EXA-DUNE: Flexible PDE Solvers, Numerical Methods and Applications

Peter Bastian[1], Christian Engwer[2], Dominik Göddeke[3], Oleg Iliev[4], Olaf Ippisch[5], Mario Ohlberger[2], Stefan Turek[3], Jorrit Fahlke[2], Sven Kaulmann[2], Steffen Müthing[1], and Dirk Ribbrock[3]

[1] Interdisciplinary Center for Scientific Computing, Heidelberg University, Im Neuenheimer Feld 368, D-69120 Heidelberg, Germany
[2] Institute for Computational and Applied Mathematics, University of Münster Orleans-Ring 10, D-48149 Münster, Germany
[3] Department of Mathematics, TU Dortmund, Vogelpothsweg 87, D-44227 Dortmund, Germany
[4] Fraunhofer Institute for Industrial Mathematics ITWM Fraunhofer-Platz 1, D-67663 Kaiserslautern, Germany
[5] Institut für Mathematik, TU Clausthal-Zellerfeld, Erzstr. 1, D-38678 Clausthal-Zellerfeld, Germany

Abstract. In the EXA-DUNE project we strive to (i) develop and implement numerical algorithms for solving PDE problems efficiently on heterogeneous architectures, (ii) provide corresponding domain-specific abstractions that allow application scientists to effectively use these methods, and (iii) demonstrate performance on porous media flow problems. In this paper, we present first results on the hybrid parallelisation of sparse linear algebra, system and RHS assembly, the implementation of multiscale finite element methods and the SIMD performance of high-order discontinuous Galerkin methods within an application scenario.

1 The EXA-DUNE Project

Many processes from science and engineering can be modelled with stochastic or parameterised partial differential equations (PDEs). Despite increasing computational capacities, many of these problems are still only solvable with severe simplifications. This is particularly true if not only single forward problems are considered, but rather uncertainty quantification, parameter estimation or optimisation in engineering applications are investigated.

Within the EXA-DUNE[1] project we pursue three different routes to make progress towards exascale: (i) we develop new computational algorithms and implementations for solving PDEs that are highly suitable to better exploit the performance offered by prospective exascale hardware, (ii) we provide domain-specific abstractions that allow mathematicians and application scientists to exploit (exascale) hardware with reasonable effort in terms of programmers' time (a metric that we consider highly important) and (iii) we showcase our methodology to solve complex application problems of flow in porous media.

[1] `http://www.sppexa.de/general-information/projects.html#EXADUNE`

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 530–541, 2014.

Software development, in the scope of our work for the numerical solution of a wide range of PDE problems, faces contradictory challenges. On the one hand, users and developers prefer flexibility and generality, on the other hand, the continously changing hardware landscape requires algorithmic adaptation and specialisation to be able to exploit a large fraction of peak performance.

A framework approach for entire application domains rather than distinct problem instances facilitates code reuse and thus substantially reduces development time. In contrast to the more conventional approach of developing in a 'bottom-up' fashion starting with only a limited set of problems and solution methods (likely a single problem/method), frameworks are designed from the beginning with flexibility and general applicability in mind so that new physics and new mathematical methods can be incorporated more easily. In a software framework the generic code of the framework is extended by the user to provide application specific code instead of just calling functions from a library. Template meta-programming in C++ supports this extension step in a very efficient way, performing the fusion of framework and user code at compile time which reduces granularity effects and enables a much wider range of optimisations by the compiler. In this project we strive to redesign components of the DUNE framework [3,2] in such a way that hardware-specific adaptations based on the experience acquired within the FEAST project [15] can be exploited in a transparent way without affecting user code.

Future exascale systems are characterised by a massive increase in node-level parallelism, heterogeneity and non-uniform access to memory. Current examples include nodes with multiple conventional CPU cores arranged in different sockets. GPUs require much more fine-grained parallelism, and Intel's Xeon Phi design shares similarities with both these extremes. One important common feature of all these architectures is that reasonable performance can only be achieved by explicitly using their (wide-) SIMD capabilities. The situation becomes more complicated as different programming models, APIs and language extensions are needed, which lack performance portability. Instead, different data structures and memory layouts are often required for different architectures. In addition, it is no longer possible to view the available off-chip DRAM memory within one node as globally shared in terms of performance. Accelerators are typically equipped with dedicated memory, which improves accelerator-local latency and bandwidth substantially, but at the same time suffers from a (relatively) slow connection to the host. Due to NUMA (non-uniform memory access) effects, a similar (albeit less dramatic in absolute numbers) imbalance can already be observed on multi-socket multi-core CPU systems. There is common agreement in the community that the existing MPI-only programming model has reached its limits. The most prominent successor will likely be 'MPI+X', so that MPI can still be used for coarse-grained communication, while some kind of shared memory abstraction is used within MPI processes at the UMA level.

Our work within the EXA-DUNE project currently targets pilot applications in the field of porous media flow. These problems are characterised by coupled elliptic/parabolic-hyperbolic PDEs with strongly varying coefficients and highly

anisotropic meshes. The elliptic part mandates robust solvers and thus does not lend itself to the current trend in HPC towards matrix-free methods with their beneficial properties in terms of memory bandwidth and/or FLOPs/DOF ratio; typical matrix-free techniques like stencil-based geometric multigrid are not suited to those types of problems. For that reason, we aim at algebraic multigrid (AMG) preconditioners known to work well in this context, and work towards further improving their scalability and (hardware) performance. Discontinuous Galerkin (DG) methods are employed to increase data locality and arithmetic intensity. Matrix-free techniques are investigated for the hyperbolic/parabolic parts.

In this paper we report on the current state of the EXA-DUNE project. As message passing parallelism is well established in DUNE (as documented by the inclusion of DUNE's solver library in the High-Q-Club[2]), we concentrate on core/node level performance. Regarding the three 'exa-avenues' identified in the project, implementations of multiscale reduced basis and high-order spectral DG methods are treated in Sections 3 and 4, hybrid parallelisation of finite element assembly and sparse linear algebra is covered in Section 2 and preliminary results for density-driven flow in porous media are shown in Section 4.

2 Hybrid Parallelism in DUNE

In the following, we introduce the 'virtual UMA node' concept at the heart of our hybrid parallelisation strategy, and ongoing current steps to incorporate this concept into the assembly and solver stages of our framework.

2.1 UMA Concept

Current and upcoming HPC systems are characterised by two trends which greatly increase the complexity of efficient node-level programming: (i) A massive increase in the degree of parallelism restricts the amount of memory and bandwidth available to each compute unit, and (ii) the node architecture becomes increasingly heterogeneous. Consequently, on modern multi-socket nodes the memory performance depends on the location of the memory in relation to the compute core (NUMA). The problem becomes even more pronounced in the presence of accelerators like MICs or GPUs, for which memory accesses might have to traverse the PCIe bus, severely limiting bandwidth and latency. To illustrate this issue, we consider the relative runtime of an iterative linear solver (Krylov-DG), as shown in Table 1: An identical problem is solved with different mappings to MPI processes and threads, on a representative 4-socket server with AMD Opteron 6172 12-core processors and 128 GB RAM. On this architecture, a UMA domain comprises half a socket (6 cores), and thus, (explicit or implicit) multi-threading beyond 6 cores actually yields slowdowns. This experiment validates our design decision to regard heterogeneous nodes as a collection of 'virtual

[2] `http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/_node.html`

Table 1. Poisson on the unit cube, discretised by the DG-SIPG method, timings for 100 Krylov iterations. Comparison of different MPI / shared memory mappings for varying polynomial degree p of the DG discretisation and mesh width h. Timings $t_{M/T}$ and speedups for varying numbers of MPI processes M and threads per process T.

p	h^{-1}	$t_{48/1}[s]$	$t_{8/6}[s]$	$\frac{t_{48/1}}{t_{8/6}}$	$t_{4/12}[s]$	$\frac{t_{48/1}}{t_{4/12}}$	$t_{1/48}[s]$	$\frac{t_{48/1}}{t_{1/48}}$
1	256	645.1	600.2	1.07	1483.3	0.43	2491.7	0.26
2	128	999.5	785.7	1.27	1320.7	0.76	2619.0	0.38
3	64	709.6	502.9	1.41	1237.2	0.57	1958.2	0.36

UMA nodes' on the MPI level: Internal uniform memory access characteristics are exploited by shared-memory parallelism, while off-node communication is handled via (classical/existing) message passing.

2.2 Finite Element Assembly

Assembling the finite element operator or the residual vector typically involves two user-level inputs: The assembler iterates through the grid cells of a given mesh, and for each grid cell a local operator is evaluated, which computes the local contributions to the global stiffness matrix or the residual vector. Following DUNE's general approach, we implement threading and vectorisation on top of the existing grid abstraction.

Globally the grid is partitioned using the existing MPI layer. Within each UMA node system threads are used to share the workload among all cores. For a user-defined number of concurrent threads the grid is locally partitioned such that each thread handles the same amount of work. On the finest level vectorisation (SIMD, ILP) is required to fully exploit the hardware. SIMD has the largest impact in the local operator which also poses the biggest challenge, as this is user code. The resulting requirement of fully exploiting SIMD in that setting without exposing users to the details of vectorisation presents an additional problem compared to the linear algebra, where the number of kernels is much smaller.

Multi-threading support is implemented on top of the existing grid interface, thus we can easily compare different strategies for the local partitioning of a mesh $\mathcal{T}(\Omega)$. Experiments are carried out on an Intel Xeon E7-4850 with 10 cores (20 hyperthreads), 2 GHz and 12 GB RAM and on an Intel Xeon Phi 5110P, with 60 cores (240 hyperthreads), 1 GHz and 8 GB RAM. Many bottlenecks for multi-threading only become visible on many-core systems like the Xeon Phi. SIMD experiments are carried out on an Intel Core i5-3340M with 2 cores (4 hyperthreads), 2.7 GHz and 8 GB RAM and a 256-bit SIMD unit (AVX). See [5] for more details. Our experiments indicate that the additional complexity of partitioning the node-local mesh into per-thread blocks that optimise properties like surface-to-volume ratio, e.g. using graph partitioning libraries like METIS or SCOTCH, does not pay off; those approaches impose prohibitive setup and memory penalties. Instead, a *ranged* partitioning strategy showed the best overall

Table 2. Comparison of different polynomial degrees k, number of threads P, and hardware X. Time per DOF t_P^X [μs] and efficiency E_P^X of the Jacobian assembly using ranged partitioning and entity-wise locking. We see a clear benefit from higher order discretisations, due to the increased algorithmic intensity.

k	t_1^{CPU}	t_{10}^{CPU}	t_{20}^{CPU}	E_{10}^{CPU}	E_{20}^{CPU}	t_1^{PHI}	t_{60}^{PHI}	t_{120}^{PHI}	t_{240}^{PHI}	E_{60}^{PHI}	E_{120}^{PHI}	E_{240}^{PHI}
0	4.59	0.74	0.54	62%	42%	59.57	1.33	1.17	1.20	75%	43%	21%
1	1.38	0.22	0.17	62%	42%	18.92	0.37	0.27	0.26	84%	57%	30%
2	1.10	0.15	0.12	72%	46%	17.12	0.32	0.21	0.19	90%	69%	38%
3	1.29	0.16	0.13	79%	50%	19.84	0.36	0.23	0.20	92%	72%	41%
4	1.52	0.18	0.15	87%	49%							
5	1.81	0.21	0.18	88%	51%							

performance. We define consecutive iterator ranges of the size $|\mathcal{T}|/P$. This is efficiently implemented using entry points in the form of begin and end iterators. The memory requirement is $O(P)$ and thus will not strain the bandwidth.

Data access is critical during the assembly, as different local vectors and local matrices contribute to the same global entries. Two approaches are possible to avoid race conditions: locking and colouring. *Entity-wise locks* are expected to give very good performance, as they correspond to the granularity of the critical sections. The downside is the additional memory requirement of $O(|\mathcal{T}|)$. With a ranged partitioning and entity-wise locking, or with colouring, we obtain good performance on multi-core CPUs and on many-core systems alike. The performance gain from colouring is negligible, but increases code complexity, so that this approach is less favourable.

Timings for ranged partitioning and entity-wise locking are presented in Table 2. As a benchmark we consider the assembly of the Jacobian and measure strong scalability. Discretisations using different polynomial orders are evaluated and the problem sizes are chosen such that the global number of unknowns is roughly the same. The results indicate the benefit of higher order trial and test functions, due to the increased arithmetic intensity in the local operator. The absolute timings show a significant issue for the Xeon Phi, which can only exhibit its full performance if the code is able to use the 512-bit wide SIMD instructions.

Vectorising computations in the local operator requires pursuing different avenues depending on the number of local DOFs / quadrature points: For high-order discretisations, good performance can be achieved by simply unrolling / vectorising the existing loops (cf. results in Sec. 4). For low-order methods this approach is only feasible if the number of DOFs / quadrature points is a multiple of the SIMD width, limiting the applicability of this technique. We thus follow a different approach to transparently add SIMD parallelism at the level of the local operator and vectorise over N elements, operating on the same local function space, and encapsulate data in a packed C++ data type. This approach is inspired by [7]; their Vc library is also used for the presented preliminary results. The packed data consists of a vector of N doubles. Using operator overloading an arithmetic operation $a \odot b$ is mapped to the component-wise evaluation $a_i \odot b_i$.

All interfaces providing local information of the N cells are now vectorised as well as the residual vector and the local matrix. In particular, information like the Jacobian of the geometric mapping and the determinant of the Jacobian are now provided for all N elements.

We investigate a 3D Q_2 discretisation of the Poisson problem with 262 144 cells and benchmark the assembly of the residual and the Jacobian on a structured grid on a single core. First results show a speedup of 1.8 (SSE, 2 lanes) and 2.6 (AVX, 4 lanes) for the Jacobian and 1.7 (SSE) and 2.3 (AVX) for the residual. This is measured without the scatter operation into the global matrix as this is not yet optimised — if we include scattering in the timing the speedup is, e.g., 1.7 for the Jacobian and AVX. Even without scattering some operations are not vectorised yet, so we do not obtain the full speedup, but we can show that it is possible to add SIMD parallelism to the local operator with only minimal restrictions for the user.

2.3 Sparse Linear Algebra and Solvers

Designing effcient implementations and realisations of solvers effectively boils down to (i) a suitable choice of data structures for sparse matrix-vector multiply, and (ii) numerical components of the solver, i.e., preconditioners.

DUNE's current matrix format, (block) compressed row storage, is ill-suited for modern hardware and SIMD, as there is no way to efficiently and generally expose a block structure that fits the size of the SIMD units. We have thus extended the SELL-C-σ matrix format introduced in [8] which is a tuned variant of the sorted ELL format known from GPUs, to be able to efficiently handle block structures [11].

As we mostly focus on solvers for DG discretisations, which lend themselves to block-structured matrices, this is a valid and generalisable decision. The standard approach of requiring matrix block sizes that are multiples of the SIMD size is not applicable in our case because the matrix block size is a direct consequence of the chosen discretisation. In order to support arbitrary block sizes, we interleave the data from N matrix blocks given a vector unit of size N, an approach introduced in [4]. This allows us to easily vectorise existing scalar algorithms by having them operate on multiple blocks in parallel, an approach that works as long as there are no data-dependent branches in the original algorithm. Sparse linear algebra is typically memory bandwidth bound, and thus, the main advantage of the block format is the reduced number of column block indices that need to be stored (as only a single index is required per block). With growing block size, this bandwidth advantage quickly approaches 50% of the overall required bandwidth.

So far, we have implemented the SELL-C-σ building blocks (vectors, matrices), and a (block) Jacobi preconditioner which fully inverts the corresponding subsystem; for all target architectures (CPU, MIC, CUDA). Moreover, there is an implementation of the blocked version for multi-threaded CPUs and MICs. While the GPU version is implemented as a set of CUDA kernels, we have not used any intrinsics for the standard CPU and the MIC – instead we rely on the

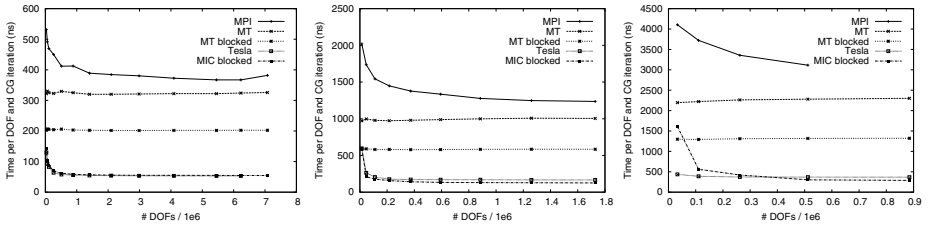

Fig. 1. Normalised execution time of the (block) Jacobi preconditioned CG solver for polynomial degrees $p = 1, 2, 3$ (left to right) of the DG discretisation. The multithreaded (MT) and MIC versions use a SIMD block size of 8. Missing data points indicate insufficient memory.

auto-vectorisation features of modern compilers without performance penalty [11]. Due to the abstract interfaces in our solver packages, all other components like the iterative solvers can work with the new data format without any changes. Finally, a new backend for our high-level PDE discretisation package enables a direct assembly into the new containers, avoiding the overhead of a separate conversion step. Consequently, users can transparently benefit from our improvements through a simple C++ typedef.

We demonstrate the benefits of our approach for a linear system generated by a 3D stationary diffusion problem on the unit cube with unit permeability, discretised using a weighted SIPG DG scheme [6]. Timings of 100 iterations of a CG solver using a (block) Jacobi preconditioner on a single-socket Intel Sandy Bridge machine (8 GB DDR3-1333 RAM, 2 GHz 4-core Intel Core i7-2635QM, no hyper-threading) which supports 256-bit wide SIMD using AVX instructions, on a NVIDIA Tesla C2070 for the GPU measurements and on a Intel Xeon Phi 7120P, are presented in Figure 1, normalised per iteration and DOF.

As can be seen, switching from MPI to threading affords moderate improvements due to the better surface-to-volume ratio of the threading approach, but we cannot expect very large gains because the required memory bandwidth is essentially identical. Accordingly, switching to the blocked SELL-C-σ format consistently yields good improvements due to the lower number of column indices that need to be loaded, an effect that becomes more pronounced as the polynomial degree grows due to larger matrix block sizes. Finally, the GPU and the MIC provide a further speedup of 2.5–5 as is to be expected given the relative peak memory bandwidth figures of the respective architectures, demonstrating that our code manages to attain a constant fraction of the theoretically available memory bandwidth across all target architectures.

3 Multiscale Methods

Our software concept for numerical multi-scale methods in a parameterised setting is based on the general model reduction framework for multi-scale problems

presented in [12]. The framework covers a large class of numerical multi-scale approaches based on an additive splitting of function spaces into macroscopic and fine scale contributions combined with a tensor decomposition of function spaces in the context of multi query applications. Numerical multi-scale methods make use of a possible separation of scales in the underlying problem. The approximation spaces for the macroscopic and the fine scale are usually defined a priori. Typically, piecewise polynomial functions are chosen on a relatively coarse and on a fine partition of the computational domain. Based on such discrete function spaces, an additive decomposition of the fine scale space into coarse parts and fine scale corrections is the basis for the derivation of large classes of numerical multi-scale methods. A variety of numerical multi-scale methods can be recovered by appropriate selection of decomposed trial and test functions, the specific localisations of the function space for the fine scale correctors, and the corresponding localised corrector operators.

To efficiently cope with multi-scale problems in multi-query scenarios, we add a further tensor type decomposition of function spaces that can be derived as a generalisation of the classical projection based reduced basis approach. Suppose that in a first step a small number of snapshots have been computed with some numerical multi-scale method for suitable parameters, e.g., chosen by a greedy algorithm based on efficient a posteriori error estimates. As a generalisation of the classical reduced basis approach, we then define a reduced approximation space as a non-linear combination of the computed snaphots. As a particular example we focus on tensor product type approximation spaces spanned by products of coarse scale functions and precomputed snapshots. A reduced multi-scale scheme is then obtained by suitable projection of the original problem onto such function spaces. A particular realisation of this approach is, e.g., the localised reduced basis multi-scale method [1].

Within EXA-DUNE we develop a unified interface-based software framework that mimics the mathematical concept for numerical multiscale methods in multi-query scenarios. Particular implementations of this framework are pursued for the multiscale finite element method as a representative of classical numerical multiscale methods and for the localised reduced basis multiscale method as a representative of the generalised model reduction approach.

Concerning the structure of the solution spaces and the resulting discrete approximation schemes, in all the above mentioned methods the global solution is decomposed into dense local solutions on coarse grid blocks, and blockwise sparse global solutions. Therefore, the general structure of approximation spaces, discrete operators and solvers is similar as for DG schemes with locally high polynomial degrees. Thus, for an efficient implementation in heterogeneous parallel environments, we can directly build upon concepts developed, e.g., for DG schemes. The realisation of the parallel multiscale methods is based on the DUNE-Multiscale module[3] and on the DUNE-gdt module[4] and builds upon the hybrid parallelism in DUNE as discussed in Section 2.

[3] `http://users.dune-project.org/projects/dune-multiscale`
[4] `http://users.dune-project.org/projects/dune-gdt`

	16-32	16-64	16-128	16-256	16-512	16-1024
Overall	1.92	3.84	7.46	14.57	28.50	54.05
Coarse Matrix Assembly	2.00	4.01	7.92	15.86	31.60	63.47
Coarse RHS Assembly	1.71	3.41	6.80	13.35	25.73	50.35
Local Problem Assembly and Solve	2.00	3.98	7.85	15.64	30.55	60.35
Coarse Problem Solve	0.41	0.80	0.77	1.01	1.96	2.16

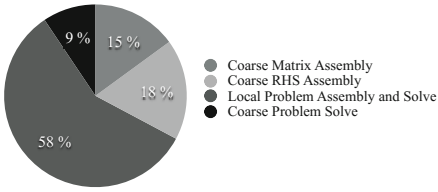

Fig. 2. Left: Strong scaling factors for different parts of the multiscale finite element (msfem) method from 16 to N cores. Right: Distribution of wall time amongst 4 heaviest callers (accounting for 99% of overall runtime) during msfem method on 1024 Cores.

In the hybrid setting, the computational grid associated with the coarse solution space is decomposed into patches (of varying size) that are then distributed to the processes using the MPI-based parallel communication interface of DUNE. On each coarse patch, a virtual local grid refinement is constructed. This locally structured grid then serves as computational mesh for the derivation of the fine scale corrections. Using the virtual grid refinement allows for fully unstructured meshes on the coarse scale while avoiding memory and bandwidth limitations on the fine scale. The fine-scale correction assembly and solve phases can then be further distributed via shared-memory parallelisation within one UMA-node using the techniques from Section 2.

In Figure 2 we demonstrate the scaling capabilities of the multiscale finite element method using an artificial 3D benchmark problem on 32768 coarse cubes, each subdivided into 4096 fine cubes. We test strong scaling on 16 to 1024 cores of our local PALMA cluster at the University of Münster. Most parts of our code show promising scaling, except for the coarse scale system solve which necessitates MPI-communication in each step of the iterative solver and therefore is inefficient for the relatively small coarse problem. Bigger meshes stemming from real-world applications will show better scaling on this part, too.

4 A First Porous Medium Flow Application

As a prototypical example for flows in porous media we consider density driven flow in a three-dimensional domain $\Omega = (0,1)^3$ given by an elliptic equation for pressure $p(x,y,z,t)$ coupled to a parabolic equation for concentration $c(x,y,z,t)$:

$$-\nabla \cdot (\nabla p - c\mathbf{1}_z) = 0, \tag{1}$$

$$\partial_t c - \nabla \cdot \left((\nabla p - c\mathbf{1}_z)c + \frac{1}{Ra}\nabla c \right) = 0. \tag{2}$$

Boundary conditions for the pressure equation are $p = 0$ at $z = 0$ and 'no flow' at all other boundaries. Boundary conditions for the concentration equation are $c = 1$ for $z = 1$, 'no flow' at lateral boundaries and 'inflow/outflow' at $z = 0$. Initial condition is $c = 0$. This system serves as a model for the dissolution of a CO_2 phase in brine, where the unstable flow behaviour leads to enhanced

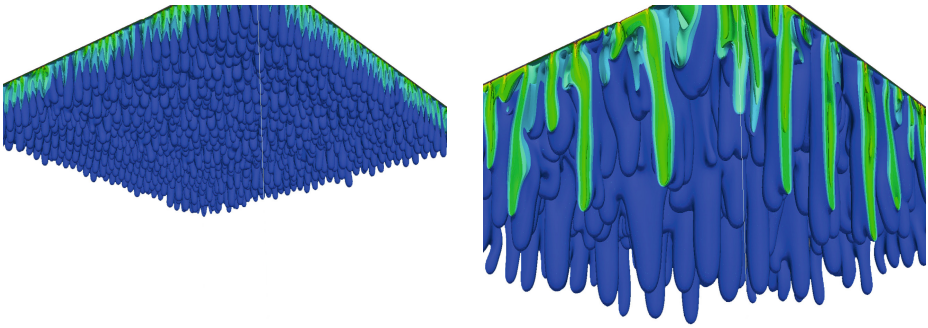

Fig. 3. Density driven flow in a porous medium in three space dimensions for $Ra = 8000$. Left: concentration at $t = 2.25$ after the onset of instability, right: concentration at $t = 4.8$ in the nonlinear regime where persistent fingers have developed.

dissolution. The system is formulated in non-dimensional form with the Raleigh number Ra as the only governing parameter. For details we refer to [13].

The system (1), (2) is solved in a decoupled fashion resulting, after discretisation in space and time, in a large and sparse linear system which is solved by algebraic multigrid and a system of ordinary differential equations. The pressure equation (1) is discretised using the cell centered finite volume (CCFV) method with two-point flux approximation on a structured, equidistant mesh. The velocity field $v = -(\nabla p - c\mathbf{1}_z)$ required in the transport equation is then reconstructed from the finite volume fluxes with lowest order Raviart-Thomas (RT_0) elements. The transport equation (2) is discretised in space with the symmetric weighted interior penalty DG finite element method [6]. For the Raleigh number and mesh sizes utilised below the grid Peclet number is of order 1 and explicit time stepping schemes for the transport equation are efficient. Using strong stability preserving explicit Runge-Kutta methods [14] we can exploit the increased arithmetic intensity of a matrix-free implementation. Figure 3 shows results of a 3D simulation on 8 Xeon E5-2680v2 10-core processors, mesh size 240^3, Q_2 DG elements ($373 \cdot 10^6$ DOF) and 16000 time steps. One time step takes $14s$.

DG methods are popular in the porous media flow community due to their local mass conservation properties, the ability to handle full diffusion tensors and unstructured, nonconforming meshes as well as the simple way to implement upwinding for convection dominated flows. The efficient implementation of high order 'spectral' DG methods relies on a tensor product structure of the polynomial basis functions and the quadrature rules on cuboid elements. At each element the following three steps are performed: (i) evaluate the finite element function and gradient at quadrature points, (ii) evaluate PDE coefficients and geometric transformation at quadrature points, and (iii) evaluate the bilinear form for all test functions. The computational complexity of steps (i) and (iii) is reduced from $O(p^{2d})$, $p-1$ being the polynomial degree and d the space dimension, to $O(dp^{d+1})$ with the sum factorisation technique, see [9,10]. This can be implemented with matrix-matrix products, albeit with small matrix

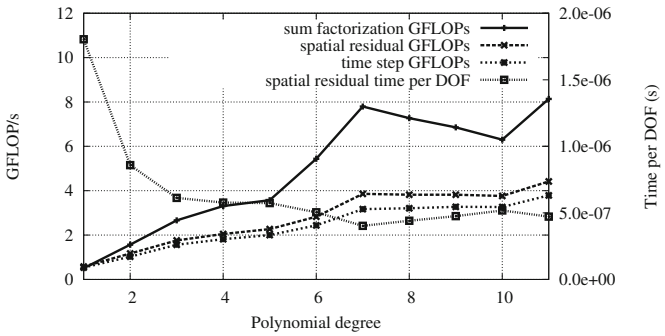

Fig. 4. Single core performance for various components of the DG method

dimensions. For the face terms, the complexity is reduced from $O(p^{2d-1})$ to $O(3dp^d)$. For practical polynomial degrees, $p \leq 10$, the face terms dominate the overall computation time, resulting in the time per degree of freedom (DOF) to be independent of the polynomial degree. This is illustrated by the finely dotted curve in Figure 4. We employ a nodal basis on Gauß-Lobatto points with under-integration on the Gauß-Lobatto points for the temporal bilinear form leading to a diagonal mass matrix. Gauß-Legendre quadrature is used for the spatial bilinear form.

Figure 4 presents performance results of the sum factorisation based 3D DG code on a single core of a Xeon E5-2680v2 for varying polynomial degree. The stand-alone sum factorisation kernel (solid line) achieves up to 8 GFLOP/s corresponding to 40% peak performance. The performance peaks at Q_7/Q_{11} with 8/12 basis functions per direction show that vectorisation is effective. The performance for the complete spatial residual evaluation and a complete time step peak at 4 GFLOP/s. These results clearly illustrate that high order methods can take advantage of modern multicore architectures and their SIMD capabilities.

5 Conclusion

This paper reports first results on introducing hybrid parallelisation and hardware-orientation into the DUNE framework. In the finite element assembly process we obtain promising results for low order methods by vectorising over several elements while for high polynomial degree good performance can also be achieved by loop auto-vectorisation. In ongoing work both approaches will be combined. On the sparse linear algebra level shared memory parallelisation and vectorisation is based on the SELL-C-σ matrix format and additionally exploits the matrix block structure. These components have already been used to speed up a multiscale finite element and a density driven flow solver.

Acknowledgements. This research was funded by the DFG SPP 1648 'Software for Exascale Computing'.

References

1. Albrecht, F., Haasdonk, B., Kaulmann, S., Ohlberger, M.: The localized reduced basis multiscale method. In: Proceedings of Algoritmy 2012, Conference on Scientific Computing, Vysoke Tatry, Podbanske, September 9-14, pp. 393–403 (2012)
2. Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Kornhuber, R., Ohlberger, M., Sander, O.: A generic grid interface for parallel and adaptive scientific computing. part II: Implementation and tests in DUNE. Computing 82(2-3), 121–138 (2008)
3. Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Ohlberger, M., Sander, O.: A generic grid interface for parallel and adaptive scientific computing. part I: Abstract framework. Computing 82(2-3), 103–119 (2008)
4. Choi, J., Singh, A., Vuduc, R.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Principles and Practice of Parallel Programming, pp. 115–126 (2010)
5. Engwer, C., Fahlke, J.: Scalable hybrid parallelization strategies for the DUNE grid interface. In: Proceedings of ENUMATH 2013 (2014)
6. Ern, A., Stephansen, A., Zunino, P.: A discontinuous Galerkin method with weighted averages for advection-diffusion equations with locally small and anisotropic diffusivity. IMA Journal of Numerical Analysis 29(2), 235–256 (2009)
7. Kretz, M., Lindenstruth, V.: Vc: A C++ library for explicit vectorization. Software: Practice and Experience 42(11), 1409–1430 (2012)
8. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A unified sparse matrix data format for modern processors with wide SIMD units. SIAM Journal on Scientific Computing 36(5), C401–C423 (2014)
9. Kronbichler, M., Kormann, K.: A generic interface for parallel cell-based finite element operator application. Computers & Fluids 63, 135–147 (2012)
10. Melenk, J., Gerdes, K., Schwab, C.: Fully discrete hp-finite elements: fast quadrature. Computer Methods in Applied Mechanics and Engineering 190(32-33), 4339–4364 (2001)
11. Müthing, S., Ribbrock, D., Göddeke, D.: Integrating multi-threading and accelerators into DUNE-ISTL. In: Proceedings of ENUMATH 2013 (2014)
12. Ohlberger, M.: Error control based model reduction for multiscale problems. In: Proceedings of Algoritmy 2012, Conference on Scientific Computing, Vysoke Tatry, Podbanske, September 9-14, pp. 1–10. Slovak University of Technology in Bratislava, Publishing House of STU (2012)
13. Riaz, A., Hesse, M., Tchelepi, H., Orr, F.: Onset of convection in a gravitationally unstable diffusive boundary layer in porous media. Journal of Fluid Mechanics 548, 87–111 (2006)
14. Shu, C.: Total-variation-diminishing time discretizations. SIAM Journal on Scientific and Statistical Computing 9, 1073–1084 (1988)
15. Turek, S., Göddeke, D., Becker, C., Buijssen, S., Wobker, S.: FEAST – Realisation of hardware-oriented numerics for HPC simulations with finite elements. Concurrency and Computation: Practice and Experience 22(6), 2247–2265 (2010)