

Enhancing OpenFOAM's Performance on HPC Systems



Thorsten Zirwes, Feichi Zhang, Jordan A. Denev, Peter Habisreuther, Henning Bockhorn, and Dimosthenis Trimis

Abstract OpenFOAM is one of the most popular open source tools for CFD simulations of engineering applications. It is therefore also often used on supercomputers to perform large eddy simulations or even direct numerical simulations of complex cases. In this work, general guidelines for improving OpenFOAM's performance on HPC clusters are given. A comparison of the serial performance for different compilers shows that the Intel compiler generally generates the fastest executables for different standard applications. More aggressive compiler optimization options beyond O3 yield performance increases of about 5% for simple cases and can lead to improvements of up to 25% for reactive flow cases. Link-time optimization does not lead to a performance gain. The parallel scaling behavior of reactive flow solvers shows an optimum at 5000 cells per MPI rank in the tested cases, where caching effects counterbalance communication overhead, leading to super linear scaling. In addition, two self-developed means of improving performance are presented: the first one targets OpenFOAM's most accurate discretization scheme "cubic". In this scheme, some polynomials are unnecessarily reevaluated during the simulation. A simple change in the code can reuse the results and achieve performance gains of about 5%. Secondly, the performance of reactive flow solvers is investigated with Score-P/Vampir and load imbalances due to the computation of the chemical reaction rates are identified. A dynamic-adaptive load balancing approach has been implemented for OpenFOAM's reacting flow solvers which can decrease computation times by 40% and increases the utilization of the HPC hardware. This load balancing approach utilizes the special feature of the reaction rate computation, that no information of neighboring cells are required, allowing to implement the load balancing efficiently.

T. Zirwes (✉) · J. A. Denev
Steinbuch Centre for Computing, Karlsruhe Institute of Technology,
Hermann-von-Helmholtz-Platz 1, Karlsruhe, Germany
e-mail: thorsten.zirwes@kit.edu

F. Zhang · P. Habisreuther · H. Bockhorn · D. Trimis
Engler-Bunte-Institute, Karlsruhe Institute of Technology, Engler-Bunte-Ring 7, Karlsruhe,
Germany
e-mail: feichi.zhang@kit.edu

Keywords OpenFOAM · Load balancing · Reactive flows · Performance optimization · Combustion

1 Introduction

Computational Fluid Dynamics (CFD) has proven to be an effective tool in engineering to aid the development of new devices or for gaining a deeper understanding of physical phenomena and their mutual interactions. One of the most popular open-source CFD tools is OpenFOAM [1]. It provides many tools for simulating different phenomena: from simple, incompressible flows to multi-physics cases of multi-phase reactive flows. In the case of combustion simulations, the interaction of chemical reactions and turbulent flow takes place on a large range of time and length scales [2, 3]. Therefore, supercomputers have to be employed to resolve all the multi-scale interactions and to gain deeper insight into the combustion dynamics by performing highly-resolved simulations.

In order to utilize the hardware provided by High Performance Computing (HPC) clusters efficiently, it is not sufficient to install OpenFOAM with its default settings. In this work, a general guideline on running OpenFOAM efficiently on supercomputers is given. Section 2 reviews the performance of OpenFOAM depending on the choice of compiler and the optimization settings as well as link-time optimization. Sections 2 and 5 give examples of the parallel scaling behavior. In the case of reactive flows, an optimal number of cells per MPI rank is discussed.

In addition to these guidelines, two self-developed methods for speeding up general OpenFOAM applications are discussed. Since the OpenFOAM applications on supercomputers are typically highly-resolved direct numerical simulations, it is likely that users will choose OpenFOAM's "cubic" discretization for spatial derivatives since it is the most accurate scheme OpenFOAM offers. In the implementation of this scheme, some polynomials are computed repeatedly which can be avoided. The necessary code changes and performance benefits are shown in Sect. 3 and the accuracy of the scheme is demonstrated with the well-known Taylor-Green vortex case [4].

The second self-developed method for improving OpenFOAM's performance improves the load balancing of reactive flow solvers. Since chemical reaction rates depend exponentially on temperature and are closely coupled with each other, they tend to be numerically stiff. This requires special treatment for the computation of the chemical reaction rates which can lead to large load imbalances. Section 4 presents the load balancing method which is done dynamically during the simulation. An application of this is given in Sect. 5, where a flame evolving in the Taylor-Green vortex case from Sect. 3 is studied.

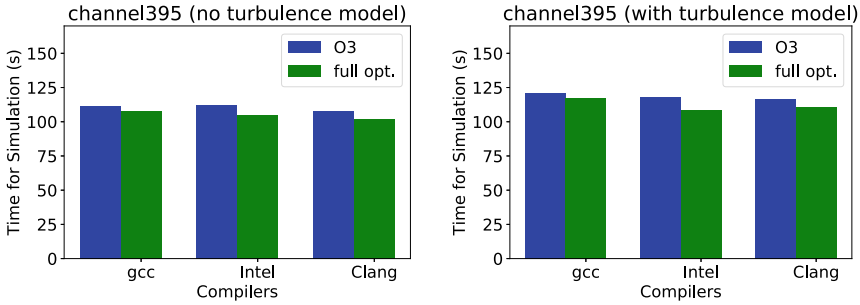


Fig. 1 Time required to compute the tutorial case `channel395` depending on the compiler and optimization settings. Simulation without turbulence model (left) and with turbulence model (right)

2 Compilers and Optimization Settings

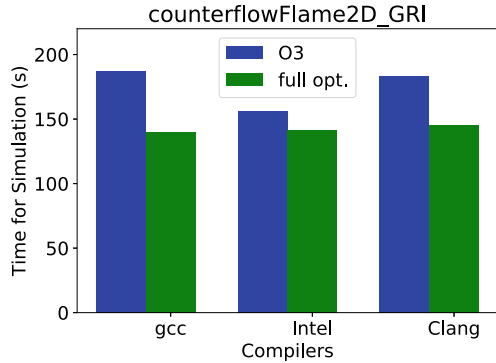
If OpenFOAM is compiled without changing any settings, it will be compiled with the `gcc` compiler. The only optimization flags given to the compiler is `O3`. This raises the question if OpenFOAM can benefit from using different compilers and more aggressive optimization options since `O3` does not generate code specifically optimized for the CPU type of the cluster. In this section, two different cases are considered:

- incompressible flow: the standard tutorial case `channel395` computed with OpenFOAM's `pimpleFoam` solver
- compressible reactive flow: the standard tutorial case `counterflowFlame_2D_GRI` computed with OpenFOAM's `reactingFoam` solver

These two cases differ in their physical complexity. For the incompressible flow case, only two conservation equations have to be solved (mass and momentum). Also, all fluid properties are assumed to be constant. The reactive flow case on the other hand simulates the chemical reactions of 52 different species. Therefore, conservation equations have to be solved for mass, momentum, energy and 51 chemical species. In addition to this, all fluid properties like density or heat capacity are computed as a function of gas mixture composition, temperature and pressure. Lastly, a large part of the simulation time stems from computing the chemical reaction rates based on the GRI3.0 [5] reaction mechanism.

Figure 1 shows the simulation time required to compute 100s of OpenFOAM's tutorial case `channel395` with OpenFOAM's `pimpleFoam` solver. The figure on the left shows the case without turbulence model and the case with turbulence model, which means that additional transport equations for the turbulence model are solved. Three different compilers are investigated: `gcc 8`, `Intel2018` and `Clang 7`. Additionally, OpenFOAM was compiled using the default optimization setting `O3` and more aggressive optimizations (full opt.). For `gcc` and `Clang`, the full optimization settings are `-Ofast -ffast-math -fno-rounding-math -march=native -mtune=native` and for the `Intel` compiler `-O3 fp-`

Fig. 2 Time required to compute the first 0.003 s of the tutorial case `counterflowFlame2D_GRI` depending on the compiler and optimization settings



`model fast=2 -fast-transcendentals -xHost`. All timings have been recorded on the ForHLR II cluster at KIT [6] with OpenFOAM 5.x in serial.

The difference between the three compilers is approximately 3%. The fastest code using only O3 as optimization option is generated by Clang, while the fastest code using the full optimization options is generated by the Intel compiler. Changing from the standard O3 to the full optimization options yields a performance increase of about 5% on average. Including the turbulence model does not change the timings significantly because most of the time is spent in the pressure correction step.

For the reactive flow case however (shown in Fig. 2), the difference between the O3 and the fully optimized build becomes significant. For the gcc compiler, switching from O3 to the full optimizations increases the performance by about 25%. The Intel compiler generates efficient code even with O3, so that the full optimizations increase the performance by about 10% for the Intel compiler. Although there are large differences between the compilers with O3, they perform approximately the same using the full optimizations.

Using the full optimization settings as described before has not shown any drawbacks in accuracy of the simulation results. Final temperature profiles for the reactive flow case for example agreed within 0.1% between the O3 build and the fully optimized build. It should be noted however, that other unsafe optimizations like strict aliasing has caused OpenFOAM to randomly crash. Therefore, using additional options to the ones shown above should be used with caution. It has also been found that link-time optimization (LTO) has no effect on performance at all. This is probably due to OpenFOAM being compiled into many different shared objects so that LTO cannot be very effective.

3 Optimizing the Cubic Discretization

OpenFOAM offers a large number of spatial discretization schemes to discretize spatial gradients for the finite volume method. Since CFD simulations on super-

Original Implementation	Optimized Implementation
<pre> 1 const surfaceScalarField kSc 2 { 3 lambda*(scalar(1) - lambda*(scalar(3) 4 - scalar(2)*lambda)) 5 }; 6 7 const surfaceScalarField kVecP 8 { 9 sqr(scalar(1) - lambda)*lambda 10 }; 11 const surfaceScalarField kVecN 12 { 13 sqr(lambda)*(lambda - scalar(1)) 14 }; </pre>	<pre> if (kScGlob.get() == nullptr mesh.changing()) { kScGlob.reset(new surfaceScalarField(lambda*(scalar(1) - lambda*(scalar(3) - scalar(2)*lambda))); kVecPGlob.reset(new surfaceScalarField(sqr(scalar(1) - lambda)*lambda)); kVecNGlob.reset(new surfaceScalarField(sqr(lambda)*(lambda - scalar(1)))); } surfaceScalarField& kSc = *kScGlob; surfaceScalarField& kVecP = *kVecPGlob; surfaceScalarField& kVecN = *kVecNGlob; </pre>

Fig. 3 Original code for computing the cubic discretization scheme from `cubic.H` from OpenFOAM (left) and optimized code version (right)

computers are usually highly-resolved, discretization schemes with high accuracy should be employed. The most accurate scheme OpenFOAM offers is the `cubic` scheme. It interpolates a quantity ϕ from the cell centers of the current cell C and the neighboring cell N to the cell face f using a third order polynomial:

$$\phi_f = \lambda\phi_C + (1 - \lambda)\phi_N + A(\phi_N - \phi_C) + B\nabla\phi_C + C\nabla\phi_N \quad (1)$$

Here, λ is a function of the distance between the cell centers. The coefficients A , B , C are themselves polynomials of λ :

$$A = 2\lambda^3 - 2\lambda^2 + \lambda \quad (2)$$

$$B = -\lambda^3 + \lambda^2 \quad (3)$$

$$C = -\lambda^3 + 2\lambda^2 - \lambda \quad (4)$$

Since these polynomials only depend on the distances between the cell centers, they only have to be computed once for each simulation if the computational mesh does not change. In OpenFOAM's implementation however, these polynomials are computed at every time step and for every transport equation. So for example if the conservation of momentum and energy use the `cubic` scheme, these polynomials are computed twice per time step. In order to avoid this, OpenFOAM's code of the cubic discretization can easily be modified so that these polynomials are computed only once at the beginning of the simulation.

Figure 3 on the left shows the code from OpenFOAM which computes the three polynomials. The code on the right is the modification. It starts by checking if the polynomials have already been computed. If not, they are computed only once. They are also re-computed if the mesh is changing during the simulation. If the polynomials have already been computed, references to the precomputed values are used on the subsequent computations.

The performance gain of this new implementation is tested with the well-known Taylor-Green vortex [4] using OpenFOAM's `pimpleFoam` solver. The case consists of a cube with side length $2\pi L$ where all boundary conditions are periodic. The

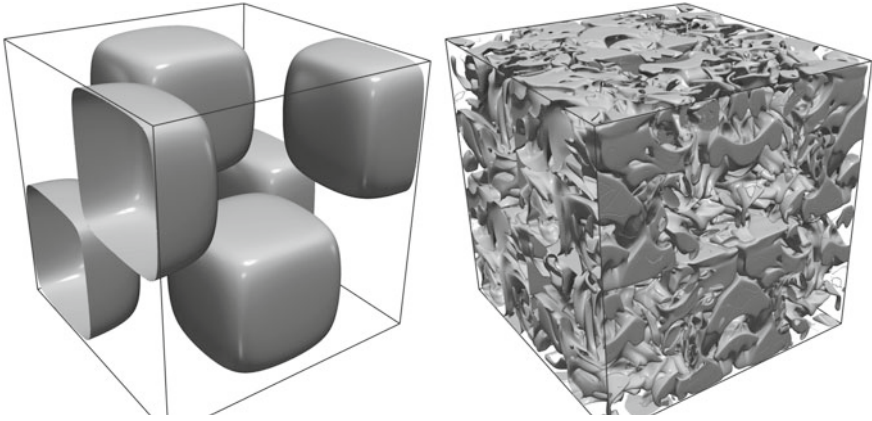


Fig. 4 Vorticity iso-surface of 0.1 s^{-1} at the start of the simulation (left) and after 20 s (right)

velocity profile at the start of the simulation is set to:

$$u_x = u_0 \sin\left(\frac{2\pi x}{L}\right) \cos\left(\frac{2\pi y}{L}\right) \cos\left(\frac{2\pi z}{L}\right) \quad (5)$$

$$u_y = -u_0 \cos\left(\frac{2\pi x}{L}\right) \sin\left(\frac{2\pi y}{L}\right) \cos\left(\frac{2\pi z}{L}\right) \quad (6)$$

$$u_z = 0 \quad (7)$$

This places counter-rotating vortices at each corner. The flow field then starts to become turbulent and decays over time. This is shown in Fig. 4.

By using the new implementation for the cubic scheme, which requires only minimal code changes, the total simulation time was decreased by 5%.

The accuracy of the cubic scheme is shown in Fig. 5 on the left. The normalized, volume averaged dissipation rate over time computed with OpenFOAM is compared with a spectral DNS code [4].

$$\varepsilon = \frac{1}{V} \int_V S : S \, dV \quad (8)$$

where S is the strain rate tensor. Both simulations use a grid with 512^3 cells; the OpenFOAM simulation was performed on Hazel Hen at the HLRS [7]. The maximum deviation is below 1%, demonstrating that OpenFOAM's cubic scheme is able to accurately simulate the decaying turbulent flow.

On the right of Fig. 5 strong scaling results for this case using 384^3 cells are shown. The numbers above the markers show the parallel scaling efficiency. The tests have been performed with OpenFOAM 1812 using the standard pimpleFoam solver.

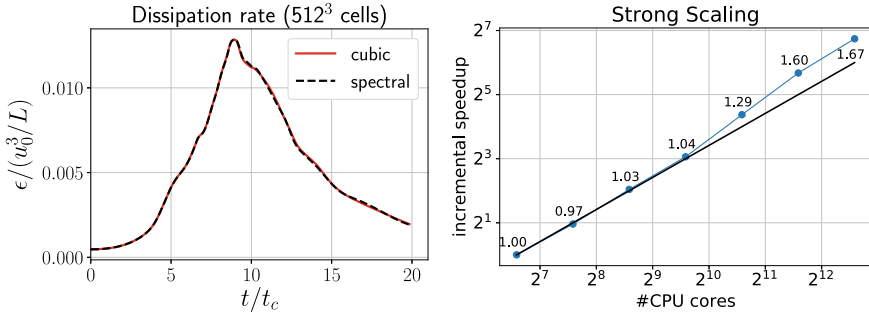


Fig. 5 Integral dissipation rate over time compared with a spectral DNS code (right) and strong scaling results (right)

4 Dynamic Load Balancing for Reactive Flows

OpenFOAM's parallelization strategy is based on domain decomposition together with MPI. By default, OpenFOAM uses third-party libraries like scotch or metis to decompose the computational domain into a number of sub-domains with approximately the same number of cells. This is the ideal strategy for most solvers and applications. For reactive flow simulations however, the chemical reaction rates have to be computed. Since the chemical reaction rates generally depend exponentially on temperature and are coupled to the rate of change of all chemical species, they are numerically very stiff and require very small time steps. In order to avoid using very small time steps in the simulation, an operator splitting approach is usually used. In this approach, the chemical reaction rates are computed by solving a system of ordinary differential equations (ODE) in each cell with special ODE solvers for stiff systems. These solvers take adaptive sub-time steps for integrating the chemical reaction rates over the CFD time step (see [8, 9] for a more detailed description of the operator splitting approach).

The size of the sub-time steps depends on the current local conditions. The lower the chemical time scales are, i.e. the faster the chemical reactions are, the more sub-time steps have to be taken. Consider for example the simulation of a turbulent flame in Fig. 6 using Sundials [10] as the ODE integrator. The light blue lines show the sub-domains into which the computational domain has been divided for the parallel simulation. It can easily be seen that the sub domains in the top and bottom row do not contain any cells where chemical reactions take place. Therefore, the computational effort for sub-domains in the middle where the flame is located is much higher. This leads to load imbalances during the simulation. In general, the position of the flame might change over time so that it is not possible to decompose the mesh once in an optimal way. Therefore, this section presents an approach to overcome load imbalances caused by chemical reaction computations that is done dynamically during runtime.

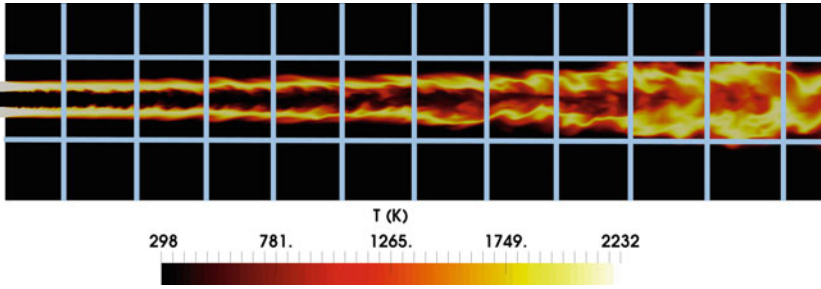


Fig. 6 Temperature field from the simulation of a turbulent flame. Blue lines show the sub-domains for the parallel simulation

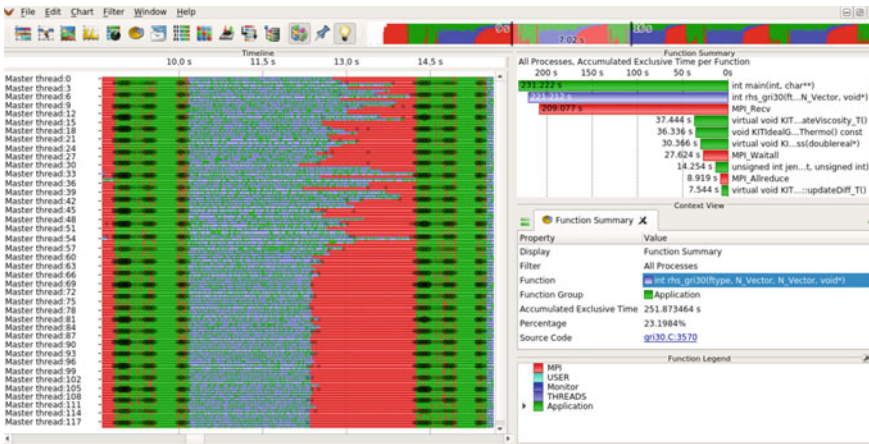


Fig. 7 Performance measurements with Score-P [11], visualized with Vampir [12]

Figure 7 shows performance measurements of a combustion simulation [9] with OpenFOAM on 120 MPI ranks recorded with Score-P [11] and visualized with Vampir [12]. The solver is a self-developed DNS extension for OpenFOAM [13]. The load imbalances can be seen on the left: depicted is one time step during the simulation. Each horizontal line represents one MPI rank. The x-axis represents time. The green areas are useful work performed by OpenFOAM, like solving the conservation equations. The gray regions show the computation of chemical reaction rates. The red areas show communication overhead. In this case, a small number of processes requires about twice as much time to compute the chemical reaction rates than most other processes. Therefore, most of the processes spend half of the simulation time waiting on the few slow processes. This of course wastes large amounts of computational resources. The more complex the chemical reaction mechanisms are (i.e. the more reactions and chemical species are considered), the more severe these load imbalances become.

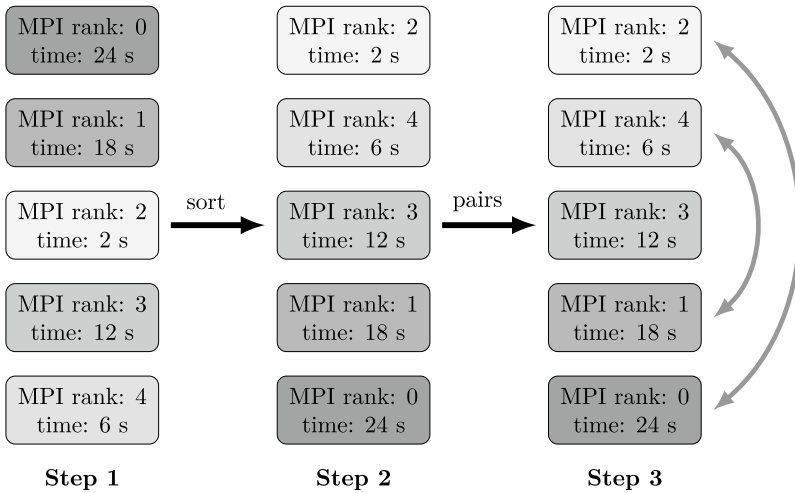


Fig. 8 Schematic drawing of the load balancing approach: Step 1: Measure time for computing chemical reaction rates. Step 2: Sort all processes by computing time. Step 3: Form pairs of processes that share their workload during the following time steps

In order to overcome this, a load balancing algorithm has been implemented which can be used with any reactive flow solver in OpenFOAM and achieves dynamic load balancing during runtime. It also exploits the fact, that the computation of chemical reaction rates does not require information from neighboring cells. This means, that the work load can be freely distributed among the processes without taking any connectivity information of the computational mesh into account, which makes the algorithm flexible and efficient.

The load balancing approach is based on the idea of forming pairs of processes that share their workload. Figure 8 shows an example of this idea: At the beginning of the simulation after the very first time step, each MPI rank measures the time it takes to compute the chemical reaction rates on its own sub-domain. The timing results are then sorted and communicated to all MPI ranks. In the example from Fig. 8, MPI rank 2 has the lowest computing time of 2 s and is therefore the first entry in the list. MPI rank 0 is the slowest with 24 s and therefore is the last entry in the list. The last step is to form pairs of processes: The fastest MPI rank forms a pair with the slowest. The second fastest forms a pair with the second slowest, and so on. The two processes in each pair will share their workload in the subsequent time steps of the simulation. Of course, communicating the timing results from all MPI ranks to all other MPI ranks is an expensive operation. If the position of the flame is rapidly changing, it might be a good idea to repeat this communication step every few time steps to make sure that each pair of processes consists of a slow and a fast process. Usually it is sufficient to check the global imbalances every few thousand time steps.

After the pairs have been formed, the slow process in the pair sends some of its cells to the fast process. After the fast process has computed the chemical reaction

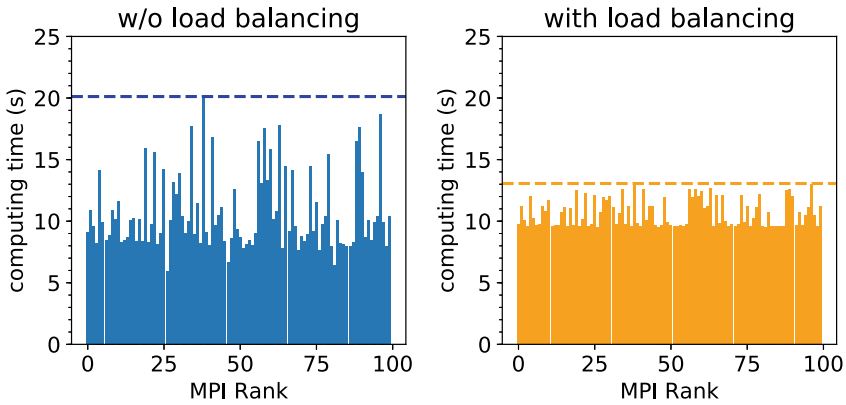


Fig. 9 Time for computing the chemical reaction rates on each MPI rank. The slowest rank determines the overall time for the simulation, shown as dashed line

rates on its own sub-domain, it also computes the reaction rates in the cells of the partner process and sends back the reaction rates in these cells along with the time it took to compute the rates. Based on this time, the number of cells that is sent between the processes is adjusted in the next time step. Due to this, the load balancing approach can adapt to changing conditions during the simulation. Because the calculation of chemical reaction rates does not depend on information in neighboring cells, the cells can be freely shared among the processes. For more information of this approach, see [9].

Figure 9 shows measurements for the time spent on computing chemical reaction rates for a case similar to the one in Fig. 7, using the self-developed DNS solver [13] and the GRI3.0 reaction mechanism, computed with 100 MPI ranks. It can be seen on the left, that some processes take much longer to compute the chemical reaction rates than most other processes. Since all processes have to wait until the last one finishes, most processes spend half the time waiting. Since the slowest process determines the overall time for computing the chemical reaction rates, 20 s per time step are needed for computing the chemical reaction rates in this simulation. Running the same simulation with the load balancing approach from above (Fig. 9 on the right) shows that the difference between the fastest and slowest processes has been drastically decreased. The ratio of the slowest to the fastest process in terms of computing time is about 4 without load balancing and about 1.3 with load balancing. The time per time step that is spent on chemical reaction rates reduces to about 13 s with load balancing, as shown by the dashed line. Often, the computation of chemical reaction rates is the most time consuming part of the simulation, so that the simulation time is reduced from 20 to 13 s per time step, saving about 40 % of the total simulation time.

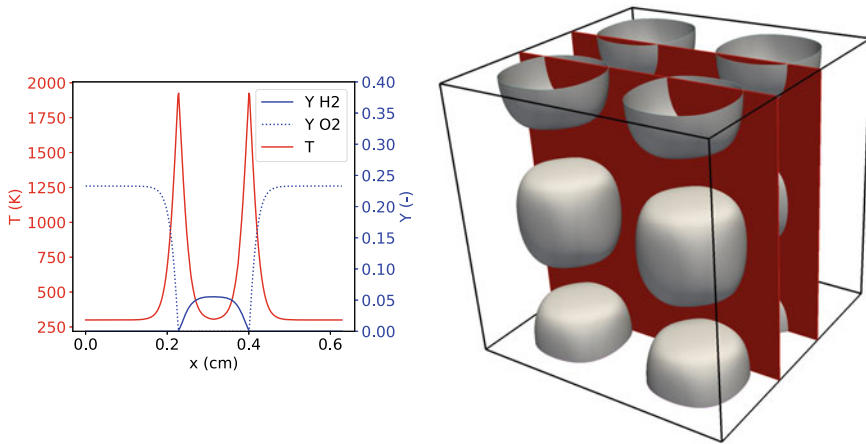


Fig. 10 Left: Initial profiles of temperature, hydrogen mass fraction and oxygen mass fraction along the centerline. Right: Iso-surface of the magnitude of vorticity of 4000 s^{-1} and iso-surface of temperature at 1800 K colored in red

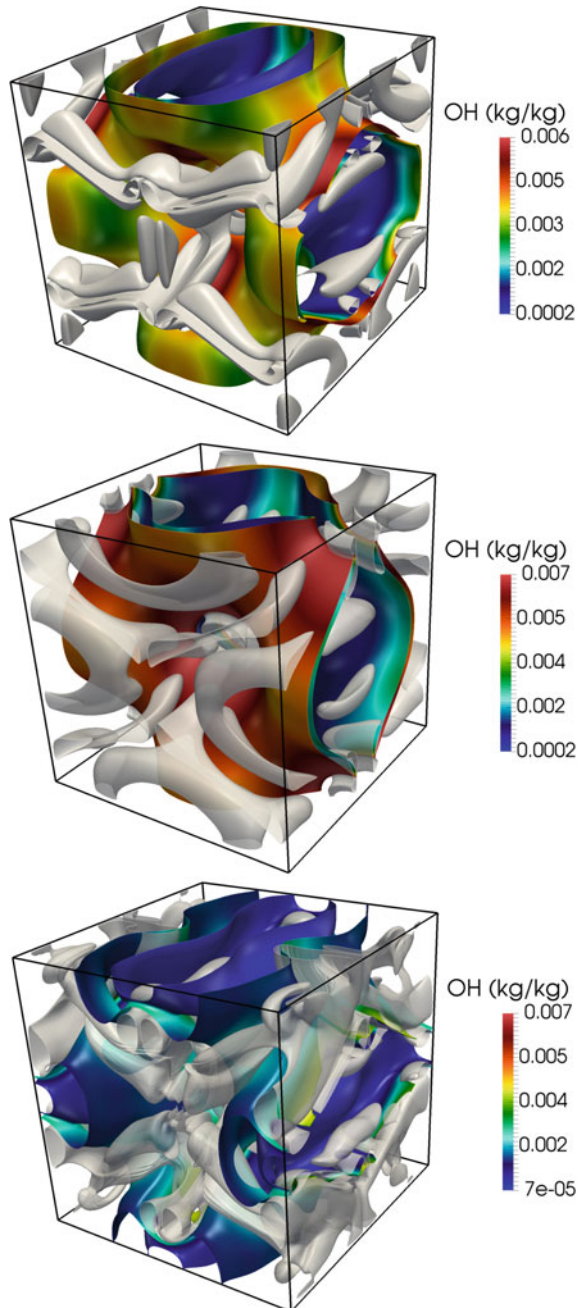
5 Reactive Taylor-Green Vortex

In this section, an application for reactive flow solvers is shown which uses both presented optimization methods: the improved cubic discretization scheme from Sect. 3 and the new load balancing approach from Sect. 4. In total, these led to a reduction of total computing time of about 20%. This application uses the Taylor-Green vortex from Sect. 3 but embeds a flame into the flow field [14]. This setup allows to study the interaction of different types of flames with a well-defined turbulent flow field.

The initial conditions for the flow are the same as for the standard Taylor-Green vortex case (see Eqs. 5 and 6). Additionally, there are now profiles for the temperature and mass fractions of hydrogen and oxygen present. The gas properties and chemical reactions are taken from a reaction mechanism for hydrogen combustion [15]. Figure 10 on the left shows these profiles along the centerline. On the right, a 3D view of the initial conditions in terms of flow vorticity and temperature can be seen. As soon as the simulation starts, the flow field begins to decay into a turbulent flow while the flame burns the hydrogen and oxygen in the domain, leading to a complex interaction between the flow field and the flame. Figure 11 shows the 1800K iso-surface colored by OH mass fraction and iso-surfaces of vorticity of the flow field during three points in time during the simulation.

The case as shown in Fig. 11 was computed on an equidistant grid with 256^3 cells. This case was used to perform a strong scaling test (Fig. 12) both with activated and deactivated computation of chemical reaction rates. The scaling tests are performed with up to 12288 CPU cores. When using up to about 1000 CPU cores or 15 000 cells per core, the scaling is slightly super linear. At 1536 cores, parallel efficiency is

Fig. 11 Iso-surface of temperature at $T = 1800$ K colored by the mass fraction of the OH radical and iso-surfaces of the vorticity magnitude of 4000 s^{-1} (shown in white) at different times: $t = 0.5$ ms (top), $t = 1.0$ ms (middle), $t = 2.0$ ms (bottom)



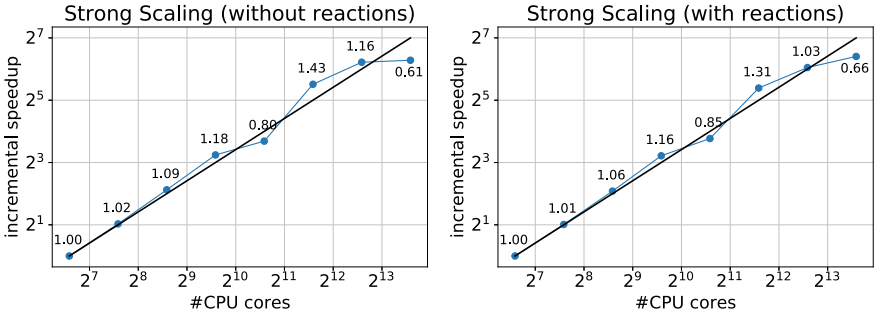


Fig. 12 Strong scaling for the reactive Taylor-Green vortex setup. Left: without chemical reactions. Right: With chemical reactions. Numbers above the markers show the parallel scaling efficiency

reduced to about 0.8 for both the reactive and non-reactive simulation. The optimum lies at 3072 CPU cores or about 5000 cells per core, where the parallel efficiency reaches 1.3 to 1.4. For larger fuels or larger reaction mechanisms, the optimal number of cells might be even lower. Due to the low number of cells, all sub-domains fit into the L3 cache of the CPU. Running the simulation with twice as many cores yields another good scaling result. But beyond this point (more than 12 000 CPU cores or less than 1500 cells per core) the scaling efficiency degrades rapidly. In conclusion, there is an optimum in terms of efficiency at about 5000 cells per core for this simulation using a relatively small reaction mechanism for hydrogen combustion [15]. Reducing the number of cells to about 2500 per core is still possible, but further reduction does not benefit the simulation time.

6 Summary

In order to utilize HPC hardware efficiently with OpenFOAM, the following recommendations are given:

- In order to maximize performance, OpenFOAM should be compiled with more aggressive optimization options like `-Ofast -ffast-math -march=native -mtune=native` for gcc and Clang and `-O3 -fp-model fast=2 -fast-transcendentals -xHost` for the Intel compiler. The effect of this on the accuracy of the simulation results was found to be negligible. The performance gain of using the full optimization options can be as large as 25 % for complex simulations and about 5 % for simple simulations with incompressible flows.
- In general, the Intel compiler generates the most efficient code, but the difference between the compilers are negligible if the full optimization options are used. Using only `O3` as optimization option shows that the Intel compiler generally outperforms gcc and Clang.

- using even more aggressive optimization options like strict aliasing should be avoided since they have led to random crashes of OpenFOAM.
- link-time optimization (LTO) was found to have no effect on performance.

For reactive flow simulations, an optimal value in terms of cells per process was found to be about 5000. This case was run with a relatively small reaction mechanism so that this number may be lower for more complex reaction mechanisms including more chemical species and reactions.

OpenFOAM's most accurate spatial discretization scheme is its `cubic` scheme. It is based on a third order polynomial interpolation from cell centers to cell faces. A comparison with a spectral DNS code has shown that this scheme is able to predict the correct dissipation rate during turbulent decay of the Taylor-Green vortex. The performance of this discretization scheme can be increased by reusing the results of the interpolation polynomials as shown in the code modification in Sect. 3. This can reduce simulation times by an additional 5%.

For simulations using reactive flow solvers, load imbalances due to the computation of chemical reaction rates can lead to large amounts of wasted HPC resources. This is demonstrated by doing a performance measurement with Score-P/Vampir. A self-developed load balancing approach, which is specifically made for the imbalances caused by the computation of chemical reaction rates, is shown to reduce overall simulation times by up to 40%. This approach can be used with any reactive flow solver. The workload is adaptively shared between pairs of processes between a slow and a fast process. This can be combined with the automatic code generation for the chemical reaction rates as described in a previous work [8] to achieve even larger performance gains.

Acknowledgements This work was supported by the Helmholtz Association of German Research Centres (HGF) through the Research Unit EMR, Topic 4 Gasification (34.14.02). This work was performed on the national supercomputer Cray XC40 Hazel Hen at the High Performance Computing Center Stuttgart (HLRS) and on the computational resource ForHLR II with the acronym DNSbomb funded by the Ministry of Science, Research and the Arts Baden-Württemberg and DFG ("Deutsche Forschungsgemeinschaft").

References

1. OpenCFD, OpenFOAM: The Open Source CFD Toolbox. User Guide Version 1.4, OpenCFD Limited. Reading UK (2007)
2. T. Poinsot, D. Veynante, *Theoretical and Numerical Combustion* (R.T. Edwards, 2001)
3. R. Kee, M. Coltrin, P. Glarborg, *Chemically reacting flow: theory and practice* (John Wiley & Sons, 2005)
4. G.I. Taylor, A.E. Green, Mechanism of the production of small eddies from large ones, *Proceedings of the Royal Society of London Series A-Mathematical and Physical Sciences*, vol. 158, no. 895, pp. 499–521 (1937)
5. G. Smith, D. Golden, M. Frenklach, N. Moriarty, B. Eiteneer, M. Goldenberg et al., Gri 3.0 reaction mechanism
6. Karlsruhe institute of technology (2018), www.scc.kit.edu/dienste/forhrlr2.php

7. High performance computing center stuttgart (2018) www.hlrs.de/systems/cray-xc40-hazelhen
8. T. Zirwes, F. Zhang, J. Denev, P. Habisreuther, H. Bockhorn, Automated code generation for maximizing performance of detailed chemistry calculations in OpenFOAM, in *High Performance Computing in Science and Engineering '17*, ed. by W. Nagel, D. Kröner, M. Resch (Springer, 2017) pp. 189–204
9. T. Zirwes, F. Zhang, P. Habisreuther, J. Denev, H. Bockhorn, D. Trimis, Optimizing load balancing of reacting flow solvers in openfoam for high performance computing. ESI (2018)
10. Suite of nonlinear and differential/algebraic equation solvers <http://computation.llnl.gov/casc/sundials>
11. Score-p tracing tool, <http://www.vi-hps.org/tools/score-p.html>
12. Vampir visualization tool, <http://www.paratools.com/vampir/>
13. T. Zirwes, F. Zhang, P. Habisreuther, M. Hansinger, H. Bockhorn, M. Pfitzner, D. Trimis, *Quasi-DNS dataset of a piloted flame with inhomogeneous inlet conditions* (Turb. and Combust, Flow, 2019)
14. H. Zhou, J. You, S. Xiong, Y. Yang, D. Thévenin, S. Chen, Interactions between the premixed flame front and the three-dimensional taylor-green vortex. *Proc. Combust. Instit.* **37**(2), 2461–2468 (2019)
15. P. Boivin, *Reduced-kinetic mechanisms for hydrogen and syngas combustion including autoignition* (Universidad Carlos III, Madrid, Spain, Dissertaton, 2011)