Proceedings of ALGORITMY 2012
pp. 11–21

EFFICIENT MATRIX-FREE IMPLEMENTATION OF DISCONTINUOUS GALERKIN METHODS FOR COMPRESSIBLE FLOW PROBLEMS

ROBERT KLÖFKORN

Abstract. We discuss the matrix-free implementation of Discontinuous Galerkin methods for compressible flow problems, i.e. the compressible Navier-Stokes equations. For the spatial discretization the CDG2 method and for temporal discretization an explicit Runge-Kutta method is used. For the presented matrix-free approach we discuss asynchronous communication, shared memory parallelization, and automated code generation to increase the floating point performance of the code.

Key words. Discontinuous Galerkin, Navier-Stokes, asynchronous communication, shard memory, code generation

AMS subject classifications. 35L65, 35G50, 35Q30, 65Y05

1. Introduction. Discontinuous Galerkin (DG) methods have been intensively studied during the last decade for various types of flow problems. For compressible flow DG methods seem to be well suited due to their stability, robustness, and efficiency especially in parallel computations. Lately, several papers discussed the parallel efficiency of DG methods in advanced applications, e.g. [12, 13, 18, 19]. In this paper we want to pick up ideas about parallelization of DG methods (first given in [2]) and ideas about proper implementation, for example in [14], to achieve high on-node performance. A further aspect is the hybrid parallelization of DG methods which will become more important on today's many-core systems. As a test problem we consider the so called *Density Current* test case from [22], a well accepted test problem for atmospheric flow.

The paper is organized as follows: In Section 2 we describe the governing equations followed by the discretization in Section 3. In Section 4 we discuss the implementation of the DG method and present numerical results. A short summary and outlook is presented in Section 5.

2. Governing Equations. The system we investigate is governed by the viscous compressible flow equations in θ-form, for example, described in [10]. For $\Omega \subset \mathbb{R}^d$, $d = 1, 2, 3$, these equations can be written in the form

$$\begin{aligned} \partial_t \boldsymbol{U} &= \mathcal{L}(\boldsymbol{U}) \qquad \text{in } (0,T] \times \Omega, \\ \boldsymbol{U}(0,\cdot) &= \boldsymbol{U}_0(\cdot) \qquad \text{in } \Omega, \end{aligned} \tag{2.1}$$

with $\mathcal{L}(\boldsymbol{U}) := -\nabla \cdot \big(\mathcal{F}(\boldsymbol{U}) - \mathcal{A}(\boldsymbol{U})\nabla\boldsymbol{U}\big) + \mathcal{S}(\boldsymbol{U})$ and suitable boundary conditions. The vector of conservative variables is $\boldsymbol{U} = (\rho, \rho\boldsymbol{v}, \rho\theta)^{\mathrm{T}}$. ρ is the density, θ the potential temperature, and $\boldsymbol{v} = (v_1, ..., v_d)^T$ the velocity field. $\mathcal{F}(\boldsymbol{U}) = (\mathcal{F}_i(\boldsymbol{U}))$ and $\mathcal{A}(\boldsymbol{U})\nabla\boldsymbol{U} = ((\mathcal{A}(\boldsymbol{U})\nabla\boldsymbol{U})_i)$ for $i = 1, ..., d$, are given as follows:

$$\mathcal{F}_i(\boldsymbol{U}) = \begin{pmatrix} \rho v_i \\ \rho v_1 v_i + \delta_{1i} p \\ \vdots \\ \rho v_d v_i + \delta_{di} p \\ v_i \rho\theta \end{pmatrix}, \qquad \big(\mathcal{A}(\boldsymbol{U})\nabla\boldsymbol{U}\big)_i = \mu\rho \begin{pmatrix} 0 \\ \partial_i v_1 \\ \vdots \\ \partial_i v_d \\ \partial_i \theta \end{pmatrix}. \tag{2.2}$$

with μ being the kinematic viscosity. The source term $\mathcal{S}$ is only acting on the last component of the velocity field, i.e. $\mathcal{S}(\boldsymbol{U}) = (0, ..., 0, -\rho g, 0)^T$ with g being the constant of the gravitation force. To close the system we define the pressure p in accordance with the ideal gas law $p = p_0 \left(\frac{\rho R_d \theta}{p_0}\right)^\gamma$, where $\gamma = c_p/c_v$ is the heat capacity ratio and c_p and c_v are specific heat capacities under constant pressure and volume, respectively. The individual gas constant is defined as $R_d = c_p - c_v$. For the standard reference pressure p_0 we choose $p_0 = 10^5$ Pa.

3. Discretization. The considered discretization is based on the Discontinuous Galerkin (DG) approach and implemented in Dune-Fem [8] a module of the Dune framework [3, 4]. The current state of development allows for simulation of convection dominated (cf. [7]) as well as viscous flow (cf. [5]). We consider the CDG2 method from [5] of up to 5th order in space and 3rd order in time for the numerical investigations carried out in this paper.

3.1. Spatial Discretization. The spatial discretization is derived in the following way. Given a tessellation $\mathcal{T}_h$ of the domain Ω with $\cup_{K\in\mathcal{T}_h} K = \Omega$ the discrete solution $\boldsymbol{U}_h$ is sought in the piecewise polynomial space

$$V_h = \{\boldsymbol{v} \in L^2(\Omega, \mathbb{R}^{d+2}) : \boldsymbol{v}|_K \in [\mathcal{P}_k(K)]^{d+2},\ K \in \mathcal{T}_h\} \quad \text{for some } k \in \mathbb{N},$$

where $\mathcal{P}_k(K)$ is a space containing polynomials up to degree k. On quadrilateral or hexahedral elements we replace $\mathcal{P}_k$ with $\mathcal{Q}_k$ build by products of Legendre polynomials of up to degree k in each coordinate.

We denote with Γ_i the set of all intersections between two elements of the grid $\mathcal{T}_h$ and accordingly with Γ the set of all intersections, also with the boundary of the domain Ω. The following discrete form is not the most general but still covers a wide range of well established DG methods. For all basis functions $\boldsymbol{\varphi} \in V_h$ we define

$$\langle \boldsymbol{\varphi}, \mathcal{L}_h(\boldsymbol{U}_h)\rangle := \langle \boldsymbol{\varphi}, \mathcal{K}_h(\boldsymbol{U}_h)\rangle + \langle \boldsymbol{\varphi}, \mathcal{I}_h(\boldsymbol{U}_h)\rangle \tag{3.1}$$

with the element integrals

$$\langle \boldsymbol{\varphi}, \mathcal{K}_h(\boldsymbol{U}_h)\rangle := \sum_{K\in\mathcal{T}_h} \int_K \big((\mathcal{F}(\boldsymbol{U}_h) - \mathcal{A}(\boldsymbol{U}_h)\nabla \boldsymbol{U}_h) : \nabla\boldsymbol{\varphi} + S(\boldsymbol{U}_h)\cdot\boldsymbol{\varphi}\big), \tag{3.2}$$

and the surface integrals (by introducing appropriate numerical fluxes $\widehat{\mathcal{F}}_e$, $\widehat{\mathcal{A}}_e$ for the convection and diffusion terms, respectively)

$$\begin{aligned}\langle \boldsymbol{\varphi}, \mathcal{I}_h(\boldsymbol{U}_h)\rangle := &\sum_{e\in\Gamma_i} \int_e \big(\{\!\{\mathcal{A}(\boldsymbol{U}_h)^T\nabla\boldsymbol{\varphi}\}\!\}_e : [\![\boldsymbol{U}_h]\!]_e + \{\!\{\mathcal{A}(\boldsymbol{U}_h)\nabla\boldsymbol{U}_h\}\!\}_e : [\![\boldsymbol{\varphi}]\!]_e\big) \\ &- \sum_{e\in\Gamma} \int_e \big(\widehat{\mathcal{F}}_e(\boldsymbol{U}_h) - \widehat{\mathcal{A}}_e(\boldsymbol{U}_h)\big) : [\![\boldsymbol{\varphi}]\!]_e,\end{aligned} \tag{3.3}$$

where $\{\!\{\boldsymbol{V}\}\!\}_e = \frac{1}{2}(\boldsymbol{V}^+ + \boldsymbol{V}^-)$ denotes the average and $[\![\boldsymbol{V}]\!]_e = (\boldsymbol{n}^+ \otimes \boldsymbol{V}^+ + \boldsymbol{n}^- \otimes \boldsymbol{V}^-)$ the jump of the discontinuous function $\boldsymbol{V} \in V_h$ over element boundaries. For matrices $\sigma, \tau \in \mathbb{R}^{m\times n}$ we use standard notation $\sigma : \tau = \sum_{j=1}^m \sum_{l=1}^n \sigma_{jl}\tau_{jl}$. Additionally, for vectors $\boldsymbol{v} \in \mathbb{R}^m, \boldsymbol{w} \in \mathbb{R}^n$, we define $\boldsymbol{v} \otimes \boldsymbol{w} \in \mathbb{R}^{m\times n}$ according to $(\boldsymbol{v}\otimes\boldsymbol{w})_{jl} = \boldsymbol{v}_j\boldsymbol{w}_l$ for $1 \le j \le m$, $1 \le l \le n$.

The convective numerical flux $\widehat{\mathcal{F}}_e$ can be any appropriate numerical flux known for standard finite volume methods. For the results presented in this paper we choose $\widehat{\mathcal{F}}_e$ to be the HLL numerical flux function described in [17].

A wide range of diffusion fluxes $\widehat{\mathcal{A}}_e$ can be found in the literature, for a summary see [1]. We choose the CDG2 flux

$$\widehat{\mathcal{A}}_e(\boldsymbol{V}) := 2\chi_e\big(\mathcal{A}(\boldsymbol{V})\boldsymbol{r}_e([\![\boldsymbol{V}]\!]_e)\big)|_{K_e^-} \quad \text{for } \boldsymbol{V} \in V_h, \tag{3.4}$$

which was shown to be highly efficient for the Navier-Stokes equations (cf. [5]). Based on stability results, we choose K_e^- to be the element adjacent to the edge e with the smaller volume. $\boldsymbol{r}_e([\![\boldsymbol{V}]\!]_e) \in [V_h]^d$ is the lifting of the jump of $\boldsymbol{V}$ defined by

$$\int_\Omega \boldsymbol{r}_e([\![\boldsymbol{V}]\!]_e) : \boldsymbol{\tau} = -\int_e [\![\boldsymbol{V}]\!]_e : \{\!\{\boldsymbol{\tau}\}\!\}_e \quad \text{for all } \boldsymbol{\tau} \in [V_h]^d. \tag{3.5}$$

For the numerical experiments done in this paper we use $\chi_e = \frac{1}{2}\mathcal{N}_{\mathcal{T}_h}$, where $\mathcal{N}_{\mathcal{T}_h}$ is the maximal number of intersections one element in the grid can have (cf. [5]). For most of the numerical results in this paper we use conforming hexahedral elements and thus $\chi_e = 3$ for all $e \in \Gamma$.

3.2. Temporal discretization. The discrete solution $\boldsymbol{U}_h(t) \in V_h$ has the form $\boldsymbol{U}_h(t,x) = \sum_i \boldsymbol{U}_i(t)\boldsymbol{\varphi}_i(x)$. We get a system of ODEs for the coefficients of $\boldsymbol{U}(t)$ which reads

$$\boldsymbol{U}'(t) = f(\boldsymbol{U}(t), t) \text{ in } (0, T] \tag{3.6}$$

with $f(\boldsymbol{U}(t),t) = M^{-1}\mathcal{L}_h(\boldsymbol{U}_h(t),t)$, M being the mass matrix which is in our case block diagonal or even the identity, depending on the choice of basis functions. $\boldsymbol{U}(0)$ is given by the projection of $\boldsymbol{U}_0$ onto V_h.

Disregarding the order of the spatial discretization we use an explicit *Strong Stability Preserving* Runge-Kutta method (SSP-RK) of third order [11]. Implicit or semi-implicit Runge-Kutta solvers based on a Jacobian-free Newton-Krylov method (see [16]) are also available for the proposed DG method and implemented in the DUNE-FEM framework. The results and implementation techniques presented in this paper can be applied to explicit, implicit, or semi-implicit methods, as long as a **matrix-free** implementation of the discrete operator $\mathcal{L}_h$ is used.

4. Numerical results. In this section we discuss aspects of the implementation of our proposed DG method from the previous section. We consider the so called *Non-Linear Density Current* test case described in [22]. In this test case one examines the evolution of a cold bubble in a neutrally stratified atmosphere.

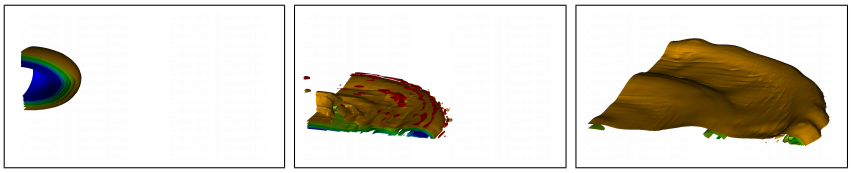

FIG. 4.1. *Results of the a 3D Density Current test case for a run on a Cartesian grid with* 18 490 *cells using the above described DG method of order* $k = 5$. *From left to right the iso-surfaces for the potential temperature* θ *at* $T = 0, 450, 900$ *seconds is shown.*

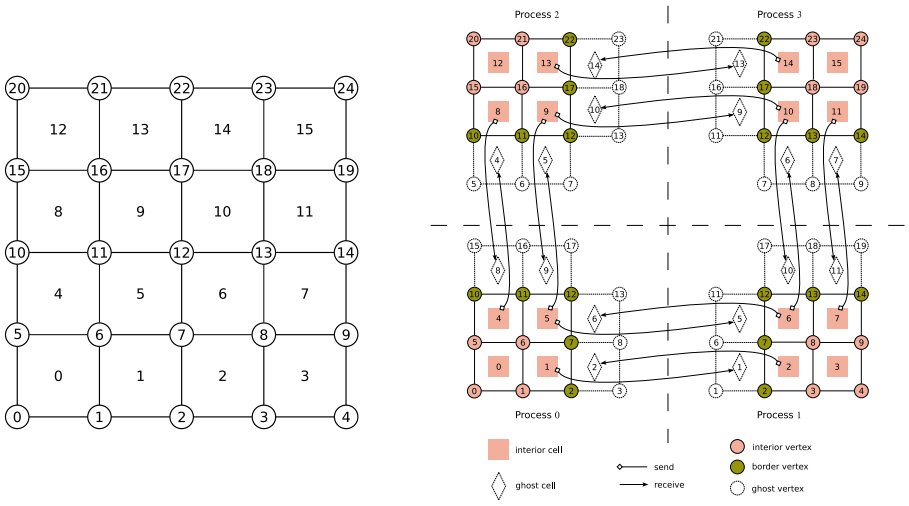

FIG. 4.2. *Serial grid split into 4 partitions including a ghost cell approach at the process boundaries. The arrows indicate the data exchange necessary for the evaluation of the discrete spatial operator* $\mathcal{L}_h$.

In Fig. 4.1 we can see the time evolution of this cold bubble. For our numerical tests in this section we will not run the full simulation. Instead we only run a few timesteps which is sufficient to get reliable information about the efficiency of the method.

We start with a detailed description of the communication procedures used during the evaluation of the spatial discrete operator $\mathcal{L}_h$. This is followed by an approach to hybrid parallelization, i.e. shard memory based micro parallelization. The section is concluded by a discussion on implementation of basis function evaluation which is the most expensive part in the evaluation of $\mathcal{L}_h$. Note once again that the following implementation suggestions rely strongly on the fact that the method is applied in the previously described **matrix-free** fashion.

4.1. Asynchronous communication. The parallelization techniques in DUNE [3, 4] and DUNE-FEM [8] are based on domain decomposition using MPI [9] for data exchange between multiple processes. The domain decomposition is usually achieved by using graph partitioning tools like ParMETIS [20]. We discuss some limitations of the DUNE grid interface at present state, namely missing intersection communication, only support of blocking communication, and possible grid traversal in communication procedures. We try to highlight possible solution for these problems.

For illustration we consider a grid decomposed into 4 domains as presented in Fig. 4.2. During the evaluation of the discrete operator $\mathcal{L}_h$ in (3.1) numerical fluxes at cell boundaries have to be evaluated. This means that for one element the information about the solution $\boldsymbol{U}_h$ on directly neighboring cells is needed. If the neighboring cell is a ghost cell, communication has to be used to obtain data of the solution $\boldsymbol{U}_h$ on this cell (see also Fig. 4.2). In fact, for the DG method it would be sufficient to only exchange values of the discrete function $\boldsymbol{U}_h$ at the process boundaries since for the evaluation of $\mathcal{L}_h$ neighboring information is only needed at the cell interfaces and not on the neighboring cell itself (see also (3.1)). Theoretically this allows to completely avoid ghost cells. However, the corresponding communication interfaces

for exchanging data on an intersection e are still missing in DUNE. Therefore, we have to rely on the ghost cell approach for the evaluation of numerical fluxes at process boundaries.

During each evaluation of the discrete spatial operator $\mathcal{L}_h$ one data transfer from interior cells to their ghost copies (see Fig. 4.2) has to be performed. The current state of the DUNE grid interface only supports blocking communication, i.e. once the communication has been started the simulation waits until all communications initiated by the same call are finished. Furthermore, the very general communication interface in DUNE does not specify whether the communication does involve iteration over grid entities or not (e.g. in `ALUGrid` [6] it involves grid traversal). For implicit solvers or even explicit solvers in non-adaptive simulations this should be avoided since grid traversal is to expensive compared to matrix-vector multiplication or similar operation that require communication. Thus communication that includes grid traversal decreases the performance of the solver drastically. This problem has been overcome in DUNE-FEM by building communication look-up tables for each discrete function space. During setup the standard DUNE communication is used to exchange the necessary linkage information to create the send-receive patterns for communication of degrees of freedom (DoF)s. As a result communication of DoFs does not include grid traversal anymore. This is described in detail in [8] including a scaling study comparing both communication strategies. The decoupling of the communication procedures from the DUNE grid interface allows to further improve the communication for allowing non-blocking or even truly asynchronous communication.

In the following we describe how the communication can be improved for our proposed DG method. Using the DUNE grid interface communication a natural way to exchange data for the evaluation of the discrete spatial operator would be an interior-ghost communication just before the evaluation of the discrete spatial operator. This guarantees that all necessary data for the evaluation of the numerical fluxes are present. In the following we call this **synchronous communication**. The drawback of this approach is, that every process has to wait until all communications have been finished before starting with the computation of $\mathcal{L}_h$. Newly investigated communication techniques such as **asynchronous communication** (cf. [21]) might drastically increase the strong scaling of the code. The basic idea is to hide network latency behind the evaluation of the element integrals since these integrals can be computed without information from other partitions. To achieve this we use the splitting of the discrete operator into element and surface integrals, i.e. $\mathcal{L}_h(\boldsymbol{U}_h) = \mathcal{K}_h(\boldsymbol{U}_h) + \mathcal{I}_h(\boldsymbol{U}_h)$, from equation (3.1). Since for the computation of $\mathcal{K}_h$ (given in equation (3.2)) on each cell K no neighboring information is needed and thus no data exchange between different processes is necessary. The improved computation of $\mathcal{L}_h$ is done in the following steps:

1. **Initiate communication** by sending interior data required by other processes for evaluation of numerical fluxes using the MPI function `MPI_Isend`. This method is non-blocking in the sense that the actual sending might not be finished on return of the method `MPI_Isend` (cf. [9]).
2. **Compute** $\mathcal{K}_h(\boldsymbol{U}_h)$ from (3.2) which only involves computation of element integrals and no data exchange is required.
3. **Receive data** for ghost cells, e.g. by using the MPI function `MPI_Iprobe` to check whether communication has been finished and if a message was received using `MPI_Recv` (cf. [9]) to copy the message to the corresponding storage on the ghost cells.

TABLE 4.1

Strong scaling and efficiency of the DG code on the supercomputer JUGENE (Jülich, Germany) ***without*** *and* ***with*** *asynchronous communication. P denotes the number of cores, $G := |\mathcal{T}_h|$ the overall number of grid elements, and $N := |V_h|$ denotes the number of degrees of freedom. $\bar{\eta}$ is the average run time in milliseconds needed to compute one timestep. $S_{512\to P} := \bar{\eta}_{512}/\bar{\eta}_P$ denotes the speed-up from 512 to P cores with respect to $\bar{\eta}$ and $E_{512\to P} := \frac{512}{P} S_{512\to P}$ being the efficiency.*

			synchronous comm.			asynchronous comm.		
P	G/P	N/P	$\bar{\eta}$	$S_{512\to P}$	$E_{512\to P}$	$\bar{\eta}$	$S_{512\to P}$	$E_{512\to P}$
512	474.6	296 630.8	68 579	—	—	46 216	—	—
4 096	59.3	37 324.9	14 575	4.71	0.59	6 294	7.34	0.91
32 768	7.4	4 634.8	5 034	13.62	0.21	949	48.71	0.76
65 536	3.7	2 317.4	—	—	—	504	91.70	0.72

4. **Compute** $\mathcal{I}_h(\boldsymbol{U}_h)$ from (3.3) which involves only surface integrals. Now, the values of the function $\boldsymbol{U}_h$ on the ghost cells are available.
5. **Accumulate** $\mathcal{L}_h(\boldsymbol{U}_h) = \mathcal{K}_h(\boldsymbol{U}_h) + \mathcal{I}_h(\boldsymbol{U}_h)$ (which is only a vector operation on the vector of degrees of freedom) and proceed as before.

In Table 4.1 we present the strong scaling of the DG code on the supercomputer JUGENE (IBM BlueGene/P, Jülich, Germany). Similar results for unsteady flow problems using DG methods were obtained by Hindenlang et al. [12] on up to 4 096 cores. We compute the Density Current in 3D using basis functions of order $k = 4$ including automated code generation (see Section 4.3) on a Cartesian grid with 243 000 cells in total. The overall memory consumption in a serial run is about 6.1 GB. This means that this problem can still be run on a standard desktop machine (at time of writing). Therefore, this test also shows the possibility of the presented DG code to scale standard test problems to a huge number of cores. In order to save computation time on the JUGENE which is only very limited available, we only compute 30 timesteps per run and compare the average run time per timestep $\bar{\eta}$. In the left part of Table 4.1 we can see that using the usual synchronous communication would not allow us to scale our problem efficiently to 32 768 cores. The efficiency is only about 21% which means that a huge portion of our run time is spend in waiting for communication of our data. We can conclude that the standard synchronous communication does not allow to scale our problem to 32 768 cores. However, the right part of Table 4.1 indicates that the use of asynchronous communication allows to scale our problem to 65 536 cores with a reasonable efficiency of about 72%. This means that the proposed asynchronous communication drastically increases the strong scaling of the code and allows also to compute problems of moderate size on a huge number of cores. Note that the run on 65 536 cores uses an average of 3.7 cells per core (G/P) resulting in an average of only 2 317.4 DoFs per core (N/P). However, the internal simple partitioning of `SPGrid` [15] in this case lead to a domain decomposition with maximal 8 elements and minimal 1 element per core which is far from being optimal. A better load balance will further increase the scalability in this case.

4.2. Shared memory parallelization. Similar to the domain decomposition presented in Section 4.1 in this section we present a shared memory parallelization for our DG solver based on POSIX threads. It turns out that this can be achieved with minimal changes to the implementation. Given a computational grid again the basic idea is to split the grid in several domains, one domain for each thread. Then each thread only computes the operator $\mathcal{L}_h$ on cells assigned to this thread. As before, the domain decomposition is obtained by using a graph partitioning approach. Since

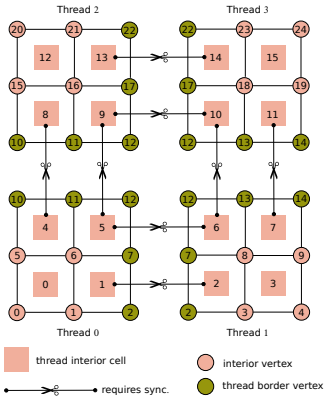

FIG. 4.3 *Serial grid split into 4 partitions for a the thread based partitioning.*

TABLE 4.2

Strong scaling and efficiency of the DG solver on a 16 core AMD Barcelona system. The shared memory parallelization has been carried out using POSIX threads. Q denotes the number of threads used. G/Q is the number of elements per thread and N/Q the number of Dofs per thread. $\tilde{F}$ denotes the ratio between intersections at the thread domain boundary and the overall number of intersections per thread. $\bar{\eta}$ is the average run time per time step and $S_{1\to Q} := \bar{\eta}_1/\bar{\eta}_Q$ denotes the speed-up from 1 to Q threads with respect to $\bar{\eta}$ and $E_{1\to Q} := \frac{1}{Q} S_{1\to Q}$.

coarse grid

Q	G/Q	N/Q	$\tilde{F}$	$\bar{\eta}$	$S_{1\to Q}$	$E_{1\to Q}$
1	72	45000	0	842	—	—
2	36	22500	0.08	416	2.02	1.01
4	18	11250	0.15	205	4.11	1.03
8	9	5625	0.38	110	7.66	0.96
16	4.5	2812.5	0.68	63	13.4	0.84

fine grid

Q	G/Q	N/Q	$\tilde{F}$	$\bar{\eta}$	$S_{1\to Q}$	$E_{1\to Q}$
1	243	151875	0	2185	—	—
2	121.5	75937.5	0.06	1135	1.92	0.96
4	60.75	37968.8	0.13	523	4.18	1.05
8	30.38	18984.4	0.28	264	8.28	1.04
16	15.19	9492.2	0.45	142	15.39	0.96

all data are available to all threads simultaneously, no complicated data migration has to be implemented. In fact it is sufficient to implement an additional filtering mechanism for the already available DUNE grid iterators. The modified iterators should only stop on elements assigned to the thread the iterator was created on. For the implementation of iterators that only traverse a thread partition we require the underlying DUNE grid to be *thread safe* with respect to operations that do not change the grid such as traversal of elements.

Instead of data communication, the difficulty here is to avoid that different threads write to the same memory address simultaneously. In our example this can in particular happen during the calculation of the numerical fluxes because the numerical flux is only computed once per intersection (using the conservation property) for efficiency reasons. This means that given the grid from Fig. 4.3 the computations of the numerical fluxes for the intersection between element 5 and 6 can causes clashes between thread 0 and thread 1 when updating the vector of coefficients. To avoid such race conditions one could introduce so called *mutex locks* that make sure that certain parts of the code are only executed by one thread at the same time. In our implementation we avoid write clashes by computing numerical fluxes at thread domain boundaries twice, once for each thread at the thread domain boundary. This way write clashes can never occur at the expense of an increased numerical complexity.

To investigate the strong scaling behavior of our DG solver we again compute 30 timesteps for a coarse grid with 72 elements and a finer grid with 243 elements. We compare the average run time per timestep to obtain scaling information. The computations have been done on a AMD Barcelona system (Processor 8347) with 4 sockets and 4 cores per socket. In Table 4.2 we present the strong scaling results for the DG solver using basis functions with polynomial degree $k = 4$ and automated

code generation (see Section 4.3). We see that the scaling up to 8 cores is very good, mostly above 95%. This can be achieved with pinning of threads to specific cores, in this case on separate sockets resulting in increased local cache sizes relative to the problem size. This explains the efficiency rates of above 1 for the first runs. Using the full system with 16 threads we see that the efficiency is decreased to a still acceptable rate of about 84% in the coarse grid case. Decomposing the domain into 16 parts results in 68% of the intersections being located at a domain boundary and thus fluxes on theses interfaces have to be computed twice. For the finer grid the scaling is slightly better, as expected, which is clear since the ratio between interior intersections and intersections at the thread domain boundary is smaller. Again we conclude that a minimal amount of around 2 500 DoFs per thread (core) is needed for efficient parallel computation of the operator $\mathcal{L}_h$.

4.3. Automated code generation. In this section we investigate the performance of our DG solver with respect to *floating point operations per second* (FLOPS).

Recalling the structure of the discrete operator $\mathcal{L}_h$ (3.1) we see that it mainly consists of integrals on elements and intersections. For convenience we only consider the simple source term integral from (3.1) with an appropriate set of quadrature points Q_K for element $K \in \mathcal{T}_h$

$$\int_K S(\boldsymbol{U}_h) \cdot \boldsymbol{\varphi} \approx \sum_{(\omega,\lambda)\in Q_K} \omega \mathcal{S}(\boldsymbol{U}_h(\lambda)) \cdot \boldsymbol{\varphi}(\lambda) \qquad \forall\, \boldsymbol{\varphi} \in V_h. \tag{4.1}$$

With $\boldsymbol{U}_h(\boldsymbol{x})_{|K} = \sum_i \boldsymbol{U}_i \boldsymbol{\varphi}_i(\boldsymbol{x})$ we see that for the evaluation of the integrals in the discrete operator $\mathcal{L}_h$ we have to evaluate, for example, all basis functions on element K for all quadrature points in Q_K. Since the interface in Dune-Fem for evaluation of basis functions allow to do this at once for all basis functions for a given quadrature it makes sense to optimize these matrix-matrix like multiplications (MMM). Similar ideas for DG methods on GPUs have been presented in [14].

For flexibility reasons the design of basis function sets and quadratures in Dune-Fem have been made such that the number of basis functions and number of quadrature points is provided *dynamically*, i.e. information that is available during run time, but not at compile time. Therefore, the for-loops in the MMM cannot be unrolled by the compiler and thus possibly resulting in non-optimal code.

To overcome this problem we implemented an approach based on automated code generation. This simply means that instead of compiling our code once, we compile it twice. First, we compile the code in the usual way and run one time step for a small problem size. During this run we determine all combinations of basis functions and quadrature points that have been used. For these combinations we are now able to provide more efficient auto-generated implementations of the expression in (4.1). In the second compilation we can now make use of the previously determined combinations of number of basis function and quadrature points which are now statically available at compile time.

In the following we compare both approaches in terms of FLOPS performance. The performance measurements have been done on an Intel Core i7 (Nehalem) CPU Q740 @ 1.73 Ghz with a theoretical peak performance of 27.728 GLFOPS. In Table 4.3 we present FLOPS measurement of our DG solver using a Cartesian grid, i.e. `SPGrid` [15], and a fully unstructured grid, i.e. `ALUGrid` [6], as computational grid. All computations use the shared memory parallelization described in Section 4.2 with 4 threads on our considered CPU with 4 cores. The FLOPS were measured with

TABLE 4.3

Performance measurement of the DG code for polynomial orders $k = 1, 2, 3, 4$ using a Cartesian grid (`SPGrid`) and fully unstructured grid (`ALUGrid`). N/G denotes the number of DoFs per element, $\bar{\eta}$ denotes the average run time per time step. GFLOPS denote the measured performance of the DG code in GFLOPS and PERF% denotes the ratio between the measured performance and the theoretical peak performance of about 27.728 GLFOPS.

`SPGrid`

		without code generation			with code generation		
k	N/G	$\bar{\eta}$	GFLOPS	PERF %	$\bar{\eta}$	GFLOPS	PERF %
1	40	136	3.08	11.1	127	3.35	12.08
2	135	526	3.90	14.06	431	4.89	17.63
3	320	1805	4.35	15.68	1241	6.45	23.26
4	625	5261	4.58	16.51	3592	6.8	24.52

`ALUGrid`

		without code generation			with code generation		
k	N/G	$\bar{\eta}$	GFLOPS	PERF %	$\bar{\eta}$	GFLOPS	PERF %
1	40	162	2.66	9.59	149	2.81	10.13
2	135	578	3.27	11.79	485	4.25	15.32
3	320	1880	4.09	14.74	1352	5.73	20.66
4	625	5318	4.37	15.75	3647	6.77	24.41

the open source tool `likwid` [23] at version 2.2.1. We see that on `SPGrid` for the version without code generation we obtain reasonable performance rates of about $11\% - 16.5\%$ of the theoretical peak performance. Using the auto generated code we are able to increase the performance to $12\% - 24.5\%$. Especially for the higher order method we are able to increase the methods performance to about $1/4$ of the theoretical peak performance. The same measurements for `ALUGrid` show a slightly inferior performance compared to `SPGrid`. Without automated code generation the performance rates are between $9.5\%-15.75\%$ whereas with automated code generation the performance can be increased to $10\% - 24.4\%$. For the higher order methods the difference between the Cartesian and the fully unstructured grid is very small, due to the high amount of DoFs per element. The results also show that using a higher order method the increased data locality leads to very good performance of the code.

5. Conclusion and Outlook. In this paper we gave a brief description of a DG solver for compressible flow. We discussed aspects of implementation (based on DUNE-FEM and DUNE) of the DG solver with respect to asynchronous communication, thread based micro partitioning, and code optimization to achieve high floating point performance. For our matrix-free implementation we could prove that using asynchronous communication even small problems can be scaled to a high number of cores (65 536 in our case). This can probably be improved by replacing the simple partitioning in `SPGrid` with a space filling cure based approach. Further improvement could be achieved by combining the MPI parallelization with the shared memory parallelization based on POSIX threads. Also the thread based parallelization showed good scaling even for small grids. Both parallelization techniques require a minimal amount of around 2 500 DoFs per core which in case of the DG method boils down to a few number of elements per core. Due to technical problems we were not able to provide results for the scaling of the hybrid version (MPI + POSIX threads) of the code on the JUGENE, yet. This is ongoing work and we expect that the hybrid version of the code will scale even better. To further increase the codes floating point performance – which is currently between 10% and 25% of the theoretical peak

performance depending on the order of the method – we will make use of the SIMD architecture of modern CPUs which has not been done yet.

Acknowledgments. The author acknowledges Gregor Gassner and Claus-Dieter Munz (both IAG, Uni Stuttgart) for helpful comments on implementation of DG methods, Georg Hager (RRZE, Uni Erlangen) and Gerhard Wellein (INF, Uni Erlangen) for fruitful discussions about code optimization. Furthermore, thanks to Slavko Brdar (AAM, Uni Freiburg) for providing the pictures in Fig. 4.1, and Martin Nolte (AAM, Uni Freiburg) and Andreas Dedner (Mathematics Institute, Uni Warwick) for helpful comments. Major part of the computation time was kindly provided by the Jülich Supercomputing Centre.

REFERENCES

[1] D.N. Arnold, F. Brezzi, B. Cockburn, and L.D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39(5):1749–1779, 2002.

[2] A. Baggag, H. Atkins, and D.E. Keyes. Parallel Implementation of the Discontinuous Galerkin Method. In *Proceedings of Parallel CFD'99*, pages 115–122, 1999.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. *Computing*, 82(2–3):121–138, 2008.

[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part I: Abstract Framework. *Computing*, 82(2–3):103–119, 2008.

[5] S. Brdar, A. Dedner, and R. Klöfkorn. Compact and stable Discontinuous Galerkin methods for convection-diffusion problems. *SIAM J. Sci. Comput.*, 34(1), 2012.

[6] A. Burri, A. Dedner, R. Klöfkorn, and M. Ohlberger. An efficient implementation of an adaptive and parallel grid in dune. In E. Krause et al., editor, *Computational Science and High Performance Computing II*, volume 91, pages 67–82. Springer, 2006.

[7] A. Dedner and R. Klöfkorn. A generic stabilization approach for higher order Discontinuous Galerkin methods for convection dominated problems. *J. Sci. Comp.*, 47(3):365–388, 2011.

[8] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive scientific computing: Abstraction principles and the Dune-Fem module. *Computing*, 89(1), 2010.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard. Version 2.2.* High Performance Computing Center Stuttgart (HLRS), 2009.

[10] F.X. Giraldo and M. Restelli. A Study of Spectral Element and Discontinuous Galerkin Methods for the Navier-Stokes equations in Nonhydrostatic Mesoscale Atmospheric Modeling : Equations Sets and Test Cases. *J. Comput. Phys.*, 227:3849–3877, 2008.

[11] S. Gottlieb, C.-W. Shu, and E. Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Rev.*, 43(1):89–112, 2001.

[12] F. Hindenlang, G.J. Gassner, Ch. Altmann, A. Beck, M. Staudenmaier, and C.-D. Munz. Explicit discontinuous Galerkin methods for unsteady problems. *Comput. Fluids*, 2012.

[13] James F. Kelly and Francis X. Giraldo. Continuous and discontinuous galerkin methods for a scalable three-dimensional nonhydrostatic atmospheric model: Limited-area mode. *Journal of Computational Physics*, 2012.

[14] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *J. Comput. Phys.*, 228:7863–7882, November 2009.

[15] R. Klöfkorn and M. Nolte. Performance Pitfalls in the DUNE Grid Interface. In A. Dedner, B. Flemisch, and R. Klöfkorn, editors, *Advances in DUNE*. Springer, 2012.

[16] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *J. Comput. Phys.*, 193(2):357–397, 2004.

[17] D. Kröner. *Numerical Schemes for Conservation Laws*. Wiley & Teubner, Stuttgart, 1997.

[18] R.D. Nair, H.W. Choi, and H.M. Tufo. Computational aspects of a scalable high-order discontinuous Galerkin atmospheric dynamical core. *Computers and Fluids*, 30:309319, 2009.

[19] P.-O. Persson. Scalable Parallel Newton-Krylov Solvers for Discontinuous Galerkin Discretizations. In *Proc. of the 47th AIAA Aerospace Sciences Meeting and Exhibit*, 2009.

[20] K. Schloegel, G. Karypis, and V Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.

[21] G. Schubert, H. Fehske, G. Hager, and G. Wellein. Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. *Parallel Processing Letters*, 21(3):39–358, 2011.

[22] J. M. Straka, R. B. Wilhelmson, L. J. Wicker, J. R. Anderson, and K. K. Droegemeier. Numerical Solutions of a Non-Linear Density Current: A Benchmark Solution and Comparison. *Int. J. Num. Meth. Fluids*, 17:1–22, 1993.

[23] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.