Integrating multi-threading and accelerators into DUNE-ISTL

Steffen Müthing[1], Dirk Ribbrock[2], and Dominik Göddeke[2]

[1] Interdisciplinary Center for Scientific Computing, University of Heidelberg, `steffen.muething@iwr.uni-heidelberg.de`
[2] Institute of Applied Mathematics (LS3), TU Dortmund, `dirk.ribbrock@math.tu-dortmund.de`, `dominik.goeddeke@math.tu-dortmund.de`

Abstract. A major challenge in PDE software is the balance between user-level flexibility and performance on heterogeneous hardware. We discuss our ideas on how this challenge can be tackled, exemplarily for the DUNE framework and in particular its linear algebra and solver components. We demonstrate how the former MPI-only implementation is modified to support MPI+[CPU/GPU] threading and vectorisation. To this end, we devise a novel block extension of the recently proposed SELL-C-σ format. The efficiency of our approach is underlined by benchmark computations that exhibit reasonable speedups over the CPU-MPI-only case.

1 Introduction

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

Software Frameworks A framework approach for entire application domains rather than distinct problem instances targets the first challenge. We are particularly interested in frameworks for the solution of PDE problems with grid-based discretisation techniques. In contrast to the more conventional approach of developing in a 'bottom-up' fashion starting with only a limited set of problems (likely, a single problem) and solution methods in mind, 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.

Target Applications and Numerical Approach Our work within the EXA-DUNE project ultimately targets applications in the field of porous media simulations. These problems are characterised by strongly varying coefficients and extremely anisotropic meshes, which mandate powerful and robust solvers and thus do not

lend themselves 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 Cholesky preconditioning and 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.

Hardware Development Future exascale systems are characterised by a massive increase in node-level parallelism and heterogeneity. 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 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. Firstly, 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 limitations. 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.

Consequences and Challenges Obviously, the necessary adaptations to the changing hardware landscape should be hidden as much as possible from the user. The conventional library-based approach to software development only addresses this challenge at the component level, forcing users to manually integrate those changes into their applications. Software frameworks aid the user with higher abstraction and integration levels, which isolate applications from hardware-specific implementation details. Nonetheless, frameworks still face the conflict between generality, flexibility and API stability on the user side and the need to adapt to new hardware and its potentially disruptive programming models 'under the hood' for optimal performance. Finding the right balance between those extremes defines the challenge of effective framework development in HPC.

Paper Contribution In the EXA-DUNE project, we pursue different avenues to preparing the DUNE framework [4,3] for the exascale era.[1] The goal is to combine the flexibility, generality and application base of DUNE with the concepts of hardware-oriented numerics as developed in the FEAST project [12]. In this paper, we report on our first results and design decisions. We focus on extending DUNE's linear solver module ISTL with architecture-aware backends for low-level constructs like vectors, matrices and preconditioners.

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

2 Design and Implementation

Our approach to enable hybrid parallelism and memory heterogeneity within the DUNE package can be categorised as follows: (1) We redesign the linear algebra part of ISTL (DUNE's linear solver library) around a novel block extension of the SELL-C-σ format [8] so that existing solvers need not be modified to benefit from CPU/GPU threading. (2) We equip PDElab (DUNE's 'user interface') with support for this format so that existing user-level code and other DUNE components can, e.g., directly assemble into this format without the need for costly format conversions. In the following, we describe our changes in a bottom-up fashion.

2.1 Operations: Linear Algebra Kernels

Independent of the architecture, performance improvements not related to mathematically superior algorithms stem from threading and vectorisation (SIMD units, UMA domains). This distinction is explicit on modern multicore CPUs and the Xeon Phi, and implicit on GPUs. We tackle the vectorisation level by creating two collections of linear algebra kernels, one based on CUDA for NVIDIA GPUs and a shared one based on Intel's Threading Building Blocks (TBB) for multi-core CPUs and Xeon Phi. Our design can easily encorporate other specialisations. Following the DUNE philosophy, all new kernels are integrated into ISTL via (new) C++ template interfaces, enabling standard architecture-dependent optimisations for data types, SIMD block sizes and data alignment independent of the interfaces. We choose not to manually program CPU vector units using compiler intrinsics because of the large maintenance overhead every time a new SIMD instruction set is released. Instead, we rely on the auto-vectorisers of modern compilers, which we feed with explicit aliasing and alignment hints for the data arrays: Both GCC 4.8 and ICC 14.0 are able to generate vectorised code of sufficient quality, as verified by inspecting the generated assembly code. The actual kernel implementation follows standard approaches as reported elsewhere in abundance.

2.2 Containers: Matrices and Vectors

One central design choice for implementing numerical linear algebra on heterogeneous architectures is the issue of matrix (and associated vector) formats. On CPUs, (block) CRS is the general format of choice so far [1,7], while the GPGPU community prefers ELL-like formats that enable a more efficient use of wide-SIMD [8]. Format conversions between architecture-optimised formats pose a severe bottleneck and should thus be avoided. Recent work by Kreutzer et al. [8] indicates a feasible solution: Their SELL-C-σ format potentially constitutes a 'best compromise' across architectures. Note that we do not implement a matrix reordering step in our adoption of the format, as our matrices have a very uniform row length distribution and thus, padding is not excessive and need not be avoided.

Other high-level PDE solver frameworks like FEniCS [9] and deal.II [2] support hybrid parallelism via MPI + OpenMP / TBB, but rely on existing libraries at the linear algebra level and do not fully support other accelerators. The linear algebra packages PETSc [1], Trilinos [7] and MTL [10] all support (at least in part) threading and/or GPUs on top of MPI, but exploiting these features from higher-level projects can prove difficult.

2.3 Block Matrices

As part of EXA-DUNE, we are investigating Discontinuous Galerkin (DG) discretisations for porous media simulations. Vectors and matrices associated with DG discretisations exhibit a natural block structure at the mesh cell level, which can be exploited by storing only block sparsity patters. As almost all of the memory required by a CRS or ELL matrix is taken up by two arrays storing the non-zero entries and their associated column indices, storing only block column indices implies a factor of $\approx \frac{1+b^2}{2}$ for a block size b in storage and bandwidth (cf. Figure 1).

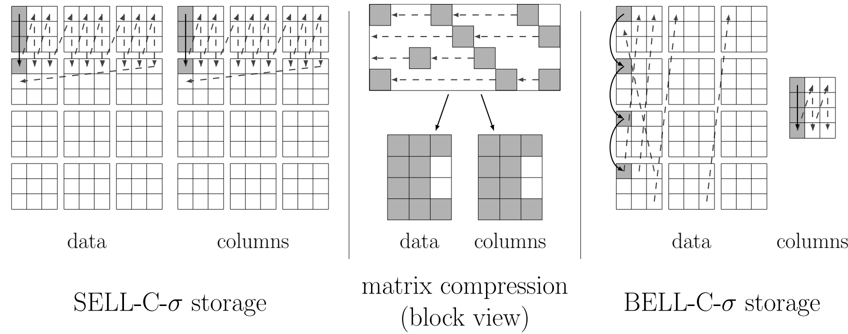

Fig. 1. Data layout of SELL-C-σ and the blocked version BELL-C-σ for a single chunk of a matrix with block size 3 and SIMD width 4. The columns are compressed and padded up to a uniform width (center). For each scheme, the figure shows the in-memory data layout of those compressed arrays as a path along the arrows. Note that SIMD chunks are not block-aligned for SELL-C-σ, while BELL-C-σ coalesces storage from 4 blocks to allow vectorisation of operations across those blocks.

Block CRS matrices work by storing the individual blocks as small, dense matrices, which is a well-known and widely-used optimisation technique that can be implemented fairly easily [1,7]. The efficient implementation proves more difficult in our setting: ELL-like formats and SIMD-awareness require coalescing a number of matrix entries that corresponds to the SIMD width. Most implementations thus introduce blocks that match the SIMD width [11], which works fine in the typical context of (GPU) performance studies, where the block structure tends to be artificially introduced as an optimization parameter, accompanied by standard padding. In our setting the block size is a property of the DG basis that we cannot influence. Thus, our implementation needs to work with arbitrary block sizes. We follow an alternative approach [5], which performs SIMD coalescing at the level of entire matrix blocks as illustrated in Figure 1. Our kernels then operate on several blocks at a time, only requiring vertical padding with empty blocks at the end of the matrix.

In order to further exploit the mathematically motivated blocking structure in our setting, we also implement a block Jacobi preconditioner on top of the blocked matrix, which performs an exact inversion of the diagonal blocks. The diagonal blocks within a SIMD group lack alignment, so the preconditioner extracts the diagonal block band in a preprocessing step and operates on this auxiliary data.

2.4 Solvers and PDELab

Due to the clear separation of algorithms and data structures in ISTL, we transparently reuse existing solver implementations on top of our new container formats without any code changes in the framework. All modifications are restricted to components that directly interact with the matrix structure, i.e., the containers themselves and the preconditioners. In keeping with the DUNE framework approach, we fully integrate our new containers into the high-level PDE toolbox PDELab by implementing a new backend interface that encapsulates the translation of user-space (i, j) indexing to the underlying data layout. As all high-level access to the containers happens via this backend interface, DUNE's existing grid and system assembly infrastructure can directly operate on the new containers, and we avoid using an intermediate matrix format that would then have to be explicitly converted to (B/S)ELL-C-σ. As a direct consequence of this tight integration with the existing solver library and high-level infrastructure, porting PDELab programs is a very straightforward process that only requires modifying the two or three lines of source code which define the active backends for vectors, matrices and preconditioners.

3 Experimental Evaluation

At the current stage of our project, we are mainly interested in validating cross-platform functionality and (relative) performance, especially in terms of our hybrid approach vs. the traditional MPI-only implementation. We can thus restrict ourselves to a standard conjugate gradient solver with a simple scalar or block Jacobi preconditioner, advanced numerical techniques like the ISTL AMG preconditioner are not necessary yet. For our measurements we adapt an existing example program from PDELab that solves a stationary diffusion problem:

$$\nabla(K\nabla u) = f \text{ in } \Omega \subset \mathbb{R}^3$$
$$u = g \text{ on } \Gamma = \partial\overline{\Omega}$$

with $f = (6 - 4|x|^2)\exp(-|x|^2)$ and $g = \exp(-|x|^2)$. A Discontinuous Galerkin discretisation is used with a weighted SIPG scheme [6]. We restrict our experiments to the unit cube $\Omega = (0, 1)^3$ and unit permeability $K = 1$.

Most of our experiments are executed 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. As this baseline machine only has a single UMA memory domain, we also run larger benchmarks on a 4-socket server with AMD Opteron 6172 12-core processors and 128 GB RAM. These CPUs internally comprise two dies with separate cache hierarchies and memory controllers, creating a fairly complex memory layout with 8 6-core UMA domains. This larger platform allows us to test the feasibility of our fundamental approach to parallelisation, drawing the line between classical message passing parallelism (MPI) and shared memory approaches at the level of a single UMA domain, but is limited to two-way SIMD. For the GPU measurements, we use an NVIDIA Tesla C2070 which comes from the same hardware generation as the AMD server. While the host platform differs from our CPU testbeds, this does not influence the results shown below as we only benchmark the CG solver, whose compute- and bandwidth-intensive components run entirely on the GPU.

We use a structured grid with a uniform mesh size h and DG spaces of order $p = 1, 2, 3$, which in 3D translates into block sizes of 8, 27 and 64, and to matrix densities with $\approx$ 56, 189 and 448 non-zero entries per row, respectively. For each space, we choose problem sizes that stretch up to the limit of available memory on our test systems, which is about 6 GB for both the multicore CPU system and the GPU card. In order to enable a fair comparison with a non-multithreaded pure MPI version of the code, we take care to choose problem sizes that can be decomposed into evenly sized subdomains without excessively large surfaces to avoid load balancing issues in the MPI version. Due to the large number of DOFs per cell for the higher-order spaces, this restriction limits us to a smaller number of samples for larger values of p. All computations are carried out in double precision floating point arithmetic.

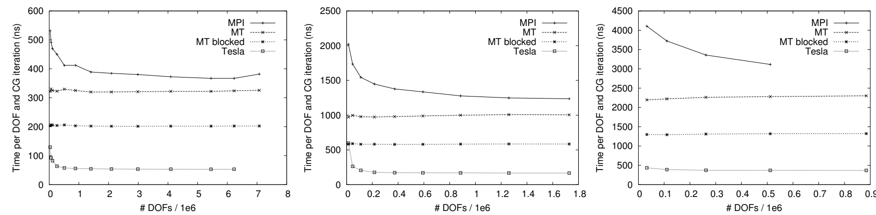

Fig. 2. Normalised execution time of the (Block-) Jacobi preconditioned CG solver for polynomial degrees $p = 1, 2, 3$ (left to right) of the DG discretisation. The multithreaded (MT) versions use a SIMD block size of 8. Missing data points indicate insufficient memory.

The blocked format is currently only available on the CPU, here we also investigate the impact of switching from a scalar preconditioner to a block version that performs an exact inversion of the diagonal blocks. All of the compared implementations use common data structures and mostly perform identical basic operations (with the exception of the block matrix version), so we can expect all of them to require the same number of CG iterations to achieve a given error reduction. Moreover, due to the constant number of matrix entries per DOF for a fixed value of p (apart from boundary effects), we can actually expect a constant time per iteration and DOF for large problem sizes (after saturating either the compute or the memory bandwidth of the system). Figure 2 illustrates that this assumption holds very well, with all shared-memory implementations quickly reaching a saturation plateau. The MPI version appears to actually become faster with growing problem sizes, this is because it is based on an overlapping domain decomposition and thus benefits from the reduction of the relative amount of overlap in the bigger problems. We can see that the dominant computation kernels of the CG solver (SpMV and the (block-) Jacobi preconditioner) are entirely limited by memory bandwidth across all examined architectures. The threaded implementation is approximately 25% faster than the MPI baseline, in line with common expectations. Switching from scalar to blocked containers yields a speedup of $\approx$ 1.6–2.0 depending on p, which is in line with the bandwidth savings of moving to a block-level column index array (the additional work by the block inversion in the CPU preconditioner has

negligible impact). Finally, the GPU gives a speedup of 3–4 over the best CPU implementation.

In order to validate our assumption that shared memory parallelism should be limited to individual UMA domains, we measure the runtime of several large benchmark problems on the AMD server with four different parallel setups, an MPI-only version with 48 single-thread processes, an optimal configuration with 8 MPI processes that each span a complete UMA domain (6 cores), and two suboptimal configurations employing 4 processes with 12 cores each (one process per socket) and one process with 48 threads (using only shared memory). For all measurements, we enforce process pinning. Ignoring the NUMA issue, we expect the timings to improve for smaller numbers of MPI processes because there is less domain overlap, reducing the effective problem size. The results in Table 1 clearly show a noteworthy improvement from the MPI-only setting to our UMA-domain approach (columns $t_{48/1}$ vs. $t_{8/6}$), with better results for higher polynomial degrees due to slightly worse surface-to-volume ratios in the MPI case. The remaining columns show that extending shared-memory parallelism across the UMA domain boundary causes a major performance breakdown by a factor of 2 in the intermediate setting and up to 6 for the worst case.

Table 1. Comparison of different MPI / shared memory partition models for varying degree p of the DG discretisation and mesh width h. For each configuration, we list timings for 100 iterations of the CG solver ($t_{M/T}$, where M is the number of MPI ranks and T the number of threads per process), and the speedups compared to the MPI-only (48/1) case.

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	192	262.8	259.5	1.01	622.6	0.42	1695.0	0.16
1	256	645.1	600.2	1.07	1483.3	0.43	2491.7	0.26
2	96	345.8	318.3	1.09	814.8	0.42	1639.5	0.21
2	128	999.5	785.7	1.27	1320.7	0.76	2619.0	0.38
3	32	120.7	70.1	1.72	183.0	0.66	622.8	0.19
3	64	709.6	502.9	1.41	1237.2	0.57	1958.2	0.36

We additionally note that we do not observe relevant differences when switching off compiler vectorisation for the CPU in our experiments: As the benchmarks are entirely bandwidth bound and as our blocking scheme ensures cache line reuse, performing the actual computations in SIMD instructions has negligible impact.

Acknowledgements

This work was supported (in part) by the German Research Foundation (DFG) through the Priority Programme 1648 'Software for Exascale Computing' (SPPEXA), grants GO 1758/2-1 and BA 1498/10-1.

References

1. S. Balay, J. Brown, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. McInnes, B. Smith, and H. Zhang, *PETSc Web page*, 2011, http://www.mcs.anl.gov/petsc.
2. W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Transactions on Mathematical Software **38**:2 (2012), 14:1–14:28.
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 (2008), 121–138.
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 (2008), 103–119.
5. J. Choi, A. Singh, and R. Vuduc, *Model-driven autotuning of sparse matrix-vector multiply on GPUs*, Principles and Practice of Parallel Programming, 2010, pp. 115–126.
6. A. Ern, A. Stephansen, and P. Zunino, *A discontinuous Galerkin method with weighted averages for advection-diffusion equations with locally small and anisotropic diffusivity*, IMA Journal of Numerical Analysis **29**:2 (2009), 235–256.
7. M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, A. Williams, and K. Stanley, *An overview of the Trilinos project*, ACM Transactions on Mathematical Software **31**:3 (2005), 397–423.
8. M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, *A unified sparse matrix data format for modern processors with wide SIMD units*, SIAM Journal on Scientific Computing (2013), accepted, Preprint: arXiv 1307.6209.
9. A. Logg, K.-A. Mardal, and G. Wells, *Automated solution of differential equations by the finite element method*, Springer, 2012.
10. J. Sick and A. Lumsdale, *A modern framework for portable high-performance numerical linear algebra.*, Advances in Software tools for scientific computing, Lecture Notes in Computational Science and Engineering, vol. 10, Springer, 2000, pp. 1–56.
11. B. Su and K. Keutzer, *clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs*, ISC'12, 2012, pp. 353–364.
12. S. Turek, D. Göddeke, C. Becker, S. Buijssen, and S. Wobker, *FEAST – Realisation of hardware-oriented numerics for HPC simulations with finite elements*, Concurrency and Computation: Practice and Experience **22**:6 (2010), 2247–2265.