Procedia Computer Science

Volume 51, 2015, Pages 413–422

ICCS 2015 International Conference On Computational Science

A Unified and Memory Efficient Framework for Simulating Mechanical Behavior of Carbon Nanotubes

Michael Burger[1], Christian Bischof[2], Christian Schröppel[3], and Jens Wackerfuß[4]

[1] TU Darmstadt, Graduate School of Computational Engineering, Darmstadt, Germany
burger@gsc.tu-darmstadt.de
[2] TU Darmstadt, Institute of Scientific Computing, Darmstadt, Germany
christian.bischof@tu-darmstadt.de
[3] University of Kassel, Institute of Structural Analysis, Kassel, Germany
schroeppel@uni-kassel.de
[4] University of Kassel, Institute of Structural Analysis, Kassel, Germany
wackerfuss@uni-kassel.de

Abstract
Carbon nanotubes possess many interesting properties, which make them a promising material for a variety of applications. In this paper, we present a unified framework for the simulation of the mechanical behavior of carbon nanotubes. It allows the creation, simulation and visualization of these structures, extending previous work by the research group "MISMO" at TU Darmstadt. In particular, we develop and integrate a new matrix-free iterative solving procedure, employing the conjugate gradient method, that drastically reduces the memory consumption in comparison to the existing approaches. The increase in operations for the memory saving approach is partially offset by a well scaling shared-memory parallelization. In addition the hotspots in the code have been vectorized. Altogether, the resulting simulation framework enables the simulation of complex carbon nanotubes on commodity multicore desktop computers.

Keywords: parallelization, vectorization, simulation, software engineering, carbon nanotubes, matrix-free solver

1 Introduction

Carbon Nanotubes (CNTs) possess several remarkable properties. They may be the basis of new methods for integrated circuit design and high-speed and power efficient logic applications with high density [8], because they can replace traditional CMOS based circuits. CNTs also exhibit useful mechanical characteristics for the design of novel materials [2]. In particular, they can resist high mechanical stresses. CNTs can be imagined as a rolled up sheet formed by hexagonal carbon rings. Figure 1 displays a tube that is rolled up around the dark red x-axis, with the center of the coordinate system in the middle.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.261

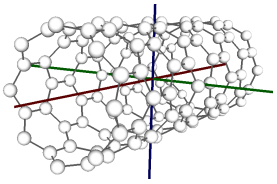

Figure 1: Carbon nanotube

The synthesis of Y-shaped junctions allows the composition of carbon nanotubes to structures of higher order called Super Carbon Nanotubes (SCNTs) ([2]). These formations are highly hierarchical, symmetric and self-similar. All calculations and principles that are described within this paper are demonstrated on CNTs, but are directly applicable to SCNTs which consist of a drastically higher amount of atoms and are very relevant for research.

Wackerfuß and Schröppel from the MISMO group[1] developed a graph algebra approach to model and represent SCNTs [9]. This procedure to create the SCNTs was implemented in Matlab and a description of topology and geometry is exported to files. These files are used as input for the MISMO-FEM code where the mechanical properties of SCNTs can be simulated. In 2003 Li et al. [4] introduced a method to compute elastic deformations of CNTs with structural mechanics and finite element simulations. This work was extended by Wan and Delale [12] by taking the bending stiffness of the graphite layer into account. Additionally Liu et al. [6] proposed the atomic-scale finite element method (AFEM) [6] which also uses the first and second order derivatives of the total potential energy within the CNTs. The concept of the AFEM has been generalized in [11], making it applicable for arbitrary atomic structures. Additionally, an alternative assembly concept has been proposed, allowing determinung the global vector and matrix in a line-by-line manner. This concept has been implemented in the MISMO-FEM code. During all of these approaches a large, sparse stiffness matrix is created. In the case of the MISMO-FEM code this matrix is used as input for the direct solver of the Pardiso library[2].

In this paper, we develop a new, unified software framework that allows the simulation of complex CNTs on commodity multicore desktop systems. While the reference solver requires several gigabytes of main memory on a reference configuration, our matrix-free scheme allocates twelve time less memory. We employ the conjugate gradients (cg) method [3] with preconditioning (PCG) for solving the equation system. The increase in calculations induced by the new procedure can be partially offset by an efficient parallelization of the code. That parallelization is very important for our version, because this part of the code is executed for every iteration of the new matrix-free solving process. Nonetheless the slowdown currently is a factor of about 10.

For comparison with our new approach of on-the-fly application of the stiffness matrix, we implemented a reference version of the solving process that precalculates and saves the matrix. It uses the same preconditioner as the matrix-free version and does a standard matrix vector multiply with compressed row storage (CRS) as underlying data-format. In the reference version the calculation of the potentials between the atoms is done only once.

In addition our software allows the interactive visualization of the SCNTs [1] where we again provide memory savings, because no export of structural data to the disk is needed anymore.

[1] http://www.mismo.tu-darmstadt.de/mismo/homemenu/welcome.de.jsp
[2] http://www.pardiso-project.org/

2 Simulating the mechanical behavior of CNTs

Our software simulates the mechanical behavior of the CNT structures. A CNT is constructed, its internal forces resulting from the interaction of adjacent atoms are determined, and, together with the external forces resulting from the specific load case, an system of linear equations is set up to compute the displacements of the atoms resulting in a static equilibrium.

In order to construct the CNT models, we apply a graph-based approach developed by Schröppel and Wackerfuß [10]. Given the topological type of CNT—in this case, the armchair variant (see [5] for details) it is possible to identify a CNT by two parameters, L_0 and D_0, associated with the length and the diameter of the CNT, respectively. Thus a CNT well be identified by the pair (D_0, L_0) within this paper. The graph-based approach allows to construct the CNT, including the adjacency list of the atoms, without the need for a computationally expensive neighborhood search algorithm.

This section briefly summarizes the AFEM method proposed in [11]. Please refer to this work for a complete account of the model assumptions and mathematical formulas.

2.1 Governing equations and linearization

A CNT is composed of a indexed family of n atoms with spatial positions $\mathbf{x}_i$. The current position $\mathbf{x}_i$ of a node results from its initial position $\mathbf{x}_i^0$ and its displacement $\Delta\mathbf{x}_i$, i.e. $\mathbf{x}_i = \mathbf{x}_i^0 + \Delta\mathbf{x}_i$. The internal energy U^{int} is given as the sum of the potentials of interactions of adjacent atoms, derived from existing generic force fields. Given a conservative external potential, the symmetric stiffness matrix $\mathbf{K}(\mathbf{x})$ is given as the Hessian matrix of the second derivatives of the internal energy w.r.t. the displacement of the atoms, while the residual vector $\mathbf{r}(\mathbf{x})$ is given by the negative of the first derivative of the total energy. Thus, $\mathbf{r}(\mathbf{x})$ is the difference of the external forces $\mathbf{f}(\mathbf{x})$ and the first derivative of the internal energy. In static equilibrium, a local minimum of the total energy is attained, i.e. $\frac{\partial U}{\partial \mathbf{x}}(\mathbf{x}) = \mathbf{0}$. In the Newton-Raphson algorithm, which is widely used to identify critical points of multidimensional functions, we obtain, with $\mathbf{K} := \mathbf{K}(\mathbf{x}^0)$, $\mathbf{r} := \mathbf{r}(\mathbf{x}^0)$, the Taylor expansion $\frac{\partial U}{\partial \mathbf{x}}(\mathbf{x}) = -\mathbf{r} + \mathbf{K}\Delta\mathbf{x} + o(\|\Delta\mathbf{x}\|)$. Thus, an estimate for $\Delta\mathbf{x}$ can be obtained from solving the linear system of equations

$$\mathbf{K}\Delta\mathbf{x} = \mathbf{r} \tag{1}$$

This estimate is also called the solution of the linearized minimization problem. In this paper, we are concerned with the calculation of the displacement $\Delta\mathbf{x}$, i.e. we perform a single iteration step of the Newton-Raphson algorithm.

As the internal energy is invariant with regard to rigid body motions, suitable boundary conditions, such as prescribed resulting displacements and forces, must be incorporated into (1) in order to obtain a unique estimate for the displacements. We note that all boundary conditions considered in this paper can be readily included at the element level and thus do not alter on the validity of the results of the following exposition with regard to the computation and assembly of the stiffness matrix.

2.2 Implementation in the context of the finite element method

In order to construct the global variables, i.e. the stiffness matrix $\mathbf{K}$ and the residuum vector $\mathbf{r}$, we employ the formalism of the finite element method. We compute element stiffness matrices $\mathbf{k}^{\text{e}}$ and element residual vectors $\mathbf{r}^{\text{e}}$, whose entries are then assembled into the global variables. The internal energy U^{int} is given as the sum of the internal energies $U^{\text{int,e}}$ of the elements.

Given that the internal energy is exclusively based on the bonded interaction forces modeled by the Dreiding potential, we ignore non-bonded interactions, such as van-der-Waals forces, as well as interactions involving atoms located more than three atomic bonds apart from each other. We denote the minimum number of bonds separating two atoms i and j, $i, j \in I$, by the distance $d(i,j)$. An atom j is said to belong to the *neighborhood* Ω_i^{e} of an atom i if $d(i,j) \leq 3$.

As a result, most of the entries $\mathbf{k}_{i,j} = \frac{\partial^2 U}{\partial \mathbf{x}_i \partial \mathbf{x}_j}$ of the stiffness matrix $\mathbf{K}$ vanish, and only the entries $\mathbf{k}_{i,j}$ for which $j \in \Omega_i^{\mathrm{e}}$ can assume non-zero values. We introduce a local index j' for the 22 atoms in the neighborhood of a *reference atom* i (see figure 2), which, in particular, gives rise to a function $\sigma : (j', i) \mapsto j$. Using local indexing and employing the *line-based* method introduced in [11], the element stiffness matrix is given as $\mathbf{k}_i^{\mathrm{e}} = \begin{bmatrix} \mathbf{k}_{i,1}^{\mathrm{e}} & \mathbf{k}_{i,2}^{\mathrm{e}} & \dots & \mathbf{k}_{i,22}^{\mathrm{e}} \end{bmatrix}$, with $\mathbf{k}_{i,j'}^{\mathrm{e}} = \frac{\partial^2 U_i^{\mathrm{int,e}}}{\partial \mathbf{x}_i \partial \mathbf{x}_{j'}}$, and the element residuum vector is given as $\mathbf{r}_i^{\mathrm{e}} = \mathbf{f}_i - \frac{\partial U_i^{\mathrm{int,e}}}{\partial \mathbf{x}_i}$. $\mathbf{K}$ is a $3n \times 3n$ matrix, $\mathbf{k}_i^{\mathrm{e}}$ is a $3 \times (22 \cdot 3)$ rectangular matrix, and $\mathbf{r}_i^{\mathrm{e}}$ is a 3-element vector.

As the function σ is not necessarily injective, i.e. more than one local index j' may be associated with a global index j, the entries in the global stiffness matrix are computed as $\mathbf{k}_{i,j} = \sum_{j' \in \{j' | \sigma(j',i)=j\}} \mathbf{k}_{i,j'}^{\mathrm{e}}$. All summands needed for a given component of the global stiffness matrix $\mathbf{k}_{i,j}$ are present in the same element stiffness matrix $\mathbf{k}_i^{\mathrm{e}}$. Therefore, the computation and assembly of the parts of the global stiffness matrix associated with the respective elements are mutually independent .

The components of the element stiffness matrix are computed as the sum of the second derivatives of the 2-, 3- and 4-atom interaction energies with regard to the displacement of the reference atom i and the displacements of the atoms j'_1, j'_2 … j'_p, $p \in \{2,3,4\}$, that participate in these interactions. In this paper, the interaction energies are given by the Dreiding potential, a generic force field introduced in [7]. Figure 3 shows the three interactions of the Dreiding potential and their respective kinematic values, i.e. the bond length r, the valence angle θ and the dihedral angle φ. For the potential functions, see Table I on p. 978 of [11].

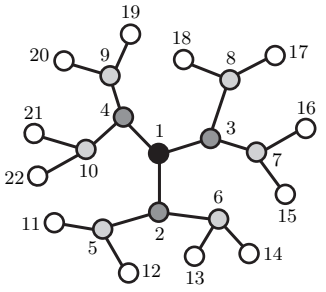

Figure 2: The neighborhood Ω^{e} of the reference atom (local index $j' = 1$) contains all the atoms present in the finite element (taken from [11]).

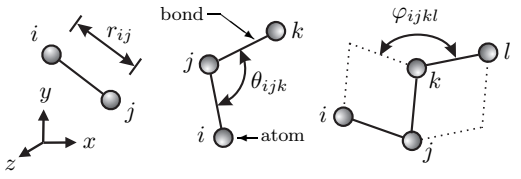

Figure 3: The three interactions of the Dreiding potential (taken from [11]). Note that, in this context, $i := j'_1$, $j := j'_2$, $k := j'_3$, $l := j'_4$.

3 Implementation

Our main goal was to avoid the construction of the whole stiffness matrix $\mathbf{K}$ in order to reduce the memory consumption of the software. We use the NAG Library[3] for our project because it provides all routines and interfaces for PCG and is parallelized for shared-memory.

The main routine that must be implemented for a PCG iterative solver returns the result of the multiplication of the stiffness matrix $\mathbf{K}$ by the residual vector $\mathbf{r}$. Due to the symmetry of the stiffness matrix, we need to consider only its upper triangular part and its diagonal. Unlike the stiffness matrix, the residual vector is stored in memory.

For the solving algorithm we run through all the nodes of the system and calculate on the fly all the 3×3 terms within ${\mathbf{k}_i}^e$ for refernce atom i. This 3×3 contributions are directly multiplied with the appropriate positions within the residual vector $\mathbf{r}$ and added to the displacement vector $\Delta\mathbf{x}$. In this way we avoid the creation of the stiffness matrix $\mathbf{K}$, because every contribution can be deleted after its product with $\mathbf{r}$ is integrated into $\Delta\mathbf{x}$.

At this point it is also checked whether one of the boundary conditions must be applied to the contribution. The contributions are only calculated if their result belongs to the upper triangular matrix or the diagonal of the matrix. If it belongs to the upper part of the matrix, the contribution is also used to calculate the result for the corresponding contribution in the lower triangular part. To assess the impact of symmetry, we also created a code version which does not exploit the symmetry within the stiffness matrix.

The second task that needs to be implemented is the creation of the preconditioning matrix $\mathbf{C}$. This matrix must on the one hand be fast to compute, but on the other hand should have similar properties as the stiffness matrix. We explore two different preconditioning matrices: The diagonal matrix and a block diagonal preconditioner that is based on the structure of the stiffness matrix. We take the 3×3 matrices around the diagonal of $\mathbf{K}$. Their diagonals make up the diagonal of $\mathbf{K}$. For each of these blocks $\mathbf{B}_i$ we do a Cholesky decomposition $\mathbf{B}_i = \mathbf{L}_i * {\mathbf{L}_i}^T$ and save $\mathbf{L}_i$ for each block. Each of this $\mathbf{L}_i$ is symmetric and positive definite. The preconditioning matrix $\mathbf{C}$ is calculated once and saved.

A third routine has to be passed to the NAG library that solves the preconditioning system $\mathbf{C} * \mathbf{r} = \Delta x$. To that end $\mathbf{C}$ needs to be inverted. For the diagonal matrix this problem is directly solved by inverting each diagonal element. For the block diagonal preconditioner, we employ the Cholesky solver *DPOSL* from the Linpack library for each equation system of a $\mathbf{L}_i$ with its three corresponding entries of the residual vector.

4 Storage and Compute Tradeoffs

To assess the memory savings possible with our approach, we estimate the memory necessary for the explicit calculation of $\mathbf{K}$ versus our matrix-free approach for a graph with n nodes (=atoms). Depending on the boundary conditions the actual number of non-zeros may slightly vary.

Every node's neighborhood contains at most 21 other atoms (see figure 2) and each atom-atom-contribution is a 3×3 matrix. This means that our resulting matrix has $3 * n$ rows and every row contains at most $3 * 22$ non-zero values. The whole matrix contains no more than 66*3*n non-zero values. With $N = 3 * n$, the amount of non-zeros in $\mathbf{K}$ is at most $66 * N$.

The next step is to take the symmetry of $\mathbf{K}$ into account. Only half of the values plus the diagonal need to be stored. Since the diagonal is of length N, we can estimate the memory requirements of $\mathbf{K}$ by:

[3] `http://www.nag.com`

$$size(K) = \underbrace{(66 * N)/2}_{\substack{half \\ values}} + \underbrace{N}_{\substack{diagonal- \\ values}} = 34 * N \tag{2}$$

Together with the stiffness matrix the values for the residual $\mathbf{r}$ and the displacement vector $\Delta\mathbf{x}$ must be kept in memory. Both vectors are of size N. Thus the reference solver needs to store at most $36 * N$ values.

In our approach, we need to store $\mathbf{r}$, $\Delta\mathbf{x}$ and the preconditioner matrix $\mathbf{C}$, which also has N entries in the case of diagonal preconditioning. Since the influence of each atom-atom-contribution is calculated one after another, each 3×3 contribution needs only to be stored temporarily. The amount of variables needed during the calculation of a contribution is constant and thus becomes negligible for large n. Thus our scheme needs to save $3 * N$ values:

$$\text{Storage demand} = \begin{cases} 36 * \text{N} & \text{for reference version} \\ 3 * \text{N} & \text{for matrix-free version} \end{cases} \tag{3}$$

As a result, our approach reduces memory requirements by a factor of about 12. So for example if we consider a $(1024, 1024)$ tube with 2 097 152 atoms and double precision floating-point values, the approach calculating $\mathbf{K}$ will consume about 1.69 GB of memory, while avoiding $\mathbf{K}$ results in 140 MB of data.

On the other hand, though, the computational load increases when calculating the matrix vector product. The matrix-free version needs to recalculate all potentials for each iteration step of the conjugate gradient method in order to execute the matrix vector multiplication. The number m of iterations that is executed increases with the size of the tubes. Figure 4 shows the correlation of the size of the tube and m, when solving the equation system to full precision.

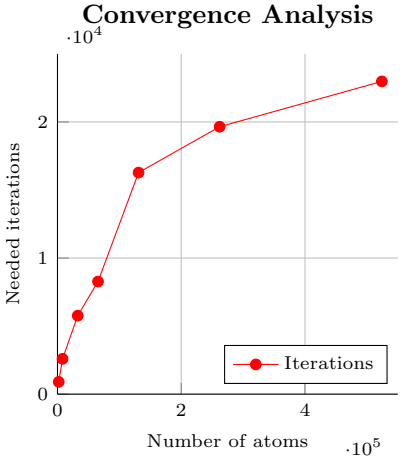

Figure 4: Correlation of the size of the CNT and the number of iterations needed to solve the system to full working precision.

The graph flattens with increasing model size, but it can be seen that the amount of calculations increases by three orders of magnitude for the matrix-free solver if a system is solved to

full precision.

5 Software Architecture

5.1 Separation of Functionality

The software separates the different components of the simulation, in order to be able to modify and develop different parts of the package independently from the others. The three main packages are: Graph modeling, calculation on the tubes and visualization.

The modeling can be further partitioned into the pure graph algebra and its operations, and the creation of the geometry information. The graph algebra operations that were already realized in Matlab (based on [9]) have been reimplemented in C++ from scratch to optimize performance, extensibility and maintainability. The data for the atoms is stored in std:maps, while their relations, i.e. the atom-bonds, are saved within bimap data-structures taken from the boost library[4]. This data-structure has the advantage that both sides of the table can be used as keys. This choice allows straight-forward implementation of the modeling part, with acceptable performance for our purpose.

However, the dominating part is the solving process with over 2000 seconds for a $(64, 64)$ tube and a single core execution. As the construction process only needs to be done once we view the bimap data-structure as a reasonable choice for now.

The software package contains a direct visualization feature based on the OpenGL interface. It is able to render tubes with more than 1 million atoms in real-time and allows free moving within the scene. The visualizer auto-adapts the visual quality to the capabilities of the underlying graphics hardware. It works directly on the std::map- and bimap-data-structures and avoids time- and storage expensive exports. Details can be found in [1].

5.2 Parallelization and Vectorization

Since we employ the procedure from [11], we avoid a priori the double counting of element contributions, although some elements overlap. This is due to the fact that a finite element considers only the influence of the neighbor atoms on the reference atom and not vice versa. Thus, calculating the potential values for an atom is independent from the computation of all others atoms. It is possible to do these calculations in parallel. The multiplication of a contribution with the residual is also independent from others. Hence, this step is also done in parallel. One needs to synchronize only at the point when the threads want to register their partial results into the displacement vector $\Delta\mathbf{x}$.

The routines for calculating the dihedral- and valence-angle terms and their first and second derivatives are the hotspots during the program execution. They consume over 85% of the total CPU time. Thus, the implementation has to make sure that these parts are vectorized.

6 Results

6.1 Test Environment

The measurements were taken on a dual socket desktop machine running Ubuntu 14.04. It is equipped with two Intel Xeon E5-2670 CPUs (Sandy-Bridge) with eight cores each. Each

[4] `http://www.boost.org`

core is equipped with 20 MB L3 Cache as well as a 256 bit AVX vectorization unit. Hyper threading is deactivated. The board has 128 gigabyte of shared DDR3 ECC RAM. The Intel C++ compiler in version 14 with optimization level Ox (i.e. full optimization) is employed. The type of vectorization is set to host-dependent, and the native OpenMP implementation from Intel is used.

6.2 Preconditioning

Several test cases have been calculated for the diagonal and the block diagonal preconditioner. Somewhat surprisingly, both preconditioners require about the same number of iterations in every case to reach the same accuracy. Since the computational load and memory demand for the block diagonal preconditioner is higher than for the diagonal preconditioner, it is not considered further.

6.3 Parallelization and Vectorization

Since the Intel C++ compiler is not able to auto-vectorize the hotspots in the code we restructured them. For the valence-angle terms we interchanged loops and integrated additional pragmas that force the compiler to vectorize the loops within the calculation. The code for the dihedral-angle and its first two derivatives has been partially replaced by AVX2-intrinsics. We measured the effect of vectorization for different tubes with our matrix-free version. The vectorized code needs 48% - 61% (increasing percentage with increasing model size) less overall run-time than the base-version, e.g. 55% for the $(128, 128)$ and 60% for the $(256, 256)$ tube. Please note, that the auto-vectorization for the rest of the code is turned on in both cases. This improvement only applies to the matrix-free version, because only in that case these potential energy terms must be recalculated for every iteration of the PCG solving process.

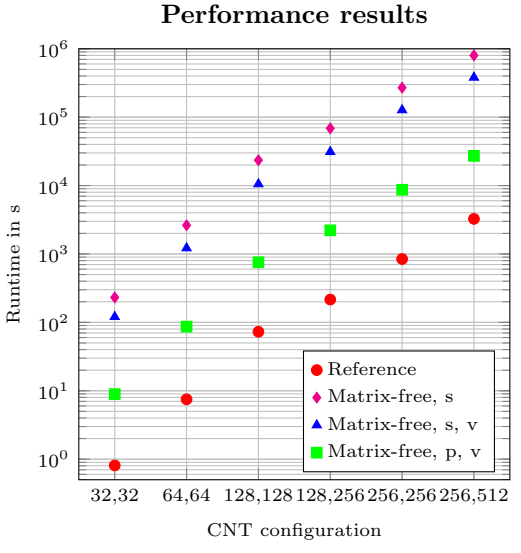

Figure 5: Runtime comparison of reference and the matrix-free version. (s=serial, p=parallel, v=vectorized)

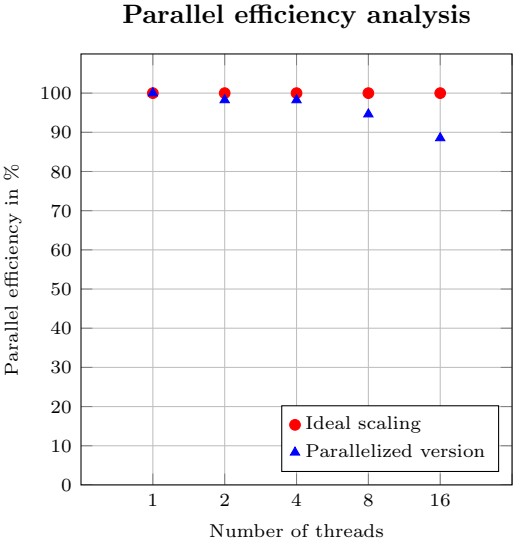

Figure 6: Efficiency analysis on the dual socket Xeon E5 system with 16 cores and disabled hyper-threading for a $(256, 512)$ tube.

Figure 5 shows the runtime of various implementations of the solving routine on the same test cases used for the preconditioner evaluation. The red dots show the reference version. It also uses PCG, but computes the matrix vector product using the explicit stiffness matrix. It runs with 1 thread in this case. The total time of the reference solver is composed by assembling the matrix directly into CRS format and executing the matrix-vector multiplication. The multiplication can be parallelized, but the reachable performance gain is limited by the inherent memory boundedness of CRS multiplications. Using 4 threads delivers the best speedup by a factor of 2.03 for the multiplication. Combined with the serial matrix assembly the reference solver running with 4 threads is 1.5 times faster than the serial version in figure 5. The magenta diamonds show the performance of the serial matrix-free version, without vectorization. Even though it requires, for the largest problem, about 20000 iterations, the actual slowdown is two orders of magnitude less. We suspect that this is due to the extremely compact memory footprint, which facilitates efficient cache reuse. The hand-vectorized version, shown in blue triangles, improves performance as described before. However, employing parallelization, the picture changes considerably. The green squares show runtime with 16 threads, and we observe that parallelization leads to a drop in execution time by an order of magnitude. Table 1 shows in detail the impact of parallelization.

CNT configuration	64 / 64	128 / 128	128 / 256	256 / 256	256 / 512
Reference / matrix-free serial	162.4	144.2	144.2	150.0	116.7
Reference / matrix-free parallel	11.5	10.4	10.3	10.3	8.33

Table 1: Ratio of the runtimes of the different solver-versions for five different CNT configurations.

The first data-row compares the reference solver with the serial, vectorized version of the matrix-free solver. The second row compares reference and parallelized, vectorized matrix-free solver, running with 16 threads. We see that in particular for larger configurations, the runtime penalty for the matrix-free version decreases, and the parallel version achieves a 12-fold decrease in memory at a factor of 8 in additional runtime cost.

To analyze the efficiency of our parallelization, we performed an efficiency analysis for the $(256, 512)$ tube on our dual socket system. Figure 6 shows the measured results. Despite the critical section, the implementation scales well. We employ the C++ STL data-structure *std::map* for storing and accessing the positions of the atoms. Although the maps are not optimized for parallel access, they apparently do not have a negative influence on the scaling behavior.

7 Conclusion and Outlook

We presented a unified framework for the simulation of carbon nanotubes. The graph algebra approach, underlying the CNT definition, allows a compact but unambiguous description of the structures. Our new matrix-free procedure considerably reduces the usage of the critical resource main memory by a factor of 12. The runtime overhead was reduced by efficient parallelization and vectorization, so that for the $(256, 512)$ tube, our approach only takes 8.3 times as long as the reference solver.

As current hardware trends see a further increase of compute versus memory capabilities, we anticipate that this tradeoff of memory savings vs compute time increase will continue

to develop in our favor. We will extend our investigation to SCNTs and especially those of higher order, which currently can't be simulated because of limited memory. In addition, we are working on a caching mechanism that should considerably reduce the recomputation cost while retaining full control over memory consumption, even for very large configurations. We also plan to integrate our solver in the MISMO-FEM code. There, the iterative nature of our approach will allow to vary the solution accuracy during the Newton iterations, further reducing the number of matrix vector multiplies needed.

Acknowledgments We thank J. Marczona for his work on optimizing the Mismo-FEM code and M. Bücker for the helpful discussions about iterative solvers and preconditioning. The work of M. Burger is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt. The work of C. Schröppel and J. Wackerfuß is financially supported by the Deutsche Forschungsgemeinschaft (DFG) via the Emmy Noether Program under Grant No. Wa 2514/3-1. This support is gratefully acknowledged.

References

[1] Michael Burger and Christian Bischof. Using instancing to efficiently render carbon nanotubes. In Miriam Mehl, editor, *Proc. 3rd International Workshop on Computational Engineering*, volume 3, pages 206–210, Stuttgart, Germany, October 2014.

[2] Vitor R. Coluci, Douglas S. Galvão, and Ado Jorio. Geometric and electronic structure of carbon nanotube networks: 'super'-carbon nanotubes. *Nanotechnology*, 17(3):617, 2006.

[3] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, December 1952.

[4] Chunyu Li and Tsu-Wei Chou. A structural mechanics approach for the analysis of carbon nanotubes. *International Journal of Solids and Structures*, 40(10):2487 – 2499, 2003.

[5] Sergey V. Lisenkov, Inna V. Ponomareva, and Leonid A. Chernozatonskii. Basic configuration of a single-wall carbon nanotube y junction of d3h symmetry: Structure and classification. *Physics of the Solid State*, 45(8):1577–1582, 2014.

[6] B. Liu, H. Jiang, Y. Huang, S. Qu, M. F. Yu, and K. C. Hwang. Atomic-scale finite element method in multiscale computation with applications to carbon nanotubes. *Physical Review B*, 72(3):035435+, July 2005.

[7] Stephen L. Mayo, Barry D. Olafson, and William A. Goddard. DREIDING: A generic force field for molecular simulations. *Journal of Physical Chemistry*, 94:8897–8909, 1990.

[8] Hongsik Park, Ali Afzali, Shu-Jen Han, George S. Tulevski, Aaron D. Franklin, Jerry Tersoff, James B. Hannon, and Wilfried Haensch. High-density integration of carbon nanotubes via chemical self-assembly, 2012.

[9] Christian Schröppel and Jens Wackerfuß. Algebraic graph theory and its applications for mesh generation. *PAMM*, 12(1):663–664, 2012.

[10] Christian Schröppel and Jens Wackerfuß. Constructing meshes based on hierarchically symmetric graphs. *Submitted for publication*, 2015.

[11] Jens Wackerfuß. Molecular mechanics in the context of the finite element method. *International Journal for Numerical Methods in Engineering*, 77(7):969–997, 2009.

[12] H. Wan and Feridun Delale. A structural mechanics approach for predicting the mechanical properties of carbon nanotubes. *Meccanica*, 45(1):43–51, 2010.