}^n$ is independent of $n$. An ideally small constant multiple of the cost of a primal flow simulation followed by the evaluation of the objective is required. Moreover, the gradient is obtained with machine accuracy. Truncation is avoided altogether eliminating unwanted numerical effects in finite precision floating-point arithmetic. The *discrete adjoint model* uses Algorithmic Differentiation whereas the *continuous adjoint model* relies on analytically obtained adjoint equations which have to be solved alongside the primal (e.g. Navier-Stokes) equations; see also [17]. For a given discretization of the computational domain (into a potentially very large number of $n$ individual *cells*) topology optimization [3, 21] penalizes cells which are undesirable for the flow pattern. This penalization is realized by adding a source term $-\alpha\mathbf{v}$ to the $k$D ($k \in \{2, 3\}$) Navier-Stokes equations, thus blocking cells with high values of impermeability $\alpha$ for the flow:

$$\mathbf{v} \cdot \nabla\mathbf{v} = \nu\nabla^2\mathbf{v} - \frac{1}{\rho}\nabla p - \alpha\mathbf{v} \tag{1}$$

$$\nabla \cdot \mathbf{v} = 0 \quad , \tag{2}$$

where $\mathbf{v} \in \mathbb{R}^k$ and $\nu, \rho, p, \alpha \in \mathbb{R}$. The discretized [8] equations can then be solved for the velocity field $U \in \mathbb{R}^{k\times n}$ and pressure field $\mathbf{p} \in \mathbb{R}^n$ by a suitable method such as the SIMPLE-Algorithm [22]. Gradient information $\partial J/\partial\boldsymbol{\alpha} \in \mathbb{R}^n$ becomes necessary to determine an $\boldsymbol{\alpha}$ which minimizes a given cost functional $J$. Given the gradient one can iteratively refine $\boldsymbol{\alpha}$ from an initial guess, for example, by applying steepest descent

$$\boldsymbol{\alpha}^{n+1} = \boldsymbol{\alpha}^n - \lambda \cdot \frac{\partial J}{\partial \boldsymbol{\alpha}^n}$$

with local line search parameter $\lambda$.

# 3 Building Blocks

Discrete adjoint OpenFOAM uses Algorithmic Differentiation (AD) [12, 16] implemented by our AD overloading tool dco/c++[2] [18]. The Adjoint MPI library AMPI extends the implied data flow reversal [15] to the Message Passing Interface (MPI).[3]

## 3.1 Algorithmic Differentiation

We consider the optimization problem $\operatorname{argmin}_{\boldsymbol{\alpha}} J(\boldsymbol{\alpha})$ for $J : \mathbb{R}^n \to \mathbb{R}$, where each function evaluation $J(\boldsymbol{\alpha})$ comprises the solution of the discrete $k$D Navier-Stokes equations forming a very large system of parameterized nonlinear equations [19]. Our notation is based on [16].

[2] `https://www.stce.rwth-aachen.de/software/dco_cpp.html`
[3] `http://www.mpi-forum.org/`

First-order AD assumes $J$ to be at least once continuously differentiable at all points of interest.[4] For a given implementation of the primal objective $y = J(\boldsymbol{\alpha})$, a corresponding (first-order) tangent code computes a directional derivative

$$y^{(1)} = J^{(1)}(\boldsymbol{\alpha}, \boldsymbol{\alpha}^{(1)}) \equiv < \nabla J, \boldsymbol{\alpha}^{(1)} > \quad ,$$

with scalar product $< ., . >$ and where $\boldsymbol{\alpha}^{(1)} \in I\!R^n$, $y^{(1)} \in I\!R$, and $\nabla J = \nabla J(\boldsymbol{\alpha}) \equiv \frac{\partial J}{\partial \boldsymbol{\alpha}} \in I\!R^n$ denotes the gradient of $J$. A (first-order) adjoint code computes

$$\boldsymbol{\alpha}_{(1)} = J_{(1)}(\boldsymbol{\alpha}, y_{(1)}) \equiv \nabla J^T \cdot y_{(1)},$$

where $\boldsymbol{\alpha}_{(1)} \in I\!R^n$ and $y_{(1)} \in I\!R$. The gradient can be obtained in tangent mode at the computational cost of $O(n) \cdot Cost(J)$, where $Cost(J)$ denotes the computational cost of a single evaluation of $J$. The same gradient can be obtained in adjoint mode at $O(1) \cdot Cost(J)$, which typically amounts to $1 - 100$ times the cost of a single evaluation of the objective. The actual factor depends on various parameters including the mode of differentiation (continuous vs. discrete adjoint), the expertise of the adjoint code developer, and the quality of the AD software tool, if one is used.

For topology optimization we use the adjoint model, because a typically very large number of inputs (all elements of $\boldsymbol{\alpha} \in I\!R^n$) are mapped onto a single output. Second and higher derivatives can be obtained by repeatedly applying tangent mode AD to existing tangent or adjoint code; see [16] for details.

Conceptually, AD is based on the fact that the given implementation of the primal objective as a computer program can be decomposed at run time into a *single assignment code*

$$\begin{aligned} &\text{for } j = n, \ldots, n+p \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \quad , \end{aligned}$$

where $i \prec j$ denotes a direct dependence of the variable $v_j$ on $v_i$. The result of each *elemental function* $\varphi_j$ is assigned to a unique auxiliary variable $v_j$. The $n$ *independent inputs* $x_i = v_i$, for $i = 0, \ldots, n-1$, are mapped onto the *dependent output* $y = v_{n+p}$. The values of $p$ *intermediate variables* $v_k$ are computed for $k = n, \ldots, n+p-1$.

The primal code is augmented with instructions for storing data which is required for the reversal of the data flow and for the computation of the local partial derivatives $\frac{\partial \varphi_j}{\partial v_i}$, for $j = n, \ldots, n+p$ and $i \prec j$. A data structure commonly referred to as *tape* is used for this purpose. This *(augmented) forward section* of the adjoint code is succeeded by the *reverse section* propagating adjoints for all $v_i$ in reverse order, that is, for $i = n+p-1, \ldots, 0$ :

$$\begin{aligned} &\left.\begin{aligned} &\text{for } j = n, \ldots, n+p+m-1 \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \end{aligned}\right\} \text{forward section (records tape)} \\ &\left.\begin{aligned} &\text{for } i = n+p-1, \ldots, 0 \\ &\quad v_{(1)i} = \sum_{j : i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_{(1)j} \end{aligned}\right\} \text{reverse section (interprets tape)} \end{aligned} \tag{3}$$

Note that the $v_j$ computed in the forward section are potentially required as arguments of local

[4] Generalizations of the theory of AD to non-differentiable functions has been the subject of ongoing research for several years; see, for example, [12].

partial derivatives within the reverse section. They are read in reverse with respect to the original order of their evaluation. The additional persistent memory requirement of the adjoint code becomes $O(n+p)$. The efficient reversal of the data flow is among the main challenges in adjoint AD. It is responsible for black-box adjoint AD typically not being applicable to large-scale numerical simulations. The available persistent memory may simply not be large enough [15].

## 3.2 Discrete Adjoint OpenFOAM

Discrete adjoint OpenFOAM is obtained by application of our overloading AD tool dco/c++ to OpenFOAM (current version 2.3.x); see [27] for details. Recent improvements include the *symbolic differentiation of the linear solvers* [9] as well as *reverse accumulation* [6]. Most importantly, the memory footprint of discrete adjoint OpenFOAM can be adapted to the specifics of the given compute platform through the use of *checkpointing* [11].

## 3.3 Adjoint MPI

MPI is the de-facto standard for writing massively parallel programs on modern distributed memory high performance computing (HPC) architectures. To ensure performance and scalability the goal of the programmer should be to keep as many computations as possible local and to communicate the minimal amount data which needs to be shared between two or multiple processes by passing corresponding messages. Adjoint MPI (AMPI) is a library which wraps MPI in order to make adjoint AD applicable to numerical simulation programs that contain MPI communication. Its ongoing development started in 2009 [28] based on theoretical work about the correctness of adjoint message passing [20] and with various improvements and extensions added over time [24, 25, 26].

A growing subset of the MPI routines have been overloaded to facilitate the reversal of the data flow for communication of adjoints. In particular, AMPI supports the message passing patterns used in OpenFOAM, namely, `MPI_Bsend`, `MPI_Recv`, and `MPI_Allreduce`. It provides the corresponding routines `AMPI_Bsend`, `AMPI_Recv`, and `AMPI_Allreduce`. Their semantics depend on their calling context; see Table 1. For example, `AMPI_Bsend` passes the primal values from the sender to the receiver in the forward section of the adjoint code by calling `MPI_Bsend`. In the reverse section adjoints need to be extracted from the primal receiver by calling `MPI_Recv`.

In addition to dco/c++ AMPI can be interfaced with other AD tools such as ADOL-C [10] and Tapenade [13]. Related work by others includes [2, 4, 5, 7].

| Primal MPI-Call | Adjoint AMPI-Call | MPI-Call in forward run | MPI-Call in reverse run |
|---|---|---|---|
| `MPI_Bsend` | `AMPI_Bsend` | `MPI_Bsend` | `MPI_Recv` |
| `MPI_Recv` | `AMPI_Recv` | `MPI_Recv` | `MPI_Bsend` |
| `MPI_Allreduce` | `AMPI_Allreduce` | `MPI_Allreduce` | `MPI_Allreduce` |

Table 1: MPI routines used in OpenFOAM, their AMPI versions, and communication patterns specific to the forward and reverse sections of the adjoint code

# 4 MPI-Parallel Discrete Adjoint OpenFOAM

With all the building blocks from Section 3 in place the actual transition to MPI-parallel discrete adjoint OpenFOAM turned out to be rather straight forward. Nevertheless, dealing with a highly complex C++ source code always adds a considerable amount of complication at the technical level. As usual, well designed components and modular software architecture are crucial for successful and sustained numerical simulation.

## 4.1 MPI-Parallelism in OpenFOAM

OpenFOAM features a very object oriented design with distributed memory parallelism in mind. All calls to MPI are encapsulated in the so-called `Pstream` library. An object to be passed from the *sender* to the *receiver* is serialized into a contiguous block of data stripped of its type information (the receiver knows what type of data to expect). Hence all data can be passed as `MPI_BYTE`, thus reducing the complexity of the low-level communication methods. Unfortunately, communication of dco/c++ data requires type information, as AMPI needs to know whether the passed data is *active* (a dco/c++ derivative type) or *passive* (for example, `int`, `string`).

## 4.2 Coupling OpenFOAM with AMPI

To enable the communication of active data we augment all calls to MPI send and receive routines to also include the type information. Thus, we are able to distinguish between active and passive data being passed. For active data a transfer as `MPI_BYTE` is no longer possible as the data layout of the primal values is no longer contiguous. Moreover, AMPI needs to track source and target of each message in order to enable the reversal of the data flow for passing adjoints from the target back to the source. This functionality is provided by the data type `AMPI_DOUBLE`.

OpenFOAM uses the previously mentioned set of basic MPI routines. More complex operations such as broadcasts are implemented in OpenFOAM in terms of these basic MPI routines. Thus inside the `Pstream` library only four different source files have to be patched in order to change all relevant MPI calls into the corresponding AMPI calls:
`Pstream/mpi/UPstream.C`: `MPI_Init()` → `AMPI_Init()`;
`Pstream/mpi/UOPwrite.C`: `MPI_Bsend()` → `AMPI_Bsend()`;
`Pstream/mpi/UIPread.C`: `MPI_Recv()` → `AMPI_Recv()`;
`Pstream/mpi/allReduceTemplates.C`: `MPI_Allreduce()` → `AMPI_Allreduce()`.
As pointed out earlier for the `MPI_Send` / `MPI_Recv` calls it has to be decided if an active or passive version is to be used. This decision is made depending on the incoming data type (floating-point data is considered as active; other data, such as `int` or `string`, is considered as passive). Moreover, repeated passive runs of the primal computation are performed if checkpointing is used. In this case passive floating-point data needs to be communicated.

Feasibility of the proposed solution is illustrated with the help of a case study in the following Section.

# 5 Case Study

As a case study we consider a 90-degree duct whose geometry can conveniently be modified by specifying parameters L1, L2, R1 and R2. The problem is scaled to different mesh sizes by an

automatic meshing algorithm which depends only on a mesh density parameter. We run the calculation on different numbers of cores and observe the scaling of run time and peak memory consumption per process.

## 5.1 Geometry, Boundary Conditions, and Solver Settings

The geometry of the computational domain is shown in Figure 1. A cost functional $J$ is defined

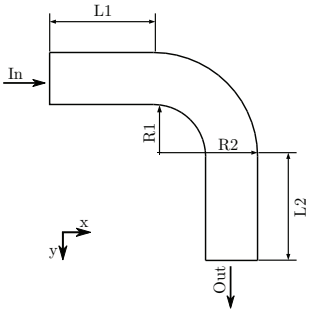


Figure 1: Sketch of the Geometry. In our case $L1 = L2 = 5\,\mathrm{m}$, $R1 = 1\,\mathrm{m}$, $R2 = 2\,\mathrm{m}$.

to minimize the total dissipation of the system:

$$J = -\int_\Gamma \left(p + \frac{1}{2}\mathbf{v}^2\right)\mathbf{v}\cdot\mathbf{n}\;\mathrm{d}\Gamma \quad ,$$

where $\Gamma \subset D$ denotes the part of the domain boundary corresponding to in- (In) and outlet (Out) and with the following boundary conditions: on inlet fixed velocity $\|\mathbf{v}_{\mathrm{in}}\| = 15\,\mathrm{m/s}$ and vanishing change of pressure in $x$ direction $\frac{\partial p}{\partial x} = 0$; on outlet fixed pressure[5] $p_{\mathrm{out}} = 0\,\mathrm{m}^2/\mathrm{s}^2$ and vanishing change of velocity in $y$ direction $\frac{\partial \mathbf{v}}{\partial y} = 0$; no-slip walls $\mathbf{v}|_{\Gamma\setminus(\mathrm{In}\cup\mathrm{Out})} = 0$. The viscosity is given as $\nu = 1.5 \times 10^{-5}\,\mathrm{m}^2/\mathrm{s}$ yielding a Reynolds Number of $\mathrm{Re} = 10^6$. A total of 300 iterations of the SIMPLE Algorithm were initialized by a potential flow solution. Binomial checkpointing (using revolve [11]) with 60 checkpoints was used for calculation of the adjoints yielding 538 passive (primal) and 300 active (adjoint by recording and interpretation of the corresponding section of the tape) iterations.

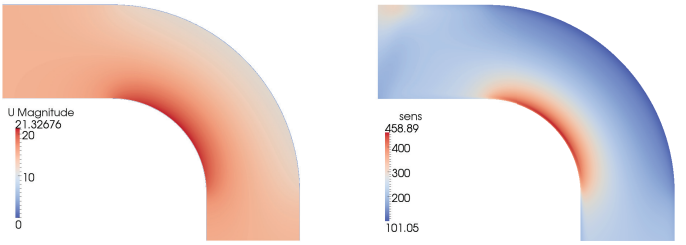


Figure 2: Results after 300 iterations of the SIMPLE algorithm (solution fully converged): Velocity field $U$ on the left and corresponding sensitivities of the objective with respect to the discrete impermeability parameter $\frac{\partial J}{\partial \boldsymbol{\alpha}}$ on the right.

[5] the pressure in OpenFOAM is scaled by the inverse density of the fluid yielding the unit $\mathrm{m}^2/\mathrm{s}^2$

## 5.2 Results on Shared Memory Machine

First we ran the test case on a shared memory machine (2x Intel® Xeon® E5-2630 @ 2.30GHz (12 cores), 128 GB RAM). A 2D mesh consisting of 260,000 hexaedra was build. The resulting flow field and the corresponding sensitivities are visualized using ParaView[6] in Figure 2. Red regions indicate high sensitivity of the objective with respect to changes in the local impermeability. The impermeability in blue regions has less impact on $J$. Absence of regions with negative sensitivity indicate missing need for placement of material to obstruct the flow in the context of topology optimization. The geometry turns out to be optimal for the chosen objective under the given constraints.

Scaling in terms of run time and memory consumption is observed in Tables 2 and 3. The overhead of the adjoint varies between 10 and 20 which is a good result considering that the adjoint run includes 538 additional primal evaluations due to checkpointing and no further optimizations have been applied. Without checkpointing the total memory requirement of the adjoint would amount to infeasible 15 TB. The speedup for the primal (no adjoints are computed, OpenFOAM version from the official repository) run is comparable with the speedup of the adjoint run. The adjoint version performs slightly worse than the primal due to the additional AMPI communications. Moreover, the tape interpretation is usually memory bandwidth bound. Hence, it cannot scale perfectly because all processes share the same memory bus.

Figure 3 counts the additional MPI calls performed due to adjoint communication. For one adjoint SIMPLE step the number of communication calls is roughly doubled (for each MPI statement in the forward section another one is executed in the reverse section). Further passive MPI calls result from additional runs of primal iterations due to checkpointing. Both the numbers of passive and active MPI calls triple for each duplication of the number of processes. For larger numbers of processes the number of MPI calls per process is likely to stagnate as not every process has to communicate with every other process but with a limited subset only.

Table 3 indicates that the memory consumption per process scales inversely with the number of processes. The additional memory consumption of the adjoint is dominated by the tape whose size scales directly with the number of floating point operations. However, the decomposition of the problem into multiple regions lowers the dimension of the local linear systems to be solved and hence the number of floating point operations performed per process.

## 5.3 Results on RWTH Compute Cluster

The same test case with a finer mesh (723,000 cells) was run on the RWTH Aachen University Compute Cluster[7]. For the purpose of testing, the mesh size was set to enable the serial execution within the available memory of the target system (one node consists of 2x Intel® Xeon® X5675 @ 3.07 GHz CPUs (12 cores), 96 GB of RAM; InfiniBand between nodes). The number of processes per node was fixed to two in order to investigate a fully distributed problem. Again, 300 SIMPLE iterations with 60 checkpoints were performed. Larger simulations require some administrative overhead in order to gain exclusive access to a larger partition of the RWTH Compute Cluster. They are the subject of ongoing work.

Run time and memory consumption for this configuration are illustrated in Figure 4. We observe a run time scaling behavior of the adjoint which is similar to the primal. Again, the adjoint increases the run time relative to the primal by a factor between 10 and 20. We believe that the increased memory bandwidth associated with running on two different cores on the same node is responsible for the superlinear scaling of the adjoint on two processes. Remember

[6] http://www.paraview.org

[7] `https://doc.itc.rwth-aachen.de/display/CC/Home`

| | primal | | adjoint | | run time factor |
|---|---|---|---|---|---|
| $n$ | run time [s] | speedup | run time [s] | speedup | primal / adjoint |
| 1 | 423 | 1.0 | 6245 | 1.0 | 14.8 |
| 2 | 200 | 2.1 | 3030 | 2.1 | 15.2 |
| 4 | 134 | 3.2 | 2194 | 2.8 | 16.4 |
| 8 | 76 | 5.6 | 1316 | 4.7 | 17.3 |
| 12 | 63 | 6.7 | 1085 | 5.8 | 17.2 |

Table 2: Run time scaling of primal and adjoint simulation on our shared memory machine for increasing numbers of processes $n$.

| | primal | | adjoint | |
|---|---|---|---|---|
| $n$ | max. memory [MB] | factor | max. memory [MB] | factor |
| 1 | 402 | 1.0 | 49895 | 1.0 |
| 2 | 222 | 1.8 | 24126 | 2.1 |
| 4 | 133 | 3.0 | 14709 | 3.4 |
| 8 | 86 | 4.7 | 6491 | 7.7 |
| 12 | 71 | 5.7 | 4847 | 10.3 |

Table 3: Scaling of memory consumption per rank of primal and adjoint simulation on our shared memory machine for increasing numbers of processes $n$.

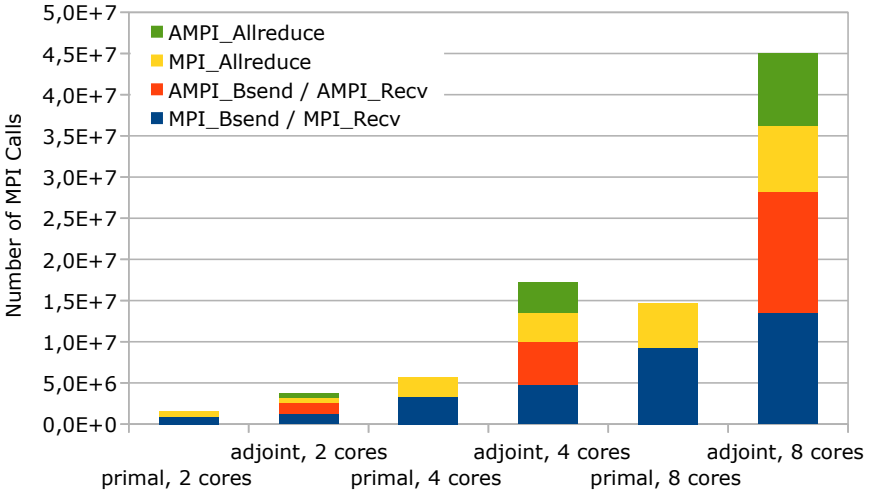


Figure 3: Number of MPI calls in primal and adjoint runs of OpenFOAM.

that the run time of the tape interpretation is usually memory bandwidth bound. Scaling in terms of memory consumption turns out to be near optimal. We observe a 30% rise in total memory usage when going from 1 to 128 CPUs. The memory consumption of the adjoint is dominated by the tape. Primal scaling stalls for $n \geq 32$ processes. The workload becomes insignificant and run time and memory consumption are dominated by constant factors due to startup time, I/O of the mesh, and case setup. The adjoint scales better due to the substantially higher work load per process. Again, larger tests with higher numbers of processes are the subject of ongoing activities.

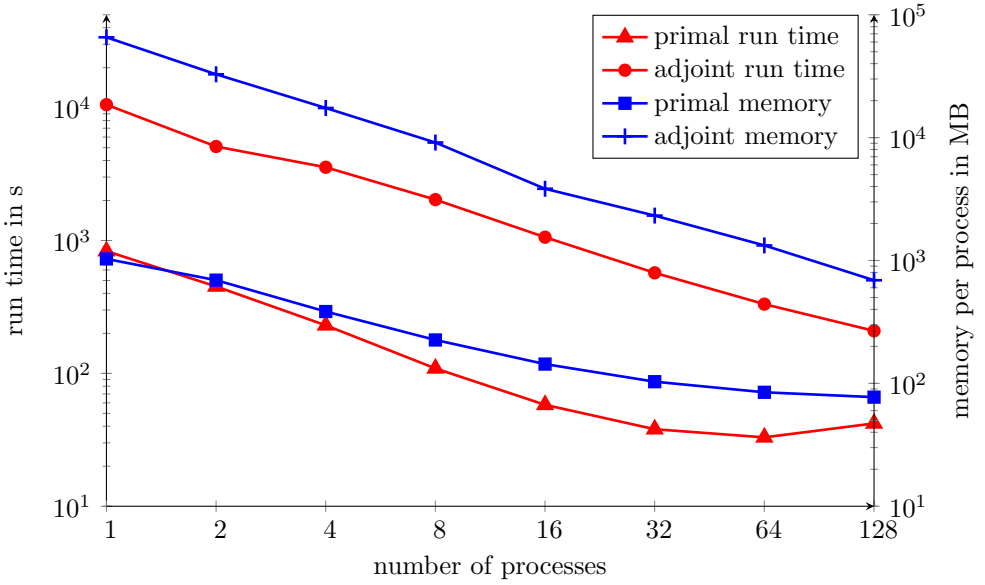


Figure 4: Run time and memory scaling on RWTH Compute Cluster

# 6 Conclusion and Outlook

An MPI-parallel discrete adjoint version of the CFD tool box OpenFOAM was presented. Our test results indicate reasonable scaling in terms of run time and next to ideal scaling in terms of memory consumption. Ongoing work targets larger instances of real-world applications for increased numbers of processes. Industrial-size problems can only be solved by combining modern HPC architectures (parallel computation and I/O) with mathematical and structural insight into the given numerical simulation. Adjoint simulations are likely to remain a considerable challenge for the foreseeable future. The ongoing rapid evolution of HPC hard- and software adds a level of technical complication not to be underestimated.

# References

[1] M. J. Behbahani, M. Behr, M. Hormes, U. Steinseifer, D. Arora, O. Coronado, and M. Pasquali. A review of computational fluid dynamics analysis of blood pumps. *European Journal of Applied Mathematics*, 20(4):363–397, 2009.

[2] J. Benary. Parallelism in the reverse mode. In *Computational Differentiation: Techniques, Applications, and Tools*, pages 137–147. SIAM, Philadelphia, PA, 1996.

[3] T. Borrvall and J. Petersson. Topology optimization of fluids in Stokes flow. *International Journal for Numerical Methods in Fluids*, 41(1):77–107, 2003.

[4] A. Carle and M. Fagan. Automatically differentiating MPI-1 datatypes: The complete story. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 25, pages 215–222. Springer, 2002.

[5] B. Cheng. A duality between forward and adjoint MPI communication routines. In *Computational Methods in Science and Technology*, pages 23–24. Polish Academy of Sciences, 2006.

[6] B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3(4):311–326, 1994.

[7] C. Faure, P. Dutto, and S. Fidanova. Odysée and parallelism: Extension and validation. In *Procceedings of The 3rd European Conference on Numerical Mathematics and Advanced Applications, Jyväskylä, Finland, July 26-30, 1999*, pages 478–485. World Scientific, 2000.

[8] J. H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, 1999.

[9] M. B. Giles. Collected matrix derivative results for forward and reverse mode algorithmic differentiation. In *Advances in Automatic Differentiation*, pages 35–44. Springer, 2008.

[10] 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.

[11] A. Griewank and A. Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, March 2000.

[12] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, chapter 6. SIAM, 2008.

[13] L. Hascoët and V. Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software*, 39(3):20:1–20:43, 2013.

[14] H. K. Nahra and Y. Kamotani. Prediction of bubble diameter at detachment from a wall orifice in liquid cross-flow under reduced and normal gravity conditions. *Chemical Engineering Science*, 58(1):55 – 69, 2003.

[15] U. Naumann. DAG reversal is NP-complete. *Journal of Discrete Algorithms*, 7:402–410, 2009.

[16] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation.*, chapter 1&2. SIAM, 2012.

[17] U. Naumann. Executive summary. In N. Gauger, M. Giles, M. Gunzburger, and U. Naumann, editors, *Adoint Methods in Computational Science, Engineering, and Finance*, pages 1–3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2014.

[18] U. Naumann, K. Leppkes, and J. Lotz. dco/c++ user guide. Technical Report AIB-2014-03, RWTH Aachen, January 2014.

[19] U. Naumann, J. Lotz, K. Leppkes, and M. Towara. Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations. *ACM Transactions on Mathematical Software*, 2015. To appear.

[20] U. Naumann, J. Utke, J. Riehme, P. Hovland, and C. Hill. A framework for proving correctness of adjoint message passing programs. In *Proceedings of EUROPVM/MPI 2008*, pages 316–321, 2008.

[21] C. Othmer. A continuous adjoint formulation for the computation of topological and surface sensitivities of ducted flows. *International Journal for Numerical Methods in Fluids*, 58(8):861–877, 2008.

[22] S. V. Patankar and D.B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *Int. J. of Heat and Mass Transfer*, 15(10):1787–1806, 1972.

[23] R.D. Reitz and C.J. Rutland. Development and testing of diesel engine CFD models. *Progress in Energy and Combustion Science*, 21(2):173 – 196, 1995.

[24] M. Schanen, M. Förster, and U. Naumann. Second-order algorithmic differentiation by source transformation of MPI code. In *Recent Advances in the Message Passing Interface*, volume 6305 of *Lecture Notes in Computer Science*, pages 257–264. Springer, 2010.

[25] M. Schanen and U. Naumann. A wish list for efficient adjoints of one-sided MPI communication. In *Recent Advances in the Message Passing Interface*, pages 248–257. Springer, 2012.

[26] M. Schanen, U. Naumann, L. Hascoët, and J. Utke. Interpretative adjoints for numerical simulation codes using MPI. In *Proceedings of ICCS2010*, pages 1825 – 1833. 2010.

[27] M. Towara and U. Naumann. A discrete adjoint model for OpenFOAM. *Procedia Computer Science*, 18(0):429 – 438, 2013. 2013 International Conference on Computational Science.

[28] J. Utke, L. Hascoët, C. Hill, P. Hovland, and U. Naumann. Toward adjoinable MPI. In *Proceedings of IPDPS 2009*, 2009.

[29] L. Wang and R. O. Fox. Comparison of micromixing models for CFD simulation of nanoparticle formation. *AIChE Journal*, 50(9):2217–2232, 2004.