Computational Mechanics (2019) 64:1339–1345
https://doi.org/10.1007/s00466-019-01713-3

ORIGINAL PAPER

Runtime optimization of a memory efficient CG solver for FFT-based homogenization: implementation details and scaling results for linear elasticity

Hannes Grimm-Strele[1] · Matthias Kabel[1]

Received: 15 October 2018 / Accepted: 23 April 2019 / Published online: 29 April 2019

Abstract
The memory efficient CG algorithm of Kabel et al. (Comput Mech 54(6):1497–1514, 2014) reduces the memory requirements of a strain based implementation of the CG algorithm following Zeman et al. (J Comput Phys 229(21):8065–8071, 2010) for solving the equations of linear elasticity by 40%. But since the Fourier wave vectors have to be recalculated at several steps of the memory efficient algorithm, the runtime increases for a straightforward implementation.. We explain how to reduce the runtime overhead to a negligible size, and show that the memory efficient algorithm scales better than the standard algorithm with up to 256 MPI processes.

Keywords Lippmann–Schwinger equation · Homogenization · Numerical methods · Linear elasticity

1 Introduction

The FFT–based algorithm of [3] is a fast and accurate method for obtaining effective properties in linear elasticity and conductivity problems. Even though its memory requirements are low due to the matrix free implementation, it can still be a limiting factor to the resolution and therefore to the accuracy of the numerical simulation. When solving the discretized system by the CG algorithm as suggested in [2] instead by a Neumann series expansion as in [3], the memory usage per voxel is increased from 144 bytes to 360, a factor of 2.5. To mitigate this additional burden while still benefiting from the fast convergence of the CG algorithm, [1] suggests an alternative implementation reducing the memory requirement to 216 bytes per voxel, a reduction of 40%.

A direct implementation of the proposed memory efficient CG algorithm increases the runtime by a factor two. We will show how the computational overhead can be considerably reduced. We will also investigate the scaling behaviour of both the standard CG algorithm and the memory efficient implementation.

✉ Hannes Grimm-Strele
hannes.grimm-strele@itwm.fraunhofer.de

[1] Department of Flow and Material Simulation, Fraunhofer ITWM, Kaiserslautern, Germany

Besides low memory consumption, scalability is a major prerequisite for an efficient FFT solver. [4] has recently compared MPI and OpenMP parallelization strategies. They show that with MPI, nearly perfect scaling can be achieved for a solver which is very similar to ours.

[5] emphasizes the dependence of the scaling of their code on the material law. Since applying the material law is a local operation and scales perfectly for linear elasticity, the scaling of the code is better if the material law application dominates overall runtime. In this sense, linear elasticity is a challenging test for the scalability of a code since in this case, the application of the material law is computationally cheap.

In this work, we focus on the staggered grid discretization from [6]. But the CG algorithm can be combined with many other discretizations as, e.g., the basic discretization from [3] and the discretizations from [8–10], to which our results apply partly, too.

1.1 Memory efficient CG algorithm

In the following, we describe the memory efficient CG algorithm from [1] for small deformations. Our aim is to solve the Lippmann–Schwinger equation

$$\varepsilon + \Gamma^0 * ((\mathcal{C} - \mathcal{C}_0) : \varepsilon) = E, \;\; \varepsilon = E + \varepsilon(u), \qquad (1)$$

where the strain ε is given by the sum of E, the prescribed macroscopic field, and a periodic fluctuation field depending on the displacement u. The simulation domain is a cuboid V with periodic boundary fluctuation conditions.

Equation (1) can be brought in the form $A\varepsilon = E$ by writing $A = \text{Id} + B$ and $B = \Gamma^0 (\mathcal{C} - \mathcal{C}_0)$ such that we can apply iterative schemes as the CG algorithm to solve it. The operator Γ^0 has the form

$$\Gamma^0 = \nabla G^0 \, \text{Div}, \tag{2}$$

where G^0 is the Green operator which is explicitly known in Fourier space. For small deformations, the strain operator $\nabla = \nabla_s$ is defined by

$$\nabla_s u = \begin{pmatrix} \frac{\partial u_1}{\partial x_1} & \frac{1}{2}\left(\frac{\partial u_2}{\partial x_1} + \frac{\partial u_1}{\partial x_2}\right) & \frac{1}{2}\left(\frac{\partial u_3}{\partial x_1} + \frac{\partial u_1}{\partial x_3}\right) \\ & \frac{\partial u_2}{\partial x_2} & \frac{1}{2}\left(\frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2}\right) \\ \text{sym} & & \frac{\partial u_3}{\partial x_3} \end{pmatrix} \tag{3}$$

for a displacement vector $u = (u_1, u_2, u_3)$.

The key idea is to save all additional arrays of the CG algorithm as displacement fluctuation fields instead of strain fields, thereby halving the memory requirements of each array except for one array which stores the final solution. The modified algorithm yields the same outcome as the naive implementation based on strain fields.

2 Implementation details

When implementing the memory efficient CG algorithm from [1], we need to define how to apply the operator

$$\widehat{G^0}\widehat{\text{Div}}\left(\text{FFT}\left(-(\mathcal{C} - \mathcal{C}_0) : \text{FFT}^{-1}\left(\widehat{\nabla_s}\right)\right)\right) \tag{4}$$

and the operator ∇_s as defined in (3) to a displacement fluctuation field. We also need to specify how to calculate the inner product from the Fourier coefficients of the displacements. As a convergence criterion, we use the simple method given in [1],

$$\frac{\left| \|\varepsilon^{\text{new}}\|^2 - \|\varepsilon^{\text{old}}\|^2 \right|}{\|\varepsilon^{\text{initial}}\|^2} < \text{tolerance}. \tag{5}$$

In the following, capital letters refer to strain fields while lower case letters refer to displacement fluctuation fields.

When we calculate the inner product on the displacement fluctuation fields, it must give the same result as when calculated on the strain fields. More specifically, if $\nabla_s q = Q$ and $\nabla_s y = Y$, we require that

$$\langle Q, Y \rangle = \langle \nabla_s q, \nabla_s y \rangle. \tag{6}$$

Our simulation domain V is a cuboid which we discretize by a voxel mesh with $N_1 \times N_2 \times N_3$ voxels. All voxels have the same constant volume. Then,

$$\begin{aligned} \langle Q, Y \rangle &= \frac{1}{|V|}\int_V Q : Y \, dX = \frac{1}{|V|}\int_V \sum_{l,m=1}^{3} Q_{l,m} Y_{l,m} \, dX \\ &= \sum_{l,m=1}^{3} \frac{1}{|V|}\int_V Q_{l,m} Y_{l,m} \, dX \\ &= \frac{1}{N} \sum_{l,m=1}^{3} \sum_{i=1}^{N_1}\sum_{j=1}^{N_2}\sum_{k=1}^{N_3} Q_{l,m}(i,j,k) Y_{l,m}(i,j,k), \end{aligned} \tag{7}$$

with the Frobenius inner product defined by $Q : Y = \sum_{l,m} Q^*_{l,m} Y_{l,m} = \sum_{l,m} Q_{l,m} Y_{l,m}$ (since Q and Y are real-valued and symmetric), and $N = N_1 N_2 N_3$. Due to Parseval's theorem, the last sum can be calculated by summing the Fourier coefficients of $Q_{l,m}$ and $Y_{l,m}$. We write $\xi = \xi(i,j,k) = \left(\frac{2\pi i}{N_1}, \frac{2\pi j}{N_1}, \frac{2\pi k}{N_3}\right)$ for the Fourier wave vector. The form of the Fourier wave vector and the constant factors in front of the sums might change depending on the implementation of the FFT library.

Continuing from (7),

$$\begin{aligned} &\sum_{l,m=1}^{3} \sum_{i=1}^{N_1}\sum_{j=1}^{N_2}\sum_{k=1}^{N_3} Q_{l,m}(i,j,k) Y_{l,m}(i,j,k) \\ &= \sum_{\xi} \sum_{l,m=1}^{3} \widehat{Q_{l,m}}(\xi)\widehat{Y_{l,m}}^*(\xi) \\ &= \sum_{\xi} \sum_{l,m=1}^{3} \widehat{\nabla_s q_{l,m}}(\xi)\widehat{\nabla_s y_{l,m}}^*(\xi). \end{aligned} \tag{8}$$

As described in [6], derivatives transform to multiplications with the Fourier wave vectors

$$k^{\pm}(\xi) = \left(k_1^{\pm}(\xi), k_2^{\pm}(\xi), k_3^{\pm}(\xi)\right) \tag{9}$$

in Fourier space. The precise form of $k^{\pm}(\xi)$ depends on the discretization method. For the staggered grid discretization, they are defined by

$$\Re\left(k_j^{\pm}\right) = \pm\frac{\cos\left(\pm\xi_j\right) - 1}{h_j}, \quad \Im\left(k_j^{\pm}\right) = \pm\frac{\sin\left(\pm\xi_j\right)}{h_j}, \tag{10}$$

where h_j is the grid spacing in direction j. The discrete form of the ∇_s and the Div operators in Fourier space depends on the discretization method, too. We focus here on the staggered grid discretization, but similar formulae can be derived in the same way for any other discretization method.

Applying $\widehat{\nabla_s}$ to a displacement q in Fourier space yields

$$\widehat{\nabla_s q}(\xi) = \begin{pmatrix} k_1^+ \widehat{q_1} & \frac{1}{2}\left(k_1^- \widehat{q_2} + k_2^- \widehat{q_1}\right) & \frac{1}{2}\left(k_1^- \widehat{q_3} + k_3^- \widehat{q_1}\right) \\ & k_2^+ \widehat{q_2} & \frac{1}{2}\left(k_3^- \widehat{q_2} + k_2^- \widehat{q_3}\right) \\ \text{sym} & & k_3^+ \widehat{q_3} \end{pmatrix}, \tag{11}$$

with the Fourier coefficients $\widehat{q}(\xi) = (\widehat{q_1}, \widehat{q_2}, \widehat{q_3})$ [6]. Using (11) and since $k_j^+ = -\left(k_j^-\right)^*$, we obtain

$$\sum_{l,m=1}^{3} \widehat{\nabla_s q_{l,m}}(\xi) \widehat{\nabla_s y_{l,m}}^*(\xi) = \sum_{l=1}^{3} \left(k_l^-\right)^* \widehat{q_l} k_l^- \left(\widehat{y_l}\right)^*$$
$$+ \frac{1}{2} \sum_{\substack{l,m=1 \\ l<m}}^{3} \left(k_l^- \widehat{q_m} + k_m^- \widehat{q_l}\right) \left(k_l^- \widehat{y_m} + k_m^- \widehat{y_l}\right)^*. \tag{12}$$

When calculating the norm, i.e., $q = y$, we can simplify this expression to

$$\sum_{l,m=1}^{3} \left|\widehat{\nabla_s q_{l,m}}(\xi)\right|^2$$
$$= \left\|k^-\right\|^2 \|\widehat{q}\|^2 + \sum_{\substack{l,m=1 \\ l<m}}^{3} \Re\left(k_l^- \widehat{q_m} \left(k_m^- \widehat{q_l}\right)^*\right). \tag{13}$$

For the inner product, we cannot simplify the expression in the same way, but we have to calculate the gradients first and sum up their product. Since the input arrays q and y contain only real data, the Fourier coefficients possess the Hermitian symmetry $\widehat{q}(\xi) = (\widehat{q}(-\xi))^*$ [7]. Therefore, the imaginary parts cancel in the final summation, and we only need to take the real parts into account. The result is, as expected, a real number.

As can be seen from Eq. (1), the displacement fields contain only information about the periodic part of the strain field. The mean of the displacement fields is always 0. When calculating the norm or the inner product through the gradients of displacements as in (6), we have to add the mean of the strain fields associated with q and y. Therefore, we save the mean of the corresponding strain field in the vectors $\widehat{q}(0)$ and $\widehat{y}(0)$ and add the inner product $\langle\widehat{q}(0), \widehat{y}(0)\rangle$ to the result. We note that $\widehat{q}(0)$ and $\widehat{y}(0)$ are always strain tensors even though q and y contain displacements.

The implementation of the inner product of two displacement fluctuation fields q and y is summarized in Algorithms 1 and 2.

Algorithm 1 Function **InnerProduct**(q, y)

```
1: $\Sigma \leftarrow 0$
2: for $i = 1, \ldots, N_1,\ j = 1, \ldots, N_2,\ k = 1, \ldots, N_3$ do
3:     $\Sigma \leftarrow \Sigma +$ VoxelProduct$(q(\xi(i,j,k)), y(\xi(i,j,k)))$
4: return $\Sigma$
```

Algorithm 2 Function **VoxelProduct**$(q(\xi), y(\xi))$

```
1: for $l = 1, 2, 3$ do
2:     $\widehat{\nabla_s q_{l,l}} \leftarrow -k_l \widehat{q_l}$, $\widehat{\nabla_s y_{l,l}} \leftarrow -k_l \widehat{y_l}$
3: for $l, m = 1, 2, 3,\ l < m$ do
4:     $\widehat{\nabla_s q_{l,m}} \leftarrow k_l \widehat{q_m} + k_m \widehat{q_l}$, $\widehat{\nabla_s y_{l,m}} \leftarrow k_l \widehat{y_m} + k_m \widehat{y_l}$
5: return $\sum_{\substack{l,m=1 \\ l\le m}}^{3} \Re\left(\widehat{\nabla_s q_{l,m}} \widehat{\nabla_s y_{l,m}}\right)$
```

2.1 Minimizing runtime overhead

In the memory efficient CG algorithm, the discretized wave vectors $k^{\pm}$ in Fourier space are needed every time we calculate an inner product, a norm, or apply one of the operators $\widehat{\nabla_s}$, $\widehat{G^0}$, and $\widehat{\text{Div}}$. If we use the formula (10), calculating $k^{\pm}$ is quite expensive, and recalculating it in each loop introduces considerable overhead compared to the standard CG algorithm.

We can decrease this overhead in two ways. We can modify the function applying the operator defined by Eq. (4) such that the parameter α, the norms $\|q\|^2$ and $\|q-w\|^2$, the inner products $\langle q, q-w\rangle$, $\langle r, q-w\rangle$ and $\langle x, q\rangle$ are all calculated "on the fly" in the same loop where $\widehat{G^0}\widehat{\text{Div}}$ is applied. Then, we can calculate

$$\delta = \left\|r^{\text{new}}\right\|^2 = \left\|r^{\text{old}}\right\|^2 + \alpha^2 \|q-w\|^2 - 2\alpha\langle r^{\text{old}}, q-w\rangle, \tag{14}$$

$$\left\|u^{\text{new}}\right\|^2 = \left\|u^{\text{old}}\right\|^2 + \alpha^2 \|q\|^2 + 2\alpha\langle u^{\text{old}}, q\rangle, \tag{15}$$

where $\left\|r^{\text{old}}\right\|^2$ and $\left\|u^{\text{old}}\right\|^2$ are known from the previous iteration. $\|u^{\text{new}}\|^2$ is needed for our convergence criterion (5). In this way, we can avoid recalculating the wave vectors $k^{\pm}$ in each of these tasks.

The resulting loop, corresponding to the application of $\widehat{G^0}\widehat{\text{Div}}$ in Eq. (4), is summarized in Algorithm 3. The wave vectors $k^{\pm}(\xi)$ are only calculated once per coefficient. We note that in each application of the **InnerProduct** function as defined in Algorithm 2, the gradient of the input vector has to be computed. Therefore, we modify the corresponding functions to take the gradient as input, and calculate the gradient itself only once.

The input for Algorithm 3 is the strain field

$$W = \text{FFT}\left(-(\mathcal{C} - \mathcal{C}_0) : \left(\text{FFT}^{-1}\left(\widehat{\nabla_s q}\right)\right)\right) \tag{16}$$

Algorithm 3 Apply $\widehat{G^0}\widehat{\mathrm{Div}}$ combined with inner product and norm calculation

```
1: $\Sigma_1 \leftarrow 0, \Sigma_2 \leftarrow 0, \Sigma_3 \leftarrow 0, \Sigma_4 \leftarrow 0, \Sigma_5 \leftarrow 0$
2: for all $\xi$ do
3:     Calculate $k^{\pm}(\xi)$
4:     $w(\xi) \leftarrow \widehat{G^0}\widehat{\mathrm{Div}}\,(W(\xi))$
5:     $\Sigma_1 \leftarrow \Sigma_1 +$ VoxelProduct $(q(\xi), (q-w)(\xi))$
6:     $\Sigma_2 \leftarrow \Sigma_2 +$ VoxelProduct $(r(\xi), (q-w)(\xi))$
7:     $\Sigma_3 \leftarrow \Sigma_3 +$ VoxelProduct $(u(\xi), q(\xi))$
8:     $\Sigma_4 \leftarrow \Sigma_4 +$ VoxelProduct $((q-w)(\xi), (q-w)(\xi))$
9:     $\Sigma_5 \leftarrow \Sigma_5 +$ VoxelProduct $(q(\xi), q(\xi))$
10: for $i = 1, \ldots, 5$ do
11:     $\Sigma_i \leftarrow$ add mean field contribution
12:     $\Sigma_i \leftarrow \Sigma_i / N$
13: $\alpha = \gamma_{\mathrm{old}} / \Sigma_1$
14: $\delta = \delta_{\mathrm{old}} + \alpha^2 \Sigma_4 - 2\alpha \Sigma_2$
15: $\|u^{\mathrm{new}}\|^2 = \|u^{\mathrm{old}}\|^2 + \alpha^2 \Sigma_5 + 2\alpha \Sigma_3$
```

in Fourier space. Furthermore, the arrays q, r and u as well as the parameter γ are used for calculating the parameters α and δ. The array w contains the output displacement field. Additionally, Algorithm 3 returns the norm of the new solution field $\|u^{\mathrm{new}}\|$ used for the convergence test.

Using Algorithm 3, the memory efficicent CG algorithm can be simplified to Algorithm 4. The function **ApplyReducedOperator** corresponds to applying Eq. (4) to a displacement fluctuation field, and **FourierGradient** to applying Eq. (11).

Algorithm 4 Efficient implementation of the algorithm from [1]

```
1: $W \leftarrow \widehat{G^0}\widehat{\mathrm{Div}}\,(\mathrm{FFT}\,(E))$
2: ApplyReducedOperator$(x, W, r)$
3: $q \leftarrow r$
4: $\gamma \leftarrow$ InnerProduct$(r, r)$
5: while not converged do
6:     Algorithm 3$(W, q, r, u, w, \alpha, \gamma, \delta)$
7:     $u \leftarrow u + \alpha q$
8:     $r \leftarrow r - \alpha(q - w)$
9:     $\beta \leftarrow \delta / \gamma$
10:     $\gamma \leftarrow \delta$
11:     $q \leftarrow r + \beta q$
12: $W \leftarrow$ FourierGradient$(u)$
13: return $\mathrm{FFT}^{-1}\,(W)$
```

For the staggered grid discretization [6], according to Eq. (10) and for each $j = 1, 2, 3$, the component $k_j^{\pm}$ of the discretized wave vector depends only on ξ_j. Therefore, these vectors can be precalculated and stored in a one-dimensional array per direction. In each loop, we only need to access the precomputed values instead of calculating them. This is also possible, e.g., for the basic discretization used in the algorithm of [3], but not with the discretization from [8] or with finite element based discretizations [9,10]. If calculating $k_j^{\pm}$ is expensive and the vectors cannot be stored in one–dimensional arrays, but in three-dimensional ones, the memory savings of the whole algorithm are lost.

Nevertheless, we cannot completely remove the runtime overhead because norm and inner product calculation are inevitably more expensive in Fourier space than in real space, as can be seen from Eq. (13). There, we need to calculate two vector norms, twelve complex products, and three real parts, compared to one vector norm in the real case.

3 Numerical results

Verification of Algorithm 4 is straightforward since the results of each iteration must coincide with the results of the classical CG algorithm up to machine precision. It is also obvious that the memory requirements are lower by 40%.

All tests in this section are run on the Beehive cluster at ITWM. The Beehive cluster consists of 166 nodes with each two eight-core Intel Xeon E5-2670 CPUs. All nodes have 64 GB RAM and are connected through Infiniband interconnect. The cluster runs on CentOS Linux release 7.5 (Linux kernel 3.10.0). For all tests, we use gcc version 6.2.0, OpenMPI 1.10.7, MPICH 3.2, and FFTW version 3.3.7. We use the compilation options

```
-mfpmath=387 -funroll-all-loops -O2 -pipe
```

We perform two strong scaling tests with up to 256 tasks where we use the staggered grid discretization. As test geometry, we choose the Berea Sandstone from [11]. The image can be downloaded from [12]. We cut out a box of 512^3 voxels in the center of the original image as shown in Fig. 1. For the solid material, we set the bulk modulus to 36 GPa, the shear modulus to 45 GPa [13], and assume linear isotropy. Then, the Berea sandstone can be treated as a linear elastic problem. We calculate one load case for which it takes 163 iterations of the CG algorithm until Eq. (5) is fulfilled with an tolerance of 10^{-4}.

3.1 Workstation test

In the first setup, we run this test with the memory efficient CG algorithm on a single node of our cluster, and compare the performance of OpenMP parallelization and both MPI libraries. The resulting performance is shown in Fig. 2 together with the runtime of the naive implementation of the algorithm parallelized with OpenMP.

In this test, the runtime with the naive implementation for the iterative solver without the FFT transformations is about a factor two higher than with the efficient implementation. We excluded the FFT runtime since it is the same for all algorithms, and its share of the total runtime depends on the problem size and the number of parallel processes.

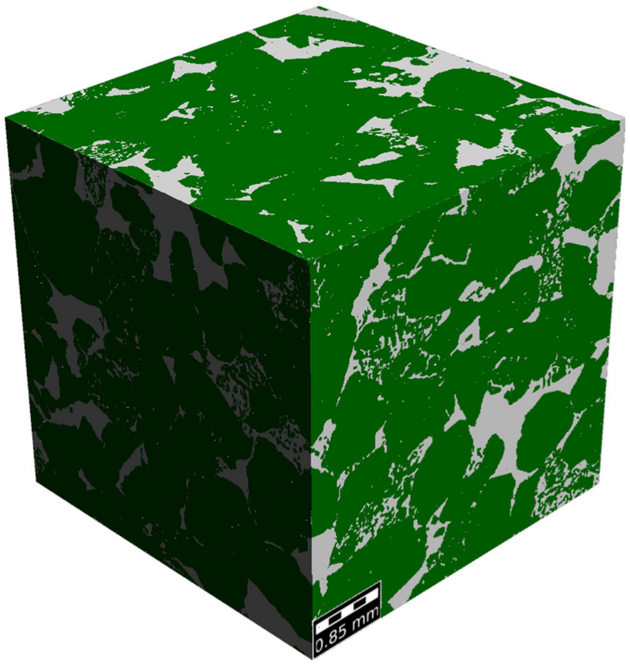

Fig. 1 Berea sandstone dataset with a binary segmentation. The image has 512 × 512 × 512 voxels and the voxel edge length is 8.444 µm

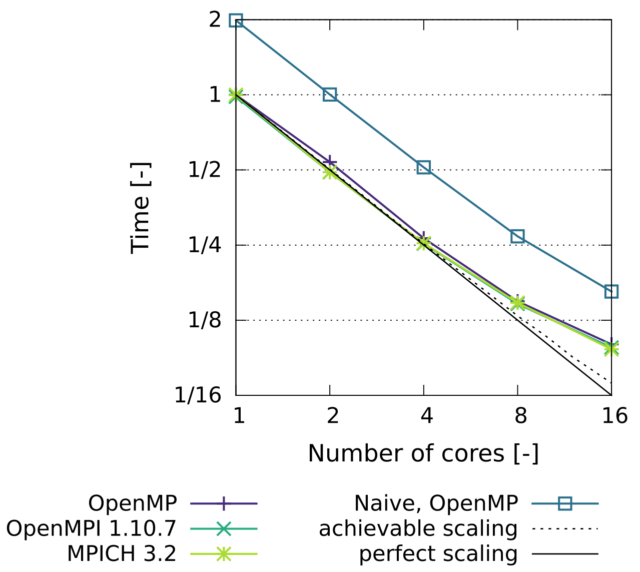

Fig. 2 Solver runtime with FFT transformations for the workstation test with the memory efficient CG algorithm on one single cluster node. Details can be found in Sect. 3.1

The scaling efficiency is generally very good. We observe that with MPI, the parallel efficiency is slightly better than with OpenMP. The choice of the MPI library does not significantly change the parallel performance. In all cases, we observe a particularly strong slowdown when going from 8 to 16 cores. This is partly due to the decreasing frequency of the cores depending on the number of "active cores" as described in [14]. Similar to their work, we could deduce the

Table 1 Empirical dependence of the core frequency on the number of active cores for the 8-core Intel Xeon E5-2670 CPU

Active cores	1	2	3	4	
frequency (GHz)	3.3	3.26	3.20	3.17	
Active cores	5	6	7	8	Base
frequency (GHz)	3.10	3.09	3.01	2.95	2.6

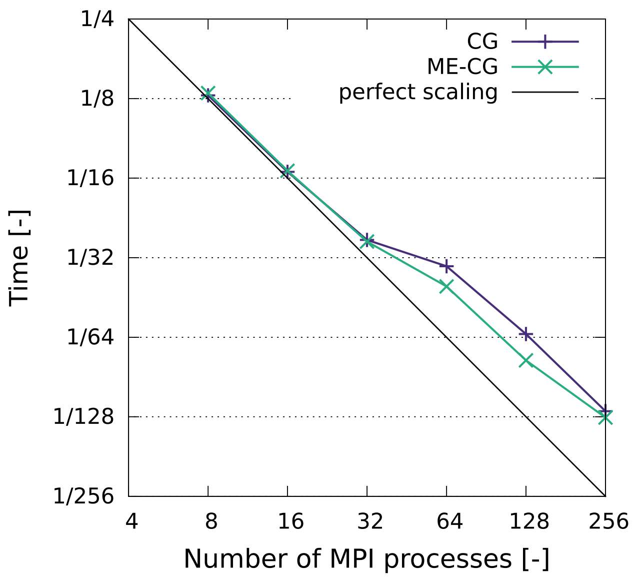

Fig. 3 Scaling results for the cluster test with up to 256 MPI processes. Details can be found in Sect. 3.2

core frequencies of the Intel Xeon E5-2670 CPUs of Beehive depending on the number of active cores using multiplication tests. The resuling clock rates are shown in Table 1. In Fig. 2, the line designated by "achievable scaling" takes the expected slowdown due to this effect into account. These results, in particular the improved performance with less active cores per node, is in accordance with the findings of [4].

While the standard CG algorithm consumed 11.62 GB of main memory, the memory consumption of the memory efficient variant was only 7.32 GB. This represents as reduction of about 40%, as predicted. $k^{\pm}(\xi)$ can be stored in three one-dimensional arrays, and therefore the additional amount of memory needed is negligible.

3.2 Cluster test

In the second setup, we repeat the same test, but using more MPI processes and comparing the standard CG with the memory efficient CG algorithm. To avoid dependency on the number of active cores, we set the number of processes per node always to 8. Since the results do not significantly depend on the MPI library, we only show data for OpenMPI 1.10.7.

Table 2 Total runtime depending on the number of MPI processes, and time spent in the FFT calls for the standard CG algorithm. The second column lists the number of slices per MPI process

Processes	Slices	Total time (s)	FFT (s)	FFT (s)
8	64	2239.87	1081.80	48
16	32	1150.57	558.29	48
32	16	636.72	326.91	51
64	8	509.52	281.86	49
128	4	289.07	153.50	55
256	2	170.80	90.35	53

Table 3 Total runtime depending on the number of MPI processes, and time spent in the FFT calls for the memory efficient CG algorithm. The second column lists the number of slices per MPI process

Processes	Slices	Total time (s)	FFT (s)	FFT (%)
8	64	2277.66	1086.00	48
16	32	1157.92	560.55	48
32	16	624.36	321.79	52
64	8	422.38	256.86	61
128	4	221.94	138.33	62
256	2	134.77	87.93	65

From Fig. 3, we observe that the scaling is still very good for both algorithms. The memory efficient CG algorithm is even slightly faster in most cases. In particular, there is no performance loss compared to the standard CG algorithm.

Due to the problem size, more than 256 MPI processes cannot be efficiently used in this test. Other decisive factors influencing the scaling is the parallel performance of the FFTW library [7]. As can be seen from Tables 2 and 3 where the total runtime and the time spent in the FFT calls is listed for both the standard and the memory efficient CG algorithm, the share of the runtime spent in the FFT calls increases with the number of MPI processes, indicating that the scaling efficiency of the FFT libraries is worse than the rest of the code. In particular, the FFT runtime does only slightly decrease when increasing the number of processes from 32 to 64.

4 Conclusions

The memory efficient CG algorithm from [1] reduces the memory requirements of numerical simulations of linear elasticity by around 40%. At the same time, it introduces a runtime overhead. Depending on the parallelization technique and the problem size, this overhead can be reduced in the range of 0 to 15% of the runtime of the standard implementation of the CG algorithm. Even though the runtime of our code is dominated by the FFT library, we obtain an impressive parallel efficiency when performing two strong scaling tests with up to 256 MPI processes.

We remark that for large deformations where the displacement gradient operator is given by

$$\nabla u = \begin{pmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \\ \frac{\partial u_3}{\partial x_1} & \frac{\partial u_3}{\partial x_2} & \frac{\partial u_3}{\partial x_3} \end{pmatrix}, \tag{17}$$

the norm calculation in Fourier space actually becomes simpler because

$$\sum_{l,m=1}^{3} \left| \widehat{\nabla q_{l,m}}(\xi) \right|^2 = \left\| k^- \right\|^2 \|\widehat{q}\|^2 . \tag{18}$$

Therefore, the runtime overhead should be even smaller in this case.

Acknowledgements The authors thank Odelin Gentieu and Antoine Motte for performing the multiplication tests to calculate the core frequencies depending on the number of active cores and for collecting the data shown in Table 1. The authors also thank Heiko Andrä and Jonathan Köbler for stimulating discussions, Christian Peter for pointing out the article of [14], and the two anonymous reviewers whose comments and suggestions helped to improve and clarify this manuscript.

References

1. Kabel M, Böhlke T, Schneider M (2014) Efficient fixed point and Newton-Krylov solvers for FFT-based homogenization of elasticity at large deformations. Comput Mech 54(6):1497–1514
2. Zeman J, Vondřejc J, Novák J, Marek I (2010) Accelerating a FFT-based solver for numerical homogenization of periodic media by conjugate gradients. J Comput Phys 229(21):8065–8071
3. Moulinec H, Suquet P (1998) A numerical method for computing the overall response of nonlinear composites with complex microstructure. Comput Methods Appl Mech Eng 157(1–2):69–94
4. Eghtesad A, Barrett TJ, Germaschewski K et al (2018) OpenMP and MPI implementations of an elasto-viscoplastic fast Fourier transform-based micromechanical solver for fast crystal plasticity modeling. Adv Eng Softw 126:46–60
5. Gelebart L, Derouillat J (2016) Scalability – AMITEX 2.3 documentation. http://www.maisondelasimulation.fr/projects/amitex/html/scalability.html. Accessed 5 May 2019
6. Schneider M, Ospald F, Kabel M (2016) Computational homogenization of elasticity on a staggered grid. Int J Numer Methods Eng 105(9):693–720
7. Frigo M, Johnson SG (2017) FFTW manual. Technical report version 3.3.7. Massachusetts Institute of Technology

8. Willot F (2015) Fourier-based schemes for computing the mechanical response of composites with accurate local fields. Comptes Rendus Mécanique 343(3):232–245
9. Schneider M, Merkert D, Kabel M (2017) FFT-based homogenization for microstructures discretized by linear hexahedral elements. Int J Numer Methods Eng 109(10):1461–1489
10. Leuschner M, Fritzen F (2018) Fourier-accelerated nodal solvers (FANS) for homogenization problems. Comput Mech 62(3):359–392
11. Andrä H, Combaret N, Dvorkin J et al (2013) Digital rock physics benchmarks. Part I: Imaging and segmentation. Comput Geosci 50:25–32
12. Krzikalla F (2012) Digital rock physics benchmarks. https://github.com/cageo/Krzikalla-2012/blob/master/images/berea/segmented-vsg.raw.gz. Accessed on 5 March 2019
13. Voigt W (1928) Lehrbuch der Kristallphysik, 2nd edn. Teubner, Leipzig
14. Verner U, Mendelson A, Schuster A (2017) Extending Amdahl's law for multicores with turbo boost. IEEE Comput Archit Lett 16(1):30–33