Scalable Hybrid Parallelization Strategies for the DUNE Grid Interface

Christian Engwer and Jorrit Fahlke

Abstract The DUNE framework provides a PDE toolbox which is both flexible and efficient. Integration of hardware oriented techniques into DUNE will be necessary to maintain performance on modern and future architectures. We present the current effort to add hybrid parallelization to the DUNE grid interface, which up to now only supports MPI parallelization. In current hardware trends, we see a transition from multi-core to many-core architectures, like the Intel PHI. Techniques which worked well on traditional multi-core CPUs don't scale anymore on many-core systems. We compare different strategies to add a thread parallel layer to DUNE and discuss their scalability and performance.

1 Introduction

Numerical software currently undergoes a dramatic change. We discuss this change in the context of simulations of partial differential equations (PDEs). Since mathematical models are growing in complexity, we seek coupled multi-physics applications, and advanced numerical methods. This calls for flexible general purpose frameworks, with a large body of functionality.

At the same time the underlying hardware is posing orthogonal challenges. The memory and power wall problems are becoming hard limitations, and further performance improvements are only achieved by using all levels of parallelism and heterogeneity. Many people believe this can only be achieved by hardware-software co-design, which contradicts the flexibility goal [5].

Simulation Software becomes more specialised and at the same time generalised. On one hand, applications need to specialise toward advanced models in order to answer more detailed questions. On the other hand, it is infeasible to write such an application from scratch, so a general basis is needed to build upon. Here *Software frameworks* play an important role to provide the required flexibility. As applications

C. Engwer (✉) • J. Fahlke
Institute for Computational und Applied Mathematics, WWU Münster, Münster, Germany
e-mail: christian.engwer@uni-muenster.de; jorrit.fahlke@uni-muenster.de

A. Abdulle et al. (eds.), *Numerical Mathematics and Advanced Applications - ENUMATH 2013*, Lecture Notes in Computational Science and Engineering 103, DOI 10.1007/978-3-319-10705-9_57

continue to grow in complexity, the need for sustainable development of software for PDEs is increasing rapidly: Modern numerical ingredients such as unstructured grids, adaptivity, high-order discretizations and fast and robust multilevel solvers are required to achieve high numerical efficiency, and several physical models must be combined in challenging applications. This is beyond the scope of an individual simulation application, but requires additional support. Frameworks like Deal.II [1], Fenics [8], or DUNE [2, 3] (the one we are focusing on) support developers by providing a rich set of numerical algorithms and mathematical models. Such frameworks are designed from the beginning for flexibility and generality. Thus users can easily extend the generic framework code with their own algorithms and models. Using modern C++ techniques DUNE supports this fusion of user and framework code at compile time, which enables many compiler optimisations and thus grants flexibility and efficiency.

Hardware is undergoing a dramatic change. Current peta-scale systems in general still follow the old paradigms of high performance compute nodes, linked by fast interconnects. Both on the low-power end and at the high performance end, future systems will differ significantly: Typical workstations and cluster nodes now comprise at least two multicore CPUs and potentially several manycore accelerators such as GPUs, and energy-efficient designs such as ARM+GPU or BlueGene-Q are gaining ground. Future systems will offer much less memory per node, and show a massive increase of parallelism inside a single node, either with a 'many conventional core' approach or by combining fewer cores with specialised accelerator designs like GPUs [6]. This is a 100 to 1,000-fold increase of parallelism within each node, combined with an ever increasing impact of the memory wall problem. While message passing will still be the communication of choice between NUMA-nodes, dedicated hierarchic layers of hybrid parallelism will be necessary to exploit instruction level parallelism (ILP) and short-vector units (SIMD).

The Challenge posed for frameworks is the adoption of these new hardware paradigms. As frameworks allow for thorough user extensions at a very fine grained level, it is much harder to support modern hardware than it is for classic coarse grained libraries like BLAS. A complete rewrite of the framework for every change in hardware is not feasible and contradicts the concept of fine grained user interfaces. Thus all changes in the framework should be hidden from the user code, or at least require only moderate changes. Keeping the generality and flexibility of software frameworks while adapting them to the hardware revolution to make use of the advertised performance improvements in a transparent way is the main challenge today.

Our aim is to combine the flexibility, generality and application base of DUNE [2,3] with the concepts of 'hardware-oriented numerics' as developed in the FEAST project [9]. The hypothesis is that advanced numerical methods are the key to enable efficient use of the underlying hardware and to maintain generality alike. The work presented in this paper is part of the EXA-DUNE[1] project.

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

2 Concepts

For PDE-based simulations the computation time is dominated by two phases, the assembly and the solving of sparse linear problems. We define a partition $\mathscr{T}(\Omega)$ of the computational domain, which we refer to as the mesh or grid. This mesh induces our FEM function space and our degrees of freedom. On each cell of the mesh local contributions to the global system matrix are computed and collected in a local matrix, then these local matrices are used to update the global matrix.

To maintain performance assembling of the linear system and the linear solver need to be accelerated homogeneously. In the following we discuss the necessary changes to the DUNE grid interface and to the assembler in DUNE-PDELab. Changes to the linear algebra are not discussed in this paper and will be incorporated later.

3 Design and Implementation

The mesh is one of the key components of DUNE. The grid interface [2] follows a generic definition [3], which can be implemented in many different ways and also allows to use existing external mesh libraries through this interface. Based on this grid interface and on the linear algebra library (DUNE-ISTL) the discretization module DUNE-PDELab provides many choices of function spaces and many different discretization schemes, which the user can easily extend or combine into complete discretizations.

Up to now DUNE only considered MPI parallelization, as suitable data decomposition is directly supported by the DUNE grid interface. As DUNE supports external grid managers and many of these were only designed for MPI and don't support hybrid parallelization, we are seeking a hybrid approach which can be implemented on top of the existing grid interface. Such an interface can be implemented in a generic fashion, but can be specialised if a specific grid implementation provides additional information.

Levels of Parallelism We plan for three levels of parallelism. For an efficient assembly of the stiffness matrix and the right hand side vector, the key is concurrent access to grid information and to associated data.

Globally the grid is partitioned using the existing MPI layer. This gives coarse grained parallelism on the level of UMA nodes, where all cores within one MPI node have uniform memory access.

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 will be locally partitioned such that each thread handles the same amount of work.

On the finest layer future extensions will make use of vectorisation (SIMD, ILP) by adapting the internal data structures used in DUNE and especially in the assembler.

Shared Memory parallelization using system threads is the main focus of our following experiments. The coarse grained message-passing level is used as it is and finer grained vectorisation level will be investigated in future work.

We introduce the concept of `EntitySets` to define iterator ranges, which describe different mesh partitions. An `EntitySet` describes a set of grid objects, e.g. a set of grid cells, which can be iterated over. For each cell and the associated sub-entities we compute indices to store data consecutively; using these indices we can directly access linear algebra vectors or matrices. As the `EntitySet` lives outside the original mesh, it can take locally varying computational costs into account.

For the local partitioning of a mesh $\mathscr{T}(\Omega)$ we consider three different strategies, where the first two are directly based on the induced linear ordering of all mesh cells $e \in \mathscr{T}(\Omega)$.

Strided: For P threads each thread p iterates over the whole set of cells $e \in \mathscr{T}(\Omega)$, but stops only at cells where $e \bmod P = p$ holds. As all P threads have to iterate over the whole grid simultaneously, they might start competing for bandwidth.

Ranged: We define consecutive iterator ranges of the size $|\mathscr{T}|/P$. 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.

General: Technically all other partitioning strategies will be handled in the same way. On structured meshes we can directly define geometric partitions, e.g. equidistant partitions along one or all coordinate axes (later called *sliced* or *tensor*, repectively). For unstructured meshes graph partitioning libraries like METIS or SCOTCH offer different strategies. We support all these by storing copies of all cells in an `EntitySet`. While this approach is the most flexible one, it is memory intensive, which might lead to cache trashing. The additional memory requirement is $O(|\mathscr{T}|)$, but the constant can be big, depending on the actual grid implementation.

Data Access is the other critical component. During assembly data races can occur, as different local vectors and local matrices contribute to the same global entries. Two approaches are possible to avoid race conditions: locking and colouring. As global locking is known to diminish performance as all threads are competing for this single lock, we discard this option right away and consider three different strategies:

Elock: *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 $O(|\mathscr{T}|)$.

Batched: *batched write* operations are a compromise between global and entity-wise locking. Threads still compete for a global lock, but the frequency of locking attempts is reduced by collecting updates in a temporary buffer. A lock is acquired when the buffer is full and all buffered updates are performed at once. The additional memory is $O(P)$ with a large constant.

Colouring: *colouring* avoids competing data access by assigning each partition to a colour such that there is no overlap between partitions of the same color. Different colors must be handled strictly in sequence, but partitions of the same color may be handled concurrently. Colouring is not meaningful for some partitioning strategies, e.g. strided and ranged. In general colouring may add to the set-up time, but requires very little memory, $O(P)$ with a small constant.

4 Performance Evaluation

Our goal is to evaluate the different strategies for system-level thread parallelization. We validate the cross-architecture scalability and formulate best practice suggestions. We restrict ourselves to the test problem of a stationary advection-diffusion equation in two dimensions

$$\nabla \cdot \{-A(x)\nabla u + b(x)u\} = f \qquad \text{in } \Omega \tag{1}$$

with Dirichlet and outflow boundary conditions and compare the performance for assembling the stiffness matrix and the residual. These are the most expensive mesh-related operations in the FE method. Equation (1) is discretized using the weighted SIPG discontinuous Galerkin method [4]. Ansatz and test space are discretized using an orthonormal P^k basis of degree k. While this scheme can actually be implemented in a completely race-free manner, doing so means that fluxes have to be computed twice.

To get a worst case estimate on the impact of different partitioning and data access strategies, we use a lightweight MPI-parallel structured mesh (`Dune::YaspGrid`) for our performance evaluations. For unstructured meshes the relative overhead will be considerably smaller and thus we expect better parallel efficiency.

We discuss timing results and scalability for the different hybridisation strategies and compare the results for a multi-core system with many-core architecture. Our experiments for the multi-core system are performed on a 4-socket (i.e. 4-UMA-node) Intel Xeon E7-4850 system at 2.00 GHz, with 10 cores per socket, 2 hyperthreads per core, 198 GB DDR3 total memory and 4×25.6 GB/s transfer rate. The many-core system is an Intel Xeon-PHI 5110P with 60 compute cores at 1.05 GHz, 4 hyperthreads per core, 8 GB total memory and 320 GB/s bandwidth. Performance is measured for the assembly of the residual and of the Jacobian, i.e. the right-hand side and the stiffness matrix. Additionally we compare results for different polynomial degrees.

On the CPU we observe good scalability for all partitioning strategies, see Fig. 1. This is in correspondence to the experimental hybrid parallelization discussed in [7]. The parallel efficiency of the residual drops down to $\sim$50 %, whereas the Jacobian keeps an efficiency $\geq$60 % up to the 10 physical cores. Hyper-threading improves the run times further, even though the efficiency drops significantly. The better

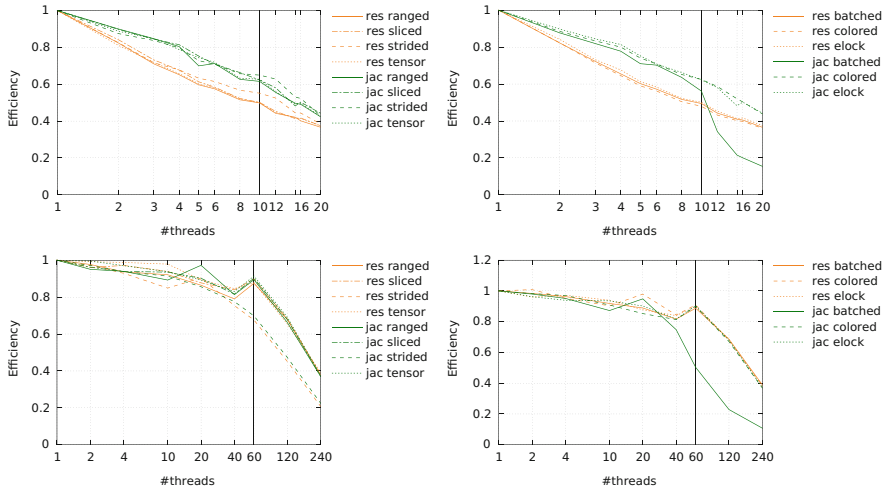

Fig. 1 Parallel efficiency of the assembly of residual (*res*) and Jacobian (*jac*) for different partitioning (*left*, entity-wise locking) and locking (*right*, sliced partitioning) strategies on CPU and PHI, polynomial degree $k = 1$

efficiency of the Jacobian is due to the increased algorithmic intensity for higher polynomial degrees.

For data access we compare batched writes, entity-wise locks and a lock-free strategy, via colouring. In our experiments the performance of entity-wise locking and colouring is comparable. Batched writes pose difficulties when assembling the Jacobian: the performance of batched writes depends severely on the size of the temporary buffer and the sparsity pattern, which means that good performance can only be achieved by tuning buffer sizes.

By comparing different partitioning strategies possible performance issues become more visible. While strided partitioning is attractive in multi-core CPU systems, it does not scale to the larger numbers of cores on the PHI. This is to be expected: for this kind of partitioning, all threads will usually operate on nearby mesh cells at any given time. On the CPU they benefit from the level 3 cache shared by all 10 cores: one thread is likely to access data that a different thread has just loaded. On the PHI each of the 60 cores has its own cache: the cores compete for the memory bandwidth to transfer cache lines, but only a small part of each cache line is actually used for computation.

Apart from strided partitioning, the choice of partitioning strategy has very little effect. This means that the memory bandwidth has not been reached for these schemes; we expect this effect to become important when element local computations are vectorized. In this case it will be necessary to further increase the algorithmic intensity—either by the use of significantly higher polynomial orders, or by using locally structured low-order computations.

Table 1 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 sliced partitioning and entity-wise locking. We see a clear benefit from higher order discretizations, 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%							

Table 1 shows the computation time per DOF and the obtained efficiency for different polynomial degrees in the DG discretization. Due to increased algorithmic intensity the efficiency increases for higher order discretizations. Hyperthreading does diminish the efficiency, but still gives a slight improvement in computation time. In computation time the Xeon PHI does not pay of, as the current implementation does not use vectorisation, which necessary to unlock the potential of the PHI.

Summary and Conclusions

We have shown that many-core architectures require additional care in designing the thread parallelism for hybrid simulations. As many mesh libraries were only designed for distributed memory, using MPI parallelization, we designed the extensions such that thread parallelism can be implemented on-top of an existing DUNE grid. Support for this additional layer was added to the DUNE-PDELab module. We emphasise that from the user point of view the changes are totally transparent and hidden underneath the discretization interface.

We demonstrated performance tests on an Intel Xeon PHI and compared with results for a 4-socket Intel Xeon E7-4850 system. With a ranged partitioning and entity-wise locking, or with colouring and the according partitioning, it is possible to provide a low overhead thread parallelization layer, which shows good performance on classic multi-core CPUs and on modern many-core systems alike. The performance gain from coloring is negligible, but increases code complexity, so that this approach is less favourable. We increased the efficiency further to $\sim$90 % by the use of higher order methods.

Supporting modern hardware paradigms is possible within a general purpose interface, without sacrificing flexibility and still obtain good performance. From the user's perspective all changes are completely transparent.

(continued)

Future work will investigate how to add SIMD and wide-SIMD support during the assembly of stiffness matrices and residuals and incorporate SIMD support in the linear algebra. Adding SIMD support is still an open issue, because in order to keep flexibility, it is no option to directly use intrinsics in the user code.

Acknowledgements This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 'Software for Exascale Computing' (SPPEXA).

References

1. W. Bangerth, C. Burstedde, T. Heister, M. Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes. ACM Trans. Math. Softw. **38**(2), 14:1–14:28 (2012)
2. P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, 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)
3. P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, O. Sander, A generic grid interface for parallel and adaptive scientific computing. part I: abstract framework. Computing **82**(2–3), 103–119 (2008)
4. A. Ern, A.F. Stephansen, P. Zunino, A discontinuous galerkin method with weighted averages for advection–diffusion equations with locally small and anisotropic diffusivity. IMA J. Numer. Anal. **29**(2), 235–256 (2009)
5. X.S. Hu, R.C. Murphy, S. Dosanjh, K. Olukotun, S. Poole, Hardware/software co-design for high performance computing: challenges and opportunities, in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, Scottsdale (IEEE, 2010), pp. 63–64
6. D.E. Keyes, Exaflop/s: the why and the how. Comptes Rendus Mécanique **339**(2–3), 70–77 (2011)
7. R. Klöfkorn, Efficient matrix-free implementation of discontinuous galerkin methods for compressible flow problems, in *ALGORITMY 2012, Proceedings of Contributed Papers and Posters*, Podbanske, ed. by A. Handlovičová, Z. Minarechová, D. Ševčovič (Publishing House of STU, 2012), pp. 11–21. http://www.iam.fmph.uniba.sk/algoritmy2012/
8. A. Logg, K.-A. Mardal, G. Wells, *Automated Solution of Differential Equations by the Finite Element Method* (Springer, Berlin/New York, 2012)
9. S. Turek, D. Göddeke, C. Becker, S. Buijssen, S. Wobker, FEAST – Realisation of hardware-oriented numerics for HPC simulations with finite elements. Concurr. Comput.: Pract. Experience **22**(6), 2247–2265 (2010)