Automated Code Generation for Maximizing Performance of Detailed Chemistry Calculations in OpenFOAM

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

Abstract In direct numerical simulation of turbulent combustion, the majority of the total simulation time is often spent on evaluating chemical reaction rates from detailed reaction mechanisms. In this work, an optimization method is presented for speeding up the calculation of chemical reaction rates significantly, which has been implemented into the open-source CFD code OpenFOAM. A converter tool has been developed, which translates any input file containing chemical reaction mechanisms into C++ source code. The automatically generated code allows to restructure the reaction mechanisms for efficient computation and enables more compiler optimizations. Additional performance improvements are achieved by generating densely packed data and linear access patterns that can be vectorized in order to exploit the maximum performance on HPC systems. The generated source code compiles to an OpenFOAM library, which can directly be used in simulations through OpenFOAM's runtime selection mechanism. The optimization concept has been applied to a realistic combustion case simulated on two peta-scale supercomputers, among them the fastest HPC cluster Hazel Hen (Cray XC40) in Germany. The optimized code leads to a decrease of total simulation time of up to 40% and this improvement increases with the complexity of the involved chemical reactions. Moreover, the optimized code yields good parallel performance on up to 28,800 CPU cores.

1 Introduction

Turbulent combustion remains the key technology for energy conversion. In 2035 fossil fuels will still provide 80% of the total energy for the world's economy [1]. Therefore, increasing the efficiency of combustion processes and reducing global pollutant emissions continues to be a major task. But because the interaction of

T. Zirwes • F. Zhang (✉) • J.A. Denev • P. Habisreuther • H. Bockhorn
Engler-Bunte-Institute/Combustion Technology, Karlsruhe Institute of Technology, Engler-Bunte-Ring. 7, 76131 Karlsruhe, Germany
e-mail: thorsten.zirwes@kit.edu; feichi.zhang@kit.edu

W.E. Nagel et al. (eds.), *High Performance Computing in Science and Engineering '17*, https://doi.org/10.1007/978-3-319-68394-2_11

flames and turbulent flows is still not fully understood [2, 3], an important research goal is to gain a deeper understanding of the underlying physics and chemistry as well as their mutual interaction in combustion systems.

Numerical simulation is a well-established tool to study combustion processes [2]. With the increasing power in High Performance Computing (HPC) it became feasible to conduct direct numerical simulation (DNS) of flames in turbulent flows. In DNS, the turbulent flow and the combustion chemistry are resolved down to the smallest time and length scales, requiring enormous computational resources. Although simulation tools for turbulent combustion become more and more efficient, DNS of turbulent flames is still limited to small simulation domains. The DNS results however deliver valuable details of the complex combustion phenomena and complement experimental data, which are limited by extreme conditions in technical combustion applications like high pressures and temperatures [4].

The predominant computing time in DNS of turbulent flames is needed to evaluate chemical reaction rates if detailed chemical mechanisms are used [5, 6]. Even for the combustion of simple fuels like methane more than 50 intermediate species and over 300 reactions are typically used to give a detailed description of the chemistry [7]. For technically relevant fuels like kerosene or gasoline consisting of higher hydrocarbons, large reaction mechanisms are being developed which contain thousands of chemical species and tens of thousands of reactions [8, 9]. In addition to solving the Navier-Stokes equations, an additional conservation equation for each chemical species has to be solved, so that the total simulation time strongly depends on the number of chemical species and therefore the complexity of the combustion chemistry. The most time consuming part of the simulation is usually the computation of chemical reaction rates, which are the source terms in the additional conservation equations for the species.

Because of this, the present work introduces on an approach for speeding up the computation of chemical reaction rates. The basis for describing the combustion chemistry are “reaction mechanisms”. They specify which species play a role during combustion and define all chemical reactions as well as their parameters. An example of a reaction mechanism for methane/air combustion is given in Sect. 3.2. In this work, a converter tool has been developed which reads general reaction mechanism files as input, restructures them, translates them into C++ source code and applies optimizations for faster evaluation of chemical reaction rates. The idea is that, instead of having a general program that can solve the chemical system for arbitrary reaction mechanisms, each mechanism is represented by specialized subroutines in the generated source code [10, 11].

In order to simulate combustion in turbulent flows and evaluate the performance gain achieved with the presented optimization method, the generated chemistry code is coupled with OpenFOAM [12], which is an open-source tool for computational fluid dynamics (CFD) and can be used for DNS of turbulent combustion [13].

2 Reference DNS Code

In previous works [14] a DNS solver for turbulent combustion phenomena was developed by coupling the general CFD tool OpenFOAM [12] with the thermo-chemical library Cantera [15]. OpenFOAM is an open-source toolbox written in C++. It has found widespread use in computational fluid dynamics during the last years, both in scientific and engineering applications, including DNS of flames. Cantera is a widely used, open-source chemistry library written in C++ which implements highly optimized routines for computing chemical reaction rates and also provides information about detailed molecular diffusion which plays an important role in combustion processes.

Figure 1 presents the structure of the developed coupling interface between OpenFOAM and Cantera. The OpenFOAM code has the task of solving the conservation equations for total mass, momentum, energy and the mass of each chemical species. The state parameters in terms of pressure p, temperature T and gas composition Y_k are used as input for Cantera's routines, which are called by the interface in order to calculate the reaction rates and other thermo-physical properties.

After coupling OpenFOAM with Cantera, additional performance gains were achieved by extracting relevant classes from Cantera and implementing them directly into an OpenFOAM library, so that Cantera is not an external dependency anymore and its methods are called by the OpenFOAM solver directly. This led to further performance improvements [16]. This coupling of OpenFOAM with Cantera represents the reference case for quantifying the performance gains when using the generated source code.

Profiling of the reference DNS code has been conducted in order to identify the performance bottlenecks. With the reaction mechanism by Kee et al. [17] for the combustion of methane, which contains only a small number of reactions and chemical species (see Table 1), chemistry computations take about 60% of the total simulation time with the setup described in Sect. 4.1. For the more detailed GRI 3.0 mechanism [7], almost 90% of simulation time are spent on the chemistry with the reference DNS code. These two reaction mechanisms will be used in the following

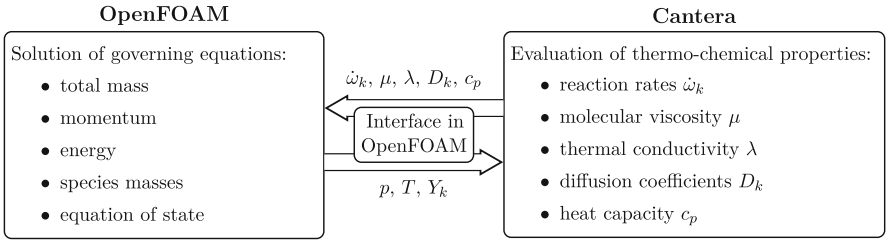

Fig. 1 Simplified chart demonstrating the coupling of OpenFOAM with Cantera in the reference case

Table 1 Reaction mechanisms for methane/air combustion used in this work

Reaction mechanism	Number of species	Number of reactions
Kee et al. [17]	17	53
GRI 3.0 [7]	58	325

sections to demonstrate the performance improvements achieved with the presented optimization method.

In order to compute the chemical reaction rates $\dot{\omega}_k$ for each species k, a number of intermediate quantities have to be calculated. First, the rates of progress $\dot{r}_r$ for each reaction r has to be computed for the forward ($'$) and reverse ($''$) reaction with

$$\dot{r}_r' = k_r' C_m \prod_k C_k^{\nu_{k,r}'}, \qquad \dot{r}_r'' = k_r'' C_m \prod_k C_k^{\nu_{k,r}''} \ , \tag{1}$$

where C_m is an effective mixture concentration in case the reaction is a three-body reaction, C_k is the concentration of the species involved in the reaction and $\nu_{k,r}$ is the stoichiometric coefficient of the species for this reaction. The rate constant k_r' for the forward reaction is either computed via Arrhenius' law, which is explained in more detail in the next section, or a more complex falloff type formulation [17]. The rate constants k_r'' of the reverse reactions are usually obtained from equilibrium constants which are computed from thermodynamic considerations. In order to obtain the species reaction rates, all rates of progress have to be added up in the following way:

$$\dot{\omega}_k = M_k \sum_r \left(\nu_{k,r}'' - \nu_{k,r}'\right)\left(\dot{r}_r{}' - \dot{r}_r''\right) \ . \tag{2}$$

M_k is the molar mass of species k. These reaction rates are not used directly in the conservation equation for the species masses. Instead, they are averaged over the simulation time step Δt:

$$\bar{\dot{\omega}}_k \approx \frac{1}{\Delta t} \int_t^{t+\Delta t} \dot{\omega}_k \, \mathrm{d}t, \qquad k = 1 \ldots N \tag{3}$$

This system of N ordinary differential equations (ODE) together with the change of temperature is solved at every time step for each cell in the computational domain. Because N can be very large depending on the complexity of the reaction mechanism, this ODE integration is the reason why the majority of computing time is spent on computing chemical reaction rates $\dot{\omega}_k$. The advantage of this method is that a higher simulation time step Δt can be used because it is not limited by the shortest chemical reaction time scales.

Using the detailed GRI 3.0 mechanism in DNS showed that typically 20% of total simulation time is spent on evaluating the exponential function `exp` (measured with perf from the Linux tools). Most of the calls to `exp` stem from the Arrhenius

law in (4). Therefore, Arrhenius' law is used in the next section to demonstrate the principles of specific optimizations that are performed during conversion from reaction mechanism input file to C++ source code.

3 Optimized Code Generation

3.1 Basic Concept of the Code Generation Approach

The basic idea in this work is to automatically generate C++ source code for each specific reaction mechanism in order to maximize performance. Two optimization steps are performed to improve the computation of chemical reaction rates:

- In the first step the structure of the chemical reaction mechanism is optimized. Redundant operations are eliminated and species and reactions are reordered and regrouped. This allows to minimize code branching and maximize reuse of cached results (see Sect. 3.2).
- The second step targets the C++ source code. The code is generated in a way that makes it easy for the compiler to optimize. For example, loops with trivial access patterns are generated to enable auto-vectorization and data is stored densely packed in memory to maximize CPU cache usage. Since all parameters from the reaction mechanisms like the number of species are compile-time constants, the compiler can make better optimization decisions with respect to inlining and loop unrolling.

In order to achieve both optimization goals, a converter tool has been developed which reads general reaction mechanism files in Cantera's `ctml` or `xml` format as input, performs the two optimization steps and automatically generates C++ source code containing all necessary routines (see Fig. 2). In contrast to Cantera's implementation which provides efficient but more general routines for computing reaction rates from arbitrary reaction mechanisms, the code generated in this way contains only routines that are specialized for one specific reaction mechanism. Instead of being scattered across different translation units, all chemistry code is visible to the compiler at once, giving it the maximum amount of information for optimizations.

Because the chemical reaction rates depend only on local mixture properties, no neighboring cell values are needed so that the generated source code does not contain any parallel communication routines. It is therefore an optimization on the node level. Compiling the generated code results in an OpenFOAM library

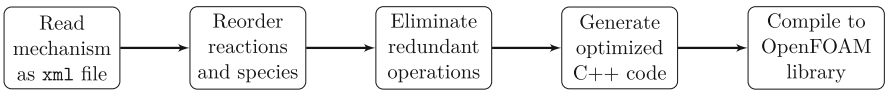

Fig. 2 Simplified overview over how the converter tool works

that can directly be used in the previously developed coupling interface described in Sect. 2 due to OpenFOAM's runtime selection mechanism. By doing this, no routines that originally came from Cantera are involved anymore in the computation of the reaction rates. But in other parts of the OpenFOAM solver, i.e. computation of thermodynamic and diffusive properties, Cantera's routines are still used. These parts are however not a performance bottleneck.

The automatically generated source code contains all necessary routines to compute the chemical reaction rates and all intermediate properties from (1) and (2). In the next section, only the part of the code regarding Arrhenius' law is shown as an example to explain the work done by the converter.

3.2 Optimized Computation of Rate Constants

As mentioned in the previous section, a considerable amount of time during the simulation is spent on evaluating the exponential function `exp` used in the computation of the rate constants k_r of each reaction via Arrhenius' law

$$k_r = A_r \exp\left(b_r \log(T) - \frac{E_r}{\mathscr{R}T}\right) \quad , \tag{4}$$

where $\mathscr{R}$ is the universal gas constant and T the temperature, which is the only variable quantity in Arrhenius' law. The parameters A_r, b_r and E_r are constants and defined for every reaction in a reaction mechanism file. Figure 3 shows a small example of the reactions defined in the GRI 3.0 reaction mechanism for methane/air combustion in standard CHEMKIN format [18], where the three Arrhenius parameters A_r, b_r and E_r are defined. In the OpenFOAM solver of the reference case, a mechanism file like this is read once at the start of each simulation.

In Cantera's original implementation, the rate constants for every reaction are computed using (4). It is however possible to compute some rate constants in a more efficient way. There are three cases, where the evaluation of the exponential function can be avoided completely. Figure 4 shows the automatically generated C++ source

Reaction	A_r	b_r	E_r
...			
O2+CH2CHO=>OH+2HCO	2.350E+10	0.000	0.00
O2+CO<=>O+CO2	2.500E+12	0.000	47800.00
H+O2+H2O<=>HO2+H2O	11.26E+18	-0.760	0.00
...			

Fig. 3 Example of three out of 325 reactions from the GRI 3.0 reaction mechanism in standard CHEMKIN format, defining the Arrhenius parameters A_r, b_r and E_r for each reaction

```
1  alignas(64) static constexpr double A[354] =
2      {3.655839600035736e+00, 9.1726385047921717e+00, ... };
3
4  alignas(64) static constexpr double b[228] =
5      {2.7000000000000002e+00, 2.0000000000000000e+00, ... };
6
7  alignas(64) static constexpr double E_R[228] =
8      {3.1501544760183583e+03, 2.0128782594366505e+03, ... };
9
10 auto invT = 1./T;
11 auto logT = std::log(T);
12 // compute the 228 rate constants for which an
13 // evaluation of the exponential function is necessary
14 for (unsigned r = 0; r != 228; ++r) {
15     auto blogT = b[r]*logT;                    // b_r log(T)
16     auto E_RT = E_R[r]*invT;                   // E_r/RT
17     auto diff = blogT - E_RT;
18     k[r] = std::exp(A[r] + diff);
19 }
20
21 // six rate constants with E_r=0 and b_r a small integer
22 double tmp0 = invT*invT;
23 for (unsigned i = 0; i != 2; ++i)
24     k[i+228] = A[i+228]*tmp0;                  // k_r = A_r T^-2
25 double tmp1 = invT;
26 for (unsigned i = 0; i != 4; ++i)
27     k[i+230] = A[i+230]*tmp1;                  // k_r = A_r T^-1
28
29 // 100 rate constants with E_r=0 and b_r=0
30 for (unsigned i = 0; i != 100; ++i)
31     k[i+234] = A[i+234];
32
33 // 20 rate constants where E_r and b_r are the same as for
34 // other rate constants that have already been computed
35 for (unsigned i=0; i != 20; ++i)
36     k[i+334] = A[i+334]*k[i+209];
```

Fig. 4 Automatically generated C++ source code for the GRI 3.0 mechanism. This example shows only the part of the generated code that computes the rate constants from (4)

code for the computation of the rate constants from the GRI 3.0 mechanism, where the following cases are considered:

- If b_r and E_r for a reaction are zero, the rate constant from (4) reduces to the constant value $k_r = A_r$ and no exponential function has to be computed (lines 30–31 in Fig. 4). In total, 100 out of 354 rate constants in the GRI 3.0 reaction mechanism have $b_r = 0$ and $E_r = 0$, see e.g. the first reaction in Fig. 3.
- If different rate constants have the same values of b_r and E_r, the exponential function is evaluated once for one of them and reused for each additional

occurrence. This applies to 40 out of all 354 rate constants from the GRI 3.0 reaction mechanism (lines 35–36 in Fig. 4).

- If E_r is zero and b_r is a small integer, Arrhenius' law reduces to $k_r = A_r T^{b_r}$ (lines 22–27 in Fig. 4). Because b_r is an integer, T^{b_r} can be replaced with a small number of multiplications. This occurs for 6 rate constants in the GRI 3.0 mechanism.

After eliminating these special cases, only 228 out of 354 rate constants have to be computed using the full Arrhenius law involving the expensive exponential function in (4) (see lines 14–18 in Fig. 4).

In the generated source code, the Arrhenius parameters are stored in contiguous arrays for maximizing data cache usage. Note that `A[r]` in line 18 is automatically stored as $\log(A_r)$ and `A[r+334]` to `A[r+354]` are stored as $\log(A_r)/\log(A_{r,\text{duplicate}})$. They are also explicitly aligned to allow auto-vectorization by the compiler. Because the number of loop iterations is a compile-time constant in the generated source code, the compiler can make better decisions about loop unrolling. During code generation all reactions are reordered so that reactions with similar properties are grouped together in the same loops over contiguous data.

In total for the GRI 3.0 mechanism, more than a third of all exponential function evaluations are omitted in the generated code. Together with the vectorization, evaluating the exponential function with the optimized chemistry source code takes only about 5% of the total computing time instead of 20% in the reference DNS code.

Computation of the rate constants k_r is only a small subset of the code needed to compute the final chemical reaction rates. For the rest of the chemistry code indicated by (1) and (2), the basic optimization principles described so far are the same: restructure and simplify the reaction mechanism and generate code that is easy to optimize. In total, the automatically generated C++ code can speed up the chemistry computation by more than 50% (see Sect. 4.3). It should also be noted that all changes done by the converter tool to the reaction mechanism, like for the evaluation of rate constants shown in this section, are equivalent in a strict mathematical sense, so that the simulation results are almost the same as in the reference case.

3.3 Choice of Compiler

The choice of compiler has an impact on the performance of the generated chemistry code too. This is illustrated for the first loop (line 14) in Fig. 4 which contains the most expensive operations and has the highest iteration count among the depicted loops. Figure 5 shows the machine code generated by the GNU compiler (g++ 6.2.0) on the left and the Intel compiler (icpc 17.0.1) on the right on the Cray XC40 Hazel Hen cluster (see Sect. 4.2 for a description of the architecture). Both machine codes are created with the flags `-std=c++14 -Ofast -march=native -mtune=native`.

g++

```
1  // b[r]*logT
2  vmulsd b(%r13), %xmm7, %xmm0
3  // E_R[r]*invT
4  vmulsd E_R(%r13), %xmm8, %xmm1
5  // A[r]+blogT
6  vaddsd A(%r13), %xmm0, %xmm0
7  // A[r]+diff
8  vsubsd %xmm1, %xmm0, %xmm0
9  // exp(A[r]+diff)
10 call   exp
```

icpc

```
// b[r]*logT
vmulpd b(,%r15,8), %ymm8, %ymm1
// E_R[r]*invT
vmulpd E_R(,%r15,8), %ymm9, %ymm3
// A[r]+blogT
vaddpd A(,%r15,8), %ymm1, %ymm2
// A[r]+diff
vsubpd %ymm3, %ymm2, %ymm0
// exp(A[r]+diff)
call   __svml_exp4
```

Fig. 5 Comparison of shortened machine code output on Cray XC40 Hazel Hen of the GNU g++ 6.2.0 (left) and Intel icpc 17.0.1 (right) compiler for the loop body in lines 14–18 of Fig. 4. Comments (`//`) show the respective C++ code

Although both machine codes look similar, the GNU compiler does not auto-vectorize the loop. This can be seen from the instructions ending in `sd` ("scalar double") instead of `pd` ("packed double"). Therefore, each instruction in the loop of the GNU compiler only operates on one double precision value whereas each iteration of the loop generated by the Intel compiler operates on four double precision values at the same time. The reason is that the Intel compiler automatically replaces the call to the exponential function `std::exp` with a call to `svml_exp4` which is a version of the exponential function defined in Intel's short vector math library (SVML) acting on four double precision values at the same time. In our tests, the Intel compiler creates machine code that performs 15–20% faster in terms of total simulation time compared to the GNU compiler for the two investigated reaction mechanisms. An additional 2% performance gain has been achieved by using the `restrict` keyword wherever possible, which is a C language feature that gives the compiler additional information about memory access. The performance and scalability tests shown in Sect. 4.3 are all measured using Intel's compiler.

4 Performance Validation

4.1 Numerical Setup

In order to assess the performance benefit of using the generated and optimized chemistry code in the context of HPC simulation of turbulent combustion, the simulation of a turbulent flame has been conducted which was experimentally studied [19]. This section gives a short description of the numerical setup and the HPC clusters.

Figure 6 on the left shows the simulation setup. It consists of the burner nozzle with a diameter of 3.5 cm through which the methane/air mixture flows, and a cylindrical region with a diameter of 0.6 m and a height of 0.6 m, representing

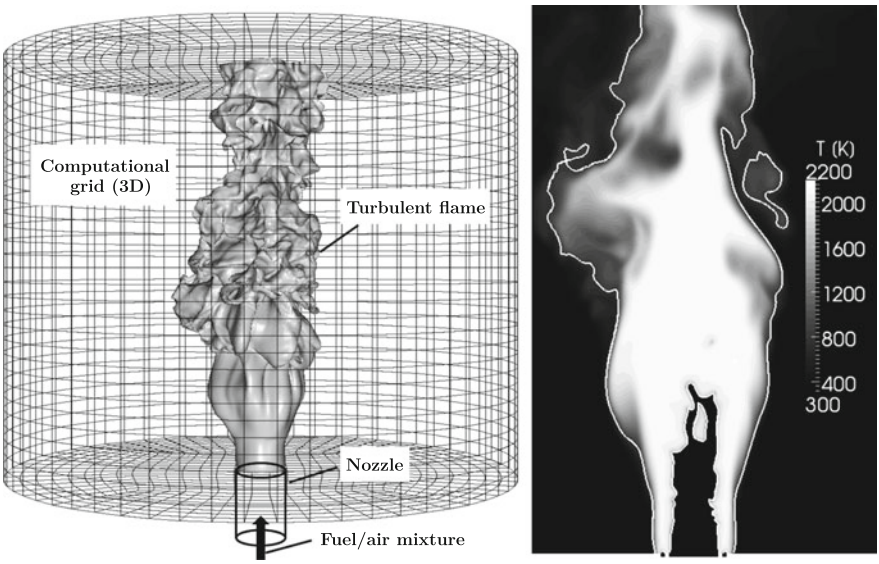

Fig. 6 Schematic drawing of the simulation setup and iso-surface of $T = 800\,\mathrm{K}$, indicating the flame surface (left), and a cut of the instantaneous temperature field (right)

the environment of the burner where the flame stabilizes. The computational grid used for the performance test is composed of 176 million cells on Hazel Hen and 76.5 million finite volumes on ForHLR II, which are refined locally where the flame burns. The governing equations are solved in OpenFOAM with the finite volume method (FVM) along with high-accuracy numerical schemes [20]. Two chemical reaction mechanisms with different complexities, as shown in Table 1, are used in order to show the general validity of the presented concept for chemistry calculations.

Figure 6 on the left illustrates the shape of the flame identified by the iso-surface of $T = 800\,\mathrm{K}$, which becomes wrinkled due to the interaction between the combustion reaction and the turbulent flow. A meridian cut though the computational domain on the right of Fig. 6 shows the temperature field. The gray line indicates the 800 K isotherm, corresponding to the contour of the flame surface depicted on the left of Fig. 6. Although not presented here, the simulation results have shown good agreement with the corresponding experimental data [19].

4.2 HPC Clusters and Software Versions

The performance gains when using the generated source code for the chemistry computations compared to the reference case are evaluated on two supercomputers.

The **Hazel Hen** cluster at the Höchstleistungsrechenzentrum Stuttgart (HLRS) is a CRAY XC40 system based on the twelve-core Intel Xeon E5-2680 v3 processor and Cray Aries network with Dragonfly network topology [21]. With its 15,424 CPUs (185,088 cores) it is currently one of the fastest supercomputers in Europe. It has 7712 dual socket nodes, each node containing a total of 24 cores and having 128 GB DDR4 memory, achieving a theoretical peak performance of 7.42 PFlops. OpenFOAM is used in version 1612+, compiled with gcc version 6.2, together with cray-mpich version 7.0.3. The OpenFOAM solver is coupled to the modified Cantera library based on Cantera 2.3.0a3. The generated chemistry source code and Cantera's implementation in the reference DNS Code are compiled with the Intel compiler version 17.0.1.

ForHLR II at the Karlsruhe Institute of Technology (KIT) has 1152 computing nodes with 64 GB RAM each [22]. Each node has 20 cores (two deca-core Intel Xeon E5-2660 v3 processors) with a total theoretical peak performance of 1 PFlops. All nodes are connected through an InfiniBand 4X EDR Interconnect. OpenFOAM is used in version 2.3.0, compiled with gcc version 4.8, together with OpenMPI version 1.10. The OpenFOAM solver is coupled to the modified Cantera library based on Cantera 2.3.0a3. The generated chemistry source code and Cantera's implementation in the reference DNS Code are compiled with the Intel compiler version 16.0.

4.3 Performance Improvement

The performance improvement by using the chemistry code generated by the converter tool instead of Cantera's implementation has first been evaluated on a single node. An interesting example of how the new chemistry code affects the performance of the simulation is shown in the table on the left of Fig. 7. Using the performance monitoring tool LIKWID, the number of cache and branch prediction misses have been recorded on ForHLR II for the detailed GRI 3.0

Type of Miss	$\frac{\#\text{Misses(reference)}}{\#\text{Misses(optimized)}}$
Branch prediction	8.1
Instruction Cache	30.8
L1 Cache	5.2
L2 Cache	7.9

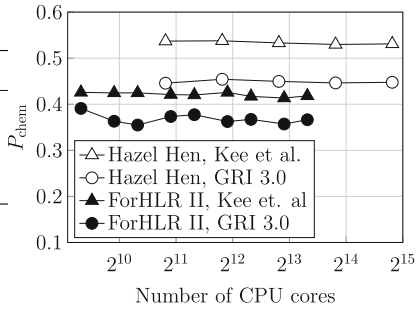

Fig. 7 Left: Relative reduction in cache and branch prediction misses for the GRI 3.0 mechanism for a serial case. Right: Performance improvement P_{chem} for only the chemistry computations

mechanism. Branch prediction misses happen when the control flow of the program is not predictable for the CPU. Cache misses happen when either data or program instructions are not in the local CPU caches when they are needed. Because in the generated chemistry code most of the functions are inlined and all loop counters and other parameters affecting the control flow are compile time constants, the number of instruction cache misses is reduced by a factor of 30. Similarly, the number of branch prediction misses is reduced by a factor of 8. Because all data is stored and accessed linearly in memory in the optimized code, L1 cache misses are reduced by a factor of 5 and L2 cache misses by a factor of 8.

Figure 7 on the right shows the time savings of only the chemistry computations for the setup described in Sect. 4.1 on ForHLR II and Hazel Hen. On ForHLR II, the simulation was run on 32 nodes (640 cores) up to 510 nodes (10,200 cores). On Hazel Hen, it was run on 75 nodes (1800 cores) to 1200 nodes (28,800 cores). The time needed for just the chemistry t_{chem} is compared between the reference case and the generated optimized chemistry code with

$$P_{\text{chem}} = \frac{t_{\text{chem}}(\text{reference case}) - t_{\text{chem}}(\text{optimized})}{t_{\text{chem}}(\text{reference case})} \quad . \tag{5}$$

With the detailed GRI 3.0 reaction mechanism, the time for the chemistry computations is reduced by approximately 45% on Hazel Hen and 35–40% on ForHLR II. Using the mechanism by Kee reduces the time for the chemistry computations on ForHLR II by 42% and on Hazel Hen by 53%, thereby saving more than half of the computing time needed for the chemical reaction rates. The improvement stays nearly constant with the number of CPU cores because the optimizations performed in the chemistry routines are not affected by communication. Large portions of the generated code are not as easily vectorizable as shown for Arrhenius' law in Sect. 3.2. For example the summation in (2) cannot be vectorized so that the total speedup stays below the optimal speedup expected from perfect vectorization. Otherwise, all loops that were expected to be vectorized have been auto-vectorized by the Intel compiler.

Because the Kee mechanism is much less complex than the GRI 3.0 mechanism, total simulation time t_{tot} is on average ten times shorter with the Kee mechanism compared to the more detailed GRI 3.0 mechanism. The chemistry part of the simulation is more than 15 times faster with the smaller Kee mechanism. There is also a difference in how much of total simulation time is spent on chemistry computations for the two investigated reaction mechanisms: chemistry calculations take 60–70% of the total simulation time with the Kee mechanism in the reference DNS code and 40–50% with the optimized code. For the GRI 3.0 mechanism, about 90% of the total simulation time is spent on computing chemical reaction rates in the reference DNS code and 85% in the optimized solver. Because much more of the total simulation time is used for the chemistry calculations with the GRI 3.0 mechanism, the overall performance gain P_{tot} is greater with the GRI 3.0 mechanism compared to the Kee mechanism, although the relative improvement

of only the chemistry computations P_{chem} in Fig. 7 is slightly better with the Kee mechanism.

$$P_{\text{tot}} = \frac{t_{\text{tot}}(\text{reference case}) - t_{\text{tot}}(\text{optimized})}{t_{\text{tot}}(\text{reference case})} \quad . \tag{6}$$

Using the generated chemistry routines, the total simulation time is reduced by $P_{\text{tot}} = 40\%$ with the GRI 3.0 mechanism on Hazel Hen and 35% on ForHLR II. For example, on Hazel Hen with 28,800 CPU cores, the average time of a time step is about 8 s with the optimized chemistry but more than 13 s with the reference case implementation. The simulation with the reaction mechanism by Kee is 25–35% faster. In order to obtain statistically converged data from DNS of turbulent combustion, typically at least 10^5 simulation time steps have to be calculated. For the present case, this would require about 6.5 million CPU core hours with the GRI 3.0 mechanism on Hazel Hen with the reference DNS code. Using the optimized chemistry routines reduces the overall simulation time by almost 40%, saving over 2.5 million core hours. For future simulations, even more detailed reaction mechanisms will be employed so that higher percentages of total simulation time are used to compute the chemical reaction rates, making the performance gains from the presented optimization technique even more important.

4.4 Parallel Performance

Figure 8 shows the incremental speedup of the DNS solver using the optimized chemistry routines for strong scaling of the setup described in Sect. 4.1. The incremental speedup is defined as $S_n = t_{\text{tot}}(n)/t_{\text{tot}}(n_0)$, where n is the number of CPU cores and n_0 is 640 CPU cores on ForHLR II and 1800 CPU cores on Hazel Hen and t_{tot} is the total simulation time for a fixed number of time steps. The

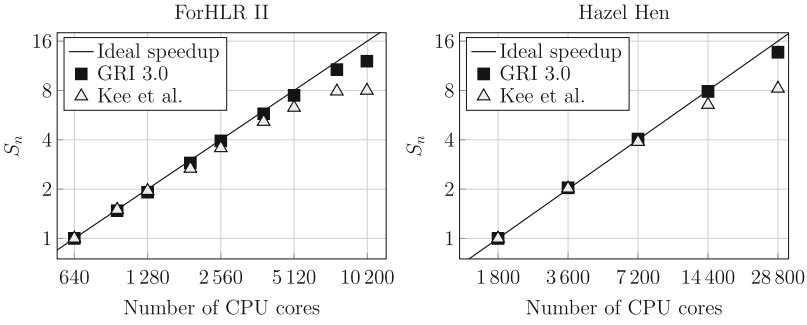

Fig. 8 Incremental speedup S_n (strong scaling): Comparison of Kee mechanism and more detailed GRI 3.0 mechanism on ForHLR II (left) with $76.5{\cdot}10^6$ cells and Hazel Hen (right) with $176{\cdot}10^6$ cells for DNS of turbulent methane/air combustion with the optimized chemistry code

simulations with the GRI 3.0 mechanism scale almost ideally. The scaling efficiency is still above 90% for 28,800 CPU cores on Hazel Hen. As mentioned before, there is no communication during the chemistry computations. Therefore, the more complex the chemistry is, the larger is the share of total simulation time without communication overhead. The simulations with the Kee mechanism scale well up to 5000 CPU cores but scaling efficiency decreases beyond that. In this case, the share of actual computation time in comparison to communication overhead decreases, leading to reduced parallel efficiency. The DNS with detailed chemistry, for example with the GRI 3.0 mechanism, is much more computationally expensive than with the Kee mechanism (see Table 1), so that a better parallel performance is achieved with the GRI 3.0 mechanism. The presented optimizations are therefore most beneficial for larger reaction mechanisms, both in terms of parallel performance and overall simulation time reduction.

It noteworthy that the scaling efficiency becomes slightly better if Cantera's chemistry implementation in the reference case is used instead of the optimized chemistry code. Because the chemistry computations take much longer with Cantera's implementation, the ratio of communication to computation becomes smaller.

5 Conclusion

This work presents an optimization technique where specialized source code is automatically generated to speed up chemistry computations in DNS of chemically reacting flows. It shows that large performance improvements are possible compared to even highly optimized libraries when source code is optimized for special cases. A converter tool has been developed which reads reaction mechanism files containing all information about the chemical reactions occurring during combustion and converts them into C++ routines. The reaction mechanism is analyzed and restructured by the converter tool to enable more efficient evaluation of the chemical reaction rates. The source code for the chemistry routines is generated in a way that is easy for the compiler to optimize and maximizes the usage of data caches and auto-vectorization. These optimizations are achieved by following the design principles of cache friendly data structures and linear data access patterns, which will become more important in the future due to increasing vector register sizes. The number of cache and branch prediction misses have been shown to be drastically reduced. The generated chemistry routines for the combustion of methane and air have been coupled with the open-source library OpenFOAM in order to perform the simulation of a realistic turbulent flame. The simulations were run on two supercomputers (Cray XC40 Hazel Hen and ForHLR II) with up to 28,800 CPU cores showing very good scalability. A decrease of up to 40% in total simulation time has been achieved compared to the reference DNS code without affecting the accuracy of the results. The method is of particular interest when applying complex chemical reaction systems instead of simplified or reduced chemistry, which is important for studying the mechanisms of flame/turbulence interaction and the generation of pollutant

emissions. The performance tuning with the presented optimization technique, together with the good scalability of the widely used OpenFOAM package, will help to investigate combustion phenomena in more detail. Consequently, the time and cost required for the development of modern combustion devices may be reduced in the future.

Acknowledgements This work was supported by the German Research Council (DFG) through Research Units DFG-BO693/27 "Combustion Noise". This work was performed on the national supercomputer Cray XC40 Hazel Hen at the High Performance Computing Center Stuttgart (HLRS) under the grant with acronym 'Cnoise' and on the computational resource ForHLR II funded by the Ministry of Science, Research and the Arts Baden-Württemberg and DFG ("Deutsche Forschungsgemeinschaft").

References

1. BP Energy Outlook, British Petroleum (2016), www.bp.com/energyoutlook
2. T. Poinsot, D. Veynante, *Theoretical and Numerical Combustion* (RT Edwards, Toulouse Cedex, 2005)
3. A. Lipatnikov, *Fundamentals of Premixed Turbulent Combustion* (CRC, Boca Raton, 2012)
4. C.K. Law, *Combustion Physics* (Cambridge University Press, Cambridge, 2010)
5. J.H. Chen, Petascale direct numerical simulation of turbulent combustion—fundamental insights towards predictive models. Proc. Combust. Inst. **33**, 99–123 (2011)
6. F. Zhang, T. Zirwes, P. Habisreuther, H. Bockhorn, Numerical simulation of turbulent combustion with a multi-regional approach, in *High Performance Computing in Science and Engineering '15*, ed. by W.E. Nagel, D.B. Kröner, M.M. Resch (Springer, Berlin, Heidelberg, 2015), pp. 267–280
7. G.P. Smith, D.M. Golden, M. Frenklach, N.W. Moriarty, B. Eiteneer, M. Goldenberg, C.T. Bowman, R.K. Hanson, S. Song, W.C. Gardiner, V.V. Lissianski, Z. Qi, GRI 3.0 reaction mechanism (1999), http://www.me.berkeley.edu/gri_mech
8. LLNL Heptane Reaction Mechanism (2012), https://combustion.llnl.gov/mechanisms/alkanes/n-heptane-detailed-mechanism-version-3
9. T. Lu, C.K. Law, Toward accommodating realistic fuel chemistry in large-scale computations. Prog. Energy Combust. Sci. **35**(2), 192–215 (2009)
10. V. Damian, A. Sandu, M. Damian, F. Potra, G.R. Carmichael, The kinetic preprocessor KPP—a software environment for solving chemical kinetics. Comput. Chem. Eng. **26,** 1567–1579 (2002)
11. K.E. Niemeyer, N.J. Curtis, pyJac, Version 1.0.1 (2016), https://github.com/kyleniemeyer/pyJac
12. H.G. Weller, G. Tabor, H. Jasak, C. Fureby, A tensorial approach to computational continuum mechanics using object-oriented techniques. Comput. Phys. **12**, 620–631 (1998)
13. S. Vo, A. Kronenburg, O.T. Stein, E.R. Hawkes, Direct numerical simulation of non-premixed syngas combustion using OpenFOAM, in *High Performance Computing in Science and Engineering '16*, ed. by W.E. Nagel, D.B. Kröner, M.M. Resch (Springer, Heidelberg, 2016)
14. F. Zhang, H. Bonart, T. Zirwes, P. Habisreuther, H. Bockhorn, N. Zarzalis, Direct numerical simulation of chemically reacting flows with the public domain code OpenFOAM, in *High Performance Computing in Science and Engineering '14*, ed. by W.E. Nagel, D.H. Kröner, M.M. Resch (Springer, Berlin, Heidelberg, 2015), pp. 221–236
15. D.G. Goodwin, H.K. Moffat, R.L. Speth, Cantera: an object-oriented software toolkit for chemical kinetics, thermodynamics, and transport processes. Version 2.3.0b (2016), http://www.cantera.org

16. T. Zirwes, Weiterentwicklung und Optimierung eines auf OpenFOAM basierten DNS Lösers zur Verbesserung der Effizienz und Handhabung. Bachelor's thesis, Karlsruhe Institute of Technology, Germany, 2013
17. R.J. Kee, M.E. Coltrin, P. Glarborg, *Chemically Reacting Flow: Theory and Practice* (Wiley, Hoboken, 2005)
18. CHEMKIN 10131, Reaction Design: San Diego (2013)
19. F. Zhang, T. Zirwes, H. Nawroth, H. Bockhorn, C.O. Paschereit, Combustion generated noise: an environment related issue for future combustion systems. Energy Technol. **5**(7), 1045–1054 (2017)
20. OpenFOAM. The Open Source CFD Toolbox. User Guide (2014)
21. Cray Inc., Cray XC40 (2016), http://www.hlrs.de/systems/cray-xc40-hazel-hen
22. ForHLR II (2016), https://www.scc.kit.edu/dienste/forhlr2.php