Hybrid MPI/OpenMP Parallelization in FETI-DP Methods

Axel Klawonn, Martin Lanser, Oliver Rheinbach, Holger Stengel, and Gerhard Wellein

Abstract We present an approach to hybrid MPI/OpenMP parallelization in FETI-DP methods using OpenMP with PETSc+MPI in the finite element assembly and using the shared memory parallel direct solver Pardiso in the FETI-DP solution phase. Our approach thus uses OpenMP parallelization on subdomains and MPI in between subdomains. We investigate the efficiency of this approach for a benchmark problem from two dimensional nonlinear hyperelasticity. We observe good scalability for up to four threads for each MPI rank on a state-of-the-art Ivy Bridge architecture and incremental improvements for up to ten OpenMP threads for each MPI rank.

1 Introduction

The solution of large nonlinear and linear problems arising from the finite element discretization of partial differential equations requires fast and highly scalable parallel solvers. Domain decomposition methods [27, 28] are constructed to handle such large, discretized problems in parallel and are well known for their robustness and scalability, especially in structural mechanics and for material science problems.

In domain decomposition methods, a geometrical decomposition of the computational domain is performed, and parallelization is based on a divide and conquer strategy. For static and quasi-static problems, an additional small coarse problem is essential for numerical and parallel scalability. In the family of domain

A. Klawonn • M. Lanser (✉)
Mathematisches Institut, Universität zu Köln, Weyertal 86-90, 50931 Köln, Germany
e-mail: axel.klawonn@uni-koeln.de; martin.lanser@uni-koeln.de

O. Rheinbach
Fakultät für Mathematik und Informatik, Institut für Numerische Mathematik und Optimierung, Technische Universität Bergakademie Freiberg, 09596 Freiberg, Germany
e-mail: oliver.rheinbach@math.tu-freiberg.de

H. Stengel • G. Wellein
Erlangen Regional Computing Center, University of Erlangen–Nuremberg, Erlangen, Germany
e-mail: holger.stengel@fau.de; gerhard.wellein@fau.de

M. Mehl et al. (eds.), *Recent Trends in Computational Engineering - CE2014*,
Lecture Notes in Computational Science and Engineering 105,
DOI 10.1007/978-3-319-22997-3_4

decomposition methods, nonoverlapping approaches such as FETI-DP (Finite Element Tearing and Interconnecting—Dual Primal), first introduced by Farhat et al. [12, 13], have reduced communication compared to overlapping Schwarz methods [27, 28]. A classical FETI-DP implementation was awarded a Gordon Bell prize in 2002 [8] for the solution of a structural mechanics problem using 3400 cores. Modified versions, i.e., inexact FETI-DP methods have scaled up to 65,536 BlueGene/P cores for elasticity problems already in 2009 [21, 24].

In pure MPI parallel implementations of FETI- or Neumann-Neumann-type iterative substructuring methods, one or more subdomains are assigned to one MPI rank and the local calculations are performed sequentially. We will introduce a second level of parallelism by using shared memory parallelism on the subdomains. This approach was also taken, e.g., for a FETI-1 method in [14, 16] based on a different shared memory direct solver [14, 15].

The robustness of FETI-DP methods partially stems from the use of sparse direct solvers as building blocks. In the past, we have used, e.g., the sequential sparse solver package UMFPACK [9], known for its robustness, as well as the solver package MUMPS [2]. In this work, we apply the more recent shared memory parallel sparse solver PARDISO [25]. Additionally, we introduce an OpenMP parallel assembly of the local finite element problems. In this paper, we present numerical results and a detailed performance analysis of our hybrid FETI-DP implementation.

In this work, we will not consider large core counts as in [19], where weak scalability results for our current FETI-DP implementation for up to 262K cores using pure MPI have recently been presented. Instead, a fixed number of subdomains and MPI ranks, e.g., 4 or 64 subdomains and ranks, is considered. We then spawn different numbers of threads for this configuration. The goal of this paper is to investigate the efficiency of this approach on a state-of-the-art architecture. In the future, we will then apply our hybrid MPI/OpenMP approach for a large number of subdomains and MPI ranks on supercomputers where pure MPI implementations do not scale well enough.

2 Description of the FETI-DP Method

The FETI-DP (Finite Element Tearing and Interconnecting—Dual Primal) method is a domain decomposition method based on the geometric decomposition of a computational domain $\Omega \subset \mathbb{R}^d, d = 2, 3$, into N nonoverlapping subdomains Ω_i. Instead of solving a single and large problem $Ku = f$ arising from a finite element discretization on the domain Ω, in FETI-type methods several smaller problems $K^{(i)}u^{(i)} = f^{(i)}$, associated to the subdomains Ω_i, are solved in parallel. Here, the locally assembled stiffness matrices are denoted by $K^{(i)}$ and the local load vectors by $f^{(i)}$. As a divide-and-conquer technique FETI type methods are thus very suitable for parallel computations. A combination of a global subassembly in a few variables and dual conditions ensures the continuity of the solution on the

interface $\Gamma := \bigcup_{i=0}^{N} \partial\Omega_i \setminus \partial\Omega$: We partition the degrees of freedom on Γ into sets of primal and dual variables, denoted by Π and Δ and also define the index set of inner and dual variables $B := [I, \Delta]$. The assembly in some primal variables u_Π is performed by means of the standard FETI-DP partial assembly operator R_Π^T; see, e.g., [21, 28]. The assembled system in the primal variables is also named FETI-DP coarse problem and can include more than just vertex constraints (c.f. [22]). To obtain continuity in the remaining interface variables we introduce the jump operator B_B, c.f. the definition in [21, 28], and Lagrange multipliers λ to enforce the continuity condition $B_B u_B = 0$ on the variables u_B. This leads to the FETI-DP master system

$$\begin{bmatrix} K_{BB} & \tilde{K}_{\Pi B}^T & B_B^T \\ \tilde{K}_{\Pi B} & \tilde{K}_{\Pi\Pi} & 0 \\ B_B & 0 & 0 \end{bmatrix} \begin{bmatrix} u_B \\ \tilde{u}_\Pi \\ \lambda \end{bmatrix} = \begin{bmatrix} f_B \\ \tilde{f}_\Pi \\ 0 \end{bmatrix}. \tag{1}$$

Here, K_{BB} is a block diagonal matrix and the blocks $K_{BB}^{(i)}$ are local to the subdomains Ω_i, while the matrices including primal variables Π are global but small. Elimination of u_B and $\tilde{u}_\Pi$ leads to

$$F\lambda = d \tag{2}$$

$$\text{where} \qquad F = B_B K_{BB}^{-1} B_B^T + B_B K_{BB}^{-1} \tilde{K}_{\Pi B}^T \tilde{S}_{\Pi\Pi}^{-1} \tilde{K}_{\Pi B} K_{BB}^{-1} B_B^T, [-1ex] \tag{3}$$

$$d = B_B K_{BB}^{-1} f_B + B_B K_{BB}^{-1} \tilde{K}_{\Pi B}^T \tilde{S}_{\Pi\Pi}^{-1} (\tilde{f}_\Pi - \tilde{K}_{\Pi B} K_{BB}^{-1} f_B). \tag{4}$$

Here, $\tilde{S}_{\Pi\Pi} := \tilde{K}_{\Pi\Pi} - \tilde{K}_{\Pi B} K_{BB}^{-1} \tilde{K}_{\Pi B}^T$ is the Schur complement on the primal variables. Finally, the FETI-DP method is the iterative solution of the preconditioned system

$$M^{-1} F\lambda = M^{-1} d \tag{5}$$

with a Krylov subspace method such as CG or GMRES. In this paper, we always use the standard Dirichlet preconditioner $M_{FETI_D}^{-1} =: M^{-1}$ which is a weighted sum of local Schur complements $S^{(i)} := K_{\Delta\Delta}^{(i)} - K_{\Delta I}^{(i)} K_{II}^{(i)-1} K_{\Delta I}^{(i)T}$; see, e.g., [21, 28] for complete notation.

3 Classical MPI Parallelism in FETI-DP Methods

MPI-parallel FETI-DP implementations usually handle one or more subdomains per MPI rank. For simplicity, let us assume that we always assign exactly one subdomain to each MPI rank. In this case, the assembly of local problems and the LU factorizations are naturally parallelizable and communication is only required during the assembly process of the coarse problem $\tilde{S}_{\Pi\Pi}$ and in each application of the jump operator B_B or its transpose.

Algorithm 3 Structure of a FETI-DP method. The assembly phase (red), LU factorizations (blue), LU forward/backward substitutions (magenta). Parts with significant MPI communication are marked in green. Some MPI communication can be hidden behind local factorizations.

FETI-DP Method

Local Assembly Phase

- **LOCAL assembly of $K^{(i)}$ and $f^{(i)}$**
- **LOCAL extraction of FETI-DP submatrices**

FETI-DP Setup Phase

- **Build B_B and R_Π as Scatter/Gather operation on vectors**
- **Assembly of $\tilde{K}_{\Pi\Pi}$ and $\tilde{f}_\Pi$**
- **LOCAL LU factorization of $K_{BB}^{(i)}$**
- **LOCAL LU factorization of $K_{II}^{(i)}$ for the preconditioner**
- **Build primal Schur complement $\tilde{S}_{\Pi\Pi} = \tilde{K}_{\Pi\Pi} - \tilde{K}_{\Pi B} K_{BB}^{-1} \tilde{K}_{\Pi B}^T$**
- **Send serial copy of $\tilde{S}_{\Pi\Pi}$ to all MPI ranks**
- **LOCAL LU factorization of $\tilde{S}_{\Pi\Pi}$**

Krylov Iteration Phase

- **Krylov iteration on $M^{-1}F\lambda = M^{-1}d$**
- **Obtain final solution by $\tilde{u} = \tilde{K}^{-1}(\tilde{f} - [B_B 0]^T \lambda$**

In general, we can split the computations in three phases: *Local Assembly*, *FETI-DP Setup*, and *Krylov Iteration*. The first phase contains the local assembly of the matrices $K^{(i)}$ and the local right-hand sides $f^{(i)}$. This process is naturally parallelizable, since no communication between the subdomains is needed.

The second phase *FETI-DP Setup* is dominated by LU factorizations of subdomain matrices $K_{BB}^{(i)}$ and $K_{II}^{(i)}$. Here, sparse direct solver packages such as UMFPACK [10, 11], MUMPS [1, 2] or PARDISO [23, 25, 26] are used. Once factorized, each application of K_{BB}^{-1} and $K_{II}^{(i)-1}$ requires only two forward-backward substitutions. The factorizations are completely parallel since all involved matrices are local to an MPI rank. The coarse operator $\tilde{S}_{\Pi\Pi}$ is also assembled in the *FETI-DP Setup* phase. The assembly can be performed in parallel using an MPI parallel matrix structure. Here, of course, MPI communication is needed. Many implementations are then possible for the solution of the coarse problem. Since the FETI-DP coarse problem is not in the focus of this paper, we only discuss the simplest choice here: $\tilde{S}_{\Pi\Pi}$ is copied to all MPI ranks, and a redundant factorization of $\tilde{S}_{\Pi\Pi}$ is performed on all ranks; refer, e.g., to [21, 24] for other possibilities.

The third phase is the Krylov subspace iteration on the preconditioned linear system $M^{-1}F\lambda = M^{-1}d$. Each iteration includes several matrix vector multiplications but the dominating parts are the forward backward substitutions caused by the applications of K_{BB}^{-1}, $K_{II}^{(i)-1}$ and $\tilde{S}_{\Pi\Pi}^{-1}$; c.f., (2), (3), and (4).

The three FETI-DP phases are illustrated in Algorithm 3.

4 Shared Memory Parallelism in FETI-DP Methods

Modern supercomputer architectures have more and more cores per node while the amount of available memory per core even tends to decrease. To utilize all cores using a pure MPI implementation, we assign at least one FETI-DP subdomain (and one MPI rank) to each core. Sometimes even an overcommit can be beneficial, i.e.,

using multiple MPI ranks per core, e.g., on BlueGene/Q; see [19]. As a result, available memory per rank can be scarce. This can enforce a decomposition into small subdomains, which may not be optimal for scalability and performance.

A shared memory parallelization of the calculations on a single subdomain allows the decomposition in fewer subdomains than available cores. Larger subdomains can then be chosen and the FETI-DP coarse problem can be kept small. In an extreme scenario only one subdomain (and one MPI rank) could be assigned to a complete compute node, although judging from the results in this paper, this may not be the most efficient choice.

To determine the hotspots of the FETI-DP algorithm and to decide which parts should be thread-parallel, we present a preliminary runtime profile of a current MPI parallel FETI-DP implementation. We have performed certain simplifications and optimizations of the assembly process, which are described in more detail in Sect. 7.1. These modifications are possible in the setting which we discuss here.

After these improvements, the *FETI-DP Setup* phase takes 62 %, the assembly process 16 %, the assembly of the right-hand side 6 %, and the Krylov phase the remaining 16 % of the average runtime of the Newton steps, c.f. Fig. 1 (left). The *FETI-DP Setup* phase is strongly dominated by the local LU factorizations, which make more than 80 % of the *FETI-DP Setup* runtime, c.f. Fig. 1 (right). All in all, the assembly and the direct solver package add up to almost 80 % of the FETI-DP runtime. Therefore, a shared memory parallelization of these two phases has priority.

In the remainder of this paper, we discuss the optimization and OpenMP parallelization of the assembly process, which is based on a parallelization of the loop over the finite elements and also investigate the use of the shared memory parallel direct solvers PARDISO.

Let us summarize our approach: We use MPI parallelism in between the subdomains, which is well-established, and we provide an additional level of shared memory parallelism on the subdomains. In an extreme case, FETI-DP with one subdomain per node is possible.

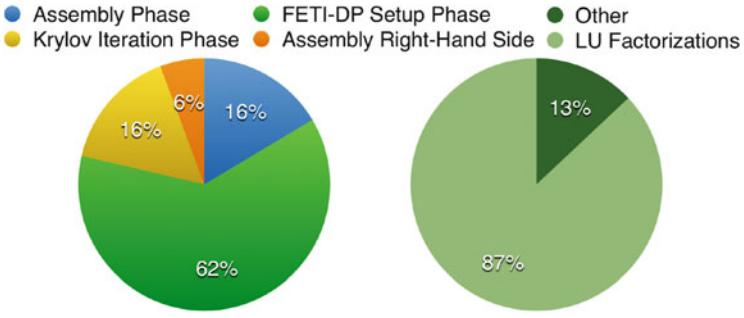

Fig. 1 *Left*: runtime profile of the FETI-DP method for a Neo-Hookean hyperelasticity problem in 2D with $m_x = m_y = 200$, using the optimized assembly; see Sect. 7.1. *Right*: distribution of the runtime in the *FETI-DP Setup* phase

5 Hybrid Parallelization in Inexact and Nonlinear FETI-DP Approaches

In this paper, we consider nonlinear hyperelasticity as a model problem and follow a standard Newton-Krylov-FETI-DP approach for its solution. Moreover, since the number of subdomains is moderate, we always use exact FETI-DP methods [12, 13, 28], i.e., we can afford to solve the FETI-DP coarse problem exactly by a sparse direct solver. The findings presented in this paper on hybrid MPI/OpenMP parallelization are, nevertheless, also valid for inexact FETI-DP methods [18, 20, 21], which remain efficient also for a large number of subdomains. Since the families of the recently introduced nonlinear FETI-DP methods [18, 20, 21] use the same algorithmic building blocks as standard exact or inexact FETI-DP methods our results also apply to these approaches. Due to space limitations, and to keep the paper self-contained, we do not report on results on these methods, here.

6 Experimental Setting

6.1 Model Problem

In all experiments in this paper, we consider nonlinear two-dimensional hyperelasticity problems. We consider a Neo-Hookean material with a soft matrix material and stiff circular inclusions; see Fig. 2 for the geometry.

The strain energy density function W is given by [30]

$$W(u) = \frac{1}{2}(K - \frac{2}{3}G)\ln^2(\det F) + \frac{1}{2}G[\mathrm{tr}(F^T F) - 3 - 2\ln(\det F)]$$

$$\text{with} \qquad K = \frac{E}{3(1-2\nu)},\ G = \frac{E}{2(1+\nu)}$$

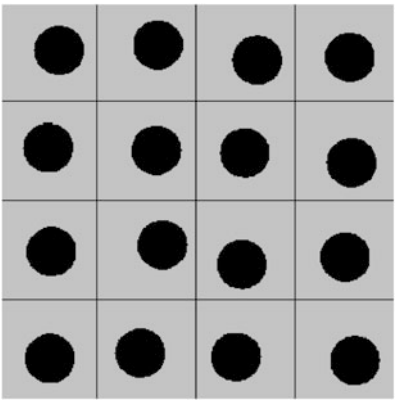

Fig. 2 Decomposition of the computational domain Ω into 16 subdomains; each subdomain has a (slightly randomly off-centered) circular inclusion of stiffer material

and the deformation gradient $F(x) := \nabla\varphi(x)$; here, $\varphi(x) = x + u(x)$ denotes the deformation and $u(x)$ the displacement of x. The energy functional of which stationary points are computed, is given by

$$J(u) = \int_{\Omega} W(u) - F(u)dx - \int_{\Gamma} G(u)ds,$$

where $F(u)$ and $G(u)$ are functionals related to the volume and traction forces. In our experiments in 2D, we choose the following material parameters E and ν, see Fig. 2 for the geometry: In the circular inclusions we have $E = 210,000$ and in the surrounding matrix material $E = 210$. We have chosen $\nu = 0.3$ in the complete domain Ω. We use a C++ implementation of a Neo-Hooke material and use the MPI-parallel infrastructure provided by PETSc [4–6].

The nonlinear elasticity problem is discretized with piecewise quadratic finite elements. Each square subdomain is discretized by $2 \times m_x \times m_y$ quadratic finite elements. This corresponds to $2(2m_x + 1)(2m_y + 1)$ degrees of freedom per subdomain. We have tested subdomain sizes of $m_x = m_y = 100$, $m_x = m_y = 200$, and $m_x = m_y = 300$ to cover a reasonable range relevant in real-life applications. But as performance results are qualitatively identical for those subdomain sizes, we consistently present performance values only for the subdomain size $m_x = m_y = 200$ to increase comparability.

We will first consider four subdomains and thus four MPI ranks, while the subdomain size is set to $m_x = m_y = 200$. If not noted otherwise, each of the four MPI processes is placed on a separate node. Appropriate measures were taken to pin processors and threads to the cores in a controlled way. The following parameters where used for the PARDISO solver:

```
-mat_pardiso_65 ${OMP_NUM_THREADS}     -mat_pardiso_68 0  -mat_pardiso_1 1
-mat_pardiso_24 1  -mat_pardiso_69 11  -mat_pardiso_2  3  -mat_pardiso_11 0
```

6.2 Hardware, Compiler, and Compiler Flags

All measurements for this report have been executed on a QDR InfiniBand cluster at RRZE. One node has two sockets, each equipped with an Intel Xeon 2660v2 "Ivy Bridge" chip (10 cores + SMT). To achieve consistent performance results, automatic frequency adjustment during socket scaling runs was prevented by fixing the processor frequency to the nominal core clock speed of 2.2 GHz. Each node has 64 GB of RAM (DDR3-1600), the available socket memory bandwidth is 42 GB/s.

We use the Intel Compiler 13.1.3.192 with optimization flags `-O3 -xAVX` and Intel MKL 11.0 Update 5. To use OpenMP within the application (additional flag `-openmp`), PETSc was built using the thread-aware version of the MPI library (Intel MPI flag `-mt_mpi`), which had no impact on the serial performance. For pinning of threads to cores and for performance analysis, we use LIKWID 3.1.1 [29].

7 Numerical Results

7.1 OpenMP Parallel Assembly

In a first quick runtime profile we observe that the assembly of the local stiffness matrix $K^{(i)}$ constitutes the major contribution to the overall program runtime (55 %; 177 s assembly, 320 s overall). Therefore, the assembly subroutine is refactored as a preliminary step before parallelization.

Our setting allows certain simplifications in the assembly. In unstructured finite element meshes the number of nonzeros in each row can largely differ. In standard CSR-type matrix storage, as implemented in the PETSc-AIJ format, the performance penalty for insufficient preallocation of memory for the rows is very high. Thus, an upper bound for the nonzeros per row has to be estimated or the number of nonzeros has to be precomputed by looping over all finite elements. The latter approach will effectively double the assembly time since the loop over all finite elements has to be traversed twice. We often use a custom, flexible sparse matrix type that does not require memory preallocation and still allows efficient insertion of entries with a cost logarithmic in the number of nonzero entries. On the other hand, this flexibility comes at the cost of non-optimal element access and overheads for conversion to the PETSc matrix format used after the assembly phase. In the setting presented here, however, it is known that the number of nonzeros in each row is bounded by 38. A simple preallocation can therefore be used and the custom matrix class can be eliminated. This results in a reduction of the assembly time by 77 % (177 s vs. 40 s). With this approach, the overall program runtime is halved (320 s vs. 157 s). As a result, the assembly is not the dominant runtime contribution any more (40 of 157 s, about 25 %). Nevertheless, we proceed with parallelizing the assembly phase.

The local stiffness matrices on the finite elements can be computed independently. Therefore, no data dependencies between iterations of the finite element loop exist, and the loop can be parallelized using `OMP_SCHEDULE=static`. After declaring all data structures to be local to each loop iteration, the insertion of the values into the target matrix with the PETSc function `MatSetValues` is protected by an OpenMP `critical` region. This is necessary, since in finite element matrices, matrix entries will generally be accessed multiple times. The serialization of the calls to `MatSetValues` exposed no drawback in terms of performance. An implementation of a matrix coloring scheme as in [17] is therefore not required in our case. This straightforward threading approach of the assembly routine shows good scalability. A speedup of 8 using ten threads per process is reached; see the solid line in Fig. 3.

In order to determine whether certain use of STL containers, such as, e.g., calls to `vector.pushback()`, within the OpenMP parallel part of the assembly routine is performance critical, we also conduct a single-thread runtime profile (using `gprof` and Intel `loopprofileviewer`).

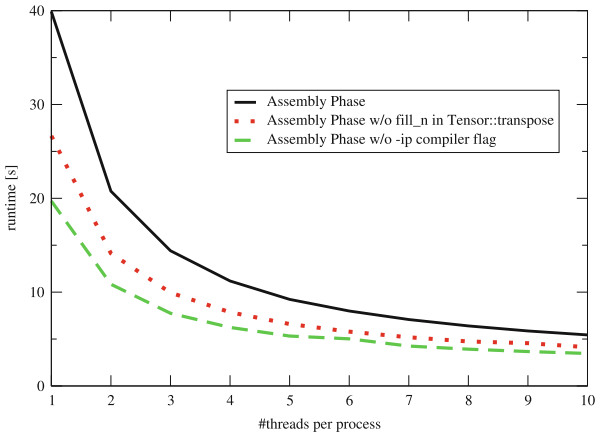

Fig. 3 Socket scaling of the OpenMP parallel assembly routine. Measurements are based on consecutive code modifications (Neo-Hookean hyperelasticity problem in 2D with $m_x = m_y = 200$)

The profiling reveals that the major part, indeed about 90 %, of the assembly runtime is spent in the computation of the element stiffness matrix. This is to be expected. This function in turn is then dominated by the innermost of three nested loops. The outermost loop is a loop over the integration points given by a numerical integration scheme. The two nested inner loops are loops over the degrees of freedom of the finite element.

For the computation of the element stiffness matrix, the transposes of matrices storing the derivatives of the finite element basis functions are required. The derivatives are implemented using a custom C++ class *Tensor*, which provides several tensor operations, such as a transpose method. Surprisingly, this transpose method of the Tensor class is the hotspot in the innermost for loop. In this method, the standard constructor is used to construct a temporary Tensor object. About 50 % (1.4 billion) of the calls to this constructor can be accounted to transpose. This constructor then initializes a Tensor element using `std::__fill_n`, which, surprisingly, turns out to be the largest runtime contribution in the whole application: According to the profile, this function makes up 25 % of the overall runtime. As the dimensions of the input and output Tensors in this case are identical, we have implemented a specialized transpose function that does not use the standard constructor to initialize the returned matrix but the implicit copy constructor, and therefore avoids excessive calls to `std::__fill_n`. A subsequent profiling run shows that this transpose now makes up only 0.05 % of the overall runtime. Single-thread assembly runtime is reduced by about one third; cf. the solid and dotted lines in Fig. 3. The reason for this, however, is not a reduction of computational complexity. The copy constructor in the specialized transpose as well as the `std::__fill_n` in the default constructor both loop over all Tensor elements. The reason is rather a reduction of code complexity which enables the compiler

to inline and optimize the modified transpose method, which it was not able to do with the original version. As other functions still use `std::__fill_n`, including the standard constructor and, e.g., the multiplication function, its share of overall runtime is now about 8 % and subject to further optimizations.

In order to get a more detailed profile we have tried to reduce code inlining by compiling the application **without** the `-ip` (inter-procedural optimization) compiler flag. This did not influence profile detail, but had an unexpected influence on performance: The original single-thread assembly runtime is halved, regardless whether the original or specialized transpose function is used; cf. the solid and dashed lines in Fig. 3. Apparently, the compiler itself is now able to perform the optimizations just described, i.e., it is able to use inlining. This is surprising, as inter-procedural optimization is intended to provide the compiler a better overview of the code. However, in the present case it caused a lack of appropriate inlining. With this, overall single-thread program runtime reduces to 134 s, where the assembly phase requires 20 s (15 %). As building PETSc without the `-ip` compiler switch seems to have no impact on performance in our setting, we now generally omit it. Assembly of the load vector $f^{(i)}$ is currently of minor impact to overall performance. Nevertheless, the steps described for the stiffness matrix here should be considered there, too, in the future.

7.2 Shared Memory Parallel Direct Solver PARDISO

To provide shared memory parallelism in the *FETI-DP Setup* phase, which is dominated by the LU factorizations, we use the shared memory parallel direct solver package PARDISO [25] from the Intel MKL library. We interface PARDISO through PETSc using a third party interface [7]. The current PETSc version 3.5 ships with an included PARDISO interface [3]. We use PARDISO to perform the sparse LU factorizations (*FETI-DP Setup* phase in the graphs) and the forward/backward substitutions (*Krylov Iteration* phase in the graphs) in a threaded parallel fashion within each MPI process. Our PARDISO options are shown in Sect. 6.1.

The runtime of the threaded PARDISO-based phases *FETI-DP Setup* and *Krylov Iteration* is shown for a single socket in Fig. 4. Both parts show a stepwise decrease in runtime and a speed-up of only 3.5–4 on ten cores. Substantial performance improvement can be seen when hitting "magic" thread counts, which are powers of two, while runtime stays constant otherwise. This behaviour is a first indication that the PARDISO phases are not limited by typical hardware bottlenecks such as the data transfer bandwidths. In Fig. 5 we confirm this assumption by showing the bandwidths utilized over the different data paths as measured with our LIKWID tool. The *Krylov Iteration* has the highest main memory bandwidth utilization but still only exploits 50 % of the maximum bandwidth of the processor. It is interesting to see that bandwidth over the on-chip data paths is almost identical to main memory bandwidth for the *Krylov Iteration*, clearly indicating that the current implementation is stream like and makes no reuse of data in L2 and L3 cache. A

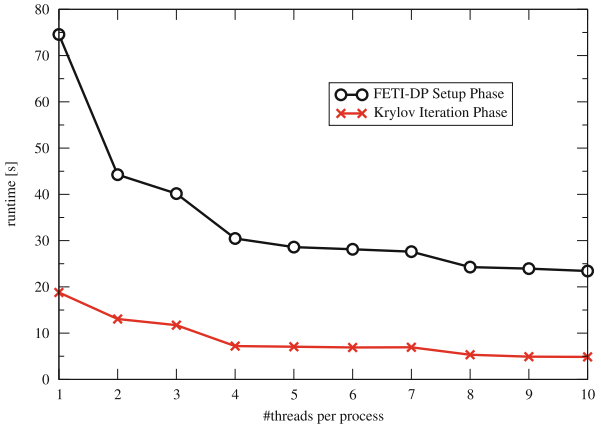

Fig. 4 Socket scaling of program parts parallelized with PARDISO (Neo-Hookean hyperelasticity problem in 2D with $m_x = m_y = 200$)

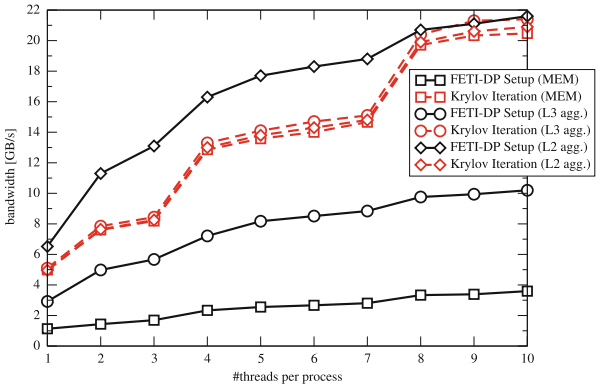

Fig. 5 Bandwidth utilization of the memory hierarchy levels of one socket for the two PARDISO parallel program parts. The different symbols mark bandwidth between memory and L3 cache (*squares*), L3 and L2 caches (*circles*), and L2 and L1 caches (*diamonds*). Cache bandwidth values are aggregated over all utilized cores. The available socket memory bandwidth of the IvyBridge system is 42 GB/s

different characteristic shows up for the *FETI-DP Setup* phase where main memory bandwidth is of no relevance but cache reuse seems to be high due to strong increase of utilized bandwidth for inner level caches. However the cache bandwidths shown in Fig. 5 for *FETI-DP Setup* are still a factor of 5x-10x away from practical hardware limitations ruling them out as potential bottlenecks. Moreover, the LIKWID analysis

has revealed a high AVX vectorization rate for *FETI-DP Setup* (which is the time critical phase) and pure scalar execution for the *Krylov Iteration*. This analysis gives a clear indication that improving the parallelization approach in the two phases is the most promising strategy to further boost performance for the direct solver phase. Work in this direction has been started with one of the developers of PARDISO (Olaf Schenk).

7.3 Runtime Study of the Full Threaded Software

We now consider the total time to solution for our Neo-Hooke material using the OpenMP threaded assembly and the shared memory solver PARDISO. We use four subdomains distributed to four MPI ranks. The four processes are placed on four nodes (one process per node, Setup 1), on the four sockets of two nodes (Setup 2), on the two sockets of one node (Setup 3) or on a single socket (Setup 4). Then we vary the number of threads, i.e., each MPI process spawned up to ten threads (Setups 1 and 2), up to five threads (Setup 3), and up to two threads (Setup 4). See Table 1 for an illustration of the hybrid setups. The results are presented in Fig. 6. Interestingly, we observe that the quantitative scaling behavior (runtime variation) is virtually independent of the hybrid configuration, i.e., the distribution of processes and threads within nodes and sockets. The parallel efficiency falls below 50 % when five or more threads are used.

Since the scaling behaviour is basically identical for the different hybrid setups (Setup 1, 2, 3, and 4), we report on detailed timings only for Setup 1; see Table 2. The scaling results are not optimal but encouraging. Satisfactory results could be achieved using up to ten threads for the *Assembly* phase adopting our simple approach of protecting the calls to `MatSetValues` by an OpenMP `critical` region, and without parallelization of the load vector assembly. The scaling of the *FETI-DP Setup* and the *Krylov iteration* phase relies on the scaling behavior of PARDISO. It is known that perfect parallel scalability of direct solvers for sparse linear systems from PDEs is hard to achieve.

From Table 2, we observe that for our hybrid MPI/OpenMP approach we obtain an overall efficiency of 58 % using four OpenMP threads per MPI rank. Investing up to ten threads per MPI rank still reduces the total execution time but at diminishing returns. Our results are thus similar to the findings in [16, Table II] for FETI-1.

In Fig. 7 the experiments from Fig. 6 are repeated but now using 64 MPI ranks and up to 640 threads. The results are qualitatively identical. Due to numerical effects the number of Krylov iterations is slightly larger. Instead of four Newton steps and between 61 and 67 Krylov iterations it now takes five Newton steps and between 209 and 211 Krylov iterations until convergence resulting in slightly higher runtimes.

Table 1 Distribution of threads (T) and processes (R, MPI ranks) on the machine for the four hybrid setups using 4 MPI ranks. Each node has two sockets, each socket holds an Ivy Bridge processor with 10 cores (C)

		Socket 0										Socket 1									
	Node	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
Setup 1	0	R0 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9										
	1	R1 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9										
	2	R2 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9										
	3	R3 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9										
Setup 2	0	R0 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	R1 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
	1	R2 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	R3 / T0	T1	T2	T3	T4	T5	T6	T7	T8	T9
Setup 3	0	R0 / T0	T1	T2	T3	T4	R2 / T0	T1	T2	T3	T4	R1 / T0	T1	T2	T3	T4	R3 / T0	T1	T2	T3	T4
Setup 4	0	R0 / T0	T1	R1 / T0	T1	R2 / T0	T1	R3 / T0	T1												

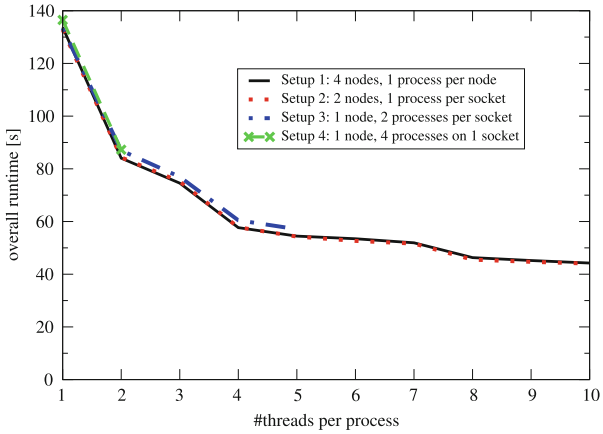

Fig. 6 Socket scaling of Neo-Hookean hyperelasticity problem in 2D with $m_x = m_y = 200$ using the four hybrid setups described in Sect. 7.3. Assembly is threaded using OpenMP, *FETI-DP Setup* and *Krylov Iteration* use the threaded PARDISO solver

Let us briefly comment on the relevance of the hybrid parallelization approach taken in this paper. In our approach using, e.g., 128 MPI ranks (and subdomains) on 128 cores is more efficient than taking the hybrid approach using, e.g., 32 MPI processes (and subdomains) and four threads (for each subdomain), i.e., we have 43.4 s for the pure MPI approach and 80.6 s for the hybrid approach.

In general, a pure MPI implementation of the FETI-DP method will scale with a high parallel efficiency for a small number of MPI ranks. From Table 2, threading on the subdomains scales from one to four threads with an acceptable efficiency of 58 %. When scaling from one to eight threads only a low efficiency of 36 % is reached. We therefore do not recommend to use eight threads with the current implementation. An efficiency of 58 % using four threads for each subdomain, however, is acceptable. Still, using cores by MPI ranks instead of threads will usually be more efficient for a small number of subdomains. However, for a large number of subdomains the time for solving the FETI-DP coarse problem becomes significant and threading on the subdomains helps to keep the coarse problem small. On large machines, such as SuperMUC, the time for the local problems and for the coarse problem have to be balanced, carefully. Moreover, on architectures like BlueGene/Q using 64 hardware threads per node by pure MPI can be challenging due to the memory restrictions of 256 MB per rank which forces the use of very small subdomains. Here, threading can be the only way to make efficient use of the hardware threads.

Table 2 Runtime in seconds of overall program and threaded phases for one up to ten threads per MPI process (Setup 1, i.e., one MPI process on each of the four nodes, Neo-Hookean hyperelasticity problem in 2D with $m_x = m_y = 200$).

Threads	Assembly (s)	Efficiency (%)	FETI-DP setup (s)	Efficiency(%)	Krylov iteration (s)	Efficiency (%)	Overall (s)	Efficiency (%)
1	19.71	100	74.56	100	18.79	100	133.69	100
2	10.84	91	44.25	84	13.05	72	84.04	80
3	7.75	85	40.16	62	11.72	53	74.56	60
4	6.24	79	30.46	61	7.20	65	57.68	58
5	5.32	74	28.60	52	7.06	53	54.46	49
6	5.02	65	28.12	44	6.90	45	53.47	42
7	4.25	66	27.61	39	6.94	39	51.91	37
8	3.92	63	24.27	38	5.33	44	46.30	36
9	3.67	60	23.94	35	4.91	43	45.19	33
10	3.46	57	23.43	32	4.86	39	44.24	30

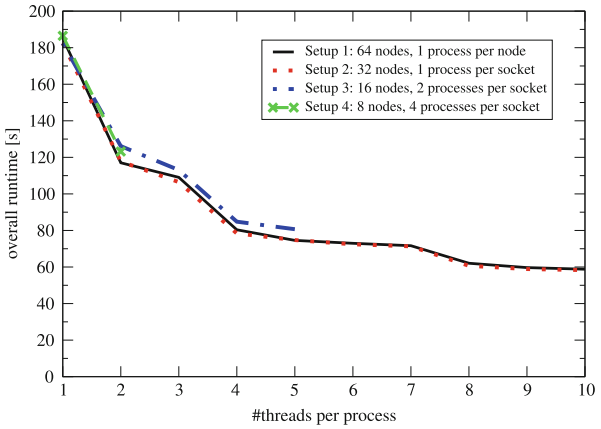

Fig. 7 Socket scaling of Neo-Hookean hyperelasticity problem in 2D with $m_x = m_y = 200$ as in Fig. 6 but using 64 MPI ranks and a maximum of 640 threads

8 Summary

We have presented a thorough profiling and code investigation in combination with a hardware bottleneck analysis for important steps of the FETI-DP method with special focus on node-level performance. It has been demonstrated that the impact of widely used C++ techniques on performance needs to be carefully investigated and rather basic code changes may result in large performance improvements. For the two phases *FETI-DP Setup* and the *Krylov iteration*, we have clearly shown that the code is "core-bound" and identified that the current parallelization strategy is the performance limiting factor on the socket level. Improving shared memory parallelization of the respective routines in the widely used PARDISO solver may not only provide additional performance for our application but will be of great interest for a large community.

Acknowledgements This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA) under KL 2094/4-1, RH 122/2-1, WE 5289/1-1.

References

1. Amestoy, P.R., Duff, I.S., l'Excellent, J.Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods Appl. Mech. Eng. **184**(2–4), 501–520 (2000)
2. Amestoy, P.R., Duff, I.S., l'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. **23**(1), 15–41 (2001)

3. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zhang, H.: Changes in the petsc 3.5 version. http://www.mcs.anl.gov/petsc/documentation/changes/35.html (2014)
4. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zhang, H.: PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory. http://www.mcs.anl.gov/petsc (2014)
5. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser, Boston (1997)
6. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Rupp, K., Smith, B.F., Zhang, H.: PETSc Web page. http://www.mcs.anl.gov/petsc (2014)
7. Bermeo, J.D.: Added support for mkl-pardiso solver. https://bitbucket.org/petsc/petsc/pull-request/105/added-support-for-mkl-pardiso-solver/commits (2013)
8. Bhardwaj, M., Pierson, K.H., Reese, G., Walsh, T., Day, D., Alvin, K., Peery, J., Farhat, C., Lesoinne, M.: Salinas: a scalable software for high performance structural and mechanics simulation. In: ACM/IEEE Proceedings of SC02: High Performance Networking and Computing. Gordon Bell Award, pp. 1–19 (2002)
9. Davis, T.A.: A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. ACM Trans. Math. Softw. **30**(2), 165–195 (2004). http://doi.acm.org/10.1145/992200.992205
10. Davis, T.A., Duff, I.S.: An unsymmetric-pattern multifrontal method for sparse lu factorization. SIAM J. Matrix Anal. Appl. **18**(1), 140–158 (1997)
11. Davis, T.A., Duff, I.S.: A combined unifrontal/multifrontal method for unsymmetric sparse matrices. ACM Trans. Math. Softw. **25**(1), 1–19 (1999)
12. Farhat, C., Lesoinne, M., Pierson, K.: A scalable dual-primal domain decomposition method. Numer. Linear Algebra Appl. **7**, 687–714 (2000)
13. Farhat, C., Lesoinne, M., LeTallec, P., Pierson, K., Rixen, D.: FETI-DP: a dual-primal unified FETI method - part i: a faster alternative to the two-level FETI method. Int. J. Numer. Methods Eng. **50**, 1523–1544 (2001)
14. Guèye, I.: Solving large linear systems arising in finite element approximations on massively parallel computers. Theses, Mines ParisTech (2009). https://tel.archives-ouvertes.fr/tel-00477653
15. Guèye, I., Juvigny, X., Feyel, F., Roux, F.X., Cailletaud, G.: A parallel algorithm for direct solution of large sparse linear systems, well suitable to domain decomposition methods. Eur. J. Comput. Mech./Revue Européenne de Mécanique Numérique **18**(7–8), 589–605 (2009). doi:10.3166/ejcm.18.589–605
16. Guèye, I., Arem, S.E., Feyel, F., Roux, F.X., Cailletaud, G.: A new parallel sparse direct solver: Presentation and numerical experiments in large-scale structural mechanics parallel computing. Int. J. Numer. Methods Eng. **88**(4), 370–384 (2011). doi:10.1002/nme.3179. http://dx.doi.org/10.1002/nme.3179
17. Guo, X., Gorman, G., Lange, M., Sunderland, A., Ashworth, M.: Developing hybrid openmp/mpi parallelism for fluidity-icom - next generation geophysical fluid modelling technology (2012). http://www.hector.ac.uk/cse/distributedcse/reports/fluidity-icom02/fluidity-icom02.pdf. Final Report for DCSE ICOM
18. Klawonn, A., Rheinbach, O.: Inexact FETI-DP methods. Int. J. Numer. Methods Eng. **69**(2), 284–307 (2007)
19. Klawonn, A., Lanser, M., Rheinbach, O.: Towards extremely scalable nonlinear domain decomposition methods for elliptic partial differential equation. Tech. Rep. 2014–13, Preprint Reihe, Fakultät für Mathematik, TU Bergakademie Freiberg, ISSN 1433-9407. http://tu-freiberg.de/fakult1/forschung/preprints (2014) [Submitted to SISC]

20. Klawonn, A., Lanser, M., Rheinbach, O.: A nonlinear FETI-DP method with an inexact coarse problem. In: Dickopf, T., Gander, M.J., Krause, R., Pavarino, L.F. (eds.) Domain Decomposition Methods in Science and Engineering. Lecture Notes in Computational Science and Engineering, vol. 22. Springer, Heidelberg (2015); Accepted for publication October 2014. Proceedings of the 22nd Conference on Domain Decomposition Methods in Science and Engineering, Lugano, 16–20 September 2013. Also http://tu-freiberg.de/fakult1/forschung/preprints
21. Klawonn, A., Rheinbach, O.: Highly scalable parallel domain decomposition methods with an application to biomechanics. ZAMM Z. Angew. Math. Mech. **90**(1), 5–32 (2010). doi:10.1002/zamm.200900329. http://dx.doi.org/10.1002/zamm.200900329
22. Klawonn, A., Widlund, O.B.: Dual-primal FETI methods for linear elasticity. Commun. Pure Appl. Math. **59**(11), 1523–1572 (2006)
23. Kuzmin, A., Luisier, M., Schenk, O.: Fast methods for computing selected elements of the greens function in massively parallel nanoelectronic device simulations. In: Wolf, F., Mohr, B., Mey, D. (eds.) Euro-Par 2013 Parallel Processing. Lecture Notes in Computer Science, vol. 8097, pp. 533–544. Springer, Berlin/Heidelberg (2013)
24. Rheinbach, O.: Parallel iterative substructuring in structural mechanics. Arch. Comput. Methods Eng. **16**(4), 425–463 (2009). doi:10.1007/s11831-009-9035-4. http://dx.doi.org/10.1007/s11831-009-9035-4
25. Schenk, O., Wächter, A., Hagemann, M.: Matching-based preprocessing algorithms to the solution of saddle-point problems in large-scale nonconvex interior-point optimization. Comput. Optim. Appl. **36**(2–3), 321–341 (2007). doi:10.1007/s10589-006-9003-y. http://dx.doi.org/10.1007/s10589-006-9003-y
26. Schenk, O., Bollhöfer, M., Römer, R.A.: On large-scale diagonalization techniques for the anderson model of localization. SIAM Rev. **50**(1), 91–112 (2008). doi:10.1137/070707002. http://dx.doi.org/10.1137/070707002
27. Smith, B.F., Bjørstad, P.E., Gropp, W.: Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, Cambridge (1996)
28. Toselli, A., Widlund, O.: Domain Decomposition Methods - Algorithms and Theory. Springer Series in Computational Mathematics, vol. 34. Springer, Heidelberg (2004)
29. Treibig, J., Hager, G., Wellein, G.: LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, pp. 207–216. IEEE Computer Society, Los Alamitos (2010). http://dx.doi.org/10.1109/ICPPW.2010.38
30. Zienkiewicz, O., Taylor, R.: The Finite Element Method for Solid and Structural Mechanics. Elsevier, Oxford (2005)