Memory-Efficient and Parallel Simulation of Super Carbon Nanotubes

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation zur Erlangung des akademischen Grades Doktor rerum naturalium (Dr. rer. nat.)
von Michael Burger, M.Sc. aus Aschaffenburg
Tag der Einreichung: 15. März 2017, Tag der Prüfung: 5. Mai 2017
Darmstadt 2017 — D 17

1. Gutachten: Prof. Dr. Christian Bischof
2. Gutachten: Prof. Dr. Martin Bücker
3. Gutachten: Prof. Dr. Jens Wackerfuß

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Informatik
Scientific Computing

Memory-Efficient and Parallel Simulation of Super Carbon Nanotubes

von Michael Burger, M.Sc. aus Aschaffenburg

1. Gutachten: Prof. Dr. Christian Bischof
2. Gutachten: Prof. Dr. Martin Bücker
3. Gutachten: Prof. Dr. Jens Wackerfuß

Tag der Einreichung: 15. März 2017
Tag der Prüfung: 5. Mai 2017

Darmstadt — D 17

Erklärung zur

Hiermit versichere ich, die vorliegende ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 15. Mai 2017

(M. Burger)

Contents

Vorwort vii

Acknowledgments ix

Zusammenfassung xi

Abstract xiii

1. Introduction and Motivation 1

2. Background 7

2.1. Important properties of super carbon nanotubes 7

2.2. Simulation methods for super carbon nanotubes and single-walled carbon nanotubes 7

2.2.1. Molecular dynamics, molecular mechanics and the atomic-scale finite element method 8

2.2.2. Model sizes of the employed super carbon nanotubes 9

2.2.3. Exploiting symmetric structures 10

2.3. Research of the MISMO group 10

3. Graph Algebra Modeling of Super Carbon Nanotubes 13

3.1. Important graph algebra operations and their geometrical meaning 15

3.1.1. Basic operations 15

3.1.2. Combined operations 19

3.2. A general algorithm for constructing super carbon nanotubes of arbitrary order . . 20

3.2.1. Construction of high level junctions 22

3.2.2. Building tubes 30

3.2.3. Correlation of the construction process and the tuples 36

3.2.4. Terminology of super carbon nanotube models 37

3.3. Identifying symmetry and hierarchy 38

4. Visualizing Super Carbon Nanotubes and the Result of Simulations 45

4.1. Principles of instanced rendering 45

4.2. Features of the visualizer 46

4.3. Rendering performance 48

5. Atomic-Scale Finite Element Method 51
5.1. The flow of the algorithm 51
5.2. Governing equations and linearization 52
5.3. Implementation in the context of the finite element method 55

6. Graph Data Structures 61
6.1. Different approaches to map tuples 62
6.1.1. Tree-based flattening (TreeGraph class) 62
6.1.2. IndexGraphs 65
6.1.3. Perfect spatial hashing 68
6.1.4. Comparison of different graph types 76
6.2. Space-saving approach to store the edges 78

7. Compressed Symmetric Graphs 83
7.1. General graph compression schemes 83
7.2. Principles behind Compressed Symmetric Graphs 85
7.3. Realization of Compressed Symmetric Graphs 86
7.3.1. Nodes on the boundaries of the tube 89
7.3.2. Nodes near the zero line 90
7.3.3. Two implementations of the retrieval procedure 91
7.4. Implementation and optimizations 94
7.4.1. Construction 95
7.4.2. Avoiding the storage of tuples 95
7.5. Achieving high compression rates 98

8. Solving for Displacements with an Iterative Approach 101
8.1. Preconditioning of the equation system 105
8.2. Parallelized reference solver 107
8.3. On-the-fly calculation of intermediate results 109
8.3.1. Memory savings 109
8.3.2. Performance optimizations 111
8.4. Software-controlled caching of intermediate results 112
8.4.1. Combination with Compressed Symmetric Graphs 113
8.4.2. Parallel preparing the cached data 115
8.4.3. Cached sparse matrix-vector multiplication 116
8.5. The case of small deformations 119

9. Results and Evaluation **123**
9.1. Test setup . . . 124
9.1.1. Test environment . . . 124
9.1.2. Different load cases . . . 124
9.2. Mechanical simulation results . . . 125
9.3. Performance measurements and comparison . . . 126
9.3.1. On-the-fly solver versus reference solver . . . 127
9.3.2. Value-symmetric solver versus reference solver . . . 131
9.3.3. Caching solver versus reference solver . . . 133
9.3.4. Caching solver with IndexGraph and Compressed Symmetric Graphs . . . 135
9.3.5. Higher order tubes . . . 138

10. Conclusion and Outlook **143**
10.1. Summary . . . 143
10.2. Future Work . . . 144

Appendices **147**

A. Summary of all Tubes **149**

B. Some Notes on Terminology of Super Carbon Nanotubes **150**

C. Background on the Simulation of Super Carbon Nanotubes **152**
C.1. The work of Coluci et al. . . . 152
C.2. The work of Li et al. . . . 153
C.3. General super carbon nanotube simulations . . . 153
C.3.1. Dependence on chirality . . . 154
C.3.2. Modeling of Y-junctions and scaling laws . . . 155

D. Performance Comparison of Different Container Structures **157**

E. Heap Consumption of the new EdgeMap **160**

F. Performance Results **163**
F.1. On-the-fly versus reference solver . . . 163
F.2. Symmetry solver versus reference solver . . . 166
F.3. Caching solver versus reference solver . . . 167
F.4. Caching solver with CSGs and IndexGraphs . . . 168
F.5. Caching solver with CSGs and IndexGraphs and varying cache . . . 169

List of Figures 172

Bibliography 174

Academic Curriculum Vitae 184

Vorwort

Doch meine Burg ist der Herr, mein Gott ist der Fels meiner Zuflucht.

Psalm 94,22

Fünf Jahre sind eine lange Zeit und der Weg bis zum Abschluss der Promotion war, wie wohl bei so vielen, nicht immer leicht und wäre alleine im stillen Kämmerchen sicher ziemlich schnell erfolglos zu Ende gewesen. Daher möchte ich versuchen diejenigen aufzulisten, ohne die das Projekt „Dr. rer. nat.“ nie von Erfolg gekrönt worden wäre. Viele Personen haben ihren Beitrag dazu geleistet.

Vielen Dank euch Mama und Papa für all die Unterstützung in den letzten 32 und besonders in den vergangenen fünf Jahren. Ohne euch wäre das niemals möglich gewesen. Jeder kann sich glücklich schätzen, wenn er Eltern wie auch hat. Danke natürlich auch an meine Geschwisterchen Thomas und Nicole, sowie meine ganze Verwandtschaft für das gezeigte Interesse an meiner Arbeit und eure aufmunternden Worte. Hier soll sich bitte auch die „Seuberts-Seite“ angesprochen fühlen, vor allem Angelika, Franz-Josef und Jakob.

Mein aufrichtigster Dank gilt meinem Doktorvater Christian Bischof für das Betreuen des Themas und es somit zu ermöglichen, dass diese Arbeit überhaupt erst geschrieben werden konnte. Danke für die hilfreichen Kommentare und Ideen im Verlauf des Projektes.

Auch muss ich bei so vielen netten Kollegen bedanken. Iris, die stets ein offenes Ohr für alle Problemchen und Schwierigkeiten hatte und bei Organisatorischem immer zu helfen wusste. Danke an Johannes, der am Anfang viel Zeit aufwenden musste um die ganzen Fragen eines Neulings erschöpfend zu beantworten und sowohl technisch als auch inhaltlich immer eine große Hilfe war. Danke an Christian für die vielen fachlichen und weniger fachlichen Gespräche, sowie die gute Zusammenarbeit in diversen Projekten und Aufgaben. Natürlich auch vielen Dank an Alexander und Jan-Patrick für die reibungslose Zusammenarbeit in der Lehre und bei Organisatorischem, den Kommentaren zur Dissertation und zum Vortrag, sowie den vielen unterhaltsamen Gesprächen und Diskussionen. Auch danke an euch Armin, Thomas, Andreas und Alexandra. Dir Nam sei für deine unerschöpfliche Mühe um die Codeoptimierung rund um SCNTs und PSH auch gedankt.

Crispin, dich möchte ich natürlich auch nicht vergessen. Vom ersten Tag der O-Phase an haben wir uns immer gut verstanden und aus einem Kommilitonen wurde bald ein Freund. Ich bin überzeugt davon, dass du mir bald auf den Pfad der Promovierten folgen wirst.

Was wäre ein Mensch ohne seine Freunde? Danke für all die vielen Jahre, die ihr es schon mit mir aushaltet. Stellvertretend möchte ich hier Sarah, Jan, Lisa, Nadine, Angelina, Max und Kevin nennen.

Dank sei auch meinen Kameraden von der Feuerwehr und von den Fingerhaklern ausgesprochen.

Sollte ich irgendjemanden vergessen habe (was ich sicherlich getan habe), so liegt dem natürlich keine Absicht zu Grunde. Nehmt es mir nicht übel! Die Aufgabe wirklich alle in diesem Text zu erwischen ist mindestens so komplex wie das Anfertigen der Dissertation selbst. Für die „fachlichen“ Danksagungen bitte die Acknowledgements beachten.

Zum Schluss natürlich ganz besonderen Dank an dich Simone, die du die letzten Jahre und vor allem Monate nicht immer die Aufmerksamkeit erhalten konntest, die du verdient gehabt hättest. Danke für dein Verständnis und deine Unterstützung. Danke für die schon bald fünf Jahre in denen ich dich an meiner Seite haben durfte und für die Zukunft auch hoffentlich immer haben darf. Ich liebe dich.

Michael Burger,
Frohnhofen, den 15. 05. 2017

Acknowledgments

I thank the MISMO group for our collaboration which enabled and motivated the work of this thesis. First of all the head of the group Jens Wackerfuß for all the fruitful discussions about the topic and that he took always time to answer my questions. I also thank Christian Schröppel for the explanations and discussions about the graph algebra and Jonas Marczona for the hours of technical assistance in using the dockSIM framework.

I thank Martin Bücker for the discussions about the preconditioning of the conjugate gradient method and the project in general, Gerhard Wellein for the discussions about data structures, and Michael Gösele for the introduction to the concept of perfect spatial hashing.

I thank Karsten Weihe, Andy Schürr and Michael Pradel for completing the committee for my defense. Thanks for your time and the challenging questions.

Finally, I thank all responsible people at the Graduate School of Computational Engineering for the opportunity of a full scholarship from 2013 until 2016.

Zusammenfassung

Kohlenstoffnanoröhren (engl. carbon nanotubes, kurz CTNs) haben seit ihrer Beschreibung in einem Nature-Artikel im Jahr 1991 viel Beachtung erfahren. Eine Kohlenstoffnanoröhren ist prinzipiell ein aufgerolltes Stück Graphen, welches man sich als ein zweidimensionales Gitter aus Kohlenstoffatomen vorstellen kann, in dem die Atome in einem Bienenwabenmuster angeordnet sind. Dieses Allotrop des Kohlenstoffs weist einige sehr interessante Eigenschaften auf, darunter eine sehr hohe Zugfestigkeit bei geringem Gewicht oder hohe Temperaturbeständigkeit. Diese Eigenschaften motivieren Forschungen in denen versucht wird, mittels CNTs neue Materialen zu erzeugen oder bestehende Materialien durch Integration von CNTs zu verstärken. Des Weiteren haben CNTs interessante elektronische Eigenschaften, da sie sich, abhängig von ihrem Syntheseprozess, bezüglich ihrer Leitfähigkeit wie Metalle oder wie Halbleiter verhalten können.

Die Synthese sich verzweigender Kohlenstoffnanoröhren ermöglicht es, diese als Verbindungsglieder für geradlinige CNTs einzusetzen und somit Netzwerke aus Röhren zu generieren. Eines dieser Netzwerke sind die im Jahr 2006 vorgestellten super carbon nanotubes (SCNTs). In diesen Strukturen wird jede Kohlenstoff-Kohlenstoff-Bindung im Atomgitter durch gleichförmige CNTs und jedes Atom mit einem Y-förmig verzweigten Röhrchen ersetzt. Diese Y-Verzweigung besitzt drei Arme mit gleicher Länge und Durchmesser, die zu ihrem Nachbararmen jeweils um 120° rotiert sind. Hieraus entsteht ein röhrenförmiges Gebilde, welches aus kleineren Röhrchen geformt wird. Es besteht die Möglichkeit diesen Konstruktionsprozess zu wiederholen. Die Kohlenstoff-Bindungen im Gitter werden hierbei nicht mit CNTs, sondern mit SCNTs ersetzt, was zu einer sehr regulären Struktur höherer Ordnung führt.

Mittels computergestützter Simulationen konnte gezeigt werden, dass SCNTs ebenso wie CNTs interessante mechanische Eigenschaften aufweisen. Sie sind noch flexibler als CNTs, was sie zu einer guten Grundlage für die Erforschung neuartiger besonders zugfester und dehnbarer Materialien macht. Andere Anwendungsgebiete sind auch mikroelektronische Schaltungen, da sich das elektronische Verhalten von SCNTs konfigurieren lässt, sowie, aufgrund ihrer Biokompatibilität, die Biologie und Medizin.

Trotz der Fortschritte bei den Syntheseverfahren für geradlinige und verzweigte CNTs liegt die Herstellung von SCNTs noch jenseits der technischen Möglichkeiten. Außerdem sind Experimente mit Nanostrukturen teuer, komplex und fehleranfällig. Deshalb sind Simulationen von SCNTs wichtig, um deren Eigenschaften vorauszusagen und Richtlinien für die experimentelle Forschung vorzugeben. Mit einer modifizierten Methode der Finiten Elemente für atomare Strukturen (engl. atomic-scale finite element method, kurz AFEM) existiert bereits ein fest etabliertes Verfahren zur Simulation von CNTs. Allerdings wächst die Modellgröße für größere

SCNTs rasch an, was dazu führt, dass der Speicher moderner Rechner für die während der Simulation erforderlichen Mehrkörper – und Gleichungssysteme nicht ausreicht. So lange die reguläre Struktur von SCNTs nicht benutzt wird, um den Speicherverbrauch der Simulationen zu reduzieren, ist die Simulation großer SCNTs auf atomarer Ebene nicht möglich. Die vorliegende Arbeit zeigt Wege auf, die Symmetrie und Hierarchie innerhalb der SCNT Strukturen auszunutzen, um somit die Simulation großer SCNTs auf atomarer Ebene zu ermöglichen. Es werden spezielle Datenstrukturen beschrieben, die es ermöglichen sehr große SCNTs mit einigen Milliarden Atomen zu speichern und effizienten Zugriff auf die Daten bieten. Dies wird in der neuartigen Graph-Datenstruktur der Compressed Symmetric Graphs umgesetzt, die dynamisch Teile der Strukturinformation der SCNTs wiederberechnen kann, anstatt sie abspeichern zu müssen.

Ebenso wird ein neuer, parallelisierter und Matrix-freier Löser für die auftretenden linearen Gleichungssysteme vorgestellt, der einen effizienten Mechanismus zum Zwischenspeichern von Matrix-Beiträgen einsetzt. Er ist doppelt so schnell wie ein Referenzlöser, der mit einer im compressed-row-storage-Format gespeicherten Matrix arbeitet und benötigt dabei nur halb so viel Speicher, wenn alle Beiträge zur Matrix zwischengespeichert werden. Es wird gezeigt, dass es die Kombination dieses Lösers mit den Compressed Symmetric Graphs ermöglicht, auf einem einzelnen Rechnenknoten Gleichungssysteme der Ordnung $5 * 10^7$ aufzustellen[1] und dabei weiterhin die vollständigen Matrixdaten im Speicher zu halten.

[1] Die entspricht einem Röhrchen mit $1.7 * 10^7$ Atomen. Das hier verwendete Modell ist also kleiner als die größten, die modelliert wurden.

Abstract

Carbon nanotubes (CNTs) received much attention since their description in Nature in 1991. In principle, a carbon nanotube is a rolled up sheet of graphene, which can be imagined as a honeycomb grid of carbon atoms. This allotrope of carbon has many interesting properties like high tensile strength at very low weight or its high temperature resistance. This motivates the application of CNTs in material science to create new carbon nanotube enforced materials. They also possess interesting electronic properties since CNTs show either metallic or semiconducting behavior, depending on their configuration.

The synthesis of branched carbon nanotubes allows the connection of straight CNTs to carbon nanotubes networks with branched tubes employed as junction elements. One of these networks are the so-called super carbon nanotubes (SCNTs) that were proposed in 2006. In that case, each carbon-carbon bond within the honeycomb grid is replaced by a CNT of equal size and each carbon atom by a Y-branched tube with three arms of equal length and a regular angle of 120° between the arms. This results in a structure that originates from tubes and regains the outer shape of a tube. It is also possible to repeat this process, replacing carbon-carbon bonds not with CNTs but with SCNTs, leading to very regular and self-similar structures of increasingly higher orders.

Simulations demonstrate that the SCNTs also exhibit very interesting mechanical properties. They are even more flexible than CNTs and thus are good candidates for high strength composites or actuators with very low weight. Other applications arise again in microelectronics because of their configurable electronic behavior and in biology due to the biocompatibility of SCNTs.

Despite progress in synthesizing processes for straight and branched CNTs, the production of SCNTs is still beyond current technological capabilities. In addition, real experiments at nanoscale are expensive and complex and hence, simulations are important to predict properties of SCNTs and to guide the experimental research. The atomic-scale finite element method (AFEM) already provides a well-established approach for simulations of CNTs at the atomic level. However, the model size of SCNTs grows very fast for larger tubes and the arising n-body and linear equation systems quickly exceed the memory capacity of available computer systems. This renders infeasible the simulation of large SCNTs on an atomic level, unless the regular structure of SCNTs can be taken into account to reduce the memory footprint.

This thesis presents ways to exploit the symmetry and hierarchy within SCNTs enabling the simulation of higher order SCNTs. We develop structure-tailored and memory-saving data structures which allow the storage of very large SCNTs models up to several billions of atoms while providing fast data access. We realize this with a novel graph data structure called Compressed

Symmetric Graphs which is able to dynamically recompute large parts of structural information for tubes instead of storing them.

We also present a new structure-aware and SMP-parallelized matrix-free solver for the linear equation systems involving the stiffness matrix, which employs an efficient caching mechanism for the data during the sparse matrix-vector multiplication. The matrix-free solver is twice as fast as a compressed row storage format-based reference solver, requiring only half the memory while caching all contributions of the matrix employed. We demonstrate that this solver, in combination with the Compressed Symmetric Graphs, is able to instantiate equation systems with matrices of an order higher than $5 * 10^7$ on a single compute node[1], while still fully caching all matrix data.

[1] This corresponds to a tube with $1.7 * 10^7$ atoms, so the models employed here were smaller than the largest models that were generated.

1 Introduction and Motivation

Carbon nanotubes (CNTs) have become popular in 1991, in particular, through an article by Iijima in Nature [1]. Although images of them had already been published in 1952 by Radushkevich and Lukyanovich (see [2, Figure 4]), this prior publication was not widely perceived by the scientific community since it was written in Russian. Hollow tubes of carbon were observed in 1976 and 1988 by Endo and Oberlin (see e.g. [3, Figure b] and [4, Figure 6a/7a]). Carbon nanotubes are, in principle, a graphene sheet that is rolled into a cylinder and can be distinguished in multi- and single-walled tubes, depending on the thickness of the employed graphene, i.e., if it has more than one or only one layer. This work focuses on single-walled carbon nanotubes (SWCNT) that were described by Iijima [5] in 1993.

Carbon nanotubes have drawn much attention because of their outstanding mechanical properties. They can resist high mechanical and thermal stresses with a tensile strength which can be up to 100 times higher than aluminum. Combined with their very high strength-to-weight ratio, they allow the construction of new composites with extraordinary strength ([6], [7]). They are also capable of transferring heat even better than diamonds [8]. Additionally, they possess an either conducting or semiconducting behavior [9] which makes them an interesting subject for microelectronic circuit design [10].

Only two years after the description of CNTs by Iijima, Chico et al. [11] reported the existence of nanotube heterojunctions. Here, so-called *defects*, i.e., non-hexagonal connections, are integrated in the graphene, interrupting the regularity of the hexagonal grid and forcing the tube to branch. Chico et al. introduced pentagon-heptagon pair defects into the SWCNT to investigate the changes in electronic behavior of the altered structures. Other groups extended those ideas by integrating different kinds of defects, like octagons or pure pentagons, into graphene walls motivated by advancements in the synthesis process ([12], [13], [14]). This resulted in Y-branching tubes whose growth became more configurable and controllable to improve their regularity and enabled the creation of structured CNT networks.

Biro [15] reported Y-branched SWCNTs with branches oriented at 120° in a growing process by fullerene decomposition on graphite (highly ordered pyrolytic graphic substrate) in 2002. In contrast to growth on templates, the decomposition process leads to an angle of about 120° between each junction arm and not to varying angles. This allows very regular CNT networks because of the equal angle between each connection point of the junction and its rotational symmetry. The Y-junctions include several heptagon defects and can also include pentagon defects.

Scuseria [16] had already theoretically proposed those regular Y-junctions consisting of three intersecting tubes at 120° in 1992 calling this structure *schwartzon*. He described two different junctions: One is built by a total of 114 carbon atoms including 6 heptagons. The other has 330 atoms containing 6 heptagons and 18 pentagons. Moreover, Scuseria already had the idea of connecting these junctions with straight tubes to form *hypermolecules* (structures that have again the outer shape of a molecule) and *hypergraphite* (a structure with graphene-like outer shape). Macroscopic structures may be achieved by repeating this process on different levels.

Motivated by those advancements, Coluci et al. [17] proposed the structure of 'super'-carbon nanotubes or super nanotubes as a new type of nanotube network in 2006. The nomenclature in the literature is not consistent. This work employes the term *super carbon nanotubes* (SCNTs) to identify these structures. They consist of SWCNTs that are connected by Y-junctions to sheets of higher order, which are called super graphene. The super graphenes are rolled to a seamless cylinder, thus forming »*a carbon nanotube made of carbon nanotubes* «[17, p. 617]. The tubes are heuristically constructed by exploiting the honeycomb symmetry: As sketched in Figure 1.1, in a graphene sheet each carbon-carbon-bond is replaced by a carbon nanotube (shown in the red box) and each carbon atom by a Y-junction (shown in the blue box). The bottom shows a SWCNT from two different perspectives: A side view and looking through it with some Y-junctions and SWCNTs being highlighted. Coluci et al. already indicated a hierarchical construction procedure, too. The carbon-carbon-bonds are not replaced by single-walled carbon nanotubes but by SCNTs to generate structures of higher order with SWCNTs representing the order 0. The tubes between junctions are called arm tubes.

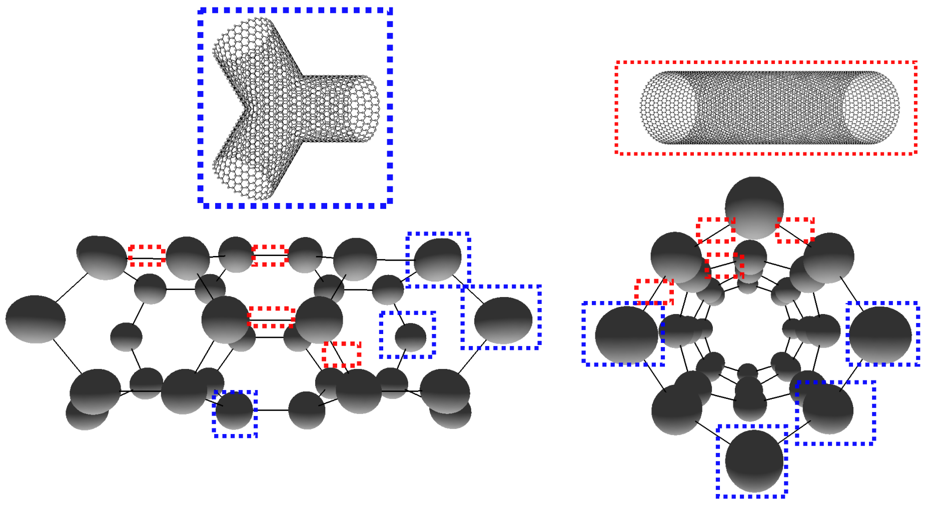


Figure 1.1.: Heuristical construction of SCNTs following Coluci et al.: Each C-atom is replaced by a Y-junction that itself consists of carbon and each C-C-bond is replaced by a SWCNT.

Based on the estimated properties of SCNTs, Coluci et al. proposed mechanics as a field of application. SCNTs may act as high strength composites or actuators with very low weight (see also [18], [19]). Furthermore, Coluci also imagined the application in biological context as catalyst for the synthesis of large biomolecules or as cavities and reservoirs. This is realistic because of their large pores and biocompatibility.

Despite the progress in synthesizing pure, defect-free SWCNTs[1] with predefined parameters and more regular Y-junctions, the production of SCNTs is still beyond technological capabilities [20]. Furthermore, real experiments at the nanoscale are expensive and complex. Hence, simulations of SCNTs are important to predict properties and to guide experimental research.

Due to the recursive construction process, the amount of data to describe an SCNT increases very fast. When increasing the order of the tube, the results of the previous order are reused as basic building blocks. To get a first impression of the problem, Table 1.1 shows the development of the SCNT size for the modeling procedure employed within this thesis. Although the starting point is a very small tube of only 64 atoms, corresponding to a graphene sheet with 21 hexagons, the resulting tube of order 4 already consists of more than a trillion of atoms. This is due to the fact that the number of atoms grows by about three orders of magnitude when increasing the order of the tube by one.

Table 1.1.: The top row shows the order of the considered tube. The second row gives the number of atoms in the corresponding tube.

Order of tube	**0**	**1**	**2**	**3**	**4**
No. of atoms	$6.4 * 10^1$	$2.8 * 10^4$	$1.2 * 10^7$	$5.2 * 10^9$	$2.2 * 10^{12}$

State-of-the-art simulations on the atomic scale within the framework of molecular dynamics (MD) or molecular mechanics (MM) require the solution of large n-body systems. Those generic methods ignore the structure of the underlying objects and require to generate the n-body systems atom-by-atom to simulate on the atomic level, i.e., we are interested in the movement of single atoms within the model. For higher order SCNTs with a large number of atoms this quickly exceeds the memory capacity of computer systems when employing the state-of-the-art methods, rendering such simulations infeasible.

However, resulting from their regular construction process, SCNTs possess a high amount of inherent self-similarity, hierarchy and symmetry. New methods must be developed to enable those simulations with available computer systems which exploit the inherent regularity of SCNTs for structure-aware data structures and improved solution algorithms. The work described in this thesis exploits those properties during construction, storage, and simulation, to drastically reduce memory-demand and greatly extend the range of feasible problem sizes on single compute nodes.

This thesis presents a novel framework for modeling, simulation, and visualization of higher order SCNTs on multicore systems and addresses the following questions:

1. How is it possible to identify the symmetry and hierarchy in SCNTs to avoid storage overhead without a time-consuming analysis of the whole structure?

[1] Defect-free SWCNTs as arm tubes are important for consistent structure and behavior.

2. How can models for large and higher order tubes be constructed and stored memory-efficient and fast?
3. Which ways exist to exploit the structural properties of SCNTs to enable a resource-saving simulation?
4. What is the best compromise between memory consumption and runtime performance in order to fully exploit the capabilities of the underlying computer system?
5. What can be done to efficiently parallelize the sparse algebra calculations during mechanical simulation?

This thesis develops new algorithms and implementations to reply these questions and is structured as follows: Chapter 2 presents the state-of-the-art methodology that was applied to simulate the mechanical behavior of SCNTs and the results that were achieved so far, with a focus on the work of the MISMO group at the University in Kassel, whose research inspired this thesis. The limitations of the state-of-the-art methods are highlighted and contrasted to the capabilities of the framework which was developed for this thesis.

Afterward, Chapter 3 describes the existing graph algebra based modeling approach for SCNTs developed by the MISMO group that employs tuples to identify nodes in the graphs. For the first time, the evolution of the tuples, the encoding process of structural information and the resulting geometry is portrayed in detail from a geometric perspective for the SCNT models of higher orders. It is explained precisely, how symmetry and hierarchy within SCNTs can be identified with the tuples, because these properties are exploited by the data structures and solvers in the thesis. The encoding of symmetry and topology within tuples is exploited for memory-efficient storage, simulation, and visualization of SCNTs. In our earlier work [21], the considerations were limited to SWCNTs and were less detailed due to space limitations.

In Chapter 4, we present a novel visualizer for graphs, based on the principles of instanced rendering. This work was published in [22]. It is directly targeted to render SCNT models and allows the inspection of large models even on slow hardware. The visualizer is capable of highlighting structural properties and represents a new application of the tuple labeling.

The following Chapter 5 summarizes the theory behind the existing algorithm to simulate the mechanical behavior of SCNTs which is the base for the simulations in the developed framework. As its contribution, the thesis highlights those properties that enable and motivate a matrix-free solution of the equation systems, like independence of calculations. In short form, the ideas were published in [23]. Moreover, this chapter investigates, explains, and compares the structure of the stiffness matrices for SCNTs up to order 3 as we have done in [24].

Chapter 6 covers the problem of indexing the tuple-based nodes in order to develop memory-efficient data structures for the SCNT graph while taking properties of SCNTs into account. A novel structure-tailored graph data type is outlined in detail, that delivers unique and compact indices for the tuples quickly and was sketched in [24]. Furthermore, the chapter demonstrates extensions to the principle of perfect spatial hashing (PSH), which were developed during the work of this thesis, and allow to apply PSH to index tuples as well as general multidimensional data. The implementation of the extensions and their effect are explained precisely as in [25]. Finally, both indexing schemes are compared with respect to several important properties.

The novel principle of Compressed Symmetric Graphs that was published in [26] is presented in Chapter 7. Exploiting the properties of SCNTs enables to dynamically recompute the structure of large parts of the tubes. This allows to avoid the storage of structural information for up to 99 % of the nodes in the SCNTs and greatly increases the range of tube sizes that can be kept in memory on a particular system. During simulation, Compressed Symmetric Graphs can serve as the underlying data structure and leave more memory for the actual calculations.

The following Chapter 8 mainly covers the new matrix-free solver that was developed during this thesis. The overhead for recomputing contributions to the stiffness matrix is compensated by an efficient caching scheme that we initially presented in [27]. Appropriate data structures for the required sparse matrix-vector multiplication and methods to efficiently parallelize the calculations are described that enable to outperform the reference implementation while only requiring half the memory. Another contribution of this chapter is the presentation of a special solver variant: it stores all required stiffness data for the sparse matrix-vector multiplication in only a few megabytes of memory by exploiting symmetric values in the matrix in the case of small deformations. The concepts behind this solver were outlined in [21].

In Chapter 9, the performance of the novel solver is compared in detail to the reference implementation, which stores the matrix in compressed row storage format, with different tubes and on two parallel computer systems. Advantages and disadvantages of different graph structures are highlighted and their effect on the total runtime of the simulation is investigated in different scenarios. For the first time, an equation system resulting from a tube of order 2 with $2 * 10^7$ atoms is instantiated, practically verifying the theoretical statements in this thesis.

Last, Chapter 10 offers conclusions and directions for future research.

2 Background

This chapter begins with a brief summary of the properties of SCNTs in Section 2.1. A detailed overview about the simulation of SCNTs can be found in Appendix C. Section 2.2 gives a short introduction to the methodology of simulating on the atomic level. Finally, the research of the MISMO group that inspired the work of this thesis is being outlined in Section 2.3.

2.1 Important properties of super carbon nanotubes

Super carbon nanotubes possess many interesting properties that were highlighted by several studies. They differ in several mechanical properties from the SWCNTs which is mainly caused by the integrated Y-junction elements. Under tension, first the angles between junction arms begin to change before the arm tubes are actually stretched [28]. This results in a high flexibility of SCNT structures [29] which allows to stretch and compress them further than SWCNT [30]. Additionally, the behavior of the junctions causes a non-linear deformation process [31], i.e., the forces that need to be applied to stretch the SCNT vary in dependence of the force already applied and rupture processes normally occur near the junctions [20]. In general, the mechanical properties seem to have a weaker dependence on the type of the SCNT than in the case of SWCNTs.

Like for SWCNTs the electronic properties depend on their construction process, i.e., which types of SWCNTs are employed as arm tubes and how the super sheet is rolled up to a tube, resulting in semiconducting or metallic behavior [20].

Finally, Coluci [20] states that the principle of SCNTs is not limited to carbon nanotubes but is also imaginable for other tube structures like boron nitride [32] or other connection types than Y-junctions like X- or T-junctions [33]. The principles presented in this thesis may be adapted to those novel structures.

2.2 Simulation methods for super carbon nanotubes and single-walled carbon nanotubes

In the following, three approaches of atomistic simulations are summarized with a focus on the atomic-scale finite element method as it is employed within this thesis (Section 2.2.1). Section 2.2.2 analyzes the model sizes investigated so far, while Section 2.2.3 presents approaches to exploit symmetry in simulated structures.

2.2.1 Molecular dynamics, molecular mechanics and the atomic-scale finite element method

Usually two methods are employed to simulate structures at the nanometer scale ($1 - 100\,\text{nm}$), namely molecular dynamics (MD) and molecular mechanics (MM). Both model the interactions between atoms and molecules while not taking into account any quantum mechanics [34]. Simulations on the atomic level, based on MD or MM, allow to predict the behavior of single atoms which is not possible for constitutive models [35]. In principle, both methods treat the atoms as particles in an n-body system and calculate the resulting movements of the atoms by empirically determined or theoretically formulated force fields, also called potentials [36]. The Brenner [37] and the Dreiding [38] potential are examples which have already been applied to SWCNT simulation. They all have in common that atoms only interact with atoms in their local neighborhood and not globally with all other atoms. Additionally, it is possible to employ the classical Newtonian mechanics and the bonds between atoms are modeled as spring elements since, in comparison to molecular dynamics, thermal and dynamic effects are also neglected in molecular mechanics. This, for example, enables us to formulate and solve the minimization problem for the global energy and to determine the equilibrium state of a system.

These facts motivated research to adapt the formalism of the finite element method (FEM) to the simulation of atomic structures. In contrast to classical finite element methods in continuum mechanics or fluid dynamics, there is no need to discretize the model into finite elements. Each atom and its relevant neighborhood, determined by the employed potential, serve as one finite element [34]. This approach is characterized as the atomic-scale finite element method (AFEM) and was presented by Liu et al. [35]. AFEM-based simulations have the advantage that they have an asymptotic complexity of $O(N)$ with N representing the number of atoms and not $O(N^2)$ like other classical methods do that are employed in MD and MM like the conjugate gradient method [35]. Furthermore, the AFEM can also be extended by the combination with other elements to take more interactions into account like, e.g., van der Waals forces [35]. It is also possible to couple AFEM with traditional continuum FE methods which allows, for example, to divide a large model into parts and to treat only critical parts with atomistic methods, while a coarser grid is applied for the non-critical parts. For example, extended and modified AFEM methods for single layer graphene sheets are presented in [39].

Atomic-scale FE methods have already been successfully applied to carbon nanotube simulation. Liu et al. [35] took five SWCNTs with 400, 800, 1600, 3200 and 48,200 atoms, fixed them at both ends and simulated a lateral force to their middle. By this example, they also empirically demonstrated the linear dependence of runtime and problem size. Additionally, they successfully investigated the bifurcation of a 6 nm-long SWCNT (resulting in several hundred atoms) when it is compressed from both ends, agreeing with previous results (see also [35]). A test for axial compression was also performed by Wackerfuß [34]. In that case an SWCNT with 320 atoms was employed and the simulation was performed with three different potentials. The comparison to the reference in the literature reveals that the results of the AFEM simulation are in very good agreement. Furthermore, Wackerfuß simulated the behavior of a tube with 800 atoms when it is twisted with two different tube versions: First, a defect-free SWCNT, and a second one containing one vacancy defect (=one missing atom) in the middle. In both cases, the AFEM leads to a stable simulation.

2.2.2 Model sizes of the employed super carbon nanotubes

In the literature, nothing is said about the required runtime or the resource allocation for the SCNT simulations. Consequently, a comparison of these existing methods in regard to their computational resources and their performance with the framework presented in this thesis is not possible.

However, some information is given for the model sizes employed. For [17] the unit cell size in the simulations ranged from 300 to 4500 atoms, while [29] investigated only a part of an SCNT containing $21,960$ atoms with molecular dynamics simulations under uniaxial tension. Wang et al. [28] even state that atomic methods cannot be used for SCNTs due to the number of arm tubes. Hence, they replace the SWCNT arm tubes by a thin shell model and employ perfect Y-junctions that totally consist of hexagons with a mesh of a few hundred triangles. Up to nine of those junctions are arranged side by side within their super graphene model. They assume that super sheets with more than nine junctions in a horizontal line would behave in the same way, since seven and nine junctions side by side already behave very similar.

A maximum of $50,000$ atoms in a model is reported for the two already covered AFEM methods in [34] and [35] . For the models in [31] a size smaller than $50,000$ atoms can be deduced from the specification of the employed SWCNT arm tubes and the fact that Scuseria junctions [16] were employed to connect those.

In contrast, in this thesis, SCNTs with up to $2.0 * 10^7$ atoms are considered for the AFEM calculations demonstrating that the methods presented in this thesis greatly increase the range of feasible model sizes.

The studies in the literature also contain only direct simulations of order 1 SCNTs, while properties for higher orders are only accessed indirectly by scaling laws [40] which exploit the fractal nature of SCNTs ([41], [42]).

The results of this thesis demonstrate that the matrix-free solver presented is prepared to execute simulations with higher order tubes which are modeled on an atomic base and which are much more complex because of the additional levels of hierarchy compared to order 1 tubes.

2.2.3 Exploiting symmetric structures

Approaches already exist that try to reduce the amount of data during calculations with matrices by exploiting symmetries in structures during the mechanical simulations. Kangwai et al. [43], for example, employed mathematical group representation theory to block-diagonalize the stiffness matrix and consequently partition it into several smaller problems. The main idea is that symmetries in structure will correlate to symmetries in the matrix like symmetric lines or blocks. The authors define symmetric structures as *»left unaltered, geometrically and mechanically, after a symmetry operation [rotations, reflections] «*[43, p. 672]. Applying these symmetry operations to parts of the modeled structure moves those parts to new spatial positions. To that end, an appropriate coordinate system based on the symmetry of the structure, needs to be found. Symmetry operations are grouped to symmetry groups and represented as matrices that can be applied at low computational costs. As a result, it is no longer necessary to fully store the stiffness matrix, reducing at the same time the complexity of the problem and the required storage as well as the computational effort.

Koohestani [44] presented a modified form of the procedure of Kangwai et al. and applied it to several use cases, e.g., to the buckling of a steel pipe. Here, the stiffness matrix was decomposed to four blocks, resulting in a reduction of the time to solve the system by a factor of 3.5.

In contrast to these two approaches, this thesis will present methods to exploit symmetry in structures without analyzing or even assembling the stiffness matrix. These methods allow to directly reveal symmetry information of the SCNT within the graph algebra based modeling approach.

2.3 Research of the MISMO group

The research-group MISMO[1] has already performed several mechanical simulations of SCNTs of order 1 based on the AFEM presented in [34]. To that end, they modeled the SCNTs within their Hierarchical Graph Meshing (HGM) method [45] which combines a self-defined graph algebra with an iterative construction process for SCNTs. The high amount of symmetry and hierarchy can be captured in the framework of the HGM method. This is also the modeling approach that is employed for this thesis and its important aspects are outlined in Chapter 3. MISMO has realized the HGM method as a Matlab script that is able to create data files containing the tube models. Additionally, the authors of [45] demonstrated that the runtime of the construction process in the HGM method depends linearly on the size of the structure, better than other mesh generation schemes. This thesis will also show that:

[1] Mechanische Instabilitäten in Selbstähnlichen Molekularen Strukturen höherer Ordnung, `https://www.uni-kassel.de/fb14bau/institute/ibsd/baustatik/startseite.html`

1. A well-designed C++ implementation of these principles can construct large tubes very fast.
2. The construction of higher order tubes is possible in combination with an appropriate data structure unlike with the native Matlab version.

Additionally, the research group MISMO developed the dockSIM framework[2], a C/C++ software solution for numerical calculations [46], which is employed to process the SCNT model files and to perform the actual simulations. Important concepts for the dockSIM framework were developed in a Bachelor's thesis [47] in collaboration of the MISMO group and the Scientific Computing group at TU Darmstadt. Furthermore, the performance of some parts of the program was improved in a student research project [48] within the same collaboration.

For this thesis, dockSIM was mainly used to generate reference results to ensure the correctness of the calculations performed by the newly developed algorithm. Moreover, dockSIM was employed for the relaxation of the input data, where the internal energy of the SCNTs is minimized. This step is required because the SCNT models resulting from the construction process within the HGM method are not in equilibrium state. In general, dockSIM is able to perform the simulation of mechanical behavior of SCNTs but the feasible model size is limited by the generic data structures and the direct solving approach employed for solving the linear equation systems.

One main goal of dockSIM is to allow the integration of different numerical methods in a consistent form by providing well-defined interfaces that allow a hierarchical composition of the actual numerical method, the employed discretization method and the constitutive equations that mathematically describe the behavior of a material. The numerical method can additionally be coupled with several external libraries, in particular, the required solver for the linear equations. This allows the user to focus on aspects of interests for him, e.g., the development of constitutive equations for new materials, while being able to reuse the existing functionality in dockSIM like reading input data or managing boundary conditions. The dockSIM framework already contains the Newton-Raphson method which also involves the required finite elements and constitutive equations to simulate super carbon nanotubes. Also other included numerical methods an implementations of different potentials may be employed to simulate the mechanical behavior of SCNTs. However, the underlying generic data structures that are not aware of the special properties of SCNTs and the PARDISO library[3] employed to directly solve the linear equation systems limit the range of feasible SCNT model sizes.

This thesis presents solutions to overcome these issues and enable the simulation of larger and higher order SCNTs.

[2] `http://docksim.de/`
[3] `http://www.pardiso-project.org/`

3 Graph Algebra Modeling of Super Carbon Nanotubes

This chapter summarizes the graph algebra construction approach employed for the SCNT models. We focus on those properties and concepts which are relevant for the graph data structures and solvers presented in later chapters.

The representation of scientific models as graphs has a very long history and was successfully applied in various problem domains [49]. The SCNT models in this thesis are also described as graphs. The modeling approach and the graph algebra construction process that lead to the SCNT models were published in detail in [45] with a focus on graph theoretic aspects. A short summary of the construction of SCNTs of order 0 with a focus on the geometrical meaning of the graph algebra operations has already been presented in [21]. Some definitions and explanations from [21] are partly reused.

The following chapter extents those considerations and presents the construction process for SCNTs of arbitrary order, for the first time with a focus on the geometrical meaning of the operations and correlates the development of the tuples during the construction with the geometric and structural information that they encode in detail. This highlights the elegance of the tuple system and the properties of the operations that can be exploited for an efficient implementation. The chosen approach may sometimes result in an intuitive but also informal description. For theoretic background and details we refer to [45]. The basic definitions for the graph theory employed are taken from [50].

The SCNT models are represented as directed graphs $G = (V, E)$. Each carbon atom corresponds to a node v (resulting from the synonym *vertex* for node) in the node set V while a set of directed edges E represents the covalent C-C-bindings. Each edge e can be written as an ordered pair of two nodes $e = (v_s, v_t)$ with v_s being the start node and v_t being the terminal node of the edge. An edge e is said to be *incident* to its start and end node. A node with no incident edge is called to be *isolated*.

During the construction, there exist also several *undirected graphs*, i.e., a graph where edges do not have a certain direction, but only have two incident vertices. In particular, the final graphs representing the tubes are undirected graphs while several directed graphs occur in some steps of the construction process. For the purposes of the graph operations, undirected graphs are modeled in the following way: For each edge $e = (v_s, v_t) \in E$ an undirected graph also contains the edge with opposite direction $e' = (v_t, v_s)$.

Two unary functions are defined which help to process the edges. Function σ returns the start node of an edge e, while function τ delivers the terminal node of the corresponding edge. Consequently, an edge e can also be given in the form $e = (\sigma(e), \tau(e))$. The functions σ and τ can also be applied to whole edge sets E as well with $\sigma(E)$ returning a set of all nodes that are the start node of an edge and $\tau(E)$ returning all nodes that are the end node of an edge, respectively.

In contrast to nearly all other graphs that can be found in the literature, the nodes within the graphs covered in this thesis are not labeled by letters or numbers, but unique m-tuples are employed to identify the nodes. Those graphs are called *tuple-based graphs* in the following. The m-tuples t are ordered sets of integers: $t = (x_m, x_{m-1}, ..., x_2, x_1)$ with m being the length of the tuple and all tuples within a graph possessing the same length. A tuple-based graph consisting of nodes with tuple length m is also defined as a graph of *dimension* m. The entry x_m is called the *leading* and x_{m-1} the *second leading* tuple entry while x_1 is the lowest entry. The distinct entries in a tuple can be accessed in a C-like fashion via the square brackets operator $t[j]$ with $j \in [1, \dots, m-1, m]$ and $t[1]$ delivering the lowest entry x_1.

Assume a set S of n tuples t_i with $i \in [1, 2, \dots, n]$, then the so-called *tuple extent* is defined as the tuple $t^{\text{ext}} = \big(\max(t_i[m]), \max(t_i[m-1]), \dots, \max(t_i[1])\big)$, that is, each entry of t^{ext} is set to the highest occurring value at the respective entry in all tuples of the set. The tuple extent determines the set of all possible tuples that is called the *tuple space*. Its size, i.e., the number of possible tuples, can be calculated by $\prod_{k=1}^{m} t^{\text{ext}}[k]$. Each tuple that is really appearing in a given graph is called *existing*. The only important thing to know about the nodes for the graph algebra is the tuple which identifies them, thus, each v can be denoted as its respective tuple t. Consequently, for simplicity the node set V is identical to the set S of existing tuples.

Another important term is the *sub-tuple* which is defined as $(x_v, x_{v-1}, \dots, x_u)$ of t with $1 \leq u < v \leq m$, i.e., $v - u$ consecutive entries of t are taken to create a new tuple, while the other entries of t are neglected. The sub-tuple function ζ_u^v is defined as $\zeta_u^v(t_i) = (t_i[v], t_i[v-1], \dots, t_i[u])$. ζ_u^v can also be applied to a whole set of tuples $\zeta_u^v(S)$ resulting in a set of sub-tuples of all tuples in S with duplicates removed.

Additionally, a lexicographic order on the tuples is defined by the algorithm in pseudocode in Listing 3.1. This order is also employed to assign a unique index to each tuple, by sorting all tuples with the function and indexing the tuples starting with 1 at the lowest one and going on consecutively.

```
tuple findGreater(tuple t1, tuple t2) {
  for(i=m; i > 0; i=i-1) {
    if(t1[i] > t2[i]) return t1;
    if(t1[i] <= t2[i]) return t2;
    }
}
```

Listing 3.1: The algorithm returns the higher tuple.

Before describing the graph algebraic operations that are required for the construction of the SCNT models, three types of base graphs need to be defined:

- Path Graph [P_l]: Create a chain of nodes with length l in which each node is connected to its successor by a directed edge.
- Cyclic Graph [Cy_l]: Create a path graph of length l and connect additionally the end node of the chain to the start node with a directed edge.
- Self-connected Graph [D_l]: Create l nodes and connect each node only to itself.
- Empty Graph [J_0]: Connect a single node with a tuple length of 0 to itself. It represents the junction element of level 0.

3.1 Important graph algebra operations and their geometrical meaning

This section summarizes the required graph algebra operations to construct SCNTs. First, the basic operations are presented, followed by the combined operations consisting of the concatenation of different basic operations. Also some details for an efficient realization in C++ are given.

3.1.1 Basic operations

The single basic unary operation within the graph algebra is the *opposite*, denoted by * after the graph identifier. The operator creates the complementary graph $G^*(V, E^*)$, i.e., a graph that has the same node set as the original G, but all edges in E^* have the opposite direction compared to E.

The first binary operation is the graph *union*, denoted by $\cup$. The union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is defined as

$$G_1 \cup G_2 = G_u(V_1 \cup V_2, E_1 \cup E_2),$$

so each node and each edge that is contained either in G_1 or G_2 is contained in the result graph. This operation is mainly employed to add additional edges to the graph during construction of SCNTs in order to connect former isolated parts.

Similarly, the *intersection* $\cap$ of graphs G_1 and G_2 is defined as

$$G_1 \cap G_2 = G_i(V_1 \cap V_2, E_1 \cap E_2).$$

This operation keeps those nodes and edges in the result graph that are members of both G_1 and G_2.

Related to the principles of graph union and intersection is the subtraction of two graphs with the graph *minus* operation denoted by $-$. For two graphs G_1 and G_2, the subtraction is defined as

$$G_1 - G_2 = G_s(V_1 - V_2, \rightarrow E_s).$$

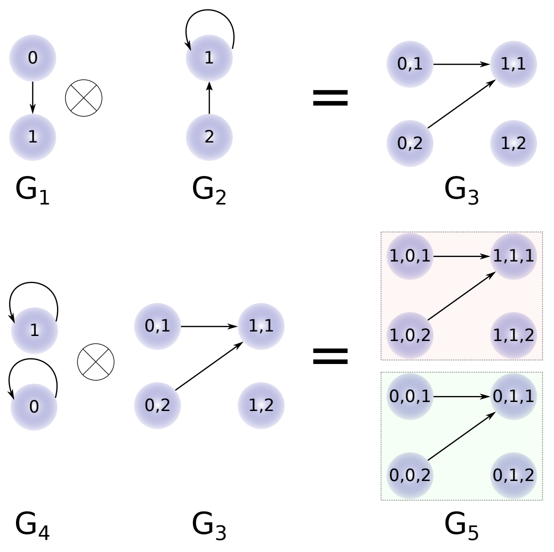
Here, the notation $\rightarrow E_s$ denotes the resulting edge set of the subtraction result G_s, which is arrived at as follows: All nodes fulfilling $V_1 \cap V_2$ are removed in G_s. Afterward, all edges e fulfilling $\sigma(e) \in (V_1 \cap V_2) \vee \tau(e) \in (V_1 \cap V_2)$ are also removed in G_s since edges starting from or pointing to non-existing nodes are invalid. As a result, the minus operator requires several search operations within the edge set. Geometrically, the graph minus can be employed to select a part of interest out of a graph, by creating a temporary graph which describes the unwanted nodes and which is subtracted from the original graph.

One very important operation is the so-called *categorical product*. It is the only basic operation that is able to increase the tuple length and is denoted as $\otimes$. Assume two graphs G_1 and G_2 with tuple length m_1 and m_2, respectively. The categorical product $G_1 \otimes G_2$ creates the result graph G_c whose node set V_c is the cross-product of the two input node sets. Its dimension m_c is determined by $m_1 + m_2$. The edges of G_c are constructed by the following rule: For all $e_1 \in E_1$ and $e_2 \in E_2$, the resulting edge $e_c \in E_c$ is determined by

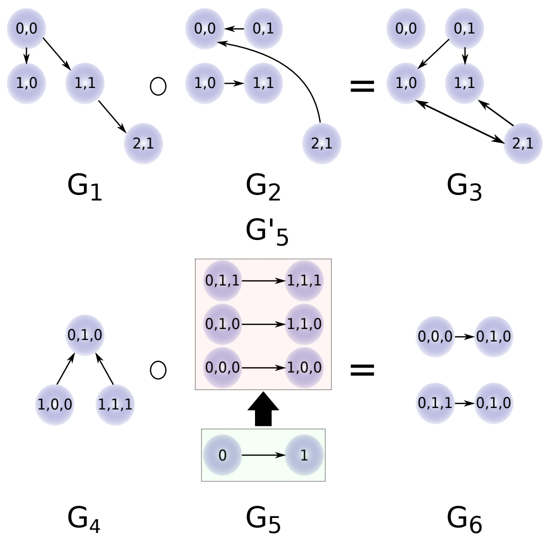
$$e_c = \Big(\big(\sigma(e_1) \times \sigma(e_2)\big), \big(\tau(e_1) \times \tau(e_2)\big)\Big),$$

where the $\times$ operator concatenates tuple pairs from $\sigma(e_1)$ and $\sigma(e_2)$, and $\tau(e_1)$ and $\tau(e_2)$, respectively. An example of two applications of the categorical product is illustrated in Figure 3.1a. The first row shows the product of two 1-dimensional graphs G_1 and G_2, resulting in the 2-dimensional graph G_3. The node $(1,2)$ in G_3 has no incident edge and is isolated from the remainder. The second calculation visualizes the main usage of the categorical product. A categorical product of the self-connected graph G_4 which is a D_2-graph with a dimension of 1 and G_3 with a dimension of 2 results in a graph that contains two non-connected parts with the same structure as G_3. They are boxed in G_5 in Figure 3.1a. They only differ in their leading tuple entry that is determined by the node labels of G_4. It is sufficient to increase the length of G_4 to the desired number to create more copies of G_3. This example also illustrates the behavior of isolated nodes. Although $(1,2)$ from G_3 is copied, its copies in G_5 remain isolated. In general, isolated nodes stay isolated in further applications of the categorical product and hence can be removed to optimize further calculations. For the generic case $G_1 \otimes G_2 = G_c$ this means that only nodes fulfilling

$$v \in (\sigma(E_1) \times \sigma(E_2)) \cup (\tau(E_1) \times \tau(E_2))$$



(a) Two examples for the categorical product of graphs. The upper image shows the case for graphs with equal dimension, while the lower image illustrates the calculation when both graphs differ in their dimension.



(b) Two examples of the graph composition. The upper image shows the case for graphs with equal dimension, while the lower image illustrates the calculation when both graphs differ in their dimension.

Figure 3.1.: Four examples of categorical product and graph composition.

are connected and need to be considered. The number of edges $|E_c|$ is $|E_1| * |E_2|$ and the complexity of the categorical product is $O(|E_1| * |E_2|)$. Finally, if t_1^{ext} is the tuple extent of G_1 and t_2^{ext} that of G_2, then the tuple extent of $G_c = G_1 \otimes G_2$ is given by $t_1^{\text{ext}} \times t_2^{\text{ext}}$ with $\times$ again representing a concatenation.

The last basic operation to cover is the graph *composition* denoted by $\circ$. For two edges $e_1 \in G_1$ and $e_2 \in G_2$ the composition is defined as

$$e_1 \circ e_2 := \begin{cases} (\sigma(e_2), \tau(e_1)) & \text{if } \sigma(e_1) = \tau(e_2) \\ \text{no edge} & \text{otherwise} \end{cases}$$

i.e., for each $e_1 \in G_1$ it needs to be tested if there exists an $e_2 \in G_2$ that ends at a node with the same tuple as the start node of e_1. This is shown in the first row of Figure 3.1b where the composition is applied to two graphs with the same tuple length. No new nodes are generated by $\circ$, but it is possible that the result graph contains nodes that previously where only part of one of the input graphs, as the node $(0,1)$ demonstrates. Moreover, it may happen that nodes, which are connected in both input graphs, are isolated in the result graph as it is the case for node $(0,0)$. The second row of Figure 3.1b shows the general case of the composition with input graphs G_4 and G_5 differing in their tuple length. We assume that the second input graph G_5 has shorter tuples. Thus, the second graph needs to be expanded, i.e., the tuples need to be adapted in their length to G_4. The procedure is similar if G_4 has the shorter tuples with the difference that G_4 is expanded in that case. If the tuples of G_4 have the length m_4 and those of G_5 the length m_5, the set of sub-tuples $S^{\text{tail}} = \zeta_1^{m_4-m_5}(V_4)$ will be created. Now, conceptually, $\left|S^{\text{tail}}\right|$ temporary graphs are generated from G_5 by concatenating one element $v^{\text{tail}} \in S^{\text{tail}}$ to all nodes $v^5 \in G_5$ by $v^5 \times v^{\text{tail}}$ and repeating this for all elements of S^{tail}. This results in $S^{\text{tail}} = \{(0,0),(1,0),(1,1)\}$ with $\left|S^{\text{tail}}\right| = 3$ for the example in Figure 3.1b and hence, three virtual non-connected copies of G_5 are generated, which are shown in the upper rectangle of the lower example in Figure 3.1b and represent the graph G_5'. Afterward, the composition $G_4 \circ G_5'$ is calculated as in the upper part of Figure 3.1b.

For the composition it is also possible to determine the tuple extent of the result graph in advance. Assuming that we have a Graph G_{short} with the shorter tuple length m_{short} compared to G_{long} with tuple length m_{long}, then, the tuple extent of $G_{\text{short}} \circ G_{\text{long}}$ and of $G_{\text{long}} \circ G_{\text{short}}$ can be constructed via the C++-code fragment of Listing 3.2 in which the extents $t^{\text{ext}}_{\text{short}}$ and $t^{\text{ext}}_{\text{long}}$ are represented as `std::vector<> longer` and `std::vector<> shorter`.

```cpp
vector<int> new_extent;
int diff = m_long - m_short;

for(int i = 0; i < m_long; i++) {
  if(i < diff) { // copy tail of the longer tuple
    new_extent.push_back(longer[i]);
  }

  else { // compare corresponding parts of the tuples
    int max_val = max(longer[i], shorter[i - diff]);
    new_extent.push_back(max_val);
  }
}
```

Listing 3.2: Constructing the tuple extent for the result graph of a graph composition before the actual calculation.

The pre-computations of dimension, extent, and numbers of edges are important to allocate appropriate efficient data structures for the operations.

3.1.2 Combined operations

There are four important combined graph operators. The unary operator υ transforms a directed graph G into an undirected one, following the above representation of undirected graphs by directed graphs. It combines the operations union $\cup$ and opposite * by $\upsilon(G) = (G \cup G^*)$.

Another unary operator is the n-fold composition of a graph G that is denoted by G^n and is an abbreviation for:

$$G^n = \underbrace{G \circ G \circ \cdots \circ G}_{n-\text{times}}$$

The binary operator *graph conjugation* $\Diamond$ combines the two operations composition $\circ$ and opposite * by $G_1 \Diamond G_2 = G_2 \circ G_1 \circ G_2^*$.

The last operator is the closure of a graph denoted by $c(G)$. It is defined as:

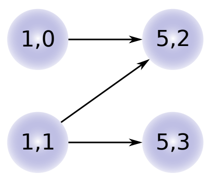
$$c(G) = \bigcup_{k=1}^{\infty} G^k$$

There always exists an $n \geq N$ for an $N \in \mathbb{N}$ for finite graphs which satisfies $c(G) = \bigcup_{k=1}^{n} G^k$ as discussed in[45, p. 10].

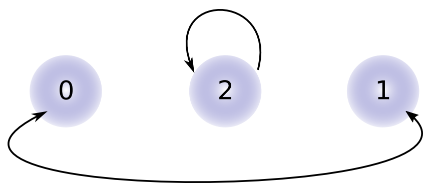
All operations based on the graph composition and their geometrical meaning are strongly connected with the concept of *transfer graphs*. In simplified terms, transfer graphs represent the relations of different node sets within different parts of the graph. When combining two graphs, it may happen that several nodes coincide in the result graph. Assume we want to mirror the graph G form Figure 3.2a horizontally with the symmetry axis running through the nodes $(5, 2)$ and $(5, 3)$, i.e., those nodes coincide between both copies of G. To that end, the graph G has already been appropriately prepared by grouping its nodes in two different sets: The nodes with leading tuple entry 1 and those with leading 5, differentiating the symmetry axis from the part that is copied.

A second requirement for a successful mirroring is a so-called symmetry graph S that describes the symmetry relations of the final graph, which is shown in Figure 3.2b. Informally, this graph tells the construction process: There are those nodes of the initial graph that do not lie on the symmetry axis and have a leading 0 in the result graph. The mirrored nodes lie in the part with 1-leading nodes. Nodes on the symmetry axis receive a leading 2.

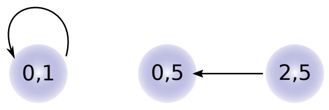
The last important entity is the transfer graph T that describes the relations between different node sets and is depicted in Figure 3.2c. In that case, it represents the fact that all nodes with a leading 5 in G will not lie in the 0-leading part in the result graph, but on the 2-leading part,



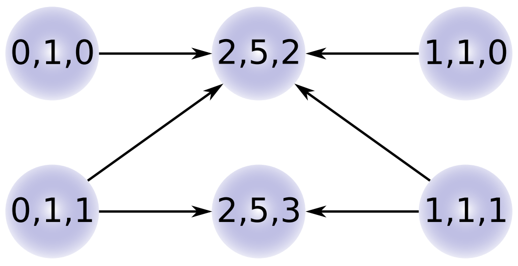
(a) Original graph G that should be mirrored.



(b) Symmetry graph S to mirror graph G.



(c) Transfer graph T to mirror graph G.



(d) Result G_m of the mirroring process of G.

Figure 3.2.: Mirroring graph G with the concept of transfer graphs.

which is the symmetry axis. The nodes with a former leading 1 are assigned to the left part of the resulting graph with a leading 0.

In general, the process of mirroring G with respect to the symmetry graph S and the transfer graph T is performed by:

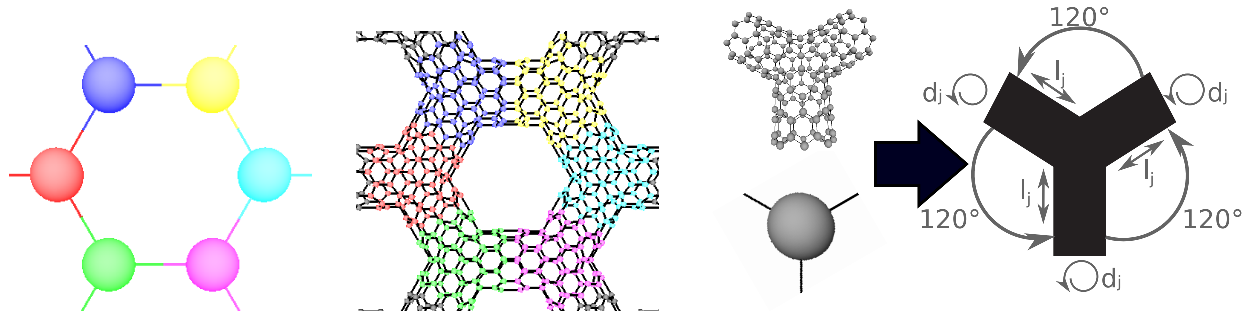
$$G_m = c(S) \circ T \circ G \circ T^* \circ c(S)^* = c(S) \circ (G \diamond T) \circ c(S)^*$$

For graph theoretic details about transfer graphs and how to construct them refer to [45]. Important for this thesis is the fact that composition, closure and conjugation operations are required several times. This results in many calls of the graph composition which is a complex operation.

3.2 A general algorithm for constructing super carbon nanotubes of arbitrary order

Methods for generating SCNTs of higher order were proposed in [24] and Section 4 of [24] is the base for this chapter. While Coluci [17] defines SCNTs as tubes formed by smaller tubes connected via Y-junctions, the work of Schröppel and Wackerfuß follows a different approach which was adapted for this thesis. Tubes of higher order are created by appropriately connected Y-junctions with extended arms. This idea is visualized in Figures 3.3a and 3.3b. Considering one single hexagon within a honeycomb sheet in Figure 3.3a, we see that each carbon atom has three neighboring atoms. The colors indicate that each C-C-bond should be assigned to one half to its left atom and to the other half to its right atom. Figure 3.3b depicts a situation where each carbon atom and its half bonds are replaced by Y-junctions. The outer shape of the resulting structure is still a hexagon. Like it is possible to create a flat honeycomb sheet out of the hexagons of Figure 3.3a, it is possible to build an equivalent grid out of those hexagons in Figure 3.3b. Since those new sheets can be used to roll up a super tube, they are called *super sheets* [17]. Analogous to the relation of CNTs and SCNTs, a standard honeycomb grid of carbon atoms is a super sheet of order 0.

Because of the properties mentioned above, the main abstraction for higher tube orders comes from the idealization of single atoms with half bonds and Y-junctions to elements with the same outer shape as shown in Figure 3.3c. Junctions of arbitrary complexity and single atoms with half bonds have the following properties in common that enable the abstraction: They possess one center of rotation and three arms of equal length l_j and equal diameter d_j. Taking one of the three arms, the other two arms are rotated around 120° and 240° within the plane this arm lies in, as shown on the right side of Figure 3.3c with all mentioned properties highlighted in gray. Those abstracted objects are called *junction element* or *element* for short if the context is clear. The geometric properties allow to arbitrarily connect the arm of one element with the arm of another element and to form regular hexagons ([41],[51]).



(a) Hexagon out of carbon atoms. **(b)** Hexagon out of level 0 junction elements. **(c)** Junctions of all levels can be abstracted as a Y-shaped figure.

Figure 3.3.: Junctions as basic elements of abstraction.

As mentioned above, super sheets are formed by replacing single atoms with Y-junctions. Thus, an important question is how to construct these Y-junctions themselves. The answer is that an iterative procedure exists that constructs (super) sheets and junction elements in an alternate fashion, starting with a single atom from which the honeycomb sheet of C-atoms is constructed. This super sheet of order 0 can be employed to create a Y-junction of level 0. This thesis follows the naming convention of the MISMO group to categorize junctions (and the so-called *base elements* presented later) in different *levels* while super tubes and their corresponding super sheets have a certain *order*. The junctions can replace single atoms and create a super sheet of level 1 that afterward is used as a base for level 1 junctions and so on. In general, an order L super sheet consists of junction elements of level $(L-1)$. Single atoms are defined as a junction elements of level (-1) to integrate into this scheme.

In principle, an SCNT of order L is a rolled up super sheet of order L created by level $(L-1)$ junctions which is cut into shape where necessary.

The theory of constructing a junction of a certain order is summarized in Section 3.2.1, while the general construction algorithm of a tube is discussed in Section 3.2.2.

3.2.1 Construction of high level junctions

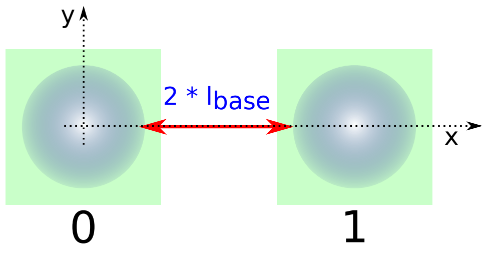
The construction of an SCNT Y-junction within the graph algebra presented was first published in [52] on a graph theoretic basis and in combination with the construction of a tube in [45] with the same focus. Like in [24], the focus of this section lies on the geometric description of the intermediate steps required, since they demonstrate the inherent self-similarity and hierarchy within the SCNT models in a descriptive fashion. An additional focus lies on encoding the construction process within the tuple system, since this information will later be exploited by the data structures and by the different solvers.

Like the construction of an SCNT of order L, the construction of the required junction elements of level $(L-1)$ is an iterative process itself. In the sequel, the procedure for a level 0 junction, starting with a single atom, is demonstrated. Independent of the actual level of the junction, the construction passes through four intermediate stages, denoted as S_1 to S_4, which are covered in the following.

The shape of the junction is determined by a tuple of two input parameters (d_x, l_x) that is explained during this section.

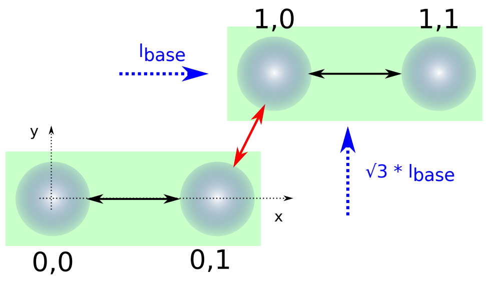
Please note that the appropriate geometry is already applied to the graphs to allow a clear visualization, although, in principle, the graph algebra itself is independent of any geometry. The way the geometry results from the tuples per node is discussed separately for each step. The geometry per step is only dependent on the respective leading tuple entry t_{val}. Additionally, for a point p its single spatial x-, y- and z-coordinates are accessed with p_x, p_y and p_z. The new point after the transformation is denoted by p' and its coordinates by p'_x, p'_y and p'_z.

Stage $\mathbf{S_1}$: S_1 on level 0 is a simple graph consisting of two tuples (0) and (1) of length 1 connected by two directed edges. In principle, during this stage, two connected copies of the junction element from the previous order are connected. In the case of order 0, these junction elements degenerate to the single node with an empty tuple of length 0 called J_0. The colors in all three images of Figure 3.4 link the parts of the respective equation to their contribution in the graph. For Equation 3.1 this means that the green part $D_2 \otimes J_0$ creates the two nodes (0) and (1) while the right part in red instantiates the red edge from (0) to (1) and vice versa. The tuple length is increased by 1 through stage S_1, since the left copy of the input graph receives a 0 as leading tuple entry and the right copy receives a 1. For the geometry, a coordinate system is indicated in gray. Please note that the construction process for the stages S_1 to S_4 only works in two dimensions. The third dimension comes into play later. The origin of the coordinate system is placed in the center of node (0) and the center of node (1) lies on the x-axis. Consequently, the underlying geometric transformation between both nodes is a translation in x-direction whose value is dependent on the so-called *base length* l_{base} as indicated in blue. With this base length the distance of two carbon atoms within graphene is modeled, which is two times the atomic radius of carbon ([53], [54, p. 227]). The values in the literature are, e.g., 67 pm based on calculations of Clementi et al. [55] or 70 pm based on experiments of Slater [56]. To realize the translation for each point the new position $p' = (p_x + t_{\text{val}} * 2 * l_{\text{base}},\ p_y,\ p_z)$ is calculated.

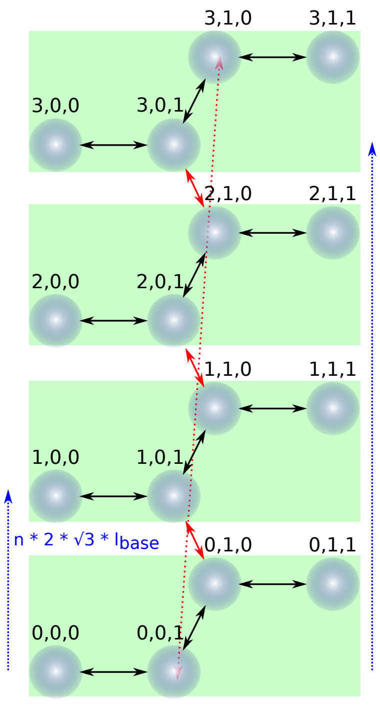


(a) Graph S_1

$$S_1 = D_2 \otimes J_0 \cup \upsilon(P_2) \qquad (3.1)$$



(b) Graph S_2



(c) Graph S_3

$$S_2 = D_2 \otimes S_1 \cup \upsilon(P_2 \otimes P_2^*) \qquad (3.2)$$

$$S_3 = D_d \otimes S_2 \cup \upsilon(Cy_d \otimes P_2^* \otimes P_2)\,,\ \mathrm{d} = 4 \qquad (3.3)$$

Figure 3.4.: The first stages steps S_1 - S_3 for constructing a sheet of order 0.

Stage S_2: S_1 is the input for S_2 that is shown in Figure 3.4b. Again, two copies of the input are created and connected from $(0, 1)$ to $(1, 0)$ and vice versa. Like for S_1, colors clarify the correlation to Equation 3.2. The left side generates the copies while the right side connects them appropriately. Since for the lower copy a leading 0 and the upper copy a leading 1 is appended to the tuple, its length is increased by 1. This time, the geometry consists of two different translations with one occurring in x-direction and the other one in y-direction as depicted by blue arrows. The position calculation is: $p' = (p_x + t_{\text{val}} * l_{\text{base}},\ p_y + t_{\text{val}} * \sqrt{3} * l_{\text{base}},\ p_z)$.

Stage S_3: d copies of S_2 are instantiated for S_3 as shown in Figure 3.4c. The parameter d determines the number of copies to be made. In the context of S_3 it is important to remember that in contrast to S_1 and S_2, not only two copies of the input graph are created but d. $d = 4$ for Equation 3.3 for the example in Figure 3.4c.

Please also note the Cy_d-graph which appears in the right side of Equation 3.3. The edges from the top node $(3,1,0)$ to $(0,0,1)$ and vice versa are created additionally by employing the Cy_d-graph which are shown as the dotted red line. These edges connect the bottom and the top of the sheet when it is rolled up later in the process. They are only required if the graphene sheet that is constructed is employed to build a CNT. If Cy_4 is replaced by P_4 in Figure 3.4c, S_3 looks identical, except for the missing edges between $(3,1,0)$ and $(0,0,1)$.

The tuple length is again increased by 1 in this step since each copy of S_2 receives a consecutive leading index from 0 up to $(d-1)$. The underlying geometric transformation is a translation in the y-direction so that the copies become stacked over each other, with the origin of the coordinate system remaining in the bottom left node $(0,0,0)$. The position of new nodes can be calculated by $p' = (p_x,\ p_y + t_{\text{val}} * 2 * \sqrt{3} * l_{\text{base}},\ p_z)$.

Stage S_4: Finally, S_4 is constructed as depicted in Figure 3.5. A second parameter l appears that determines how many copies of S_3 are created. l also depends on the type of junction that is constructed. Equation 3.4 also contains a cyclic graph that ensures the connection of the bottom with the top and vice versa shown as colored dotted lines. These edges are removed by replacing the cyclic graph Cy_d by a path graph. Analogous to S_3, each copy of the input graph receives a new leading entry from 0 to $(l-1)$.

It is obvious that S_4 represents a honeycomb grid whose size is determined by d and l. The height and the width can also be expressed in terms of l_{base} as shown by the gray numbers around the grid in Figure 3.5. At the bottom, we see that each copy of S_3 increases the length of the sheet by $(2+4)$ times the base length. The only exception is the last copy of S_3 that is only $(2+3) * l_{\text{base}}$ long. This results from the fact that for this instance of S_3 no diagonal edge is required to connect it to its successor like, for example, the edges $(k,0,1,1) \rightarrow (k+1,0,0,0)\ \forall k \in [0, l-2]$ do. As a consequence, the length of the sheet can be expressed in terms of l by $(l * 6 - 1) * l_{\text{base}}$. Accordingly, on the left side of Figure 3.5, we see that the height of the sheet can be calculated in dependence of d by $(d * 2 * \sqrt{3} - 1) * l_{\text{base}}$ since the last copy of S_2 at the top again represents an exception with a shorter length compared to the other copies. The geometric transformation applied in S_4 is a translation of the copies of S_3 along the x-axis. To that end, the translation $x_{\text{new}} = x_{\text{old}} + 6.0 * l_{\text{base}} * t_{\text{val}}$ is applied with $6.0 * l_{\text{base}}$ being the length of one instance of S_3.

Several incomplete hexagons on the left and the right boundary exist which are surrounded by the blue boxes in Figure 3.5. They need to be removed before the construction can proceed with a regular grid. Of course, the edges ending in or starting from those nodes also must be eliminated within the cutting process. In terms of the graph algebra, a graph that only contains the boundary nodes is created and subtracted from S_4 via the graph minus operator. Overall, the construction from S_1 to S_4 increases the length of the initial input graph, i.e., the junction of the previous level, by 4 new entries.

Figure 3.5 also demonstrates another important concept. As already mentioned in the beginning of this section, a lexicographical order is defined on the tuples to map each tuple to a unique global index. Bearing that in mind, we can identify several properties about the distribution of the global index within the grid. The first point is that it starts with value 1 at the bottom left of the cut grid at node $(0,0,0,1)$. This is the case for all order 0 sheets. Furthermore, recall that S_4, in principle, consists of many copies of S_2. The four nodes within S_2 are consecutively numbered from left to right since the two leading entries $x_4 = t[4]$ and $x_3 = t[3]$ always are identical and $(x_4, x_3, 0, 0) < (x_4, x_3, 0, 1) < (x_4, x_3, 1, 0) < (x_4, x_3, 1, 1)$. The fact that several copies of S_2 are stacked within S_3 leads to the situation that, in general, the global index increases within S_3 from bottom to top. In particular, the difference in the global index is always 3 between nodes $(x_4, x_3, 1, 0)$ and $(x_4, x_3 + 1, 0, 1)$, which are connected by additional diagonal edges. In the last step, S_4 vertically lines up a varying number of S_3 copies letting the global index increase from left to right. These considerations about the distribution of the global index are important for analyzing the simulation algorithm and its implementation.

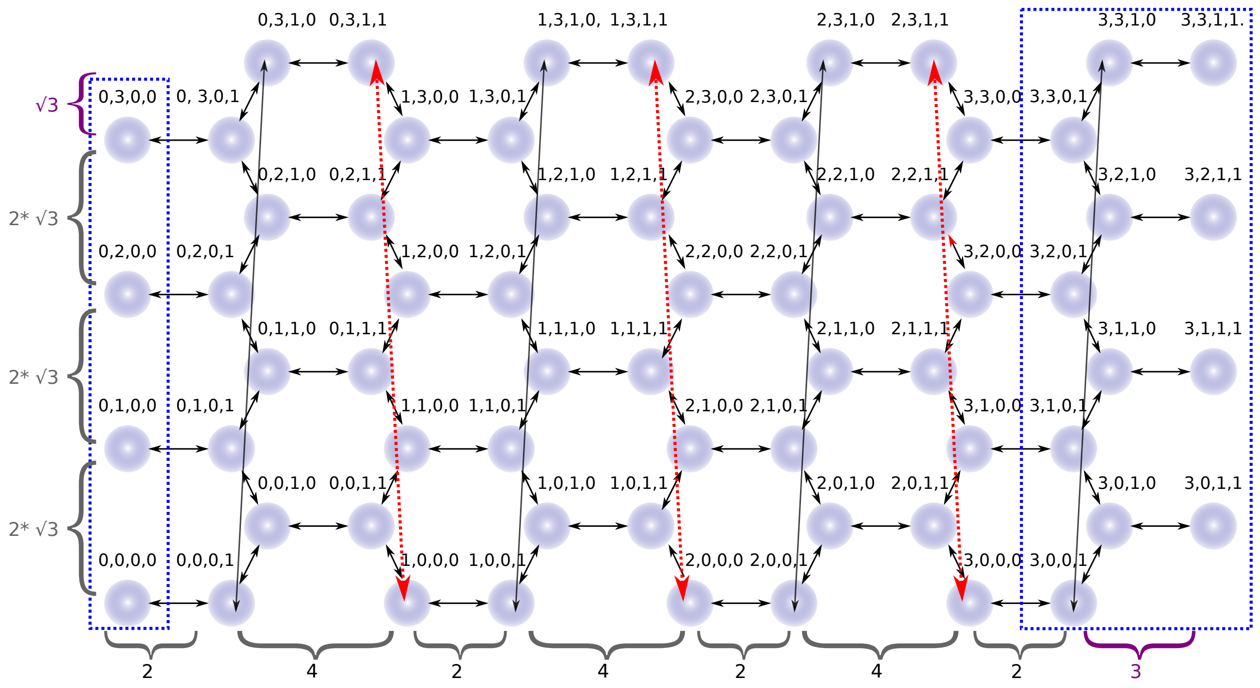
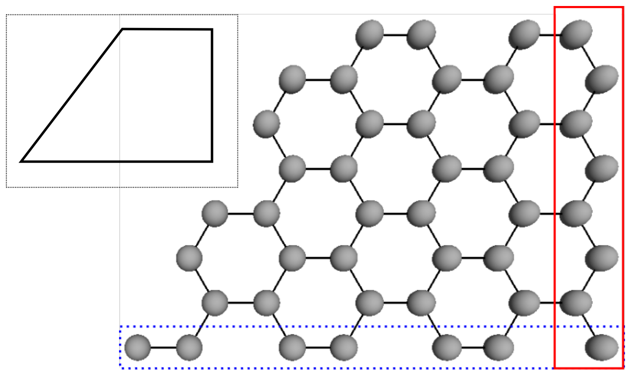


Figure 3.5.: Step S_4 during construction of an order 0 sheet. The picture shows the graphene sheet that needs to be cut into shape by removing the incomplete hexagons within the blue boxes.

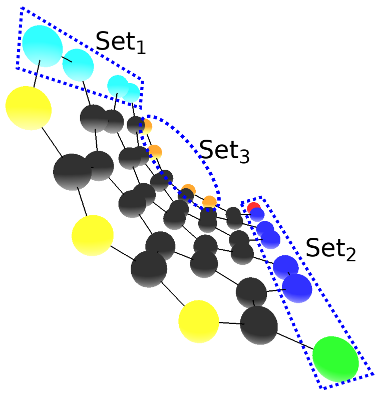
$$S_4 = D_l \otimes S_3 \cup \upsilon(P_{l-1} \otimes (D_d \cup Cy_d) \otimes P_2^* \otimes P_2^*)\,,\; d = 4,\; l = 4 \tag{3.4}$$

Construction of the base element: One required part to create a junction element of level L is a so-called *base element* that is a cut-out of a super sheet of order L as represented by S_4. One example for a base element of level 0, that serves as a building block for a Y-junction of level 0, is depicted in Figure 3.6a. The small insert on top shows an abstract view of this structure.

There is a dependence of the input parameters (d_x, l_x) for the junction configuration on the two parameters d and l that determine how many copies of S_2 are instantiated in S_3 and how many copies of S_3 are instantiated within S_4, respectively. The relations are: $d = \text{floor}\left(\frac{3*d_x+1}{2}\right) + 1$ and $l = \text{floor}\left(\frac{l_x}{2}\right)$. The floor-operation creates truncated values of the fraction calculation. It is important to note that only integer inputs for d_x and l_x lead to feasible junction configurations.



(a) Flat level 0 base element with $d_x = 2$ and $l_x = 6$. The dotted blue line indicates influence of l_x and the solid, red line the one of d_x, respectively.



(b) Bent order 0 base element with $d_x = 2$ and $l_x = 6$ with full geometry applied, and consisting of 44 nodes.

Figure 3.6.: A base element of order zero.

Additionally, (d_x, l_x) have a directly visible influence on the shape of the base element that is indicated in Figure 3.6a: On the one hand, the number of nodes along the zigzag-line on the right side of the base element, being surrounded by the solid, red box, is given by $3 * d_x + 2$. On the other hand, the number of nodes on the horizontal line of the base element, being highlighted by the dotted blue box, is given by l_x+1. Please note, that the rightmost node, which also belongs to the vertical zigzag line, is considered for both values. Thus, it can be deduced that the depicted base element in Figure 3.6a results from the parameter set $(d_x = 2, l_x = 6)$ with seven nodes on the horizontal and eight nodes on the vertical line. For feasible junctions, there is also the prerequisite $l_x > d_x$ since, otherwise, the general shape of the base element would be violated.

Concerning the geometry of the base element, it is already bent around the x-axis as shown in Figure 3.6b while Figure 3.6a ignores this bending for sake of clarity. The transformation of this bending only depends on the actual positions of the nodes within graph S_4 and is very similar to the wrapping process of the super sheet around an axis that are described in the following section. Consequently, bending the base element is skipped here.

As a last step of constructing a base element, the nodes are internally partitioned into different sets, as indicated in Figure 3.6b, where each of the six colors represents a different node set.

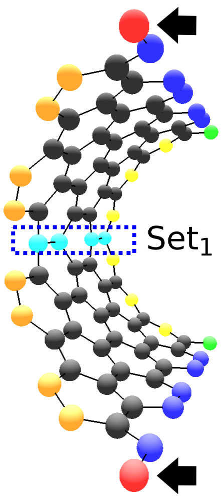
These node sets are distinguished by adding another fifth tuple dimension. The interior set is the largest set, shown in black (refer to [45] for details), whose nodes receive a 0 as a fifth entry. These nodes are all be contained in the final junction graph. In contrast to those, the remaining sets are the connection line or single connecting nodes to other base elements and coincide with corresponding sets and points of neighboring base elements. Several combined sets of nodes are labeled in Figure 3.6b with Set_1, Set_2 and Set_3 since they are important for the remainder of the junction construction process. Altogether, the junction construction added five tuple entries to the input graph so far. The new fifth tuple entry does not contribute to the geometry, i.e., no transformation is associated with it. However, it impacts two other domains. The first one is the lexicographic order of the tuples that differ if the fifth tuple entry is ignored. The second one is a drastic reduction of the density of the tuple space. Let t_{tail} be the part of the tuple following the new partitioning entry t_{part}. Then, no two tuples with the same t_{tail} but different t_{part} exist.

The construction of a Y-junction is internally divided into three stages, where the first two stages result in the intermediate graphs J_1 and J_2, and the third one creates the final junction. All these stages employ geometric mirroring and rotation operations of existing parts. This is mainly realized by the graph algebra operation of conjugation (see Section 3.1.2) combined with transfer graphs and the partitioning of nodes into the sets from the last step. The transfer graphs describe the relation of different node sets within neighboring base elements. They determine the nodes that coincide in the resulting graph and the way the remainder is connected. In total, this results in 27 calls of the composition operation during the construction of a general junction of level L starting with a base element of level L.

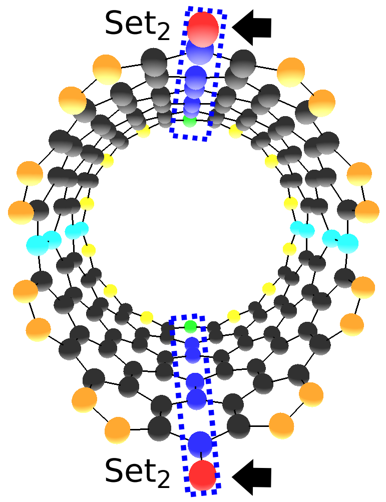
Stage $\mathbf{J_1}$: The incoming base element is mirrored and connected at the former cyan Set_1 from Figure 3.6b to create J_1, i.e., these nodes of the original graph and its mirrored copy coincide in the result graph J_1, as shown in the box of Figure 3.7a. Therefore, J_1 has four nodes less than the sum of nodes of two base elements in the depicted example of Figure in 3.7a. One tuple entry is appended in this stage whereby the nodes that only belong to the original base element receive a leading 0 and those that are part of the mirrored base element receive a leading 1. Coincident nodes in the rectangle (Set_1) receive a leading 2. The underlying geometric transformation is a mirroring on the xy-plane, realized by the calculation $z_{\text{new}} = (-1)^{t_{\text{val}}} * z_{\text{old}}$. This means that the nodes with leading 0 as well as the coincident nodes which lie on the symmetry plane do not move. J_1 represents a half junction arm. Please note the two nodes at the bottom and on top which are marked by an arrow: These two nodes coincide in all junction arms of the final junction and thus, lie on the symmetry axis of the junction.

Stage $\mathbf{J_2}$: J_1 is mirrored in order to create the graph J_2 which results in the graph depicted in Figure 3.7b. Now, it contains four base elements. The sets which coincide in that stage are highlighted by boxes. In that case, they consist of the nodes that belonged to the instances of Set_2 in the two copies of J_1. Set_2 formed the horizontal line at the bottom of the base element whose number of nodes is determined by $(l_x + 1)$. Consequently, J_2 has $2 * J_1 - 2 * (l_x + 1)$

nodes. In an analogous fashion to the procedure in J_1, the nodes in the original J_1 receive a leading 0, those in the generated copy a leading 1 and the coinciding nodes a leading 2. Here, the plane for the mirroring is the xz-plane and this operation can be realized by the coordinate transformation $y_{\text{new}} = (-1)^{t_{\text{val}}} * y_{\text{old}}$, which again does not affect the 0-leading and the 2-leading part that lies on the symmetry plane. J_2 represents a junction arm.



(a) Graph J_1, the half junction arm, is created by mirroring a base element and merging the former cyan colored sets shown in the dotted box.



(b) Graph J_2, one of three junction arms, composed of four base elements, i.e., two J_1 graphs.

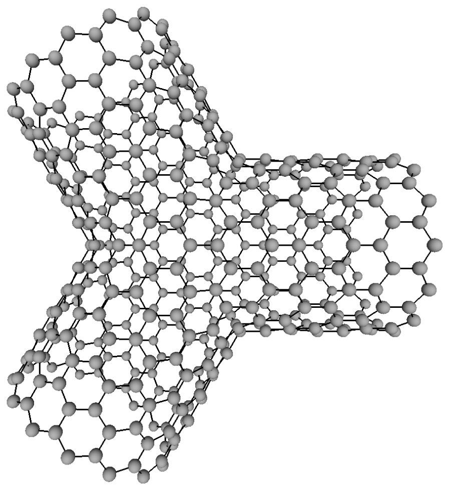
Figure 3.7.: A half and a complete junction arm of level 0.

Stage J$_3$: The last stage creates two clones of the junction arm, connects them appropriately and rotates the first cloned arm by 120° and the other by 240°. Red and orange nodes of neighboring instances of S_2 in Figure 3.7b coincide for the connection of the three arms. In that case, the geometric operation applied is a rotation around the z-axis. Mathematically, this can be described by applying the following matrix to the coordinates of each point:

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

One tuple entry is appended during the rotation procedure which identifies the position of the respective node within the junction. If it lies within the original arm, it has a 0 as leading entry, and a 1 or 2 if it is located within one of the clones. Additionally, a 3 indicates that the node is part of the coincident nodes between arm 0 and arm 1, while a 4 means that the nodes belong to the coincident nodes between arm 1 and arm 2. Finally, those nodes between arm 2 and arm 0 possess a 5 at this tuple position. Nodes on the symmetry-axis of the junction have a leading 6.

Now, J_3 is constructed that consists of 434 single nodes and represents the junction with configuration $(d_x, l_x) = (2, 6)$ as shown in Figure 3.8a. The final tuples are eight entries longer than the input tuples. Hence, the tuples of the depicted level 0 junction have a length of eight, since the construction started with a single node with an empty tuple. It is also demonstrated that J_3 consists of twelve base elements and thus, is a highly regular structure. However, the junctions have one special property. In contrast to the order 0 super sheet that is employed to construct the first base element, the nodes in J_3 are not only arranged in a regular hexagon mesh, but there are also some irregularities in the form of octagons as visualized in Figure 3.8b. The small insert at the bottom shows the orientation of the junction and the position of the camera that is looking at the red point which represents the highlighted red octagon in the junction. Three of those octagons occur within a junction at the positions where different junction arms are connected. In the insert, the respective position of the other two octagons is marked by green dots. The difficulty for the simulation that arises from these octagons is explained in Section 5. They change the structure of the grid around several nodes.


(a) Level 0 junction element for constructing an order 1 SCNT consisting of 434 nodes with junction parameters $d_x = 2$ and $l_x = 6$.

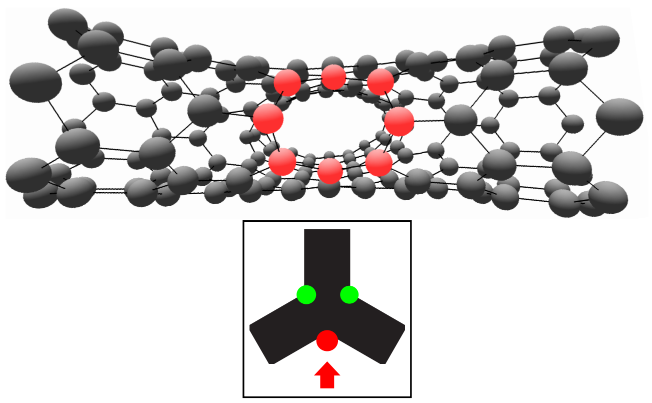

(b) One of the three octagons contained within the hexagon grid is highlighted in red. The insert visualizes the scene from above and marks the position of the octagons as well as the viewing direction.

Figure 3.8.: Details about level 0 junctions.

3.2.2 Building tubes

In principle, the construction process of the super tubes is the same for all orders. Incoming junction objects are used as a base for different, intermediate graphs that, in the end, are employed to build a super sheet that is rolled up to a tube. Additionally, the intermediate graphs have the same structure, i.e., shape as those for the construction of a base element of order 0. They are labeled with S_1^t up to S_4^t. However, there are some differences for orders higher than 0, resulting from the replacement of single nodes as basic units for construction by junctions of level $(L-1)$ to construct a tube of order L.

For example, in case of higher orders it is not the native distance between carbon atoms that is relevant for the tube geometry, but the distance between centers of two junctions of level $(L-1)$. Figure 3.9 shows for a $(1,3)$ junction of level 0 how this can be realized. The blue line shows the distance between both centers of rotation for the two junctions. Red nodes indicate a path between both centers which we follow to determine the distance in terms of l_{base}. In case of Figure 3.9, its length equals to $7*2*l_{\text{base}}$ and is calculated as follows: Between two connected nodes on the blue line, the distance in horizontal direction is $2*l_{\text{base}}$. The same is valid for the nodes on the magenta line. Additionally, we need the distance in diagonal direction, where jumps from the blue line to the magenta line occur. This distance is l_{base}. Now, following the path determined by the red nodes from left to right, this results in $(2+1+2+1+2+1+2+1+2+1+2+1+2)*l_{\text{base}} = 20*l_{\text{base}}$ for the depicted case of $l_x = 3$. In general, this line has the length $(2*(3*l_x+1))*l_{\text{base}} = (6*l_x+2)*l_{\text{base}}$ which solves the problem of determining the distance between junctions of level 0. We call the half distance between two centers of level 0 junctions $l_{\text{base-lvl-0}}$.

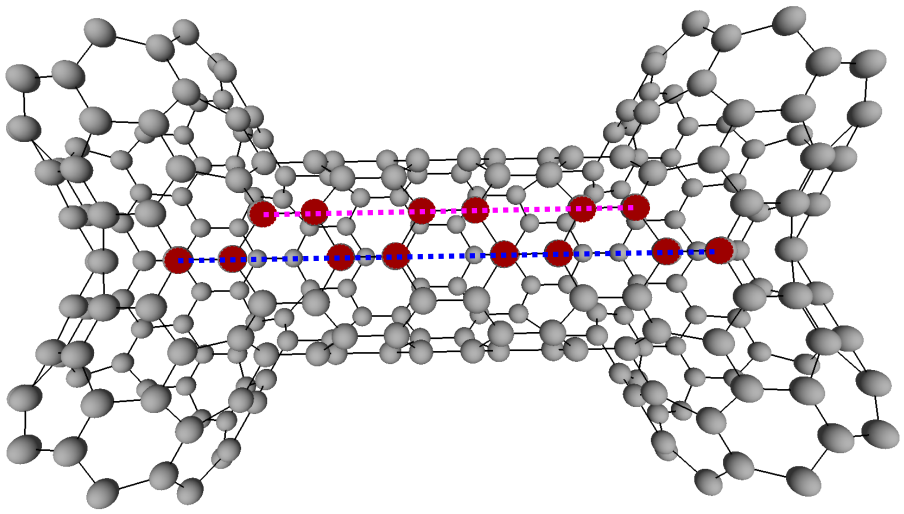


Figure 3.9.: Determining the distance between two level 0 junctions with parameters $(1,3)$. The dotted blue line indicates the actual distance between two centers.

Now, to deal with higher orders, it is sufficient to imagine that each red node can be replaced by an appropriate Y-junction. In case of level 1, the horizontal distance between two adjacent nodes on one of the dotted lines is not $2 * l_{\text{base}}$ but $2 * l_{\text{base-lvl-0}}$. Hence, the overall distance between two level 1 junctions is $(6 * l_x + 2) * l_{\text{base-lvl-0}} = (6 * l_x + 2) * (6 * l_x + 2) * l_{\text{base}}$ in that case. Consequently, for the generic case of level L junctions we see the exponential dependency: $(6 * l_x + 2)^L * l_{\text{base}}$ to calculate the half distance between the centers which is $l_{\text{base-lvl-L}}$ and which is then applied in the geometry calculations instead of l_{base}.

The way the different junctions are connected is the second difference in the construction of higher order tubes compared to the order 0 case. Instead of directly connecting neighboring nodes, an additional graph G_{connect} needs to be introduced, that is aware of the tuples of the nodes at the boundary of the junction. Because of the regular construction process, all junctions are structurally identical and, independent of the arm that is considered, the tuples at the boundary of the arms are equivalent at the corresponding positions between different arms except the leading entries. This means that there is only one G_{connect} for all connections that need to be determined and considered during the super sheet construction. Such a G_{connect} is shown in Figure 3.10 from two different perspectives. On the right side, it is rotated by 90° compared to its position between the involved junctions, while it is only slightly rotated on the left side. The outer shape of G_{connect} is a ring. One half of the nodes in this ring belongs to the left junction and the other to the right one.

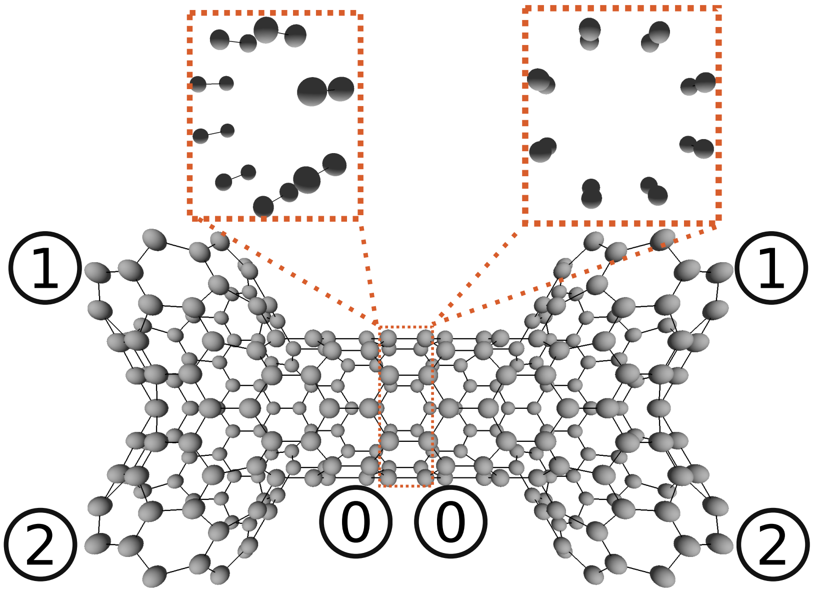


Figure 3.10.: G_{connect} within the first step of construction S_1^t.

The third change is also visible in Figure 3.10. In contrast to single atoms, the junctions are not axisymmetric and consequently their orientation matters. For the first construction step S_1^t, this means that the second junction needs to be mirrored before moving it along the horizontal axis and connecting it.

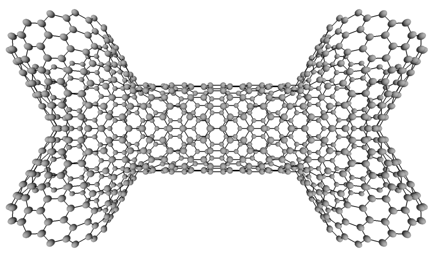
For the graph algebraic formula to calculate S_1^t these three differences result in:

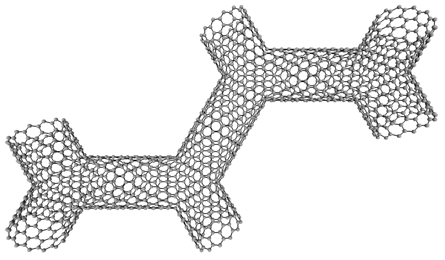
$$S_1^t = D_2 \otimes J_L \cup \upsilon(P_2 * D_1 * G_{\text{connect}}),$$

where J_L is the highest level junction element, in contrast to the empty graph J_0 for the order 0 case. The right side beside the $\cup$ establishes the connections between the two junctions. The usage of D_1 guarantees that the nodes at the boundary of both arms with internal number 0 are connected. Because of the mirroring, both 0-arms are opposed to each other. The numbering scheme for the arms is given in Figure 3.10 by the numbers in the circles.

The geometric transformation is split into two parts: The mirroring is realized by $p_x^{'} = (-1)^{t_{\text{val}}}$, which mirrors the copy of J_L with leading tuple entry 1 at the yz-plane, but the initial copy with leading 0 stays untransformed. Afterward, the translation in x-direction is performed by $p_x^{'} = 2 * l_{\text{base-lvl-L}} * t_{\text{val}} + p_x$.

The other stages S_2^t, S_3^t and S_4^t are adapted accordingly. There, no further mirroring occurs, but arms with different numbers have to be connected. For S_2^t, shown in Figure 3.11b, the arm 1 of the lower copy of S_1^t is connected to arm 2 of the upper copy. The Figures 3.11a and 3.11b demonstrate graphically that the first two stages of the tube construction process, S_1^t and S_2^t, lead to graphs that are structurally identical to S_1 and S_2 concerning the necessary amount of copied parts and their relations to each other.


(a) Graph S_1^t


(b) Graph S_2^t

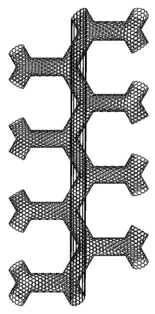

(c) Graph S_1^3

Figure 3.11.: First three steps for the construction of an order 1 sheet.

Figure 3.11c depicts the stage S_3^t. In contrast to S_3, the number of instances of S_2^t does not depend on the junction parameter d_x, but on the tube parameter d_0. The correlation of d_0 and the number of copies d_{tube} of S_2^t is simply $d_{\text{tube}} = d_0$. In the depicted case it is $d_{\text{tube}} = 4$. Again, the top is connected to the bottom, resulting in a structurally identical graph S_3^t compared to S_3.

The graph S_4^t, shown in Figure 3.12, represents a honeycomb grid formed by Y-junctions in which l_{tube} copies of S_3^t are aligned horizontally. There is again a direct connection of the number of copies l_{tube} and the input parameter l_0 that is $l_{\text{tube}} = \text{floor}\left(\frac{l_0}{2} - 1\right)$. As a consequence, it is also possible to express the number of junctions at the bottom of the sheet and on the right boundary in dependence of d_0 and l_0. The dotted blue rectangle at the bottom contains l_0 junctions, while the zigzag line on the right within the solid, red box consists of $2 * d_x$ junctions. Like for S_4, a cutting process removes some parts at the left and the right boundary to guarantee that only complete (super) hexagons are part of the final sheet. The nodes to be removed are surrounded by the orange rectangles.

We again analyze the distribution of the global indices for the individual tuples within the super sheet S_4^t before finalizing the construction, as we did for S_4 in the order 0 case. The principles are the same with the numbering starting at the lower left and growing from left to right in S_2^t, from bottom to top in S_3^t and from left to right in S_4^t. Additionally, the property can be added that the global indices within a junction form a closed set in the sense that the global index of the nodes contained within a junction are all smaller or all be greater than those in another junction. One main difference is that the node count within S_1^t and S_2^t is not constant anymore since the parameters (d_x, l_x) determine the junction size. This is particularly important in S_3^t when the copies of S_2^t are connected, since the distance between the global index of the nodes which are connected is not a constant value of 3 as in S_3, but may become arbitrarily large when increasing d_x and l_x.

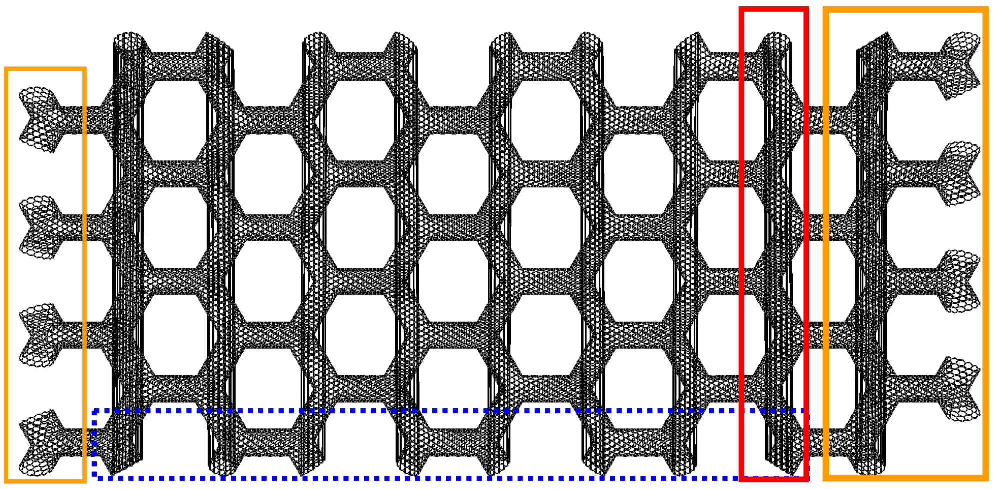
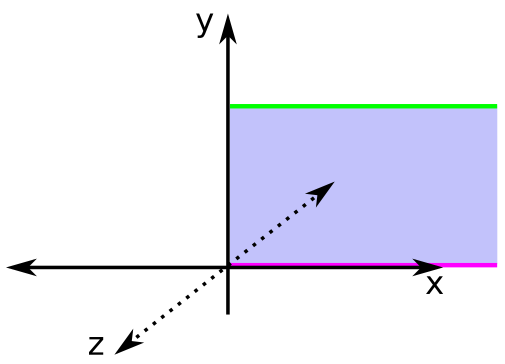


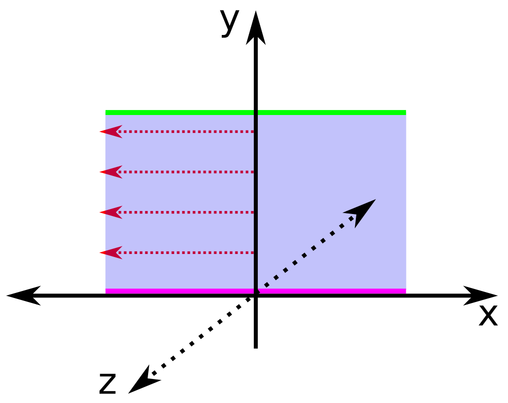
Figure 3.12.: The graph S_4^t, which is a super sheet for an order 1 tube with parameters $(d_x, l_x, d_0, l_0) = (2, 6, 4, 8)$. The solid, red rectangle visualizes the impact of d_0 and the dotted blue rectangle the one of l_0, respectively. Orange parts will be cut.

The last and main step for the geometry of the tube is to roll up the sheet around the x-axis. Here, the procedure for an order 0 sheet is discussed exemplarily, since it already contains all necessary information to comprehend the principles for the wrapping and demonstrate its elegance, but it is more straightforward to describe and visualize. The wrapping process for order 0 is divided into three steps that are discussed in the following.

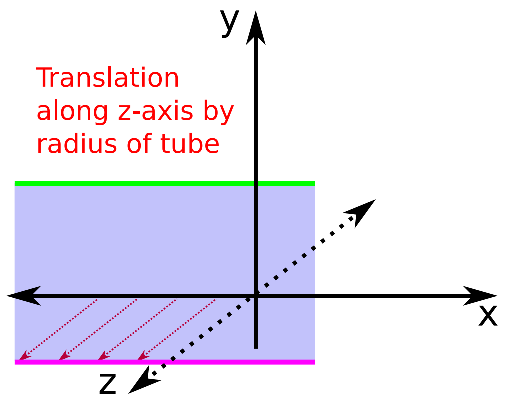
In its initial orientation, the left side of the sheet lies on the y-axis and its bottom on the x-axis. This is shown in Figure 3.13a. The axes are labeled at their respective end that is pointing to the positive values. The blue rectangle is a schematic for the honeycomb sheet. The thickness of the sheet is l_{base}. Its bottom (magenta) and its top (green) are colored to allow an easier tracking of the single movements during the wrapping.



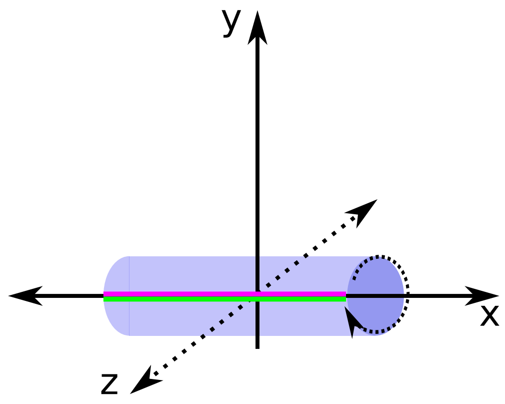
(a) The initial orientation of the sheet before starting the actual wrapping process.



(b) The sheet is moved along the x-axis to center it with the y-axis.



(c) The sheet is moved along the z-axis by the later radius of the tube.



(d) The sheet is rolled up around the x-axis and centered on the y-axis.

Figure 3.13.: The four basic geometric transformations to roll up a sheet around the x-axis.

Step 1: The first step is a translation along the x-axis that aligns the middle of the rectangle to the y-axis as shown in Figure 3.13b. To that end, the length in x-direction of the overall sheet Len_{tube} is calculated with the knowledge of the number of instances of S_3 that is l_{tube} and the base length, as it has already been demonstrated by $Len_{\text{tube}} = (l_{\text{tube}} * 6 - 1) * l_{\text{base}}$. To center the grid with the y-axis it has to be moved against the x-axis by $\frac{Len_{\text{tube}}}{2}$ as indicated in Figure 3.13b by the arrows.

Step 2: The next step is to move the grid along the positive z-direction. The distance by which it is moved is the later radius r of the tube. It is dependent on the total height Hei_{tube}, i.e., extent in y-direction, of the sheet that we can determine by $Hei_{\text{tube}} = (d_{\text{tube}} * 2 * \sqrt{3} - 1) * l_{\text{base}}$.

Step 3: In this final step, the now correctly aligned sheet is actually wrapped around the x-axis. Consequently, each node keeps its x-coordinate while the values for y and z may change. Hence, the problem is reduced to a 2-dimensional problem to map each discrete point $p = (y, z)$

on a line g to a point $p' = (y', z')$ on the orbit of a circle C with radius r and its center on the x-axis at $(p_x, 0, 0)$. The idea is visualized in Figure 3.14. For a line of length $2 * \pi$ and a unit circle, there is a straightforward solution. An angle α directly corresponds to the y-coordinate of the point on g and needs to be plugged into the following equation that defines the position $p' = (y', z')$ on the orbit:

$$y' = \sin(\alpha) \qquad\qquad z' = \cos(\alpha)$$

There are two green points in Figure 3.14 whose mapping is depicted. Point p_1 is the point where g contacts C as a tangent which is mapped to itself by the coordinate transformation. Point p_2 lies at a third of the length of g, resulting in the fact that the radian measure for this movement on the orbit is $\frac{1}{3} * 2 * \pi$. This corresponds to an angle of $\alpha = 120°$ that is shown in the circle C in Figure 3.14 in red.

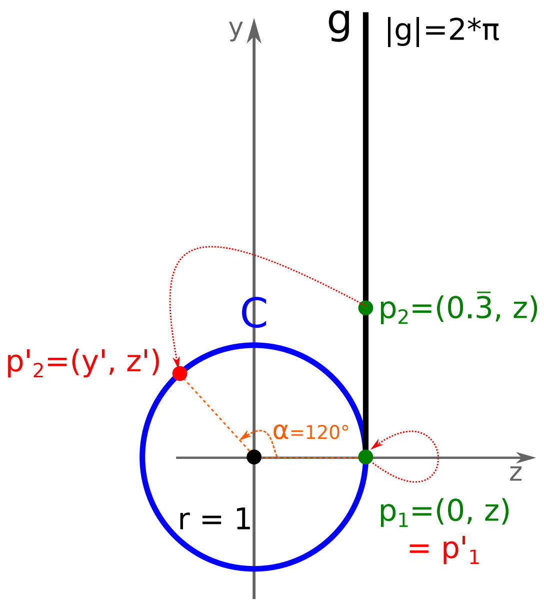


Figure 3.14.: Wrapping a line around a point on a circular path.

However, for the folding of super sheets there exist two additional problems. First, the radius r of the circle C is unequal to 1 and hence the circumference c_{circle} differs from the unit circle. Second, the length of g, $\left|g\right|$, is unequal to $2 * \pi$. As a consequence of the first problem, the radius of the resulting tube and the heights of the input sheet need to be synchronized. Since $c_{\text{circle}} = 2 * r * \pi$ it follows that $r = \frac{c_{\text{circle}}}{2*\pi}$ and that is exactly the way how r for the tube is determined. The height of the sheet can be calculated by the tube parameters and is equal to c_{circle} (length of the red arrows in Figure 3.13c) which solves the first problem. The approach to tackle the second problem is to map the length of line $\left|g\right|$ to the range $[0, 2 * \pi]$ which is equal to set the range of the angle α to $[0, 2 * \pi]$. This is done by calculating $\alpha = \frac{1}{r} * 2 * \pi$. It is $r = p_z$ and $p_{i_z} = p_{j_z} \forall p_i, p_j \in g$ since in the case of a non-unit circle its radius r has to be taken into account for the movement on the orbit. The mapping of points $p_i \in g$ to elements p'_i on the orbit of C is given by:

$$y' = z * \sin(\frac{1}{z} * y) \qquad\qquad z' = z * \cos(\frac{1}{z} * y)$$

The result is shown in Figure 3.13d. As visualized, the former magenta bottom line of the rectangle stays at its position all the time, while the former green line at the top nearly does a full move around the x-axis. Finally, it lies directly beneath the former bottom.

The actual wrapping process of a super sheet is indeed the same as for the graphene. The only thing to change is to plug in $l_{\text{base-lvl-L}}$ for the calculation of the super sheet's width and height and thus modifying the radius calculation of the tube.

To sum up this section: An SCNT model of order L is constructed within a two stage procedure that first iteratively constructs a Y-junction of level $(L-1)$ and combines these elements in a second step to the final tube. The shape and size of junctions are defined by a pair of parameters (d_x, l_x), while the diameter and length of the tube are determined by the parameters (d_0, l_0). Consequently, a specific tube of order L is unambiguously determined by the configuration:

$$(d_x, l_x, d_0, l_0)^L$$

which is the notation for this thesis. For order 0 tubes the configuration is abbreviated by (d_0, l_0) since d_x and l_x do not have an influence in that case. The most important parameters of all tubes employed in this thesis can be found in Appendix A, because depending on the focus of the different chapters, different tubes are suitable for highlighting important points.

3.2.3 Correlation of the construction process and the tuples

The last section mentioned all points within the construction step that change the length of the tuples which are required to identify a node within the structure. The construction of a base element increases the tuple length by five. Using these base elements to create a junction adds another three elements, resulting in eight new entries for the construction of a junction of level $L+1$ starting from one of level L. The final step, taking a junction and creating the tube, adds four additional entries. The general shape of a tuple of length m that represents an SCNT of order L can be summarized as shown by Equation 3.5.

$$\begin{array}{ll} \text{tube part} & \left\{ \underbrace{x_m\ x_{m-1}\ x_{m-2}\ x_{m-3}}_{\text{tube order L}} \right. \\ \text{junction part} & \left\{ \begin{array}{l} \underbrace{\underbrace{x_{m-4}\ x_{m-5}\ x_{m-6}}_{\text{topology junction arms level L}-1}\quad \underbrace{x_{m-7}\ x_{m-8}\ x_{m-9}\ x_{m-10}\ x_{m-11}}_{\text{topology base elements level L}-1}}_{junction\ level\ L-1} \\ \dots \\ \underbrace{\underbrace{x_8\ x_7\ x_6}_{\text{topology junction arms level 0}}\quad \underbrace{x_5\ x_4\ x_3\ x_2\ x_1}_{\text{topology base elements level 0}}}_{junction\ level\ 0} \end{array} \right. \end{array} \tag{3.5}$$

The first line represents the *tube part* of the tuple which is always four entries wide, while the remainder represents the *junction part* whose length is always a multiple of eight. It can be further divided into parts which code for a specific junction level. This regularity within the tuple system is exploited by the data structures to store the graphs, which is presented in Section 6.

3.2.4 Terminology of super carbon nanotube models

The regular construction process results in SCNT models with self-similar parts. A very important term in that context is a *ring*, which is a building block that appears recurrently. This is shown in Figure 3.15 for a $(1,4,8,8)^1$ tube with $22,528$ nodes. Each ring is colored differently and vertical lines indicate a partition. One particularity of a ring is that all nodes in it have the same leading tuple entry. Hence, we name the rings by the value of the leading tuple entries starting with the tuples with a 0 as leading entry from the left. Consequently, a k-leading ring is the abbreviation for «the circular part of the tube that contains all tuples with a k as leading tuple entry». The differently colored parts in Figure 3.15 can also readily be identified in the sheet. Although in that case the bottom and the top are not directly neighbored yet, the groups of nodes with the same leading tuple entry are also called rings for simplicity.

By convention, the tubes are always assumed to be aligned to the x-axis after the construction process has finished. The origin of the coordinate system lies in the middle of the tube and the negative x-coordinates are on the left side. We see that the parameter 8 for the tube length l_0 in $(1,4,8,8)^1$ results in $l = 5$ different rings. In general, there is the correlation $l = \text{floor}\left(\frac{l_0}{2} + 1\right)$ between l_0 and the number of rings as demonstrated in Section 3.2.2.

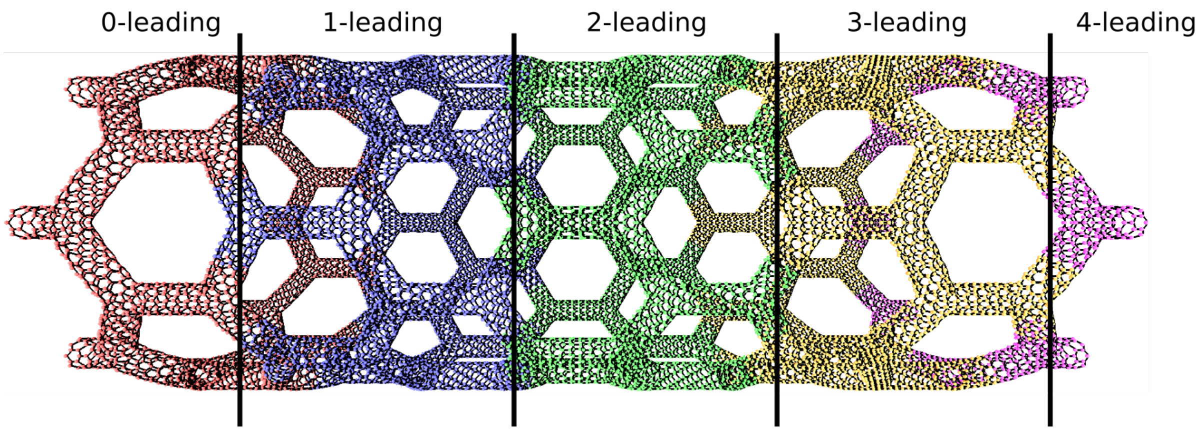
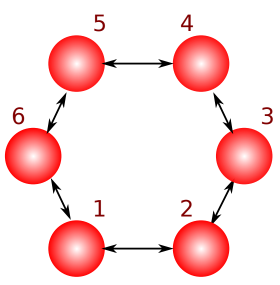


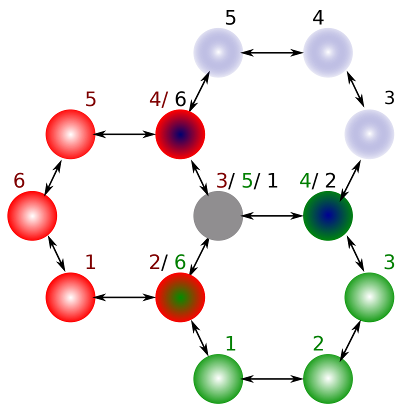
Figure 3.15.: An SCNT of order 1 with its different rings and their leading index.

3.3 Identifying symmetry and hierarchy

The last section mentioned that the rings are parts which reappear along the x-axis of the tube. This indicates a kind of translational symmetry. Recalling a single hexagon like shown in Figure 3.16a, it is possible to number all nodes starting from 1 up to 6. In that case, the lower left corner is chosen as start and the numbering is done counterclockwise. The node number 1 is connected to node 2 on its right side and with 6 on the upper left. Now, imagine that this hexagon is placed within a honeycomb grid as shown in Figure 3.16b. The three hexagons are colored differently and nodes belonging to more than one hexagon are drawn in the mixture of corresponding colors. Because of its connections to the neighboring hexagons the node 3 in the red hexagon is also part of both other hexagons. It has position 5 in the lower green one as well as position 1 in the blue hexagon at the top.



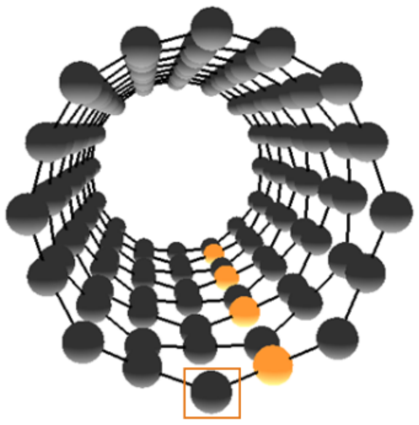
(a) Counterclockwise numbering of nodes within a hexagon.



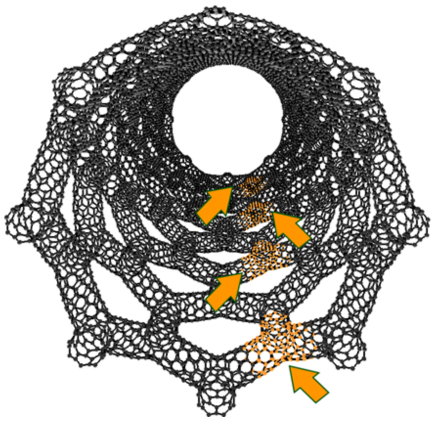
(b) Three hexagons and their numbered nodes within a honeycomb grid.

Figure 3.16.: Orientation of hexagons.

Now, assume that these three hexagons are part of the 1-leading ring. Then, there exist corresponding parts in the following rings as well, i.e., there is a corresponding node to the gray node that has the same relative position in the three corresponding hexagons it belongs to. This kind of repetition occurs in translational direction along the x-axis, since the tubes are aligned to this axis. This kind of self-similarity is called *translational symmetry*. For a whole tube the translational symmetry is visualized in Figure 3.17a for order 0 and in Figure 3.17b for order 1.



(a) Translational symmetry within an order 0 SCNT.



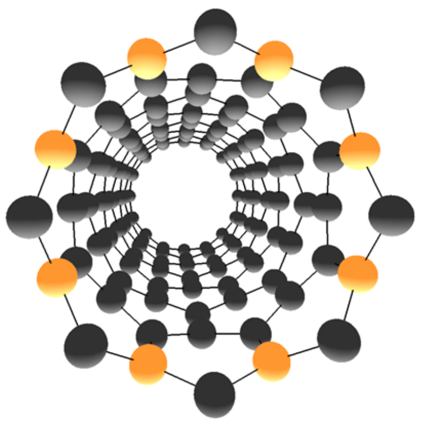
(b) Translational symmetry within an order 1 SCNT.

Figure 3.17.: Translational symmetry for order 0 and 1 tubes.

In the case of order 0, single nodes can be related by translational symmetry in the aforementioned sense that the relative position of a node within its adjacent hexagons is identical like for all the orange nodes in Figure 3.17a, lying on a line parallel to the x-axis. The first node in this line lying closest to the view point, which is equivalent to having the most negative x-coordinate, is called the *base-symmetry* node for translational symmetry. This is because all nodes following on the orange line are embedded in the same local structure of the grid. All other nodes on this orange line are the *translational symmetric* nodes or *symmetric* nodes if it is clear from the context that translational symmetry is referenced. Now, imagine that the orange line is moved clockwise by one node, so that it starts at the node which is surrounded by the rectangle afterward. Here, the first node in the line differs in its connectivity from all the others resulting in an anomaly. It is only connected to two other neighbors since it lies directly on the boundary of the tube. The same may happen at the other end of the tube. These nodes that lie in a line with symmetric nodes but possess a different number of incident edges are called *non-symmetric* nodes. The symmetric nodes can easily be identified by the tuple system: The translational symmetric counterpart of a node in a certain ring lies at the corresponding position in the other ring. Consequently, all symmetric nodes have the same value at all tuple entries except the leading one.

Figure 3.17b demonstrates that these principles are also applicable to tubes of higher order. In that case, it is additionally possible to define symmetric Y-junctions as they are highlighted in orange. They have the same positions and orientations within the super hexagons as their symmetric counterparts. Following the definitions in Section 3.2, they are called symmetric elements, base-symmetry elements and non-symmetric elements, respectively. All nodes within symmetric elements have the same value for the tuple entries $x_{m-1}, x_{m-2}, x_{m-3}$ concerning Equation 3.5. Furthermore, within two symmetric elements there exist symmetric nodes at the corresponding positions that again only differ in their leading tuple entry. Independent of the order of the tube, the amount of translational symmetry increases with the length of the tube. This can be seen in Figure 3.17a. If two additional rings are appended to the existing tube, the line of symmetric orange nodes will also receive two new members. The same happens for symmetric junctions when increasing the length of the tube in Figure 3.17b.

A second type of symmetry is visualized in Figures 3.18a and 3.18b. The pattern of hexagon orientation not only repeats when following the length of the tube along the x-axis, but this also happens when following the diameter around that axis. Recapitulating the numbering of nodes within hexagons in Figure 3.16b, we see that all orange nodes in Figure 3.18a belong to two different hexagons. They always occupy position 1 in one hexagon and position 5 in the other one. This type of symmetry is called *rotational symmetry*, since it can be resolved by rotating around the x-axis.


(a) Rotational symmetry within an order 0 SCNT.

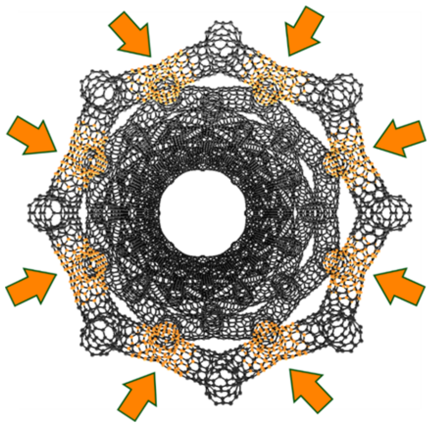

(b) Rotational symmetry within an order 1 SCNT.

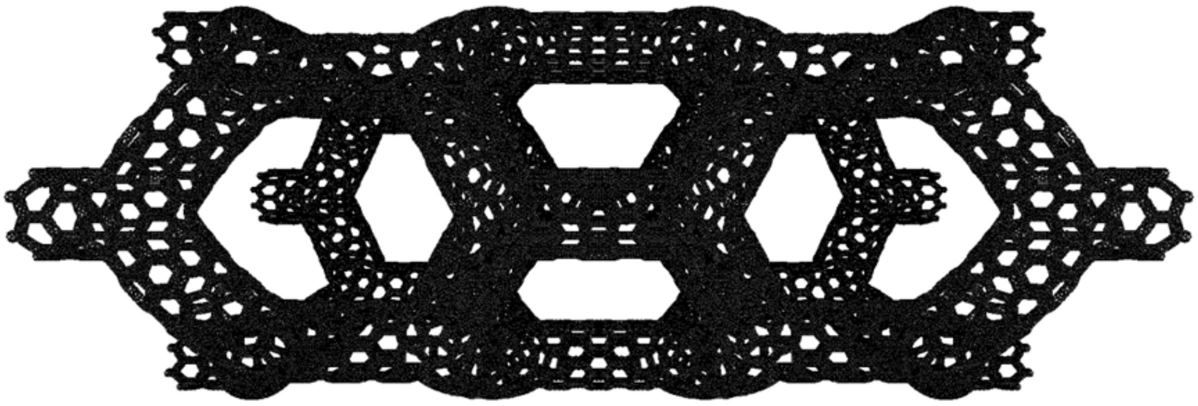
Figure 3.18.: Rotational symmetry for order 0 and 1 tubes.

As in the case of translational symmetry, it is possible to determine the rotational symmetry of two nodes by looking at their tuples. Since the second leading entry determines the placement of a node along the y-axis in the former sheet, it also determines the position on the circular arc around the x-axis. Several nodes lie on the same line around the x-axis of a tube if and only if they were stacked over each other in the original sheet. Therefore, rotationally symmetric nodes only differ in their second leading tuple entry. The base-symmetry node is defined as the symmetric node within the circular arc that has the lowest entry at its second leading entry.

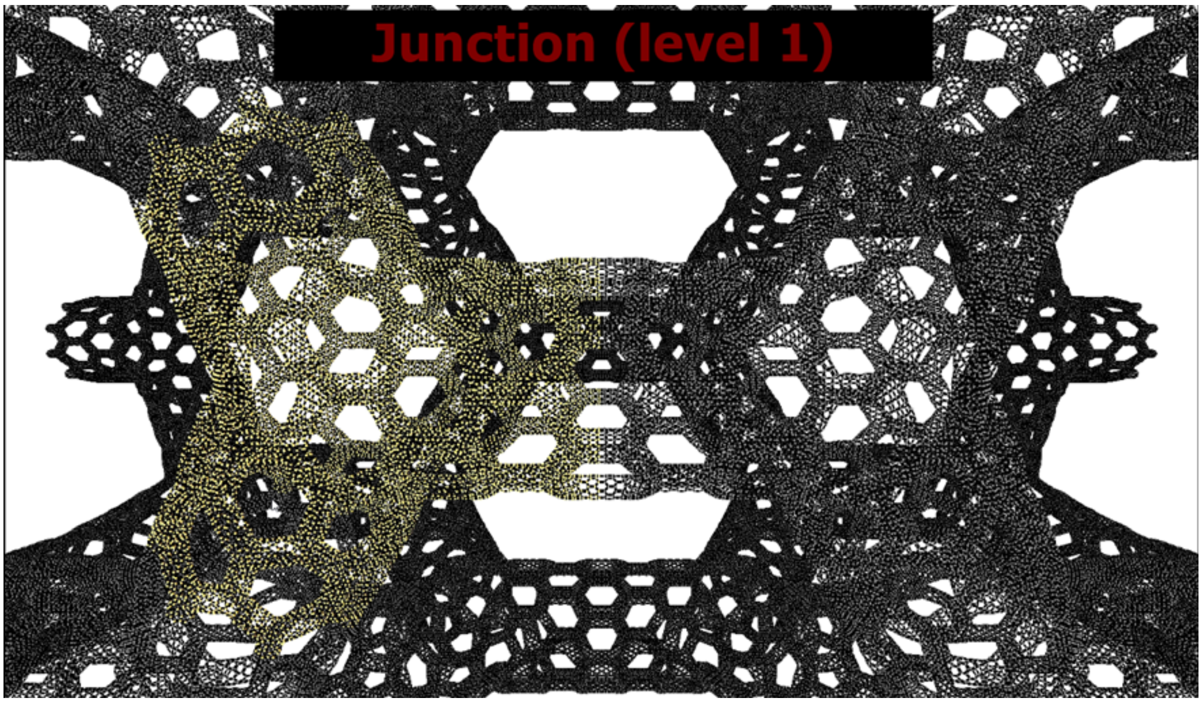
As in the translational case, the concept of rotational symmetry can be extended to whole symmetric elements as demonstrated by Figure 3.18b. All highlighted junctions and their adjacent ones have the same orientation compared to their symmetric counterparts.

These two kinds of symmetry are called *structural symmetry* within this thesis. This symmetry has the important property that it remains constant even if the tube is deformed, since it is based on the structure of the neighborhoods and the orientation of the hexagons. Assuming, for example, that the tube is stretched or twisted, then the aforementioned relations still hold. This is a very important property of SCNTs and is exploited in the implementation of the graph data structures and the solvers of this thesis.

Finally, we consider the hierarchy within SCNTs. The regular construction process leads naturally to a very hierarchical structure that can be traversed from top to bottom as shown for an order 2 tube by Figures 3.19a until Figure 3.19d.

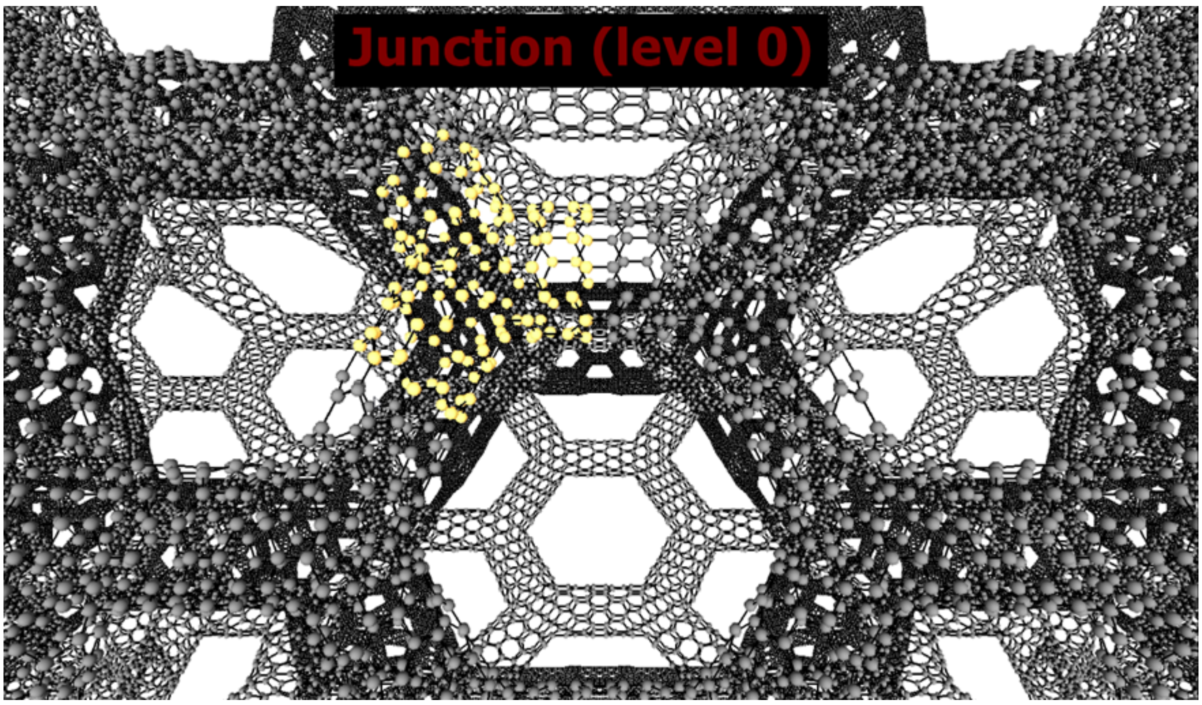


(a) Total view of an order 2 SCNTs.

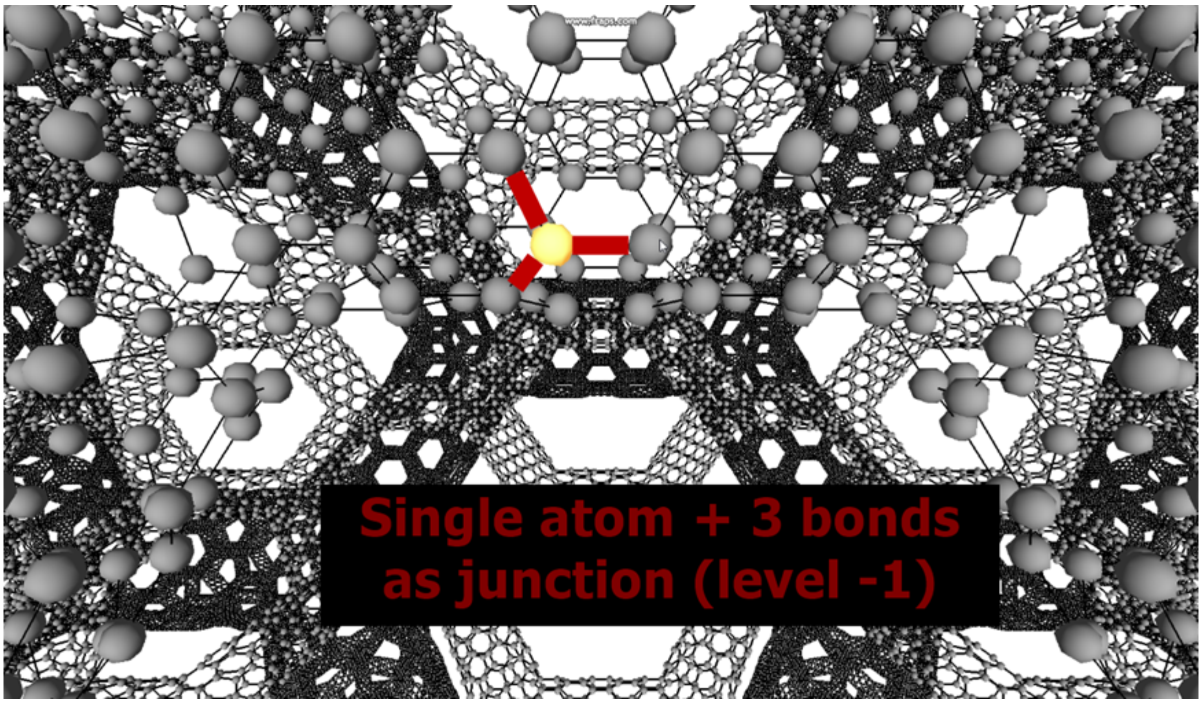


(b) Zoom to level 1 junctions.

The tube is composed of many identical level 1 junctions. They consists themselves of smaller level 0 junctions which are all equal among themselves. The lowest level is composed of the single atoms. The hierarchy can also be resolved within the tuple system. Recalling the tuple structure of Equation 3.5, the tube part of the tuple allows the determination of the positions of the highest level junctions within the tube in Figure 3.19a. Afterward, the junction part is processed level by level, so in blocks of length eight, to walk through the remaining levels of the hierarchy. Since each junction level has the same configuration, the tuple space within the different levels is structured and occupied very similarly, resulting in regularities within the distribution of the occurring tuples. This is another important property.



(c) Zoom to level 0 junctions.



(d) Zoom to the level of single atoms.

Figure 3.19.: Different levels of hierarchy in an order 2 tube.

4 Visualizing Super Carbon Nanotubes and the Result of Simulations

The framework developed in this thesis also includes the visualization of the simulated SCNTs. An early version of this visualization software was published in [22]. The visualizer employs the principle of instanced rendering. This enables it to cope with a large number of nodes within the graphs rendered. Visualizing the tubes allows visual debugging, i.e., errors within the construction process or implausible simulation results become obvious quickly without analyzing large vectors with position or connectivity data. The visualizer also allows to highlight structural properties like symmetry or hierarchy by exploiting the tuple system. Thus, the software presents another elegant application of the tuple system and shows that its use is not limited to the construction and simulation of SCNTs. The applicability of the included visualizer is demonstrated by the fact that all pictures and videos of SCNTs[1] for this thesis were generated with it.

4.1 Principles of instanced rendering

Instancing was integrated into the OpenGL-API in 2009 in version 3.3 [57, p. 128 - 139]. The technique allows to draw several elements with the same geometry with slight differences. A forest is a sample scenario that, in principle, consists of few different base trees that often reappear in the scene at varying orientation, scaling and shading. Instanced rendering enables a reduction of the OpenGL API-calls by separating the geometry of an object from its position and orientation in the scene. Thus, the creation of n instances of an object is possible by using one instanced draw-call. This drastically reduces the CPU load since OpenGL-API calls are costly[2]. The host only needs to pass the triangle mesh of the object itself and additional information like the position of the models, their material etc. in separate buffers to the video memory (VRAM) of the graphics card and not all triangles for each instance of an object separately when instanced rendering is employed.

The visualizer targets the rendering of graphs. Hence, it only needs to cope with two different kinds of objects: Spheres that represent the nodes of a graph and lines to draw the edges. This also motivates the decision to employ instanced rendering since only one model for spheres is used and repeated several million times within the scene. The graph is rendered in a two-step approach. First, the connections between the nodes are drawn as `GL_LINES`, a primitive within OpenGL. The second step adds the spheres modeled by a triangle mesh. Model transformations are applied within GLSL-vertex-shader (OpenGL shading language) programs by the GPU (for

[1] Some examples can be found online: `https://www.youtube.com/channel/UCxKMz5tvGWFjMIYdyB0-_9Q`
[2] The same holds for DirectX.

GLSL refer to [58]). The GLSL code conforms to version 3.3, which allows all graphics cards compatible with the OpenGL 3.3 specification to execute the renderer (this version is required to use instanced rendering anyway). The GLEW library [3] is employed in the 64 bit version to create, compile and bind the shader programs.

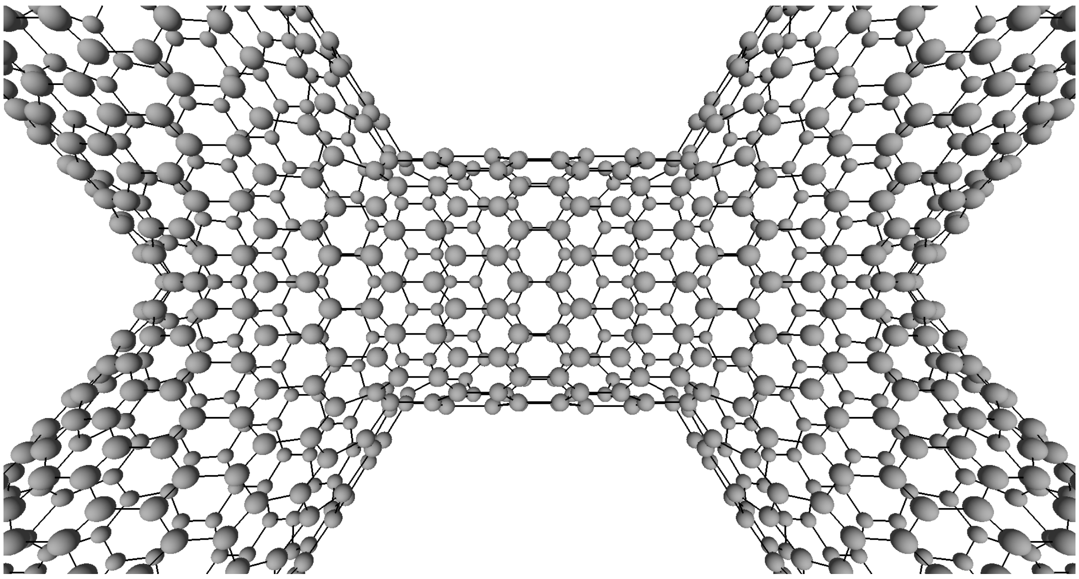
Work load is moved from the CPU to the graphics card by the usage of instancing combined with hand written shader code. This makes sense, since even the integrated graphics chips of modern mobile processors offer enough performance to render relatively large SCNT models with the presented renderer (see Section 4.3). The calculation of illumination is also done by shading programs. There are two different shaders for the two distinct components of the model. The one for spheres is based on the Phong-shading-model [59], while the lines are drawn in a predefined color. Phong-shading consists of three different contributions of illumination called ambient, diffuse and specular with increasing complexity for their calculation in this order. The latter two contributions can be deactivated in the shader programs to improve the performance on slower graphics cards.

4.2 Features of the visualizer

The main feature of the visualizer is an adaptive management of model quality. The average frame rate is measured for a fixed time-interval. If it decreases under a predefined threshold, the quality of the sphere models will be reduced. During its initialization, the visualizer reads the sphere model in five different levels of detail starting from 15 triangles up to 720 which correspond to five different quality levels of the renderer with quality level 5 employing the sphere model consisting of 720 triangles. Thus, a change in rendering quality only involves the replacement of the small triangle model in the graphics memory, while all other information, i.e., edges, colors or transformations can be kept. Rendering of spheres can be skipped at all for slow graphics cards or very large graph models and only the edges are drawn. This kind of rendering is called quality level 0. Figure 4.1 compares quality level 5 with quality level 1 and demonstrates that even level 1 is sufficient to identify the geometry and to investigate the SCNT models.

The visualizer also allows an arbitrary rotation of the scene and a zooming to parts of interest. It can also highlight several parts or properties of interest like specific elements within the tube, symmetry or hierarchy relations between elements (see for example the figures in Section 3.3) or nodes that are influenced by boundary conditions during the simulation by exploiting the tuple system. The tuple system is a very elegant way to identify such properties without creating additional information or analysis. Additionally, the visualizer is capable of exporting graph

[3] http://glew.sourceforge.net


(a) Part of an order 1 graph rendered in quality level 5.

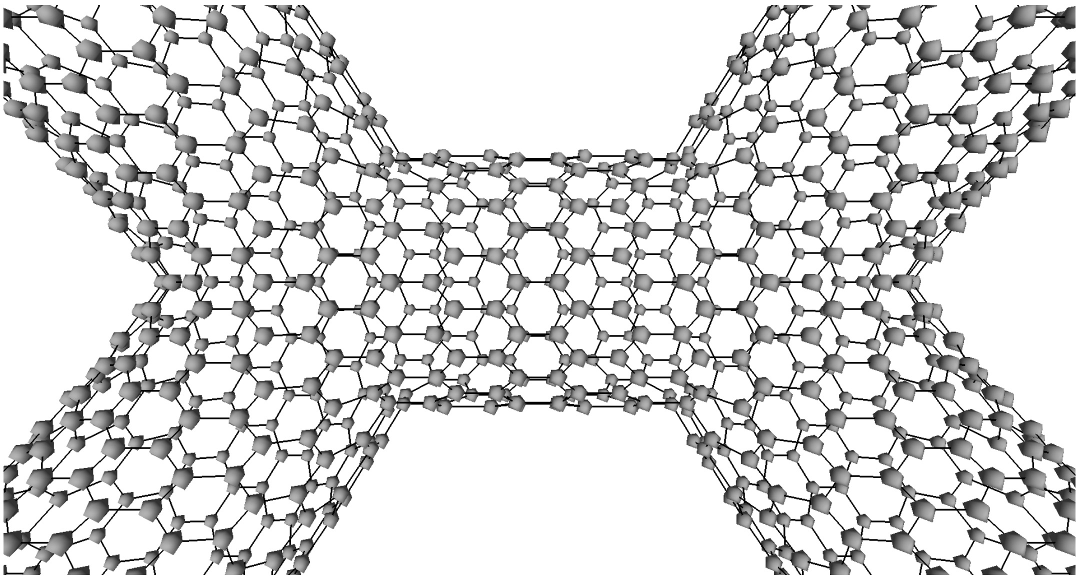

(b) Part of an order 1 graph rendered in quality level 1.

Figure 4.1.: Comparison of two different renderer quality levels.

models to the vtk-format that is, amongst other formats, processable by the tool ParaView[4]. This software may be employed for further investigation of the structure.

Finally, the visualizer can cope with multiple simulation steps and display them successively allowing the creation of simulation videos.

4.3 Rendering performance

Two tubes are visualized on two different graphics cards, the mainstream desktop card RX480 with 8 GB VRAM from AMD and the integrated Intel HD4000 in the mobile CPU i7-3520M, which represents the lower bound of available graphics solutions, to evaluate the performance of the renderer. The renderer is configured to color the rings in the respective tubes in different colors to enable all of its features. The scene is visualized in FullHD resolution and the camera is oriented in that way that both ends of the tube have contact to the boundary of the screen, so that all nodes are in the field of view. The frame rate is calculated as the median number of frames per second (fps) over 15 seconds and a minimum number of about 10 fps is judged sufficient for comfortable use of the renderer. Table 4.1 shows the performance results.

Table 4.1.: Summary of the frames per second achieved for two tubes at varying quality levels L_0 to L_5.

level 1 $(1,11,10,8)^1$ **tube**, $8.2 * 10^4$ atoms						
	L5	**L4**	**L3**	**L2**	**L1**	**L0**
RX480	60.7	166.8	251.6	480.0	829.2	3098.4
HD4000	7.1	23.5	51.1	74.3	125.8	203.9

level 2 $(1,3,6,6)^2$ **tube**, $1.2 * 10^6$ atoms						
	L5	**L4**	**L3**	**L2**	**L1**	**L0**
RX480	4.6	13.3	18.8	38.4	72.7	511
HD4000	0.6	1.7	4.1	6.2	13.5	28.4

The RX480 has no problems for the smaller tube at the different quality levels at all and even the HD4000 can cope with the second highest quality level. For the large $(1,3,6,6)^2$ tube, the RX480 reaches the target of 10 fps from the second highest quality level on. It has to cope with $8.5 * 10^8$ triangles on quality level 5 and reaches nearly 5 fps while on quality level 4, there are still $2.8 * 10^8$ triangles that have to be processed. But the scene can be also be investigated with the HD4000 at a lower quality level which would be sufficient for visual debugging and checking whether the geometry is plausible.

Altogether, the integrated visualization tool allows a convenient investigation of the models even on slow graphics cards without the need of writing data to disk or to use additional tools. The properties of the tuple system are exploited to enable structure related highlighting that simplifies the evaluation of the models. The renderer can also be employed to render general

[4] http://www.paraview.org/

graphs when its interface is used to transfer the graph data. This enables its integration in other codes as well. The code is available upon request.

5 Atomic-Scale Finite Element Method

Structural analysis of SCNTs involves the solution of equation systems with the so-called stiffness matrix. One contribution of this thesis is the development of a matrix-free approach to solve this system: The stiffness matrix is employed as operator in an iterative solver but need not be assembled. A detailed description of the overall numerical approach and of the atomic-scale finite element method can be found in [34]. The following Section 5.1 describes the algorithm on a high abstraction level. Mechanical and mathematical details of the method are explained in Sections 5.2 and 5.3 focusing on those properties which motivate a matrix-free approach and enable an efficient parallelization of the resulting implementation. Matrices are represented by capital letters in bold (e.g. $\mathbf{A}$, $\mathbf{K}$)[1], vectors in bold lower case letters (e.g. $\mathbf{x}$, $\mathbf{b}$) and scalars by lower case letters (e.g. n, α).

5.1 The flow of the algorithm

This section briefly summarizes the overall procedure that is applied to simulate the mechanical behavior of SCNTs. The pseudocode for the general method which is employed in the simulations is shown in Algorithm 1.

Algorithm 1 General simulation algorithm

```
 1: for s=1, STEPS do
 2:     Prepare next step
 3:     for i=1, MAX do
 4:         instantiate equation system
 5:         solve equation system
 6:         if CONVERGENCE then
 7:             stop inner loop
 8:     end
 9:     update global values
10: end
```

This procedure is referenced as the Newton-Raphson method in the following and it is able to solve a non-linear system of equations.

We see two nested loops which have a predefined maximum of steps called *STEPS* and *MAX*. The outer loop iterates over the single simulation steps. If loads are applied to the tube that would heavily deform it, they must be applied step by step. Otherwise the solution of the

[1] An exception are the element stiffness matrices introduced in the following with lower case bold letters to be conform with the literature.

arising equation system might fail because of numerical instabilities. Consequently, the second line determines the changes in the loads for the next step, before the inner loop starts. Here, a system of linear equations is instantiated depending on the actual positions of the atoms within the model and solved in the following statement. Afterward, it is checked whether the result satisfies a predefined convergence criterion. If the criterion is fulfilled, the resulting deformation of the tube will be integrated in the actual atom positions and the next step can be processed. If the result is not accurate enough, the equation system will be re-instantiated with consideration of the calculated atom movements for the current step so far. So, in general, the resulting movements from the inner loop become smaller with each iteration for a feasible configuration. In principle, the maximum number of iterations for the inner loop *MAX* is optional if the calculation converges, but helpful for non-converging calculations.

In this thesis, we mainly ignore the outer loop and only instantiate and solve the linear equation system within the inner loop, i.e., we are not dealing with a non-linear problem. The exceptions are explicitly referenced as multi-step simulations where a non-linear equation system is solved.

5.2 Governing equations and linearization

Some short introductions to the employed simulation algorithm have already been given in [23] and [27] and we employ the definitions and nomenclature of these publications. We describe the whole Newton-Raphson method enabling the solution of non-linear equation systems also we only instantiate the linear equation system in the inner loop once and solve it unless stated otherwise.

The models of the simulated SCNTs are composed of n indexed nodes. Each node is associated with its spatial position $\mathbf{x}_i$ which results from an embedding of the respective material points $\mathbf{X}_i$. These current positions of the nodes are determined by the sum $\mathbf{x}_i = \mathbf{x}_i^0 + \Delta\mathbf{x}_i$, of their initial position $\mathbf{x}_i^0$ and the actual displacement of the node $\Delta\mathbf{x}_i$, resulting from inner and outer forces acting on the nodes. There are three degrees of freedom for every node which result from the displacements of the nodes in the three spatial directions.

The internal energy U^{int} is 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. The residual vector $\mathbf{r}(\mathbf{x})$ is given by the negative of the first derivative of the total energy U, which is the difference of the external forces $\mathbf{f}(\mathbf{x})$, e.g., some external tension at the ends of the tube, and the first derivative of the internal energy U^{int}. 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, we obtain the Taylor expansion $\frac{\partial U}{\partial \mathbf{x}}(\mathbf{x}) = -\mathbf{r} + \mathbf{K}\Delta\mathbf{x} + o\left(\|\Delta\mathbf{x}\|\right)$ with

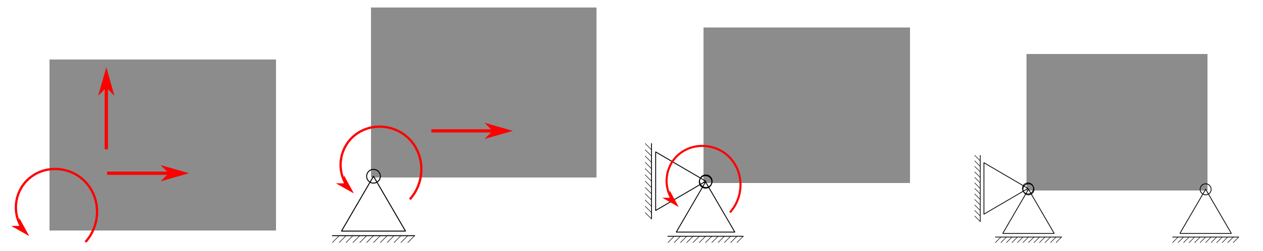
$\mathbf{K} := \mathbf{K}\left(\mathbf{x}^0\right)$ and $\mathbf{r} := \mathbf{r}\left(\mathbf{x}^0\right)$. Thus, an estimate for $\Delta\mathbf{x}$ can be obtained from solving the linear system of equations

$$\mathbf{K}\Delta\mathbf{x} = \mathbf{r} \tag{5.1}$$

This estimate is also called the solution of the linearized minimization problem. In the most cases within this thesis we are concerned with the calculation of the displacement $\Delta\mathbf{x}$, i.e., a single iteration step of the Newton-Raphson algorithm is performed. Nevertheless, there are some exceptions, as already mentioned in Section 2.3, where several steps were calculated within the dockSIM framework.

As U^{int} is invariant with regard to rigid body motions and $\mathbf{K}$ singular, suitable boundary conditions must be incorporated into the system of Equation (5.1) to regularize $\mathbf{K}$ and to obtain a unique estimate for the displacements.

We want to consider the case of performing a relaxation on an SCNTs for a closer look on the boundary conditions. From a mechanical point of view, the problem comes from the fact that the SCNTs can move as a reaction to the load without deformation. It is important to pin some of the nodes to prevent such unwanted movements. This problem is shown for the simplified 2*D* case in Figure 5.1a through Figure 5.1d. A rigid rectangle can move in the xy-plane in x- and y- direction and can be turned by a turning movement as indicated by red arrows in Figure 5.1a. The number of possible movements is reduced to two by employing one floating bearing on the left side, i.e., movements in one translational direction are prevented but still possible in the remaining translational direction and turning movements are still allowed as shown in Figure 5.1b. If now the same node is additionally secured against translational movements in x-direction by a second floating bearing, as done in Figure 5.1c, only the turning moment remains for the plane. This last possibility can be removed by applying a third floating bearing at the right side as demonstrated in Figure 5.1d. In general, three floating bearings are required to prevent unwanted movements in the 2D case [60, p. 117-121].



(a) Rigid body in the plane without bearing.

(b) Rigid body in the plane with one floating bearing.

(c) Rigid body in the plane with two floating bearings.

(d) Rigid body in the plane with three floating bearings.

Figure 5.1.: Correct applications of bearings to immobilize a plane against rigid body movements.

Now, the analogous problem needs to be solved for SCNTs in three dimensions. In that case, not only three movements are possible but there are six in total with three possible translations (one along each axis) and three turning moments. Six different bearings are required to appropriately mount the tube as visualized in Figure 5.2. Here, one bearing attached to the node at the upper left mounts the respective node against movements in all three spatial directions. The opposite node at the bottom left is not allowed to move in y-direction. Finally, a third bearing at the top right node prevents movements in y- and z-direction. This is one possible combination of bearings to avoid unwanted rigid body motions, resulting in a regular stiffness matrix.

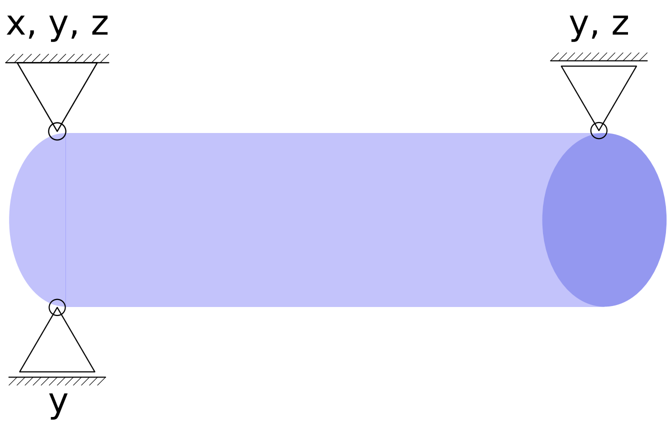


Figure 5.2.: Appropriately mounted SCNT with six bearings to prevent rigid body movements.

The boundary conditions resulting from mounting the tube are given as Dirichlet conditions $\Delta\mathbf{x}_i = \bar{\mathbf{u}}_i$, which prescribe the displacement of material points $\mathbf{X}_i$. Mathematically, this means that the corresponding lines and unknowns in the matrix $\mathbf{K}$ are decoupled from the remainder of the equation system. Assume we want to set a Dirichlet condition for the second degree of freedom for the node with index 1 within the model to simulate, i.e., a bearing in y-direction is applied as depicted in Figure 5.2 on the bottom left.

This procedure is shown in Equation 5.2. The first three rows of the matrix and in the residual vector belong to the node with index 1 and the second line to its second degree of freedom. These relations are explained in detail in the next section. The corresponding entry in the residual vector is set to zero to prescribe a movement of 0, i.e., fix the node in y-direction, Additionally, all entries in the corresponding row and column of the matrix are set to 0 except the entry in the second column which is set to 1. This guarantees that the second entry in $\Delta\mathbf{x}$ is evaluated to zero during the solution process and, thus, there is no movement of these nodes in this direction.

$$\underbrace{\begin{pmatrix} k_{1,1} & 0 & k_{1,3} & k_{1,4} & k_{1,5} & k_{1,6} & \cdots & k_{1,n-2} & k_{1,n-1} & k_{1,n} \\ 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ k_{3,1} & 0 & k_{3,3} & k_{3,4} & k_{3,5} & k_{3,6} & \cdots & k_{3,n-2} & k_{3,n-1} & k_{3,n} \\ \vdots & \vdots & & & \cdots & \cdots & & & \vdots & \vdots \\ k_{n-2,1} & 0 & k_{n-2,3} & k_{n-2,4} & k_{n-2,5} & k_{n-2,6} & \cdots & k_{n-2,n-2} & k_{n-2,n-1} & k_{n-2,n} \\ k_{n-1,1} & 0 & k_{n-1,3} & k_{n-1,4} & k_{n-1,5} & k_{n-1,6} & \cdots & k_{n-1,n-2} & k_{n-1,n-1} & k_{n-1,n} \\ k_{n,1} & 0 & k_{n,3} & k_{n,4} & k_{n,5} & k_{n,6} & \cdots & k_{n,n-2} & k_{n,n-1} & k_{n,n} \end{pmatrix}}_{\text{stiffness matrix } \mathbf{K}^{n\times n}} \times \underbrace{\begin{pmatrix} x_1 \\ \mathrm{x}_2(=0) \\ x_3 \\ \vdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{pmatrix}}_{\text{displacement } \Delta\mathbf{x}^n} = \underbrace{\begin{pmatrix} r_1 \\ 0 \\ r_3 \\ \vdots \\ r_{n-2} \\ r_{n-1} \\ r_n \end{pmatrix}}_{\text{residual } \mathbf{r}^n} \tag{5.2}$$

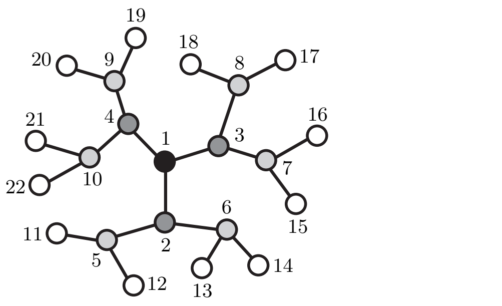
Neumann conditions $\mathbf{f}_i$ are the second type of boundary conditions that correspond, in this context, to the external forces. We assume that the external forces acting on a given node are invariant with regard to deformations of the structure. One example of external forces is tension. In that case, a movement for specific nodes, e.g. at the boundary, is prescribed by adding the value to the respective entries of the residual $\mathbf{r}$.

5.3 Implementation in the context of the finite element method

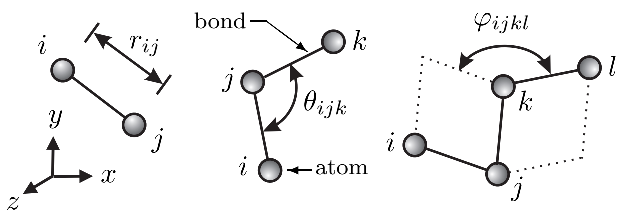
The formalism of the finite element method (FEM) is employed to construct the global variables, namely the stiffness matrix $\mathbf{K}$ and the residual vector $\mathbf{r}$. Element stiffness matrices $\mathbf{k}^{\mathrm{e}}$ and element residual vectors $\mathbf{r}^{\mathrm{e}}$ are computed and assembled into the global variables. The total internal energy U^{int} is given as the sum of the internal energies $U^{\mathrm{int,e}}$ of the elements.

The internal energy is exclusively based on the bonded interaction forces modeled by the Dreiding potential, a generic force field introduced in [38], while non-bonded interactions, such as van-der-Waals forces, are ignored as well as interactions involving atoms located more than three atomic bonds apart from each other. The minimum number of edges separating two nodes n_i and n_j in the graph is denoted by the distance $d\left(i,j\right)$. A node n_j is said to belong to the *neighborhood* Ω_i^{e} of n_i if $d\left(i,j\right) \leq 3$. For a grid in which each node has at most three outgoing edges, like it is the case for the carbon grid within an SCNT model, this means that each node has at maximum three different neighbors, resulting in the structure for neighborhoods that is shown in Figure 5.3a. A local index j' for the 22 nodes in the neighborhood of a *reference node* n_i is introduced giving rise to a function $\sigma : \left(j', i\right) \mapsto j$, i.e., σ returns the global index of node n_j that lies on the local index $j' \in [1, 2, \ldots, 22]$ in the neighborhood of the reference node n_i.

The number of nodes in the neighborhood of nodes lying on the boundary of the grid may be lower. If all possible 22 neighboring nodes exist for a certain node n_i, i.e., each of the 22 local indices in Figure 5.3a is assigned to a global index, the neighborhood of n_i is called *complete*. Consequently, all other neighborhoods are called *incomplete neighborhoods*. Following the terminology of [34, p. 980], the nodes $j' = [2, 3, 4]$ are the first, $j' = [5, \dots, 10]$ are the second and $j' = [11, \dots, 22]$ are the third neighbors, respectively.



(a) The neighborhood Ω_i^e of the reference node i (local index $j' = 1$) contains all the atoms present in the finite element (taken from [34]).



(b) The three atomic kinematics of the Dreiding potential. Note that, in this context, $i := n_i$, $j := n_j$, $k := n_k$, $l := n_l$ (taken from [34]).

Figure 5.3.: Details of the Dreiding potential.

These neighborhood objects correspond to the finite elements of the atomic-scale FEM (AFEM). Hence, there are n finite elements e_i with i identifying the reference node in an SCNT with n nodes. Each finite element e contributes one element stiffness matrix $\mathbf{k}^e$ to the global matrix $\mathbf{K}$, and one element residual vector $\mathbf{r}^e$ to $\mathbf{r}$. The components of the element stiffness matrix are computed as sum of the second derivatives of the 2-, 3- and 4-atom interaction energies with regard to the displacement of the reference node n_i and the displacements of the other up to three nodes n_j, n_k, and n_k that may participate in these interactions. The interaction energies are given by the Dreiding potential already mentioned. Figure 5.3b shows the three types of interactions of the Dreiding potential and their respective kinematic values: the bond length r (2-atom interaction), the valence angle θ (3-atom interaction) and the dihedral angle φ (4-atom interaction). The internal energy $U^{\text{int,e}}$ related to one element e is given by the sum of Equation 5.3 [34, Eq. 20]. The potential functions are defined in Table I on p. 978 of [34] and omitted for brevity.

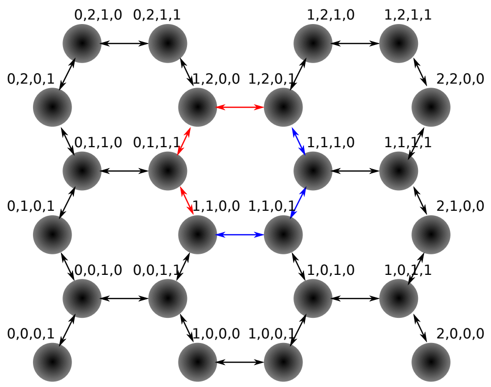
$$U^{\text{int,e}} = -f_1 * \Delta x_1 + \underbrace{\sum_{\mathscr{L}^e_{1j}} V_{1j}(r_{1j})}_{\text{bond length}} + \underbrace{\sum_{\mathscr{L}^e_{1jk}} V_{1jk}(\theta_{1jk}) + \sum_{\mathscr{L}^e_{i1k}} V_{i1k}(\theta_{i1k})}_{\text{valence angle}} + \underbrace{\sum_{\mathscr{L}^e_{1jkl}} V_{1jkl}(\varphi_{1jkl}) + \sum_{\mathscr{L}^e_{i1kl}} V_{i1kl}(\varphi_{i1kl})}_{\text{dihedral angle}} \tag{5.3}$$

Here, $-f_1$ denotes the external force that is applied to the reference node n_1. It can be seen that there are several instances of bond length calculations within the sum. The $\mathscr{L}^e_{ij[kl]}$ symbols denote small sets of possible configurations for the different summands of the Dreiding potential. One very important fact of the finite element approach of Wackerfuß is that although the finite elements overlap, i.e., nodes of Ω_i^{e} may also be contained in Ω_j^{e} of another node n_j, the calculation of the potential assures that contributions are not considered multiple times. This is reached by only taking the influence of the neighboring nodes in Ω_i^{e} on reference node n_i into account and not vice versa ([34, p. 981]). So, there are exactly three possible bond length calculations within the finite element. With the local node indexing this results in the list: $\mathscr{L}_{1j}^{e} = \{(1,2),(1,3),(1,4)\}$ ([34, p. 984]), with the reference node always being the first entry. Two cases have to be distinguished for the valence angle. In the first case, the reference node n_1 is the start of the chain with length three, resulting in six possible contributions to the valence angles like $(1,2,5)$ or $(1,4,10)$. In the second case, there are three valence angles where n_1 lies in the middle of the chain: $\mathscr{L}_{i1k}^{e} = \{(2,1,3),(3,1,4),(4,1,2)\}$. For the dihedral angle there is also the case that n_1 is the first node in the resulting 4-node chain with $\left|\mathscr{L}_{1jkl}^{e}\right| = 12$ and the second case that it is the second node in the chain with $\left|\mathscr{L}_{i1kl}^{e}\right| = 12$. This gives 36 possible contributions to the sum of the Dreiding potential in total. It must be determined which contributions actually are present for each element e (for the complete list see ([34, Eq. 24/25]).

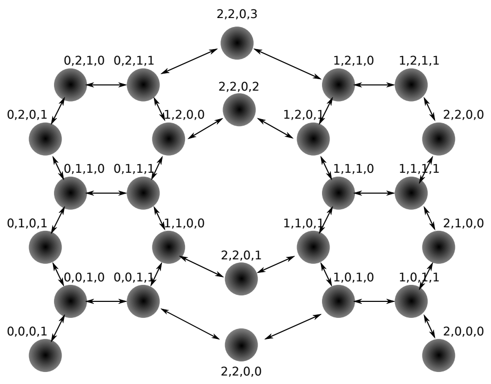
Consequently, each contribution $\mathbf{k}^{\mathrm{e}}$ to the stiffness matrix related to a finite element e with reference node n_i with its local index 1 inside the finite element is defined as the second derivative of the Dreiding potential for this element: $\mathbf{k}^{\mathrm{e}} = \frac{\delta^2 U^{\mathrm{int,e}}}{\delta \mathbf{x}_1 \delta \mathbf{x}_\beta}$ with $\beta = [1,\ldots,22]$, which gives: $\mathbf{k}^{\mathrm{e}} = \left[\mathbf{k}_{1,1}^{e} = \frac{\delta^2 U^{\mathrm{int,e}}}{\delta \mathbf{x}_1 \delta \mathbf{x}_1}, \mathbf{k}_{1,2}^{e} = \frac{\delta^2 U^{\mathrm{int,e}}}{\delta \mathbf{x}_1 \delta \mathbf{x}_2}, \cdots, \mathbf{k}_{1,22}^{e} = \frac{\delta^2 U^{\mathrm{int,e}}}{\delta \mathbf{x}_1 \delta \mathbf{x}_{22}}\right]$. Each of these 22 terms $\mathbf{k}_{1,j'}^{e}$ represents a 3×3 matrix which is called *stiffness contribution* in the following. Hence, in total $\mathbf{k}^{\mathrm{e}}$ is a $3 \times (22 * 3)$ rectangular element stiffness matrix that we call *stiffness contribution line* or just *contribution line* if it is clear that the stiffness matrix is referenced.

As a result of these definitions, only those 3×3 blocks in $\mathbf{K}$ corresponding to a $\mathbf{k}_{i,j'}^{e}$ with $\sigma\left(j',i\right) \in G$ can assume non-zero values. Thus, most entries in $\mathbf{K}$ vanish and no row may consist of more than $3 * 22$ non-zero values. All summands needed for a given component of the global stiffness matrix $\mathbf{k}_{i,j'}$ are present in the same element stiffness matrix $\mathbf{k}^{\mathrm{e}}$. Therefore, the parts of the global stiffness matrix associated with the respective elements are mutually independent. This is an important property which allows an efficient parallel realization of the matrix-free solver. Additionally, all boundary conditions considered in this thesis can be readily included at the element level and thus, do not alter the validity of the results of the aforementioned exposition with regard to the computation and assembly of the stiffness matrix.

Some global indices appear in several positions of the neighborhood, i.e., more than one local index j' may be associated with a global index j as the function σ is generally not injective for the case of SCNT graphs. Hence, the assembling algorithm has to avoid double counting of contributions. To that end, the neighborhoods store a vector which contains indices of the neighborhood nodes but removes duplicates. As stated above, the algorithm limits non-zeros per row to 66, but the actual number is still somewhat lower. Figure 5.4a demonstrates this fact. The neighborhood of node $(1,1,0,0)$ shows that there are two paths that lead to node $(1,2,0,1)$ which are highlighted in red and blue. Consequently, $(1,2,0,1)$ appears two times in the neighborhood. The same holds for the nodes $(0,1,0,1)$ and $(1,0,0,1)$ which can be reached by two paths from $(1,1,0,0)$. All other nodes can only be reached by one path. Altogether, the neighborhood of $(1,1,0,0)$ consists of 19 different nodes. This is the case for all nodes with a complete neighborhood in a honeycomb grid that do not lie at the boundary of the grid.



(a) An excerpt of a honeycomb grid with tuple-based node labeling. Two paths of length 3, marked in red and blue, lead from $(1,1,0,0)$ to $(1,2,0,1)$.

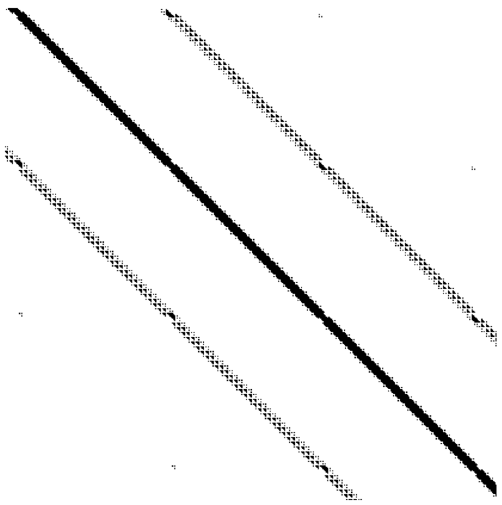


(b) An excerpt of a honeycomb grid containing an irregularity with tuple-based node labeling. 20 different nodes lie in the neighborhood of node $(1,1,0,0)$.

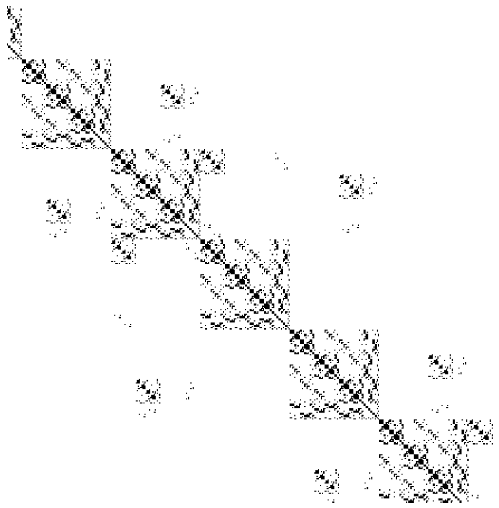
Figure 5.4.: Nodes appearing several times within the same neighborhood.

The level 0 junction elements are a special case. As demonstrated in Section 3.2.1, connecting different junction arms results in irregularities within the hexagonal grid and three octagons are introduced per junction. A schematic grid for such a situation is shown in Figure 5.4b. Focusing again on $(1,1,0,0)$, we see that $(0,1,0,1)$ and $(1,0,1,0)$ can be reached by two different paths of length 3, while all other nodes are reachable from $(1,1,0,0)$ via a unique path. From this follows that nodes adjacent to an octagon may have up to 20 different nodes in their neighborhood. This is also the upper limit within all possible SCNTs, since even the topology of the smallest possible junction prevents that a node can be adjacent to more than one octagon. Hence, no row can have more than $20 * 3$ non-zero entries within the stiffness matrix of an SCNT model. Furthermore, these octagons prevent a regular stencil to run over the model which would be conceivable for a regular honeycomb sheet of order 0.

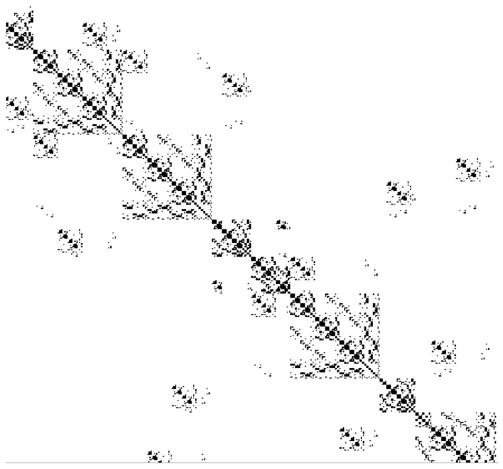
Finally, we consider the structure of the stiffness matrix itself. To that end, Figures 5.5a - 5.5d plot the non-zero patterns resulting from tubes of different order reaching from 0 up to 3. Each point in these plots represents a 3×3 stiffness contribution whose position in the matrix corresponds to the global indices of the two involved nodes. In all cases, the picture is zoomed to the diagonal of the matrix because otherwise all pictures would be just white or look like a single diagonal because of the high sparsity and large total size, like, e.g., $7 * 10^7$ rows for the matrix of Figure 5.5d.



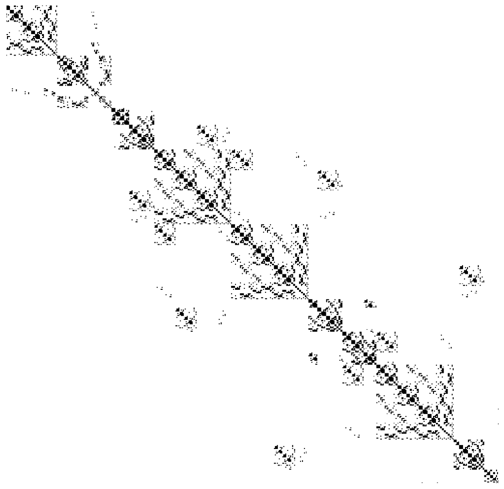
(a) Sparsity pattern of stiffness matrix K for order 0 tubes, zoomed to diagonal.



(b) Sparsity pattern of stiffness matrix K for order 1 tubes, zoomed to diagonal.



(c) Sparsity pattern of stiffness matrix K for order 2 tubes, zoomed to diagonal.



(d) Sparsity pattern of stiffness matrix K for order 3 tubes, zoomed to diagonal.

Figure 5.5.: Sparsitiy pattern of the stiffness matrix for tubes of orders 0 - 3.

Mainly three bands are noticeable for the matrix resulting from a tube of order 0 that represent the distance of nodes belonging to one neighborhood. The upper and lower bands can be explained by the line of the original grid where top and bottom nodes are connected to create the tube. Reference nodes lying at the former top of the sheet have a global index that is considerably higher than those of the bottom nodes. The main parameter that influences the structure of the bands is the chosen diameter d_0 for the tube. A higher diameter increases the distance between the global indices at the top and at the bottom causing a larger gap between the entries directly around the diagonal of the matrix and those within the two bands.

The matrices for the tubes of order 1, 2 and 3 look very similar near their diagonal and reflect the high self-similarity and hierarchy of SCNTs also on the matrix level. The pictures can not show hat the number of entries lying farther away from the diagonal increases with the order. This also means that the entries are spread more widely over the rows of the matrix. In the higher order case, the configuration of the junction elements also becomes relevant. As discussed in Section 3, the maximum distance between the global indices of connected nodes grows with the parameters d_x and l_x of the junctions since only one junction can be the direct neighbor to another one in terms of global indices. Thus, high values for (d_x, l_x) also increase the grade of distribution of the non-zero entries within the stiffness matrix.

The order in which the global indices of the neighbors of a reference node are assigned to the local indices $[1, 2, \ldots, 22]$ is not identical to the arrangement of their respective column positions from left to right in the matrix for tubes of all orders. This means that processing the contributions in the order of the neighborhood results in many jumps within the rows of $\mathbf{K}$, which is not amenable to performance in multicore environments.

6 Graph Data Structures

In this chapter, we develop the underlying graph data structures which are employed within the simulation framework presented in this thesis. We demonstrate how they exploit the symmetry and hierarchy in the SCNT graphs, thus enabling a memory-efficient storage even of large graphs and providing fast access to the information stored.

In principle, the creation of an SCNT graph model can be divided into two different steps. The first one is the actual *graph construction phase* in which the geometric algebra operations are performed that were described in Section 3. From the data point of view the node and edge set need to be stored in what we call **NodeMap** and **EdgeMap** and the graph algebra operations can be performed on these sets. The main challenge during the construction phase is to assign each node, or to be more exact each tuple, a unique index value to avoid repeating storage of tuples and to uniquely identify the nodes. This index is called the *serial index*. It is used to access the nodes within the **NodeMap** and their position data. Additionally, the edges are represented as a pair of two serial indices with the first one identifying its start node and the second one its end node. In terms of implementation, the serial index is the key for the registered values within various map structures. Although it is possible to directly employ vectors that represent the tuples as keys as each tuple is unique, measurements in the following Section 6.1.3 demonstrate that this option leads to a drastic slowdown of the program performance. Moreover it has a high memory overhead which makes this approach infeasible. Hence, it is required to find a mapping $M : \mathbb{N}^m_{\geq 0} \to \mathbb{N}_{\geq 0}$ of the tuples to a corresponding serial index.

The mapping must have several properties to be suitable for the application. First of all, the serial index assigned to a tuple must be unique to avoid conflicts within the data structure. Additionally, the mapping should be resolvable with low effort to allow efficient access to the data. The maximum size of the serial indices is determined by the range of unsigned 64 bit integer values, i.e., there must be no case where the mapping leads to indices exceeding this range. In that context, the term *index space* should be defined. It includes all values that lie between the lowest and the highest serial index which are assigned by the mapping M. If one of the values within this range is actually used by M as index for a tuple, this index is called *occupied*. The *top index* of an index space is determined by the highest occupied index.

We complete the description of the SCNT model creation process before discussing the details about possible mappings. Additional information has to be created after the graph construction phase finishes to enable an efficient simulation based on the graphs. This second phase is called *data creation phase*. Its first task is the assignment of the *global index* already defined as a second type of indexing for actually existing and registered nodes. In contrast to the serial index, it is always dense and has a range from $[1, 2, \ldots, n]$ with n being the number of nodes in the graph. This procedure is called *flattening* since the nodes are also identifiable by this flat global index which cannot directly be mapped back to the original tuples afterward. The global index is only set for the final tube and reflects the relation of nodes to the entities, i.e., rows and columns of vectors and matrices within the solution process of the equation system. Finally, neighborhoods are constructed for all nodes ending the data creation phase of the SCNT model.

In the following Section 6.1, three types of mappings for tuples are explored. They are integrated in three different graph classes and implement a common interface for graph data types. The mappings employed differ in their computational requirements, their top index in the index space and their binding to the structure of SCNTs. Section 6.2 demonstrates how the serial indices can be combined with the knowledge about SCNT models to implement a compact and efficient container for the edges of the graph.

6.1 Different approaches to map tuples

This section introduces the **TreeGraph** (Section 6.1.1) and the **HashGraph** (Section 6.1.3) as two graph data structures that internally employ two different kinds of mappings $M : \mathbb{N}^m_{\geq 0} \to \mathbb{N}_{\geq 0}$ with a generalizable approach. They are called *the tree-based flattening* and *perfect spatial hashing*. Section 6.1.2 presents the **IndexGraph** as a third option which uses a custom-tailored mapping. The three approaches are compared in Section 6.1.4 regarding their top index and their access performance to the information.

6.1.1 Tree-based flattening (TreeGraph class)

A straightforward way to map a tuple to a serial index is to treat the tuples as leaves of a tree, which is based on the tuple extent of the respective graph. The tuple extent is computed as a by-product of the graph construction phase. This concept called tree-based flattening is realized in the **TreeGraph** class. Each entry of the tuple extent corresponds to one level of this tree. The number of branches to the next level is determined by the entry in the extent at the respective position. The tuple extent is processed from the leading entry to the lowest entry. So, the first level of the tree corresponds to the leading entry, while each leaf corresponds to one tuple. The set of all leaves contains all tuples of the tuple space. The intermediate nodes are labeled with values which the tuple can attain at the respective entry. All leaves are numbered with a *leaf number* from left to right. This leaf number corresponds to the serial index assigned by **TreeGraph** . The number of leaves can directly be calculated by the product of all entries of the tuple extent. These concepts are visualized in Figure 6.1 for an exemplary tuple extent of $(3, 3, 4)$. Only some possible paths and leaves are drawn for reasons of clarity and comprehensibility. Nevertheless, it is obvious that every node on level i has the same number of successors on level $i + 1$.

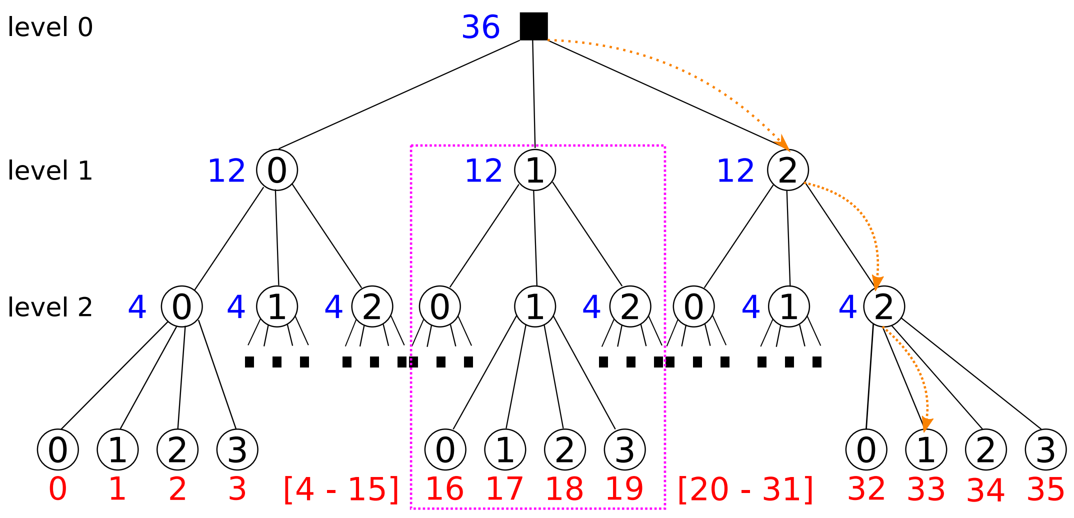


Figure 6.1.: Tree corresponding to tuple extent $(3, 3, 4)$. Serial indices for the leaves are shown in red below them, while the blue numbers beside the intermediate nodes give the number of leaf nodes in the sub-tree, rooted at the respective intermediate node. The orange arrows visualize the mapping of the tuple $(2, 2, 1)$ to its serial index 33.

Calculation of the serial index can be imagined as following the corresponding path in the tree and looking up the leaf number for the respective tuple after the tube is constructed. This procedure is shown for tuple $(2, 2, 1)$ and its index 33 in Figure 6.1 by orange arrows.

The indexing trees are built implicitly by creating the required information to serialize a tuple. This information mainly includes the number of leaves that belong to a sub-tree within the whole tree. This is also indicated in Figure 6.1 by blue numbers on the left side of some intermediate nodes. One sub-tree with 12 leaves and its root in level 1 is highlighted by dotted magenta lines for illustration. It is sufficient to store a single value per level in the array subtree_size because of the constant number of successors for nodes within one level.

The idea behind the calculation of the serial index is comparable to the linearization of a multidimensional array: Assuming, for example, a 3-dimensional array A[size_dim_2][size_dim_1][size_dim_0], the corresponding linearized index for the access A[val_dim_2][val_dim_1][val_dim_0] using a 3-vector can be calculated by val_dim_0 + val_dim_1 ⋆ size_dim_0 + val_dim_2 ⋆ size_dim_1 ⋆ size_dim_0. The function to map a graph tuple tpl to its serial index is shown in Listing 6.1. In case of flattening a tuple, the array subtree_size automatically contains the equivalent to the products size_dim_i ⋆ size_dim_i–1 ⋆ . . . ⋆size_dim_0 of the array case. In this way, their recalculation for every tuple can be avoided since they represent the number of leaves in the sub-trees of a respective level (see again blue numbers in Figure 6.1). The required memory can be neglected since even for a tube of order 3 only 28 values need to be stored.

```
long TreeGraph::serializeTuple(const vector<int>& tpl) {
long val_len = tpl.size();
long serial_idx=0;

for(long i=val_len-1; i >= 0; i=i-1) {
  // search entry for this level
  // work from top to bottom
  int val = tpl[i];

  // calculate the number of elements in other sub-trees that must be neglected
  serial_idx += (val * subtree_size[i]);
}
return serial_idx;
}
```

Listing 6.1: Procedure to calculate the serial index for a tuple tpl within **TreeGraph**.

This demonstrates that the tree-based flattening is very space-efficient. Additionally, the calculation is very fast since it mainly consists of several summations and product calculations. Read-accesses to both arrays tpl and subtree_size are consecutive in backward direction and are only dependent on the loop index i. This fact and the short length of the arrays enables an efficient prefetching of values by the compiler. The serial indices created by the tree-based method also reflect the lexicographic order defined on the tuples. This avoids time-consuming sorting or index-mapping operations. Moreover, this mapping is also able to preserve symmetry relations as shown, for example, by all sub-trees starting at level 1 in Figure 6.1. Following the same paths after a different root node (e.g., always following the rightmost path) leads to tuples with the same remainder. This would represent translational symmetry between different rings for a tree with 4 levels corresponding to a CNT.

However, the main drawback of the tree-based flattening is also visible in Figure 6.1. The tree always contains all tuples that can be created within a given tuple extent, i.e., also all unoccupied tuple positions. Thus, for sparsely occupied tuple spaces, a large index space is reserved. Assume a graph with two connected nodes $(0,0,2)$ and $(2,0,0)$ within the extent $(3,3,4)$. They are mapped to the serial indices 2 and 20, although there are no other nodes. Comparable situations may occur, e.g., if the right and the left side of the tree are removed by cutting operations on the graph, resulting in large unoccupied regions in the index space. This can cause the generation of indices for large SCNT models which exceed the range of 64 bit integers and lead to the failure of the indexing scheme.

The method presented is similar to parts of the algorithm for indexing statistical datasets from Ng and Ravishankar [61]. They encode records $r = (a_1, a_2, \dots, a_n)$ in database tables as tuples of integer values and apply the mapping process $\phi(r) = \Sigma_{i=0}^{n}\left(r[i] * \prod_{j=0}^{i-1} \left|A_j\right|\right)$, where A_j is the highest a_j at position j in all records r, to define a lexicographic order on their data-tuples. This leads to identical indices as the tree-based flattening. Their approach goes one step further and partitions the tuples into smaller groups and only encodes differences between tuples within those groups. However, these steps are database oriented and cannot be directly applied to the tuple-based graphs.

6.1.2 IndexGraphs

The **IndexGraph** (IG) is a graph data type which is able to reduce the range of serial indices by up to several orders of magnitude compared to tree-based flattening because of its structure-tailored tuple-to-index mapping. Recalling the correspondence of the entries x_i of a tuple with length m to the topology of an order L tube and the junctions at the different levels from Equation 3.5 in Section 3.2.3, the tuple can be divided into a tube part and a junction part as again shown in Equation 6.1.

$$\begin{array}{ll} \text{tube part} & \left\{ \underbrace{x_m \;\; x_{m-1} \;\; x_{m-2} \;\; x_{m-3}}_{\text{tube order L}} \right. \\ \text{junction part} & \left\{ \begin{array}{l} \underbrace{x_{m-4} \;\; x_{m-5} \;\; x_{m-6} \;\; x_{m-7} \;\; x_{m-8} \;\; x_{m-9} \;\; x_{m-10} \;\; x_{m-11}}_{\text{junction level L−1}} \\ \dots \\ \underbrace{x_8 \;\; x_7 \;\; x_6 \;\; x_5 \;\; x_4 \;\; x_3 \;\; x_2 \;\; x_1}_{\text{junction level 0}} \end{array} \right. \end{array} \tag{6.1}$$

The mapping of **IndexGraph** takes into account that the occupation of the index space for the junction part is very sparse, while the space of sub-tuples representing the tube part is relatively densely. Hence, both parts are treated differently. The hierarchy of SCNT models is also taken into account.

Figure 6.2 summarizes the principle of the serial index calculation by the **IndexGraph**. The left side shows how the sub-tuple for the tube is treated with tree-based flattening as described in the previous section. This is a good choice here because of the high density of the tuple space for the tube part. In the depicted case, the tube consists of four junctions indicated by the red leaves.

The right side of Figure 6.2 shows how the junction part is processed. It is divided into the different junction levels within the hierarchy, and each sub-tuple of length 8, corresponding to a junction level, is coded into a distinct map. In the following, for illustration purposes, we consider the tree on the left side of Figure 6.2 as an implicit tree which represents a junction level, although it only contains four and not eight tree levels and the map structure which is appended to the leaves needs to be ignored. Nevertheless, the principles are the same for a tree with a depth of eight. The procedure for each junction level works as follows:

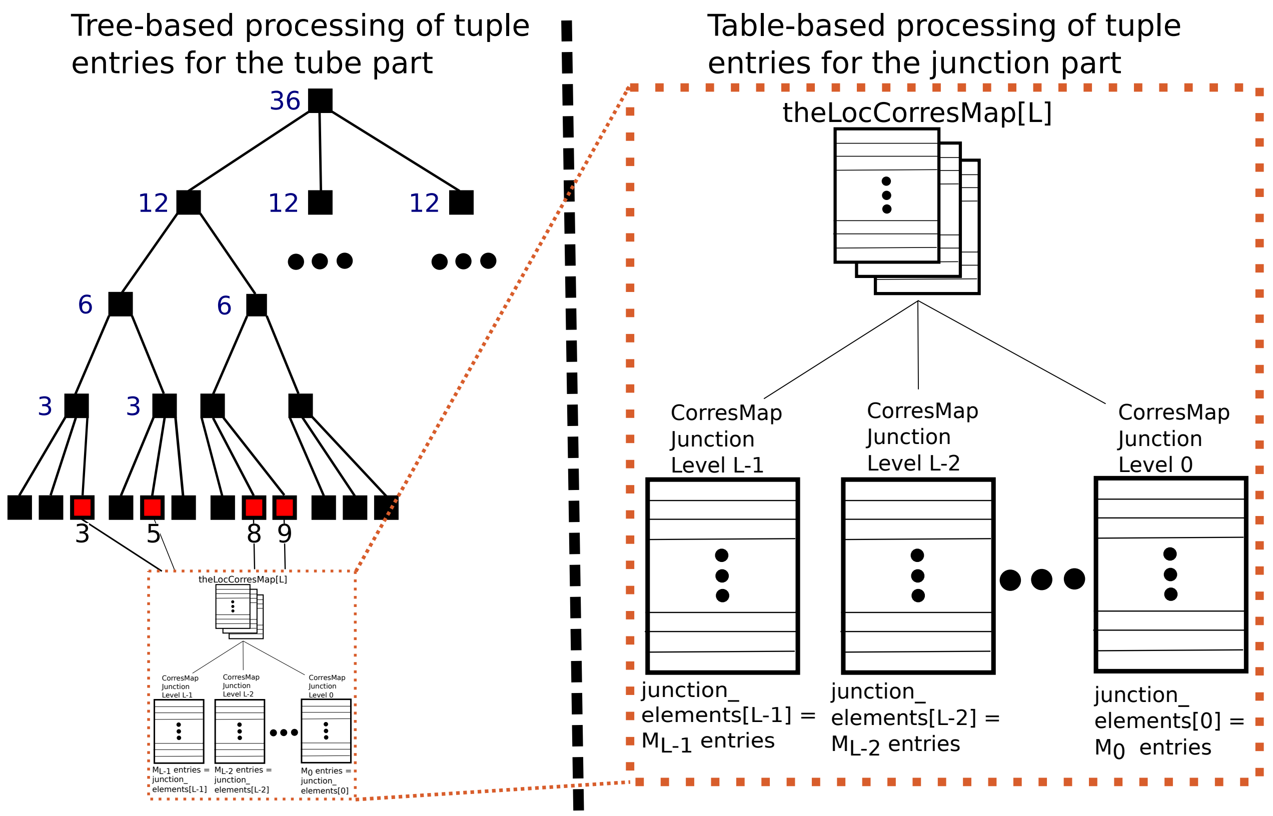

Tree-based processing of tuple entries for the tube part
36
12
12
12
6
6
3
3
3
5
8
9
Table-based processing of tuple entries for the junction part
theLocCorresMap[L]
CorresMap Junction Level L-1
CorresMap Junction Level L-2
CorresMap Junction Level 0
junction_ elements[L-1] = M_{L-1} entries
junction_ elements[L-2] = M_{L-2} entries
junction_ elements[0] = M_0 entries

Figure 6.2.: The left side shows the tree-based indexing for the tube part of the SCNT. The right side indicates that the junction part is coded within a hierarchy of maps corresponding to the distinct junction levels.

First, an implicit tree representing the tuple space for the sub-tuples of a certain junction level is created and for each sub-tuple a leaf number is assigned, which is called local_ser_idx, because it is only unique within the junction level considered. The local_ser_idx never causes an overflow, because of the upper bound of 8 for the number of tree levels and the fact that the extent of the corresponding sub-tuple does not contain high values. Also pay attention that we have two different kinds of levels here: The level of the junctions resulting from the construction process, and the level within the implicit trees from the root to the leaves.

Second, all n' actually existing tuples are identified. They are numbered by a consecutive index from $1 \ldots n'$ starting from the left, i.e., the implicit order determined by the local_ser_idx is kept. This correspondence of the local_ser_idx and the consecutive index is stored in a table for each junction level separately. This results in an array of L local correspondence maps called theLocCorresMaps for a tube of order L which is shown on the right side of Figure 6.2. They are realized as hash maps. This map then contains four entries for the depicted tree: $3 \rightarrow 1, 5 \rightarrow 2, 8 \rightarrow 3$ and $9 \rightarrow 4$.

The number of its entries is for each map stored in an array called junction_elements, also shown in Figure 6.2. Now, it is possible to look up a dense index for each element of a junction level. These maps are of relatively small size compared to the overall number of nodes in SCNTs. They typically contain several hundred up to one thousand elements. The correspondence map for level 0, for example, for the $(2,6,8,8)^2$ tube with 24 million nodes has 434 entries and the map for level 1 has 680 entries. This corresponds to 0.002 % and 0, 003 % of the total nodes, respectively. The situation is comparable for a tube of order 3 with configuration $(2,3,4,5)^3$ and 240 million nodes, i.e., one order of magnitude more nodes than the previous case of order 2: 182 entries ($7.6 * 10^{-5}$ %) for level 0 and 350 ($1.4 * 10^{-4}$ %) for levels 1 and 2. Therefore, the additional memory overhead caused by the maps is negligible.

The hierarchy and self-similarity of the SCNTs can be exploited by the fact that, in principle, all junction levels are of the same structure with some differences in level 0. In practice, this means that only the correspondence maps for level 0 and 1 need to be constructed while the latter one can be reused for all following levels. In total, the calculation of the serial index requires L look-ups in the small correspondence maps for an SCNT of order L, resulting in equal calculation time for all contained tuples in **IndexGraph**.

If both sides of Figure 6.2 are instantiated, the serialization of a tuple, i.e., the calculation of the serial index ser_idx for the whole tuple, works as shown in Listing 6.2. The local_ser_idx of the sub-tuple is calculated for each junction level by a call of function calculateSerialIndexPerLevel (line 8), starting from junction level 0. A factor is calculated (line 10) which is the product of the number of elements within each junction level covered so far. Here the local_ser_idx is translated to the corresponding consecutive index by a look-up in the correspondence map. Now, the different consecutive indices per level are combined again comparable to the linearization process of a multidimensional array. Each junction level corresponds to one of the dimensions of that fictive multidimensional array in the case presented. The intermediate results are summed up in the variable junc_idx which, after the last iteration of the **for**-loop of line 6, contains the index representing the contribution of the junction part of the tuple tpl to the overall serial index ser_idx.

The resulting index for the junction part is minimal since the index space of each junction level is “compressed“ to the minimum value by the correspondence maps and only those minimum values are combined. The tube part is serialized by tree-based flattening in line 16 of Listing 6.2. It is again assumed that these two indices correspond to a two-dimensional array that needs to be linearized to combine the tube and the junction index, resulting in the serial index for the input tuple within the **IndexGraph** (line 17).

IndexGraph also preserves symmetry information like the tree-based flattening does. This is shown on the left side in Figure 6.2 where all leaves of the tree are connected with the box. This stands for the following: The tree-based flattening for a CNT has the tuples of nodes on its leaves and thus indexes single atoms. In contrast, the implicit tree in **IndexGraph** manages and sorts junction elements. Junctions that only differ at the level 1 entry in the tree are symmetric by translation, while those only differing at the second level entries in the tree are symmetric by rotation. Their serial indices follow the lexicographic order of the tuples as another result of this. Depending on the actual step of the algorithm the structure may look differently during

```
long IndexGraph::serializeTuple(const vector<int>& tpl) const {
  // Get the first part that results from the junctions of different levels
  long junc_idx = 0; // Index contribution of the junction part to the overall
      serial index
  long local_ser_idx = 0; // Serial index of sub-tuple for a junction level
  long factor = 1;
  for(int lvl=0; lvl < scnt_level; lvl++) {
    // Find leaf number of the corresponding sub-tuple
    local_ser_idx = calculateSerialIndexPerLevel(tpl, lvl);
    // How many elements were covered so far
    factor = factor * junction_elements[lvl];
    // Look up consecutive index and integrate it into linearization procedure
    junc_idx += theLocCorresMap[lvl][local_ser_idx] * factor;
  }

  // Get the contribution to the serial index that results from the tube part
  unsigned long long tube_idx = calculateSerialIndexForTube(tpl);
  unsigned long long ser_idx = junction_elements[scnt_level]* factor * tube_idx +
      junc_idx;

  return ser_idx;
}
```

Listing 6.2: Calculation procedure of a serial index for a tuple tpl within IndexGraphs.

the construction process with its different intermediate graphs. The tree on the left side may reach a depth up to eight since this is the length of a sub-tuple that encodes a junction. No correspondence map is created until a full junction is constructed. Otherwise, a step-by-step reconstruction of the map is required which is very time-consuming. But after the junction is completed, it will not be altered again and it makes sense to compress it.

6.1.3 Perfect spatial hashing

Perfect spatial hashing (PSH) is a technique, presented by Lefebvre and Hoppe [62], to compress multidimensional sparse position data sets in graphical applications. When interpreting the tuples of the SCNT graph as multidimensional position data and applying tree-based flattening on the resulting hashed data points, the techniques of [62] can be employed to solve the mapping problem. This section demonstrates the extensions that need to be integrated into the original PSH algorithm to enable coping with multidimensional data.

First, Section 6.1.3.1 introduces the principles behind PSH and its terminology which are mapped to the application domain of tuple-based graphs. Second, Section 6.1.3.2 describes the implementation and the way it copes with the challenges resulting from high dimensional data. Last, Section 6.1.3.3 demonstrates the improvements in the quality of the hash function, resulting from our extensions of the algorithm.

The initial version of **HashGraph** was developed in a Bachelor's thesis together with Nam Nguyen [63]. The implementation was further extended and recently published in [25]. Large parts of this section were used for this publication, so they appear there literally.

6.1.3.1 Theory of perfect spatial hashing

Perfect spatial hashing originally works on a d-dimensional domain U where $d \in [2,3]$ because only $2D$ and $3D$ graphics are considered. In the case of tuple-based graphs we have a d-dimensional problem where $d \in [4,12,20,28]$ because tuples with these lengths are required to represent SCNTs of order 0 to 3. The entries can assume $\overline{u}$ discrete entries from $\{0,1,\ldots,\overline{u}-1\}$ in each dimension. Thus, the domain U can have $u=\overline{u}^d$ data points in total at positions p_i when fully occupied. In the case of tuple-based graphs, the cardinality of U is determined by the tuple extent t^{ext} with $u=\prod_{k=1}^{m} t^{ext}[k]$. For tuple-based graphs, U contains all possible tuples t that can be created within the given tuple extent t^{ext}. This is equal to the tuple space.

A subset $S \subset U$ with cardinality n denotes all those positions p_i that are occupied in U, corresponding to all pixels that have colors differing from the background color in an image. The subset S corresponds in the tuple space to all actually existing tuples t_i within a given graph G. The density ρ of the data is given by the fraction $\rho = \left|S\right|/\left|U\right| = \frac{n}{u}$.

Each position p_i is associated with a data record $D(p_i)$. This could be the actual color of the pixel in the $2D$ picture and other information, for example, its transparency. Accordingly, each tuple t_i in tuple-based graphs is associated with the information of a node $D(t_i)$ like its position or its neighborhood list.

Perfect spatial hashing tries to map the sparsely defined data $D(p_i)$ with $p_i \in S$ to a record in a dense hash map H with a hash function $h : D(p_i) \rightarrow H\left[h(p_i)\right]$. *Perfect* hashing means that there are no collisions in the hash map, i.e., for all $i,j \in [1,2,\cdots,n]$ and $i \neq j$ it needs to be true that $h(p_i) \neq h(p_j)$. This goal can be achieved through the combination of two imperfect hash functions h_0 and h_1, meaning that collisions may occur in both, with an offset table Φ. This table has $\overline{r}^d = r$ entries. The resulting hash table H has $\overline{m}^d = m$ slots and the perfect hash function h is described by Equation 6.2.

$$h(p) = h_0(p) + \Phi\left[h_1(p)\right] \bmod \overline{m}. \tag{6.2}$$

The procedure to fetch two tuples t_1 and t_2 with the help of h is visualized in Figure 6.3. It is assumed that both tuples collide in h_0. Therefore, Φ requires entries for both $h_1(t_1)$ and $h_1(t_2)$, which resolve the collision and do not lead to further collisions with other tuples.

The two imperfect hash functions involved are defined as $h_0 = M_0 p \bmod \overline{m}$ and $h_1 = M_1 p \bmod \overline{r}$ with M_0 and M_1 being $m \times m$ matrices. Since Lefebvre and Hoppe [62] found out that M_0 and M_1 can simply be set to $m \times m$ identity matrices, the hash function evaluation of h_0 and

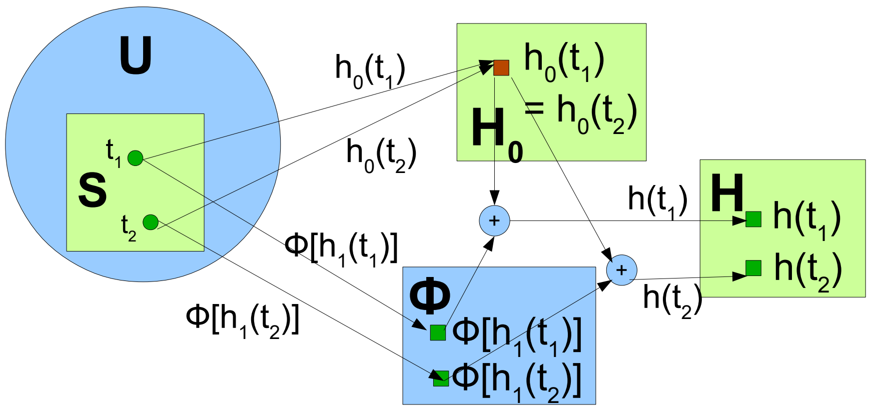


Figure 6.3.: Fetching the data for two tuples t_1 and t_2 in the case of a collision in h_0 (Figure based on Figure 2 of [62]).

h_1 reduces to an application of the mod $\overline{m}$ and mod $\overline{r}$ calculation to each entry of our tuples. The authors of [62] add the recommendation to avoid that $\overline{m}$ and $\overline{r}$ have a common divisor.

After these theoretic concepts, the remainder of this section evaluates the applicability of PSH to the tuple-based graphs and highlights several aspects that need to be taken into account for an efficient implementation.

PSH compresses spatial multidimensional data into small, dense tables. The tuple extent of the hashed tuples is much smaller than the original one which also means that the density of the hashed domain is significantly higher. This allows the derivation of a serial index on the base of the hashed domain with tree-based flattening. The property of the data set to allow random access is also preserved by PSH. The random access is important during the simulation of SCNTs, for example, when accessing neighbors of nodes. Very important is the fact that PSH avoids collisions in the hash table and thus makes the calculated serial indices unique, which is a prerequisite for the simulation to work. In contrast to most other perfect hash functions, PSH seeks also to create a minimal hash function. This implies that all slots in the hash table H are occupied. Applying the tree-based flattening algorithm to the hashed tuples in those tables results in a consecutive serial indexing. Another positive property of PSH is the combination of two hash functions h_0 and h_1 with an offset table Φ. Hence, a query for a node pointer always requires two search operations in two tables, resulting in a more uniform access time compared to other hashing schemes. The process of resolving collisions leads to varying access times there.

The concept of PSH also suffers from some drawbacks compared to other hashing schemes, but they are no severe problems for tuple-based graphs. The fact that PSH only works on a static data set is no limitation since the structure of the SCNTs and the contained nodes are static during the simulation. The same holds for the problem that the construction of the offset table Φ is complex and the process may fail, requiring several restarts. However, its runtime is not that crucial since this pre-computing step needs only to be done once before the simulation.

In particular, as it is possible to store the hashing scheme for an SCNT configuration and to reuse it for all simulations that are performed with this tube.

Some other aspects can be simplified in comparison to the original algorithm from [62]. In contrast to spatial positions, tuples are discrete data, removing the need for discretization and rounding. Furthermore, there is no need to work with normalized data since CPUs have direct integer support[1]. There is neither a problem with false positives, because the graph algebra construction process of SCNTs avoids those cases in advance. A false positive in the sense of PSH and tuple-based graphs is that a tuple $t_{\nexists}$ is queried which lies within the given tuple extent but is not actually existing in the graph. Then, it may happen that PSH returns a false entry $t_{\exists}$ from H since $h(t_{\nexists})$ is coincidentally equal to $h(t_{\exists})$. Lefebvre and Hoppe have to tackle this problem by introducing an additional integer value per hash table entry, increasing the memory requirements for the hashed data.

However, there are two main differences to the work of Lefebvre and Hoppe which the algorithm has to cope with for tuple-based graphs. First, PSH only was applied to $2D$ and $3D$ problems since it emerged within the context of computer graphics and not up to 28 dimensions like for the SCNT models. The following sections demonstrate the influence of this difference. Second, only domains with quadratic or cubic shape were employed for all the tests in [62] which considerably simplifies the problem and, in particular, leads to minimal hash functions more easily.

In general, several other spatial hashing schemes exist, but they can not be employed in the case of tuple-based graphs for different reasons. Garcia et al. [64] present a GPU and Pozzer et al. [65] a CPU algorithm with shorter construction time and faster access to the data than PSH. However, both do not stick to absolutely perfect hash functions anymore but allow a few collisions. The CPU schemes proposed by Buckland [66] and Hastings [67], which, in contrast to PSH, employ dynamic data structures, suffer from the problem of imperfect hashing, too. Alcantara et al. [68] propose a parallel, spatial hashing whose construction should be magnitudes of orders faster than PSH. However, it is not feasible for high dimensional data, because of the need to encode the multidimensional data within single integer values which is impossible for vectors with 28 entries.

6.1.3.2 Implementation

The following three subsections highlight the extensions that are applied to the original PSH algorithm and outline several important points for an efficient CPU realization since PSH was designed for the execution on GPUs in its original implementation.

[1] This is also the case for modern GPU architectures but not for those that Lefebvre and Hoppe worked with.

6.1.3.2.1 Creation of the offset table and minimization of the size of H

The construction of the offset table Φ is realized with a trial and error approach as proposed in [62]. Nevertheless, several aspects are changed to optimize the speed of the table construction and the quality of the result, especially the memory requirements and the value of the top index. First of all, the concept of constant values $\overline{r}$ and $\overline{m}$ is abandoned. Instead, vectors $\overrightarrow{r} = (\overline{r}_1, \overline{r}_2, \ldots, \overline{r}_d)$ and $\overrightarrow{m} = (\overline{m}_1, \overline{m}_2, \ldots, \overline{m}_d)$ with different entries $\overline{r}_k$ and $\overline{m}_k$ for each dimension are allowed[2]. This is done to reduce the values of r and m which are, in that case, calculated by $r = \prod_{k=1}^{d} \overline{r}_k$ and $m = \prod_{k=1}^{d} \overline{m}_k$, respectively. High values of r lead to a large Φ since $h_1(t)$ maps each t to a distinct slot, while a large m results in high serial indices and low density.

Before the actual construction of Φ can start initial values for the vectors $\overrightarrow{r}$ and $\overrightarrow{m}$ have to be found in a first step. To that end, two heuristics are employed. Initially, all $\overline{m}_k$ are set to the smallest possible $\overline{m}$ satisfying $\overline{m} \geq \sqrt[d]{n}$ following the prerequisites given in [62]. Afterward, all entries $\overline{r}_k$ are set to the smallest possible $\overline{r}$ satisfying $\overline{r}^d \geq \sigma n$ with $\sigma = 1/2d$, which is also proposed in [62]. Now, the value of m is compared to n, i.e., the sparsity of H is evaluated. If m is at least a factor of 3 higher than n, the entries within the respective tuple extent t^{ext} are searched that have the greatest distance to $\overline{m}$. At these positions, the entry $\overline{m}_k$ is decremented by 1 as long as $\prod_{k=1}^{d} \overline{m}_k \geq n$. The factor 3 is an empirically determined value. The chance to reach a significant improvement is very low for lower values.

In the second step, a first feasibility check is performed that ensures that there are no two tuples t_1, t_2 with $h_0(t_1) = h_0(t_2) \wedge h_1(t_1) = h_1(t_2)$, because it is impossible to construct a perfect hash function in those cases. This check can be performed very efficiently with an additional table $T^{\text{col}-\text{h}_1}$ that is required for the next step anyway. It stores those sets of tuples that are mapped to the same hash value by h_1. For each tuple t_i, the hash-value $h_1(t_i)$ is calculated. Now, a lookup on table $T^{\text{col}-\text{h}_1}$ is performed that has $h_1(t_i)$ as its key values. If the key $h_1(t_i)$ does not exist, a new entry in form of a vector is inserted. The vector has a reference to t_i as its only entry. In the case that an entry already exists for $h_1(t_i)$, the reference to t_i is appended to the existing vector. In this way, the tuples are divided in sets with a typical size $\ll 100$. It is sufficient to test every tuple t_i against the other tuples in this set for a collision in h_0, since only nodes within the same set are candidates for violating $h_0(t_1) = h_0(t_2) \wedge h_1(t_1) = h_1(t_2)$. If there is only one collision, $\overrightarrow{r}$ and $\overrightarrow{m}$ are not feasible and need to be updated, which enables an early exit. The algorithm primarily tries to adapt the $\overline{r}_k$ values, since it is the main goal to create minimal serial indices for the tuples. This means that H must be as compact as possible and, therefore, the m_k values should not be increased. The adaption of the $\overline{r}_k$ is realized by searching those entries which have the greatest distance to their respective entries in t^{ext} in $\overrightarrow{r}$ and incrementing them by one.

2 Only in this subsection about PSH vectors are marked by arrows on top of the variable to avoid confusion with the constant entries and other variables.

Out of the tuples resulting in collisions, k tuples are picked at random and the colliding positions in the $\overline{r}_k$'s are also incremented as long as all k collisions are resolved. The hope is to avoid many more, ideally all, collisions due to these changes. The values presented in the remainder of this thesis were obtained with $k = 1$, resulting in the most optimization steps but the lowest possible values for r and m. Also $k = 10$ was tested, resulting in a faster construction but slightly higher values for r. Afterward, the second step is repeated until it succeeds.

The third step tries to construct an offset table Φ for the given $\overrightarrow{r}$ which resolves all collisions in h_0. To that end, the heuristic of [62] is applied and the procedure tries to find offsets for the biggest colliding sets of U in $T^{\text{col}-\text{h}_1}$ at first. It is easy to deduce the order in which the sets should be processed from this auxiliary table. An offset is randomly generated for each set and tested for feasibility. As soon as an appropriate offset is found, i.e., it resolves existing collisions without creating new ones, it is inserted into Φ. Then, the algorithm proceeds with the next largest set, until all collisions are covered. In the case that it is not possible to find an appropriate offset for a set c_l, i.e., the l's set that is processed, the algorithm applies backtracking. The offset chosen for the last set c_{l-1} is changed to another possible entry and c_l is processed again. If this loop does not succeed, the offset for c_{l-2} is changed. After several unsuccessful attempts the algorithm adapts the values for $\overrightarrow{m}$ and $\overrightarrow{r}$ and restarts the current step with the new values.

This three step procedure is repeated until a perfect hash function is found or canceled when m and r become too large to achieve a sufficient reduction because of the resulting large size of H.

6.1.3.2.2 Dynamic data structures and 1-dimensional keys

In the original PSH, static multidimensional arrays are used for the tables Φ and H, since the algorithm was designed to work on GPUs. The size of these tables is statically allocated and the total number of allocated entries is $\overline{m}^d$ and $\overline{r}^d$, respectively. The hashed 2- and 3- dimensional coordinates of the points are used to access the values in Φ and H. This is feasible under the simplifying assumption that the problem domains are always quadratic or cubic. However, this procedure wastes too much memory for longer tuples with unequal dimensions Table 6.1 visualizes this fact for several tubes prior to the cutting process that is why it differs slightly from the tube data in other chapters. For example, nearly $1.6 * 10^7$ entries are statically allocated for Φ for the tube $(1, 6, 12, 18)$[1] while only $1.7 * 10^4$ entries are required, a difference of three orders of magnitude. To cope with this problem, dynamic data structures are employed, namely the `std::unordered_map`. They only store the existing values and allow access in constant time.

Another issue is the usage of tuples and their hashes as keys for Φ and H, respectively. This causes high access times and consumes a lot of memory. A detailed discussion of this problem can be found in Appendix D. To tackle this problem, the extended PSH also uses serialized indices as keys for the maps, which are calculated on the base of the tuples and the knowledge of h_0, h_1, m and r. So, the keys for Φ result from applying h_1 to the current tuple t and then

Table 6.1.: Comparison of the number of actual tuples in the graph with the number of entries that would be statically allocated in Φ and H if PSH is applied without extensions.

Configuration	No. tuples	Entries in Φ	Entries in H
$(1,4,8,4)^1$	$1.7 * 10^4$	$6.6 * 10^3$	$2.6 * 10^2$
$(1,4,8,14)^1$	$4.5 * 10^4$	$6.6 * 10^3$	$2.6 * 10^2$
$(1,6,12,18)^1$	$1.3 * 10^5$	$1.6 * 10^7$	$5.3 * 10^5$
$(2,6,12,18)^1$	$1.7 * 10^5$	$1.6 * 10^7$	$5.3 * 10^5$
$(2,6,12,116)^1$	$1.0 * 10^6$	$1.3 * 10^{14}$	$1.7 * 10^7$
$(2,3,6,12)^2$	$5.6 * 10^6$	$4.3 * 10^7$	$6.6 * 10^4$
$(1,2,8,8)^2$	$1.0 * 10^6$	$9.5 * 10^{13}$	$1.0 * 10^6$
$(1,2,4,4)^3$	$2.5 * 10^7$	$2.3 * 10^{13}$	$2.7 * 10^7$

executing tree-based flattening with $\overrightarrow{r}$ as the corresponding tuple extent. In this way, only the serialized index needs to be stored. This means that for SCNTs of order 3 only one 64 bit-value is required instead of 28 16 bit-values. The storage for the keys is reduced by a factor of 7 from 530 MB to 75 MB for $1 * 10^7$ entries in H. Additionally, long tuples cause high construction- and access-times. Generating a map with $1 * 10^7$ randomized 28-vectors as keys and values requires about 30 s on the test system while the same test is done in about 10 s, if vectors are serialized and inserted in an integer-based map. As a result, the extended PSH version avoids both drawbacks of vectors as keys by exploiting the knowledge about the tuples.

6.1.3.2.3 Density threshold and different compression modes

It is obvious that the construction of a perfect hash function always succeeds if $\overline{r}$ or $\overline{m}$ are large enough. In the first case, Φ may contain n entries, i.e., one for every possible tuple. In the second case, H has so many slots that for each tuple one slot can directly be assigned without collisions. In those cases, H reserves much more slots than $\left|U\right|$. The calculated serial indices resulting from $\overline{r}$ and $\overline{m}$ are in the same range or even much higher than those calculated on the original domain and, therefore, they are impractical.

An analysis of those cases reveals that they normally appear if domain U is occupied relatively densely. This means that $\left|S\right|$ is not much smaller than $\left|U\right|$. A density threshold $\rho_{\text{thres}} = \left|S\right|/\left|U\right| < 0.15$ is defined to exclude those cases. It is not tried to construct a full perfect hashing function for all domains with higher ρ_{thres}, because of the low probability of finding a sufficiently compact one. Instead, one of the following two options is chosen:

1. Try a perfect hashing on the whole junction part, i.e., the part of the tuple after the four highest entries $\left[x_m, \dots, x_{m-3}\right]$.
2. Ignore $\left[x_m, \dots, x_{m-3}\right]$, partition the remainder in blocks of length 4 and try to construct a perfect hash for each block.

The ideas behind these two additional options are the following: The first $\left[x_m, \dots, x_{m-3}\right]$ entries, representing the tube part, are usually relatively dense and thus can prevent a compact hash function. So, in case of failure, it makes sense to ignore them as a first step. If it is still not possible to find a sufficiently compact hash function, the second option exploits the fact that always eight consecutive entries of a tuple represent the information of one junction level. Always the leading four entries within these eight entries are much sparser occupied than the remaining four. Thus, option two normally leads to the situation that each second block of length four is hashed by PSH while the remaining blocks are indexed with tree-based flattening as they are dense. Compression of the whole domain is called *mode 1*, compression of the junction part *mode 2* and the dividing into sub-blocks and trying to hash those *mode 3*.

As a measure of efficiency of the hash junction, we define the reduction factor $\psi = |U|/|H|$. For mode 2, ψ is equal to the reduction factor ψ^{juncs} that is achieved on the junction part of the tuple. For mode 3 the overall reduction factor can be calculated as the product of the ψ_i resulting from all k blocks of length four:

$$\psi = \psi_1 \cdot \psi_2 \cdots \psi_k = \frac{|U_1|}{|H_1|} \cdot \frac{|U_2|}{|H_2|} \cdots \frac{|U_k|}{|H_k|}$$

6.1.3.3 Evaluation of the algorithmic extensions

Now, we investigate how the presented algorithmic extensions effect memory requirements and the reduction factor compared to the original PSH version. To that end, PSH was applied to all tubes that are presented in Table 6.2 and the results are summarized there. The three configurations marked by an asterisk are hashed with mode 2 while all others are hashed with mode 1.

Table 6.2.: Summary of the tested tubes. Fhe second column shows the number of tuples to hash for each configuration. Columns $r = \overline{r}^d$ and $m = \overline{m}^d$ give values achieved by the original PSH algorithm while r^* and m^* give the new values resulting from our extended procedure. The columns r/r^* and m/m^* give the factor by which the number of entries in both tables can be reduced by the extensions. %Φ shows the fraction (used slots in Φ) / (No. tuples) while the last column gives the time to construct PSH in seconds, using the extended procedure on phase one nodes of the Lichtenberg cluster.

Configuration	No. tuples	$r = \overline{r}^d$	r^*	r/r^*	$m = \overline{m}^d$	m^*	m/m^*	%Φ	Con.(s)
$(1,4,8,4)^{1*}$	$1.7*10^4$	$6.6*10^3$	$9.6*10^2$	6.9	$2.6*10^2$	$2.6*10^2$	1	99%	<1
$(1,4,8,14)^{1*}$	$4.5*10^4$	$6.6*10^3$	$9.6*10^2$	6.9	$2.6*10^2$	$2.6*10^2$	1	99%	<1
$(1,6,12,18)^1$	$1.3*10^5$	$1.6*10^7$	$9.4*10^5$	17.0	$5.3*10^5$	$1.6*10^5$	3.3	13%	<1
$(2,6,12,18)^1$	$1.7*10^5$	$1.6*10^7$	$9.4*10^5$	17.0	$5.3*10^5$	$2.4*10^5$	2.2	13%	<1
$(2,6,12,116)^1$	$1.0*10^6$	$1.3*10^{14}$	$2.7*10^6$	$4.8*10^8$	$1.7*10^7$	$1.0*10^6$	17	6%	4.7
$(2,3,6,12)^{2*}$	$5.6*10^6$	$4.3*10^7$	$3.3*10^5$	130	$6.6*10^4$	$6.6*10^4$	1	97%	<1
$(1,2,8,8)^2$	$1.0*10^6$	$9.5*10^{13}$	$2.0*10^7$	$4.8*10^6$	$1.0*10^6$	$1.0*10^6$	1	38%	11.8
$(1,2,4,4)^3$	$2.5*10^7$	$2.3*10^{13}$	$6.9*10^9$	$3.3*10^4$	$2.7*10^7$	$2.7*10^7$	1	75%	370

The development of the number of slots and entries in Φ is not crucial for performance but for the memory consumption of the PSH structure. Table Φ is the main difference concerning the required storage compared to the **IndexGraph** whose small tables can be neglected. Additionally, the product of all entries in $\overrightarrow{r}$ must be small enough to allow tree-based flattening without overflows. The total number of slots in Φ is much smaller when employing the vector $\overrightarrow{r}$ with varying entries instead of $\overline{r}^d$ for all tubes. The best demonstration of the efficiency of the algorithmic extensions is the tube $(2, 6, 12, 116)^1$ which is characterized by highly varying entries within its tuple extent $t^{\text{ext}} = (5\,9\,12\,2\,2\,|\,6\,3\,3\,8\,4\,4\,2\,2)$. This constitutes a difficulty for the original PSH. The constraint to determine a uniform $\overline{r}$ forces the algorithm to choose $\overline{r} = 15$ in order not to violate the prerequisite $\nexists t_1, t_2 \in S \mid h_0(t_1) = h_0(t_2) \wedge h_1(t_1) = h_1(t_2)$. This value is actually higher than those for the tubes of order 2 and 3 with more or longer tuples. In the extended algorithm, the original $\overline{r}^{12} = 15^{12} = 1.3 * 10^{14}$ is replaced by the new vector $\overrightarrow{r} = (15\,3\,3\,3\,|\,3\,3\,3\,3\,3\,3\,3\,3)$ reducing the number of slots by nearly eight orders of magnitude.

The second to last column of Table 6.2 shows how many slots are used in Φ in relation to the overall number of tuples in the respective tube. Nearly every tuple is hashed to a different slot for the three tubes processed with hashing mode 2. That is not an issue, since in those cases the number of tuples to hash is equal to the number of nodes within a junction. Their number is much smaller than the total number of nodes in the tube, e.g., 176 compared to $1.7*10^5$ for the tube $(2, 6, 12, 18)^1$. As already mentioned, the size of r influences the size of Φ with a higher r leading to more entries. Hence, the optimization of r can also help to reduce the size of Φ. This can be demonstrated for the tube $(2, 6, 12, 116)^1$ where the number of entries is reduced from $2.5 * 10^5$ to $6.5 * 10^4$. The percentage of slots used compared to the number of tuples decreases from 25 % to 6 %. There is also a reduction from $6.4 * 10^5$ to $3.8 * 10^5$ entries for the tube $(1, 2, 8, 8)^2$.

It is also very important that m can be reduced by one order of magnitude as well in the case of tube $(2, 6, 12, 116)^1$ by replacing the uniform $\overline{m} = 4$ with vector $\overrightarrow{m} = (4\,4\,2\,2\,|\,4\,4\,4\,4\,4\,4\,2\,2)$, which increases ψ from 5 to 73. $|H|$ could also be decreased for the tubes $(1, 6, 12, 18)^1$ and $(2, 6, 12, 18)^1$.

As explained, the runtime for the constructing a perfect hash function plays a subordinate role but was also measured. The last column of Table 6.2 shows that for nearly all tubes, the perfect hash function can be generated within a few seconds and can consequently be neglected. About 6 minutes are required only for the order 3 tube. We are confident that there is potential to further optimize the runtime of the construction process.

6.1.4 Comparison of different graph types

This section compares the reduction factors ψ of the three described graph types, i.e., their indexing schemes. ψ is for **IndexGraph** defined analogously as for PSH by the fraction of the highest serial index that occurs within an instance of **IndexGraph** and $|U|$. The tree-based flattening only serves as reference since its index space is always $|U|$. Hence, the reduction factor is 1. The most important point about the tree-based flattening is that its principle is contained in **IndexGraph** for indexing the tube part of the tuple as well as the nodes within each junction level. Within PSH and **HashGraph**, tree-based flattening calculates the indices on the new hashed domain. Hence, the focus lies on the comparison of the top index in the index spaces resulting from **IndexGraph** and **HashGraph**. Figure 6.4 summarizes the results for these two graph classes on various tubes. Additionally, it shows the ideal reduction factor that is calculated by dividing $|U|\,/\,|S|$. For tubes whose corresponding bar for PSH is marked by an asterisk, the algorithm uses compression mode 2 because it delivers the best result.

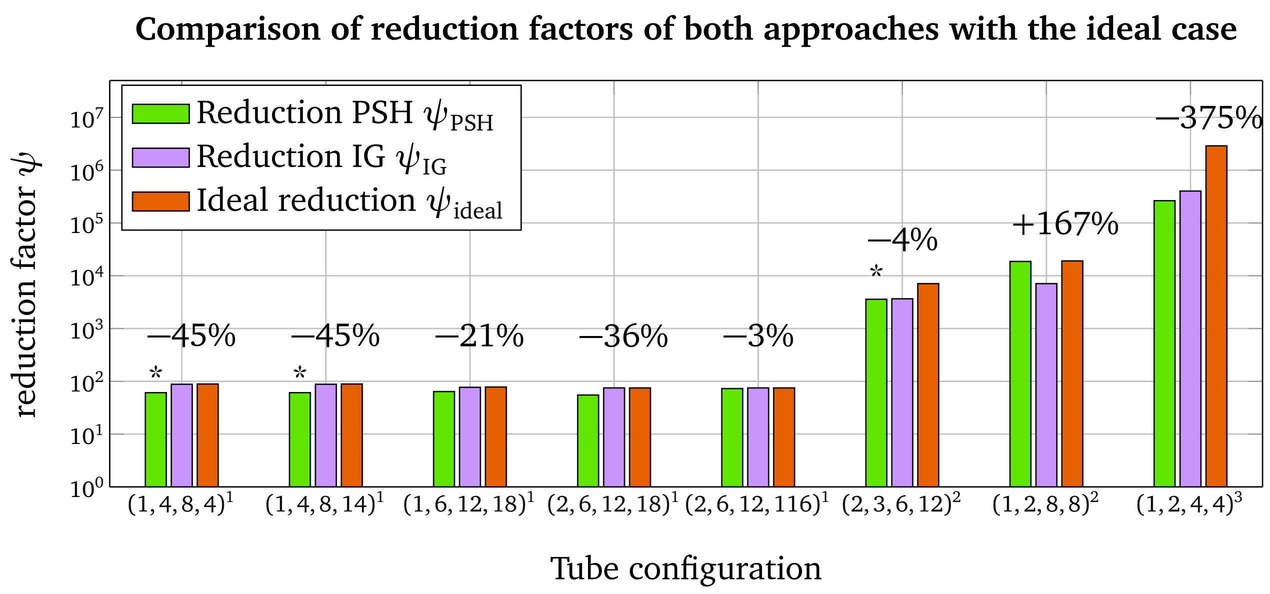


Figure 6.4.: The reduction factors achieved by PSH and the structure-tailored IndexGraphs for different tube configurations. The percentage over the bars gives the relative difference between IndexGraphs and PSH. * denotes tubes hashed in mode 2.

In most cases, PSH is able to achieve a reduction factor in the same range as the IndexGraphs does for all tested orders of tubes. Remember that we are talking about reducing the index space by several orders of magnitude compared to tree-based flattening. This relativizes differences of a factor of 2 or 3 between both mapping approaches. In principle, two particularities show up in the results: The only instance where PSH performs significantly worse is the tube of order 3 where the reduction factor is about four times lower than for IndexGraphs. In contrast, the reduction is higher for PSH as for IndexGraphs and nearly reaches the optimum (reduction of 18662 versus the optimum 19109) for tube $(1,2,8,8)^2$. The tube $(1,2,8,8)^2$ is a kind of best-case scenario for **HashGraph**. It consists of $1,024,000$ nodes which is very close to $2^{20} = 1,048,576$ with 20 being the tuple length for an SCNT of order 2. In this case, $\overline{m} = 2$ is chosen, resulting in a hash table H with $1,048,576$ available slots. In combination with an appropriate

Φ, all nodes can be assigned a slot, resulting in an occupation of 98% and hence, a nearly minimal hash function. Such a minimal function would lead to the ideal reduction factor.

The influence on the solver in PSH was also investigated. In a first test, a search of about $5 * 10^7$ random tuples was performed on different tube configurations with **HashGraph** and with **IndexGraph**. The results show that the search time for PSH is higher by a factor of about 1.5 with values ranging from 1.26 to 1.67 on these synthetic benchmarks. However, the influence of PSH on our overall solver is not that significant. The performance difference is under 10% For single threaded execution. Additionally, PSH has a negative influence on the scaling behavior of the application, so that the difference increases to about 20% when running with four threads. This difference remains when running with 8 and 16 threads.

To sum up: Although PSH is a general scheme, it delivers very good results in compression of the index space compared to the structure-tailored **IndexGraph**. The extensions proposed to the original algorithm have a very positive effect on the hashing quality, feasibility and speed of creation. Hence, the extended PSH is a good candidate to be tested for indexing other multidimensional data as well. However, the compression results of **IndexGraph** are more constant and better in nearly all cases which stems from their strong connection to the SCNT structure. Simultaneously, the access time to the data in **IndexGraph** is lower. As a consequence, **IndexGraph** is employed as underlying indexing scheme within the tests of this thesis.

6.2 Space-saving approach to store the edges

The presented way of indexing tuples is used extensively within the framework developed in this thesis. During construction, the edges require a lot of storage. There exist three times more edges than nodes for an SCNT model. It is also required to iterate over all edges or to randomly access them during the execution of graph algebra operations. This section compares the behavior of a self-developed and structure-aware edge-container to a boost::bimap from the boost library [69] storing all edges without exploiting the structure. Simply put, a bimap is an ordered map which allows to use either the left or the right side as keys to access corresponding values and vice versa. The bimap employed to store edges has unsigned 64 bit integers on the left and the right side, while the left side represents the start index of the edge and the right side its end index, respectively. The motivation to chose the boost::bimap as a reference for comparison arises from the fact that this implementation was the underlying data structure for the storage of edges in publications [21], [23], [24] and [27].

Several properties for these graphs can be assumed if it is known that a graph represents an SCNT or intermediate graphs during construction of SCNTs. The first and most important point is that the occurring graphs only contain nodes that have at maximum four incident edges. Although the nodes in the final SCNTs have only up to three incident edges, there are some nodes in temporary graphs with one additional incident edge. The second point is that most graphs also contain opposite edges to the forward edges, i.e., the edges inverted direction. In the final SCNT model, all edges have a counterpart in opposite direction, but again some intermediate steps are exceptions.

Exploiting these properties allows to reduce the amount of storage but requiring to deal with the aforementioned exceptions. This results in a highly configurable container for the edges whose complexity is hidden behind the **EdgeMap** class. Its interface allows to add, remove and search edges as well as to create iterators over all edges. There mainly lies a hash map behind this interface which contains special **EdgeEntry** objects encoding the information. The hash map has the big advantage that it does not create as much overhead as a bimap. Another feature can be deduced from the construction process: There are several steps in which some nodes need to be removed from the graph which we summarize as cutting processes. All affected nodes are removed before another query option is executed on the graph during this processes.

All edges with $e = (n_1, n_2)$ where the serial index of n_1 is smaller than that of n_2 are defined as forward edges within the graphs. Consequently, all other edges whose index of n_1 is bigger than that of n_2 are opposite edges. The main feature of the **EdgeMap** is that it separates the incident nodes of an edge from its direction. That means, if either the forward edge or the opposite edge exists, the edge is stored and the direction is marked by a flag. Only another flag is set if the opposite edge needs later also to be stored, avoiding additional storage.

This behavior is realized by the **EdgeEntry**s. These objects store the main information about the edges and they are themselves stored within the hash map. The serial index of the node that is incident to the edges in the respective object is the key of this internal map to address the **EdgeEntry** objects. **EdgeEntry**s are only created for nodes that have at least one forward edge. For example, assume a graph $G = (V, E)$ with $V = \{1, (3, 8\}$ and $E = \{(1, 3), (3, 8), (8, 3)\}$ where $\{1, 3, 8\}$ are the serial indices corresponding to the tuples of the respective nodes. In such a case, only two **EdgeEntry**s are created since the only outgoing edge $(8, 3)$ for node 8 is an opposite edge and must be encoded in the **EdgeEntry** corresponding to node 3. This principle results from the fact that nearly all (but not all) edges in the graphs occurring during construction have opposite edges and it allows to represent the exceptions, too.

Each **EdgeEntry** consists to one part of a dynamic array edges that stores the serial indices for the end nodes of the edges. The default size of the edges array is set to two because most nodes have 3 incident edges of which only two are forward edges in average. In the case that a third or a fourth edge needs to be inserted, the size of the array is dynamically increased by one. This procedure guarantees that only the minimal amount of space for the node indices within **EdgeEntry** is allocated while the costs for reallocation are small since only few nodes are affected. This exploits the knowledge about the edge structure within the occurring graphs.

The second part of an **EdgeEntry** is a std :: bitset[3] that encodes all other additional information. Its space efficiency is the main advantage of a std :: bitset in comparison to a boolean array, since

[3] `http://www.cplusplus.com/reference/bitset/bitset/`

each entry in an array requires one byte of storage in C++. In contrast, a std :: bitset always requires a multiple of four bytes (32 bit). For example, it requires 32 bit for $[1, \ldots, 32]$ entries and 64 bit for more than 33 but less than 64 entries. The std :: bitset is conceptually split into four parts: The first part encodes the possible four forward edges, while the corresponding four possible opposite edges are encoded by the second part. Part three stores the number of registered edges in edges. Finally, part four stores the information whether the array has already been resized, so if edges has two, three or four slots at the moment. In total, 12 entries are required to store all information in the std :: bitset. The resulting size of 32 bit for the std :: bitset is three times less than that of a comparable **bool**-array with 12 slots which requires 96 bit of storage.

When inserting or deleting an edge e, the respective **EdgeEntry** must be determined by taking the smaller value of the incident nodes of e and looking it up in the map. This lookup is possible in constant time in hash maps. Afterward, the small edges array is searched for the second value of e sequentially. The corresponding position in edges is then used to check and toggle the flags for forward and opposite edges. After each insertion or deletion, the edges array is sorted to avoid gaps in it. In the case that the last edge is removed from an **EdgeEntry**, the object is deleted from the hash map.

The complexity of iterating over the edges is also hidden behind the **EdgeMap** interface. Internally, the iterator itself must work in a two stage process: The outer stage runs over all entries of the internal hash map and the inner stage over the edges encoded in an **EdgeEntry**. The fact that the elements in the **EdgeEntry** are sorted simplifies the iteration and speeds up the access.

One difficulty in **EdgeMap** is the removal of nodes from the graph and the deletion of all edges incident to it. This operation requires the call of the member function erase for the left and the right side with the index of the removed node as argument for the boost::bimap. Hence, the removal is done automatically by the boost library. The situation is more complex for the new **EdgeMap**: One part of the edges can be removed directly by searching for the index of the removed node in the hash map. If a corresponding **EdgeEntry** is found it is deleted. However, the second part of the edges is harder to identify, since they may be distributed in other **EdgeEntry** objects that are assigned to a node with lower index. Hence, searching in the **EdgeEntry**s with lower node indices for the occurrence of the node index and removing it there is a straightforward way to remove the remaining edges. Of course, this linear search results in a slow removal process.

The structure of SCNTs opens ways to speed up this procedure with little memory overhead. There only is a fixed number of possible offsets between the node to remove and the index of the **EdgeEntry** containing incident edges due to the regularity of the structure. Thus, an analysis is performed within the **EdgeMap** prior to the first call of a remove operation that identifies all possible offsets, counts the number of their occurrence and sorts the offsets in descending order of their frequency in the std :: vector offsetvec. If, now, one node should be removed, the **EdgeEntry** s for the possible offsets are processed in the aforementioned order. The critical point is the size of offsetvec. It depends for SCNTs of the same order on the junction configuration, i.e., the values of (d_x, l_x). The difference is at least a factor of 8 for different orders with the same junction configuration for the tested configurations. In contrast, the tube parameters (d_0, l_0) do not have any effect on the size of offsetvec. Table 6.3 shows the development of the number of possible offsets for different tubes. It is obvious that their number and thus the search overhead becomes very high for order 2 and 3 tubes. However, it should be noted that the number of possible offsets is much lower than the overall number of nodes. For example, the tube of order 3 consisting of $(2, 4)$ junctions contains over $3.0 * 10^8$ nodes, while there are $14,621$ possible offsets. Therefore, the exploitation of the knowledge of the structure reduces the size of search space by a factor of over $20,500$. This allows to remove single nodes with acceptable performance while the additional storage for these $14,621$ integers is very low compared to the more than $3.0 * 10^8$ nodes and $9.0 * 10^8$ edges of this example graph.

Table 6.3.: Size of the vector offsetvec for the possible offsets in the **EdgeMap**. The first row shows the junction configuration (d_x, l_x) that was used for the respective data row.

Junction Config.	Order 0	Order 1	Order 2	Order 3
(1,2)	4	36	352	2890
(1,4)	4	51	485	4491
(2,4)	4	73	1033	14621
(4,8)	4	132	3039	-

The demonstrated procedure drastically increases the runtime but if more nodes need to be removed, as it is the case for the cutting process of the tubes, it is still slower than the boost::bimap approach. However, as investigation of the construction algorithm reveals, the cutting processes forms a closed unit. No other accesses on the graph structure, like reads or iteration-operations, take place during the removal of several nodes. This motivates a different approach when more nodes need to be removed. Instead of directly deleting the edges belonging to a removed node, the respective node index is stored in an additional associative container nodes_to_remove. Simultaneously, a flag marks the whole **EdgeMap** as non-synchronized. Following removal operations merely insert more nodes in nodes_to_remove. The first reading operation to the **EdgeMap** triggers a synchronizing operation within the map. Depending on the size of nodes_to_remove, either all nodes are removed individually by single delete operations, as already explained, or a specialized delete method is called, if the number of entries in nodes_to_remove exceeds a particular threshold.

This specialized method iterates over all edges in the **EdgeMap** and performs two tests on the **EdgeEntry**s. The first one checks whether the key of the entry is contained in the nodes_to_remove container and it deletes the whole **EdgeEntry** if this is the case. This lookup can be done in constant time. If the key is not listed in nodes_to_remove, each entry in the edges array is searched in nodes_to_remove and the corresponding edge is removed if it is found. An additional optimization for the SCNT graphs and the final cutting of the boundaries is also integrated. Due to the fact that the serial indices in **IndexGraph** have the same order as the global indices, all nodes with small indices lie near the left end of the tube while the highest indices lie near the right end. Hence, there is a large distance between the highest index on the left side $\text{left}_{\text{high}}$ that needs to be removed and the lowest index on the right side $\text{right}_{\text{low}}$, respectively. Consequently, no **EdgeEntry** for a key that lies between $\text{left}_{\text{high}}$ and $\text{right}_{\text{low}} - \text{max}_{\text{offset}}$ is affected by the cutting operation, where $\text{max}_{\text{offset}}$ is the largest entry in offsetvec. The non-affected **EdgeEntry**s can be skipped. This optimization reduces the number of edges to test, especially for long tubes.

The total heap-memory consumption of the two different **EdgeMap** variants was investigated with the tool *massif*[4] from the *valgrind* tool suite. The heap profiling tool massif takes several snapshots during the execution, so that the measurements may slightly vary for every run. Hence, the following results are rounded to 5 MB steps, which covers all results attained. The construction is stopped after applying the geometry for this test and no flattening and neighborhood construction is executed.

The values for the $(1,4,8,8)^2$ tube with 4 million nodes are presented as an example. The heap-memory consumption for the program with the bimap-based **EdgeMap** is about 1,575 MB. 1,000 MB of these belong to the **EdgeMap** while the remaining 575 MB are occupied by the node information and some additional data. In contrast, the construction with the new **EdgeMap** based on **EdgeEntry**s requires only 770 MB in total. The part for the edges is only 195 MB in that case while the remainder is again 575 MB. The storage for edges can further be divided into the space required for the **EdgeEntry** elements, which is about 160 MB and 35 MB for the internal hash table structure. Consequently, the new map reduces the storage for the edges by a factor greater than 5 from 1,000 MB to only 195 MB. The reduction for the whole graph is a factor of 2 (from 1,575 MB to 770 MB) after the construction phase. These values are also the average for the investigated tubes. Detailed measurements can be found in Appendix E.

A comparison for the runtime required for the construction of the $(1,4,8,8)^2$ tube was also performed. While the bimap approach requires 48 seconds, the construction phase of the tube with the **EdgeEntry** version completes after 25 seconds, i.e., in roughly half the time. An analysis of the runtime behavior reveals that the insertions in the bimap require a high amount of time, especially when the bimap is already large. In contrast, the underlying unordered map behind the **EdgeEntry** version does not show this drawback. More performance results can be found in Appendix E.

Altogether, the new **EdgeMap** structure is able to halve the memory required during the construction phase while the performance is doubled. It supports most of the graph algebra operations on the level of serial indices, further reducing the memory overhead and speeding up the computation. The explicit processing of tuples is required in a few cases only.

[4] `http://valgrind.org/docs/manual/ms-manual.html`

7 Compressed Symmetric Graphs

The graph that represents an SCNT needs to be kept in memory during the actual simulation and therefore, it is worth to minimize the space that is required for the graph data. To that end, we present the custom-tailored data structure *Compressed Symmetric Graphs (CSGs)* in this chapter. It reduces the storage of structural information, i.e., neighborhood information by dynamically recomputing parts of the tube. We demonstrate that CSGs avoid the storage of up to 99% of the structural information for a lot of SCNT models after the data creation phase.

CSGs exploit the fact that the structure of the tubes stays constant even under load. In other words: The neighbors of a node n_i will not change even if the tube is being deformed. Additionally, CSGs combine exploitation of translational and rotational structural symmetry, thus multiplying their effects. CSGs can replace the graph data structures presented in Chapter 6 during the actual simulation of the SCNTs y implementing a common interface.

The theory behind CSGs was published in [26]. This chapter is based on this publication and its definitions. Some extensions to the original publication are highlighted and several points are covered more detailed.

7.1 General graph compression schemes

There exist different ways to compress general graphs and graphs with some special properties in the literature. Most procedures arise from the field of database compression. Hence, we evaluate whether these methods can also be employed for an efficient compression of tuple-based graphs.

Feder and Motwani [70] presented a graph compression algorithm for directed and undirected graphs that allows several graph algorithms to be adapted to the compressed graphs. This can speed up tasks like finding matchings or the solution of the all-pair shortest path (APSP) problem mainly on dense graphs, i.e., finding the shortest path between all possible pairs of nodes in a graph [71]. The main goal of the algorithm of Feder and Motwani is to reduce the number of edges. To that end, a graph is partitioned into a number of bipartite cliques and the order of these cliques, i.e., the number of nodes in them, is minimized. According to the authors, their compression algorithm is able to reduce the number of edges m^* in the compressed graph compared to the original number of edges m and number of nodes n by:

$$m^* = O\left(\frac{m}{k(n,m,\delta)}\right) \qquad \text{with } k = \lfloor \delta * \frac{\log(n)}{\log(2*n^2/m)} \rfloor \text{ and } 0 \leq \delta \leq 1$$

There is always a trade-off between runtime and quality of compression, since the runtime complexity $O(m * n^{\delta} \log^2(n))$ also depends on the choice of δ. However, it may happen that the resulting n^* of the compressed graph is even higher than the original n since the procedure is targeted for dense graphs, i.e., $m \gg n$. This behavior makes the algorithm unsuitable for the graphs representing SCNTs where we always have $m \approx 3 * n$.

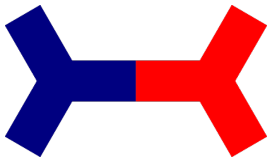
Chen and Reif proposed a lossless method to compress trees as well as undirected and directed acyclic graphs [72]. The compressed data still allows efficient operations like searching for nodes and has a similar structure to the original graph. They demonstrate their procedure for binary trees but state that it can be readily extended to general trees as well. The input trees are parsed in breadth-first search (BFS) and divided into sub-trees which are then compressed with the dictionary based Lempel-Ziv compression [73], employing a suffix tree ([74], [75]) as main data structure. Suffix trees enable an efficient pattern matching in strings. Undirected graphs are compressed by a two pass procedure. In the first pass, the graph is transformed into a BFS tree and in a second pass over the original graph, the back-edges are covered, compressed, and unified with the original BFS tree. Chen and Reif state that this may sometimes result in a bigger size of the compressed graph than the original size which is not acceptable in the case of tuple-based graphs.

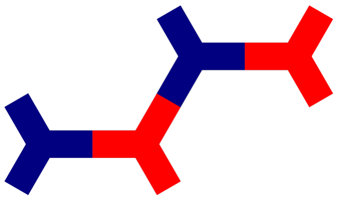
The Graph Summarization, another method to compress graphs, was presented by Navlakha et al. [76]. The technique uses similarities in the link structure and unifies nodes to supernodes if they are connected to the same set of nodes. The original graph is represented by two parts. The first one is a graph summary that is a graph itself. Each node of this graph summary corresponds to a set of nodes in the original graph. Additionally, a list of edge-corrections is stored. The technique can be used for lossless compression. Internally, the Minimum Description Length (MDL) principle [77] is employed that exploits regularities in an input string to compress it. Navlakha et al. achieve compression ratios between about 1.25 and 3.3 for the four investigated data sets. These values are not sufficient for our SCNT models.

As many other graph compression schemes, Graph Summarization is employed in the area of web and network graph compression (see e.g. [78], [79]) to compress huge graph data sets and allow efficient adjacency queries. Other known graph compression schemes in this area are the reference encoding proposed by Boldi et al. [80] and graph clustering [81].

In contrast to all above mentioned approaches, CSGs are custom-tailored to SCNTs. Hence, they achieve high compression rates, combined with a fast calculation of the compression. They directly work on the neighborhood structure and the node- and edge-level, respectively. Therefore, there is no need to fully analyze the input graph or even to work on the binary level. Furthermore, it is even not required to construct the full tube before instantiating the compressed graph. This is a significant advantage over all presented approaches, where the whole graph is needed initially.

The Y-junctions are the basic building block for SCNTs of higher order and, at the same time, the basis for the CSG-related symmetry considerations. They appear in two different orientations within SCNTs as demonstrated in Figure 7.1a. Idealized junction elements can represent single nodes or Y-junctions of arbitrary order. The orientation of the junction highlighted in blue in Figure 7.1a is called type 1 whereas the junction colored in red is of orientation type 2. This stage is labeled as S_1^t during the construction process described in Section 3.2. The next construction step leads to S_2^t in which S_1^t is copied, moved in xy-direction and connected to the original junction. A schematic drawing of a general S_2^t is shown in Figure 7.1b. In the following, this structure is called *Mirrored Z-like Structure (MZS)* due to its shape. These MZSs which always consist of four Y-junctions are the basic concept within CSGs as they reappear over and over again within the tubes. It is important to note that all nodes that lie within an MZS have the same leading and second leading tuple entry. No node outside a particular MZS has the same combination of these two entries. Hence, an MZS can unambiguously be identified by a tuple of two values: (leading entry, second leading entry).


(a) Two Y-junctions in a different orientation as they appear within SCNTs. This configuration corresponds to the intermediate construction step S_1^t for a tube of order L.


(b) A mirrored Z-like structure (MZS), consisting of four Y-junctions. It corresponds to the step S_2^t of the construction of SCNTs.

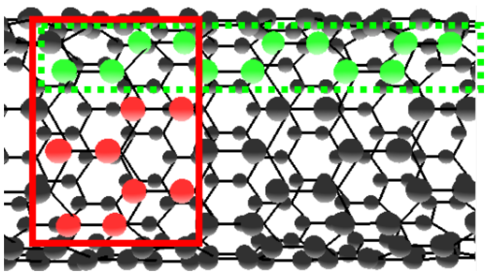

(c) Image selection of an order 0 tube. Vertically stacked (red) MZSs are rotational symmetric, those on the horizontal line (green) are translational symmetric.

Figure 7.1.: Details for general Mirrored Z-like Structures.

The leading two tuple entries are indicators for translational and rotational symmetry as demonstrated in Section 3.3. Thus, all MZSs are closely related with others by symmetry relations. Figure 7.1c demonstrates this fact for an order 0 tube. There are three vertically stacked MZSs related by rotational symmetry within the solid box on the left side. Three MZSs are depicted that are related by translational symmetry, visualized in a horizontal line at the top of Figure 7.1c within the dotted rectangle.

Following our conventions, the tube is aligned with the x-axis and the leading indices of the tuples increase from left to right. On the left side, some other nodes can be noticed in Figure

7.1c beside the rectangles. They belong to the left boundary of the tube and all of those nodes are identified by tuples with a leading 0 and 1. Hence, the following three MZSs contain tuples with leading 2, 3 and 4 respectively. The right boundary is formed by the two rings with the highest leading indices. We assume for the rotational symmetry that the identifying second index grows from bottom to top on the front side of the tube.

Combinations of rotational and translational symmetry, in this thesis called *combined symmetry*, have to be resolved successively, as indicated in Figure 7.2. This combined symmetry is the main principle behind GSCs and is exploited to compress the graphs. Analog to the definition of base-symmetry elements for translation and rotation, base-symmetry elements for combined symmetry can be defined. We assume that the base-symmetry node for combined symmetry for the node $(4, 5, 0, 1)$ is searched. In the first step, it is required to identify the base-symmetry node for translation for $(4, 5, 0, 1)$. This is indicated by the magenta arrows in Figure 7.2 and the node $(2, 5, 0, 1)$ is retrieved. The second step resolves the rotational symmetry and identifies $(2, 3, 0, 1)$ as the base-symmetry node for rotation of node $(2, 5, 0, 1)$, indicated by the red arrows. At the same time, it is the base-symmetry node for combined symmetry of node $(4, 5, 0, 1)$.

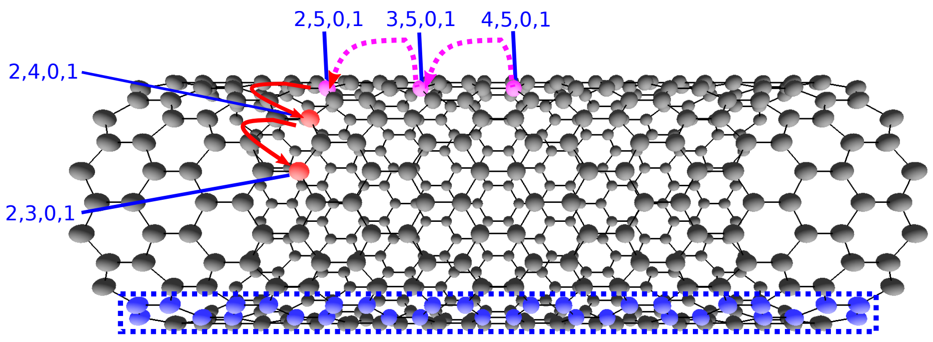


Figure 7.2.: Resolving combined symmetry with jumps in translational and rotational direction. Connection of top and bottom of the former graphene in blue.

The main idea behind CSGs is that not only the considered nodes can have symmetry-relations to other nodes in the graph but, at the same time, all nodes in their neighborhoods underlie the same relation. Hence, if the symmetry-relation of the considered node is determined, its neighborhood can be directly and efficiently constructed with this information. This removes the need to store the neighborhood lists of all nodes in the graph.

7.3 Realization of Compressed Symmetric Graphs

It is possible to implement the resolution of combined symmetry with offset calculations within the symmetric parts of the SCNT due to the regularity of SCNTs and an appropriate lexicographical ordering. This procedure is described in the sequel. The special cases for nodes near the left and right boundary are described in Section 7.3.1. The problem of nodes in regions where the former top and the bottom of the corresponding sheet are connected is covered in Section 7.3.2. This thesis already discussed ways of employing tuples to identify symmetry in Section 3.3. In that case, these two points do not arise. However, working directly on the tuples is a time-intensive procedure as [21] demonstrated, which is why the fast index calculations are preferred for the realization of CSGs.

There is just one offset for translation and one for rotation in the symmetric area of the SCNT that lead from one symmetric node to another. These offsets can be determined directly during the construction process. They are dependent on the parameters (d_0, l_0, d_x, l_x) that are chosen for the tube construction. The number of nodes within one junction element, defined as a_{junc}, is dependent on d_x and l_x. A jump to the next ring is required to move from one translational symmetric node to another adjacent one. The number of nodes within a ring is given by $\text{off}_{trans} = d_0 * 4 * a_{junc}$ because d_0 determines how many MZSs are stacked per ring. Furthermore, each MZS consists of four junctions, and each junction element consists of a_{junc} nodes. Jumps from one MZS to stacked neighbors are required to resolve rotational symmetry. Hence, $\text{off}_{rot} = 4 * a_{junc}$ nodes need to be omitted. Thus, it is necessary to count the jumps between adjacent rings j_{trans} for translational and between neighboring MZSs for rotational symmetry j_{rot} in order to resolve combined symmetry. The total offset to move from one node to its base-symmetry node for combined symmetry is $j_{trans} * \text{off}_{trans} + j_{rot} * \text{off}_{rot}$. The order in which these resolution steps are performed has no influence on the result in theory.

In addition to off_{trans} and off_{rot}, the ranges of the base-symmetry nodes for translation and rotation need to be stored, respectively. They are required to determine whether a ring is reached that contains base-symmetry nodes for translation after performing a jump. Going on to the non-symmetric rings at the boundary needs to be prevented. Similarly, it must be prevented that jumps lead to MZSs which lie near the critical line that connects top and bottom of the graphene for rotational jumps.

For the example tube $(16, 16)$ in Figure 7.2, it is $d_0 = 16$ and $a_{junc} = 1$ since junction elements are single nodes. This means that $\text{off}_{trans} = 16 * 4 * 1 = 64$ and $\text{off}_{rot} = 4$ in this case. In the global index system, node $(4, 5, 0, 1)$ has index 262. If we apply the resolution procedure for combined symmetry as illustrated above, we first have to resolve the translational symmetry. To that end, a jump in translational direction is performed arriving at $(3, 5, 0, 1)$ with index 198. Afterward, another jump to node $(2, 5, 0, 1)$ with index 134 is required, since 198 lies outside of the range of base-symmetry nodes for translation. In a second step, the rotational symmetry is resolved. We start with a leap to $(2, 4, 0, 1)$ (index 130) and a check reveals that $(2, 4, 0, 1)$ is not in the range of base-symmetry nodes for rotation, resulting in another leap in rotational direction reaching $(2, 3, 0, 1)$ with index 126. This node is the base-symmetry node for combined symmetry. In total, the resolution process requires the subtraction of $(2 * 64 + 2 * 4) = 136$ from the start index 262. Four boundary checks are performed during the resolution process.

The neighborhood of node $(4,5,0,1)$ can now be constructed from the neighborhood of $(2,3,0,1)$ by just copying it and adding an offset of 136 to the index of each neighbor node of $(2,3,0,1)$. This permits to define the term *regular neighborhood* in the following way:

Definition 1 *Two neighborhoods N_{n_1} and N_{n_2} are called regular if they are both complete and $N_{n_1}[i] - N_{n_2}[i] == N_{n_1}[j] - N_{n_2}[j]\ \forall i,j \in [1,22]$.*

Put differently, if we know the indexing scheme of the regular neighborhood N_{n_1}, we can easily deduce the indexing scheme of N_{n_2} by just adding or subtracting an offset. A neighborhood not fulfilling the definition above is called an *irregular neighborhood*.

Only one regular neighborhood needs to be kept in memory while all regular neighborhoods of symmetric nodes can be dynamically determined, if requested. Figure 7.3 gives a first visual impression of the compression that CSGs state to realize. It shows all those 6352 nodes whose neighborhood needs to be stored explicitly out of the original 39414 nodes of a $(1,4,8,14)^1$ tube. The remaining 33062 nodes whose neighborhood can be dynamically reconstructed are omitted in the picture.

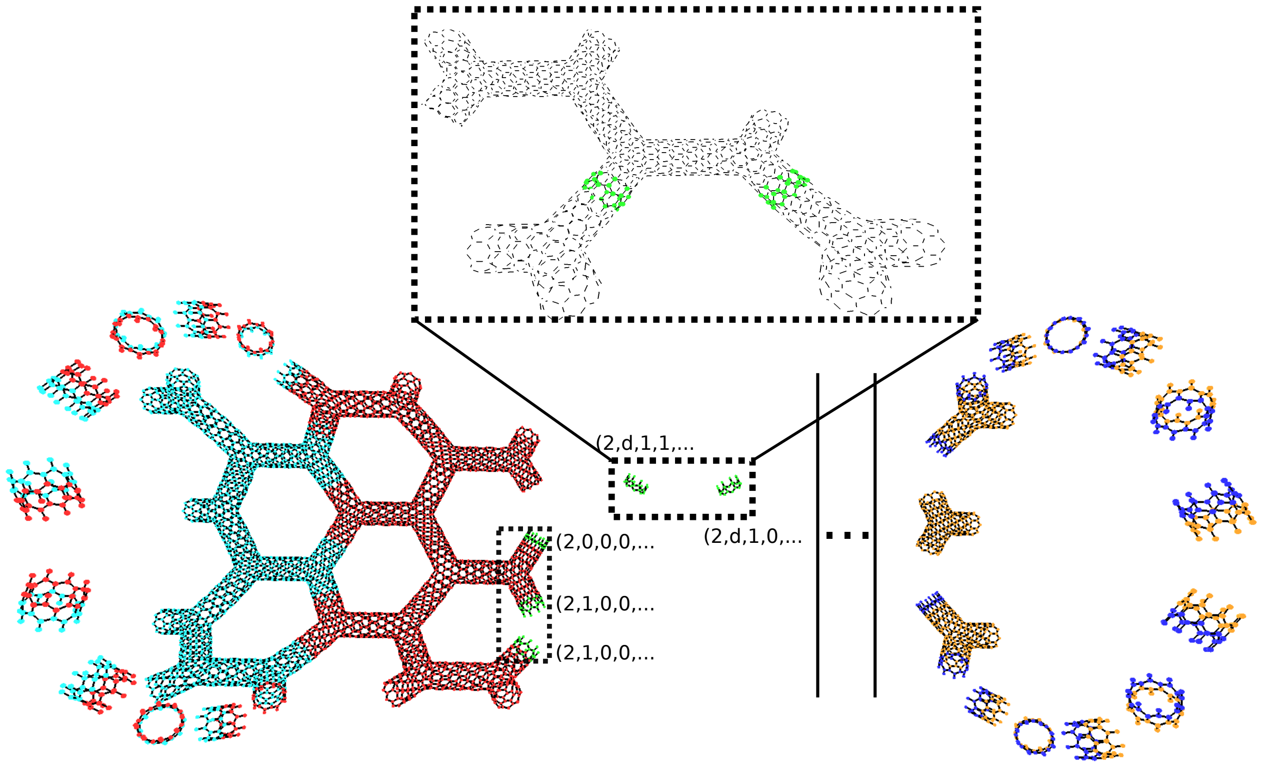
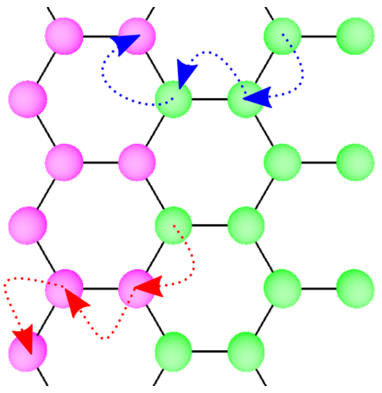


Figure 7.3.: Elements of a $(1,4,8,14)^1$ tube whose neighborhoods a CSG needs to store explicitly. Different colors indicate the different rings (cyan=0-, red=1-, green=2-, blue=$(l-1)$- and orange=l-leading ring). Dots symbolize that there exist more rings. The black letters show the first four entries of the tuples in the respective parts with the abbreviation $d = (d_0 - 1)$.

7.3.1 Nodes on the boundaries of the tube

Nodes lying on the boundary region belong to the non-symmetric nodes. Thus, they possess an incomplete neighborhood or have nodes in their neighborhood which have incomplete neighborhoods themselves. Hence, the neighborhoods for boundary nodes are irregular and the MZSs which form the boundary rings are also incomplete. Fortunately, this does not mean that all neighborhoods of those nodes need to be stored, because incomplete MZSs within those boundary rings all have the same structure. This is visualized in Figure 7.4a, where a cutout of the boundary part of the unfolded sheet for the $(16, 16)$ tube of Figure 7.2 is drawn. On the right side, we see the complete MZSs of the ring with leading ones in green. The remainder of the sheet on the right side of the green nodes is omitted. The left side shows the incomplete MZSs of the 0-leading ring in magenta. It is possible to determine an offset which represents a jump in rotational (vertical) direction from one incomplete MZS to its neighbor, because of the identical structure of those MZSs. This offset is called $\mathrm{off}_{\mathrm{rotleft}}$. In the case of Figure 7.4a it is $\mathrm{off}_{\mathrm{rotleft}} = 3$. Figure 7.4b, in which an excerpt of a super sheet for a $(1, 4, 16, 16)^1$ tube is depicted. It demonstrates that the same holds for tubes of higher order: For this structure $\mathrm{off}_{\mathrm{rotleft}}$ is 528 while its regular offset in the symmetric part is $\mathrm{off}_{\mathrm{rot}} = 704$. Of course the same is possible on the right end with a respective offset $\mathrm{off}_{\mathrm{rotright}}$ for tubes of all order.


(a) Incomplete MZSs on the left boundary of a sheet for a $(16, 16)$ order 0 tube. The two chains of arrows indicate that nearly each node with a 1-leading tuple (green) has nodes with an irregular neighborhood in its own neighborhood.

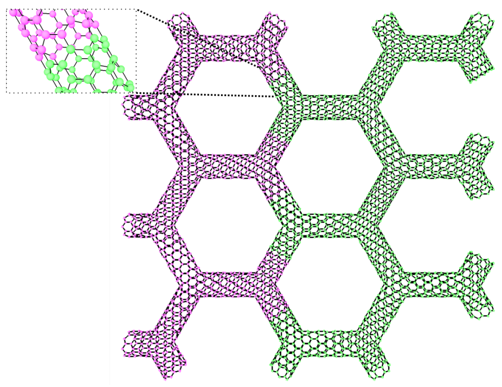

(b) Incomplete MZSs on the left boundary of an unfolded sheet for a $(1, 4, 16, 16)^1$ tube. The insert shows that in the case of higher orders only a few 1-leading nodes (green) have nodes with irregular neighborhoods in their own neighborhood.

Figure 7.4.: Adjacent MZSs in tubes of order 0 and 1.

Figures 7.4a and 7.4b also demonstrate another important point: Tubes of order 0 require a special treatment compared to all tubes of higher order. As Figure 7.4a indicates with the red and blue chains of arrows, nearly each node with a 1-leading tuple (green nodes) has nodes in its neighborhood that themselves possess an irregular neighborhood. This implies that those nodes have also an irregular neighborhood and can not be used as base-symmetry nodes.

Consequently, the CSGs assume that base-symmetry nodes for tubes of order 0 start at the ring with 2-leading nodes. The same holds for the last two rings, meaning that the last symmetric nodes can be found in the ring with the $l-2$-leading tuples.

However, the situation is different for tubes of higher order. Of course, all nodes in the ring with the 2-leading nodes are symmetric because no chain of length 3 can be constructed connecting a 2-leading node to a non-symmetric one. In the case of higher orders, even most of the 1-leading nodes have a regular neighborhood. Hence, they can be used as base-symmetry nodes. There are only few exceptions, which are highlighted in Figure 7.4b. The zoom-in to the connection of MZSs from the 0-leading and the 1-leading ring indicates that near to the connection-line there are some green 1-leading nodes that have magenta colored nodes in their resulting neighborhood. Consequently, those neighborhoods are irregular. Not all of them need to be stored because of rotational symmetry between those nodes. Only one of those MZSs is saved, because it is capable of serving as base-symmetry element for rotation in the 0-leading ring.

7.3.2 Nodes near the zero line

Apart from the boundary-nodes, a second category of nodes exists which requires special treatment. These are those nodes that are adjacent to the so-called *zero-line*. In a (super) sheet, the zero-line runs through those chemical bonds that connect the bottom of the sheet to its top when the sheet is rolled up. The zero line for the $(16, 16)$ tube is shown in Figure 7.5 as dotted blue line. The magenta nodes above the line are the former nodes at the bottom of the sheet and the green nodes those at the top, respectively. Global indices increase from the bottom to the top within a ring of a sheet. Nodes adjacent to the zero line have either low or high indices. Thus, all nodes above the zero line that have at least one node below the zero line in their neighborhood possess irregular neighborhoods. The same holds the other way round. Consequently, these nodes can not be used as base-symmetry nodes for rotation.

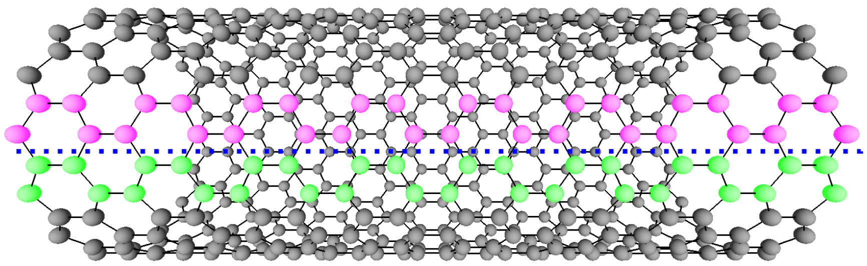


Figure 7.5.: Zero line within a $(16, 16)$ tube shown in blue with the bottom of the graphene in magenta.

However, the irregularity within the neighborhoods of nodes near the zero line is constant along the x-axis. Hence, we can avoid to store them all. Only those nodes are required which are affected by the zero line and lie in one of both boundary rings or in the base-symmetry ring.

Now, reconsider Figure 7.3. For the 0-leading part in cyan, all those nodes are stored that have a 0 or a $(d_0 - 1)$ as second leading tuple entry because of their neighborhood to the zero line. Additionally, all nodes with a 1 as second leading entry are saved as they serve as base-symmetry nodes for rotation within the 0-leading ring. There are also some small additional 0-leading parts around the circumference of the SCNT. These are regions where 0-leading nodes have 1-leading nodes in their neighborhood. They are stored since neighborhoods of those nodes are irregular.

The 1-leading part in red contains nearly all base-symmetry nodes for combined symmetry for the remainder of the tube. It stores all MZSs with a 0, 1 and $(d_0 - 1)$ at the second leading entry to avoid problems with the zero line. Additionally, those few nodes with connections to 0-leading tuples are saved.

In the 2-leading ring, only very few nodes need to be stored. They are drawn in green and are highlighted by dotted rectangles. The numbers beside the highlighted parts show the first four entries of the corresponding tuples in these parts. The insert demonstrates the context of the green nodes for the part in the middle of the grid. In principle, two kinds of problems arise if those green parts are left out. First, we discuss the case in which the $(2, 0, 0, 0, \dots)$ and $(2, 1, 0, 0, \dots)$ nodes are not present. Assume that we search the neighborhood for node $(3, 0, 0, 0, \dots)$ where the $(\dots)$ part is identical to one of the green $(2, 0, 0, 0, \dots)$ nodes. The algorithm would perform a step in translational direction, reaching node $(2, 0, 0, 0, \dots)$ and go on to $(1, 0, 0, 0, \dots)$ since $(2, 0, 0, 0, \dots)$ is not registered. $(1, 0, 0, 0, \dots)$ is connected to 0-leading nodes which would corrupt the neighborhood retrieved. Hence, the first group of green nodes prevents such cases.

The meaning of the second group of green nodes with tuples $(2, (d_0 - 1), 1, 0, \dots)$ and $(2, (d_0 - 1), 1, 1, \dots)$ is explained in the following section. Dots in the figure represent further rings that follow the 2-leading part in the direction of the x-axis. No structural information needs to be stored for those rings.

The nodes stored next lie in the $(l - 1)$-leading ring. Most of the nodes within this ring are symmetric, except those that have l-leading nodes in their neighborhood which belong to an incomplete and thus irregular ring.

The incomplete MZSs with 0, 1 and $(d_0 - 1)$ at the second leading entry are saved for this last irregular l—leading ring, analogously to the 0-leading part. The same holds for those parts that are connected to the $(l - 1)$-leading ring on the circumference of the tube through neighborhood relations as well.

7.3.3 Two implementations of the retrieval procedure

The actual algorithm for retrieving combined symmetry, as illustrated in Figure 7.2, is somewhat more complex. In principle, there are two ways to generate the CSGs. The first one is to regard the MZSs as an indivisible unit and to always store the whole MZS, if it contains nodes with irregular neighborhoods. This leads to very fast construction and easy retrieval of symmetric nodes since an MZS can directly be added to the compressed graph, if the existence of one irregular node in it is proven. Furthermore, the retrieval of symmetry information would be very fast since the structure only needs to check whether the base-symmetry MZS containing the searched node exists. This procedure works well for tubes of low order with a large length or diameter since only less than 5% of extra nodes need to be saved. However, the situation changes considerably when the order is increased. This normally results in lower values for d_0 and l_0 and simultaneously in more nodes within each MZS. In those cases, the difference in the number of stored neighborhoods can exceed 20% which is not reasonable. Hence, the CSGs try to store as few nodes as possible and the storage of incomplete MZSs is allowed. However, this means that not all base-symmetry nodes are located in the same MZS or not even in the same ring. In SCNTs of order 1, most base-symmetry nodes lie in the 1-leading ring, but a few nodes in this ring have connections to the 0-leading ring and have irregular neighborhoods. In that case, the corresponding base-symmetry nodes lie at the left side of the 2-leading ring. These numbers are higher for order 0 by one due to their small, constant number of four nodes in each MZS. The CSGs contain two different ways to search for the index of a base-symmetry node and both distinguish three different cases depending on the input node whose neighborhood is requested:

1. The node lies in the 0-leading ring of the tube.
2. The node lies in the l-leading ring of the tube.
3. The node lies in the remainder of the tube.

Both ways only differ in case 3 being the most frequent and also computationally most expensive. The index of the requested node sym_idx is passed to the function calcBaseIdx. A simplified C++ pseudo code version for the first approach is shown in Listing 7.1. It is not initially clear in which ring the base-symmetry node can be found due to the existence of incomplete MZSs. Hence, both, the 1-leading and the 2-leading ring need to be covered.

```
int CompressedGraph::calcBaseIdx(int sym_idx) {
if(neigbormap.find(sym_idx))
  return sym_idx;

// Go to ring 2 and search in it
int idx_base = resolveTransSymmetryToRingTwo(sym_idx);
if(neigbormap.find(idx_base))
  return idx_base;

int idx_before_rot = idx_base
```

```
  idx_base = resolveRotSymmetryInRingTwo(idx_base);
  if(neigbormap.find(idx_base))
    return idx_base;

  // Undo rotation
  idx_base = idx_before_rot;

  // Go to ring 1 and search in it
  idx_base = resolveTransSymmetryToRingOne(sym_idx);
  if(neigbormap.find(idx_base))
    return idx_base;

  idx_base = resolveRotSymmetryInRingOne(idx_base);
  if(neigbormap.find(idx_base))
    return idx_base;
}
```

Listing 7.1: Search for the index of a base-symmetry node within CSGs with the first method.

Consequently, retrieving the neighborhood information for a node requires up to five accesses to the internal hash table. One possibility to speed up the access to node information is to switch the order of the resolution of rotational and translational symmetry and to store several additional nodes which is realized in the second way, shown in Listing 7.2. There are two small, additional parts in the 2-leading ring at the center of Figure 7.3 within the dotted rectangle. They are also shown in the insert. These nodes belong to the upper part of the MZS with the highest second leading index and have tuples of the form $(2,(d_0-1),1,0,\ldots)$ and $(2,(d_0-1),1,1,\ldots)$. They are connected to the MZSs $(2,0)$. These are nodes starting with $(2,7,1,\ldots)$ in the case of the depicted $(1,4,8,14)$[1] tube. They are needed because of the slightly modified calcBaseIdx routine that is shown in C++ pseudo code in Listing 7.2. In contrast to Listing 7.1, only four instead of five find-operations take place in the hash table.

```
  int CompressedGraph::calcBaseIdxOptimized(int sym_idx) {
  if(neigbormap.find(sym_idx))
    return sym_idx;

  // Go to ring 2 and search in it
  int idx_base = resolveTransSymmetryToRingTwo(sym_idx);
  if(neigbormap.find(idx_base))
    return idx_base;

  idx_base = resolveRotSymmetryInRingTwo(idx_base);
  if(neigbormap.find(idx_base))
    return idx_base;

  // Go to ring 1 and search in it
```

```
    idx_base = resolveTransSymmetryToRingOne(sym_idx);
    if(neigbormap.find(idx_base))
      return idx_base;
}
```

Listing 7.2: Optimized search for the index of a base-symmetry node.

The following example on the $(1,4,8,14)^1$ tube demonstrates why additional nodes are required for the second search procedure. Assume that the combined symmetry of one of the nodes starting with the sub-tuple $(4,7,1,\dots)$ should be resolved. For the first procedure this means that the translational symmetry to ring 2 is resolved in a first step, arriving at the corresponding node $(2,7,1,\dots)$. Since this node is not stored, jumps in rotational symmetry are performed leading to node $(2,1,1,\dots)$, which is not saved either. The algorithm goes back to the index of $(2,7,1,\dots)$ and from there to $(1,7,1,\dots)$ in translational direction. This node is stored and its index is returned as base-symmetry node corresponding to $(4,7,1,\dots)$.

The first two steps, checking $(2,7,1,\dots)$, which is not existent, and afterward moving to $(2,1,1,\dots)$ that neither exists, are identical for the second procedure of Listing 7.2. The following step in translational direction leads to node $(1,1,1,\dots)$. Now, there are two cases. Assume that the insert in Figure 7.3 shows the corresponding part in the 4-leading ring which is feasible since 2- and 4-leading rings have completely the same structure. In the first case, the searched node lies above the green nodes in the remainder of the MZS $(4,7)$. Since all nodes in this area possess a regular neighborhood, the resolution process finds the correct match with node $(1,1,1,\dots)$ in MZS $(1,1)$. In the second case, the input node lies in one of both green areas. Then, the returned node in MZS $(1,1)$ is infeasible since the green nodes have irregular neighborhoods because of their adjacency to the zero line.

For this example tube, only 48 nodes exist in the green region which is equal to less than 0.8% of the total nodes of the whole CSG. About $11,000$ nodes need to be stored additionally for higher order tubes like $(2,3,4,6)^2$ with $800,000$ nodes. This corresponds to about 1.4% of the overall number of nodes. In general, the higher the order of the tube and the larger the junctions are in relation to the length of the SCNT, the higher is the percentage of extra nodes. The performance difference is lower than 10% which results from the fact that the base-symmetry node is found in the last find-operation in both ways. Consequently, the CSGs are configured to use the algorithm from Listing 7.1 for the tests in this thesis and to store as few nodes as possible. This further optimizes the memory management of the application while the alternative way can be chosen at compile time.

7.4 Implementation and optimizations

CSGs are realized as own class **CompressedGraph** which implements the same interface as **TreeGraph**, **IndexGraph** and **HashGraph** already presented. This, in particular, allows to iterate over all nodes and to randomly access nodes, hiding the complexity of their construction (Section 7.4.1) and in retrieving additional node information (Section 7.4.2).

7.4.1 Construction

Construction of a CSG requires some additional memory in its current implementation. Initially, an **IndexGraph** is instantiated which represents the highest level junction in the tube. Afterward, the following rings or parts of it are constructed employing the IndexGraph junctions as building blocks, because these rings contain all non-symmetric and base-symmetry nodes: The 0-, 1-, $(l-1)$- and l-leading rings are constructed completely, while the MZSs $(2,0)$, $(2,1)$ and $(2,d)$ are added for the 2-leading ring. In the last step, a filtering is performed that removes all symmetric nodes from the 1—, 2- and $l-1$-leading rings. Additionally, it takes out those nodes in the two boundary rings which can be reconstructed by employing the boundary offsets $\text{off}_{\text{rotleft}}$ and $\text{off}_{\text{rotright}}$ for rotational symmetry.

7.4.2 Avoiding the storage of tuples

In addition to savings in storage of the neighborhood information by dynamically reconstructing parts of it, CSGs also help to further reduce the overall memory consumption by avoiding the explicit storage of tuples. They exploit the fact that each tuple can be mapped to its unique serial index and that a reverse mapping is unique as well. Additionally, not each caller is interested in the tuple when querying a node. Actually, most calls during the simulation have the goal of retrieving the position or the neighborhood, while the tuples are mainly of interest during construction of the tube, data preparation and visualization.

Hence, just the serial index is stored and the request-procedure can be configured to reconstruct and return the tuples on demand. This method does not negatively affect the speed of the simulation.

In this way, a lot of memory can be saved. Assume a very large $(2,6,4,4)^3$ tube with more than $5.2 * 10^9$ nodes. Its tuples have a length of 28 since the tube is of order 3. Employing 16 bit short integers for the single entries of the tuple vector results in about 270 GB of occupied memory for the tuples, which we can avoid. The advantage further grows with increasing size of the tube and especially its order.

The CSGs adapt the procedure to map tuples to a serial index from the **IndexGraph** as explained in Section 6.1.2 and realize an inverse transformation. To that end, the tube part and the junction part, which contribute to the overall serial index, are covered differently. The original calculation of the serial index from the tuple processes the tuples from the lowest entry to the leading entry. First, the junction part is taken into account and then the tube part. The unfolding procedure needs to turn around the order and resolves the tube part to recover the tuples correctly.

The algorithm only needs to know the structure of the implicit tree representing the tube part to be able to reverse the mapping of a tuple tpl to its serial index ser_idx. This information is contained within an array subtree_size storing the number of nodes in the sub-trees of the four levels of the tree, as explained for the tree-based flattening in Section 6.1.1. Then, the tube part can be reconstructed by the code in Listing 7.3 in reversed order.

```
int act_idx = ser_idx;
for (int i = tpl.size() - 2; i >= 0; i=i-1) {
  int tpl_entry = act_idx / tube_subtree_size[i];
  tpl.push_back(tpl_entry);
  act_idx = act_idx - tpl_entry * tube_subtree_size[i];
}
```

Listing 7.3: Reconstructing the tube part of a tuple in CSGs

Consider the tree of Figure 6.1 with its tuple extent $(3,3,4)$. This implies that the corresponding tree_subtree_size[1] array has four entries $\{1, 4*1, 3*4*1, 3*3*4*1\} = \{1,4,12,36\}$[2]. In order to restore the tuple for the serial index 33, Listing 7.3 executes three iterations with the state after each iteration being presented in Table 7.1.

Table 7.1.: Reconstructing the tuple of node $(2,2,1)$ with index 33 of Figure 6.1 with the procedure given in Listing 7.3.

	before loop	after 1st iteration	after 2nd iteration	after 3rd iteration
act_idx	33	9	1	0
tpl_entry	/	2	2	1
tpl	/	$\{2\}$	$\{2,2\}$	$\{2,2,1\}$

The resolution process of the junction part is somewhat more complex. The input for this step is the act_idx that is calculated in the last loop iteration of Listing 7.3. The idea is outlined in Figure 7.6 which shows the initial situation in the junction part of the tuple on the left side in which the junction part is compressed into a hierarchy of maps in **IndexGraph**.

The idea behind the reconstruction of the sub-tuple for the junctions is that junction levels are processed one after the other. To that end, the information for the corresponding level is temporarily uncompressed in the hierarchy from top to bottom. Uncompressing a junction level means that the implicit tree with depth 8 is reconstructed that encodes a specific junction level. In Figure 7.6 we see the implicit tree for junction level $L-1$ in the middle and one for junction level $L-2$ on the right side.

It is important to know how many different tuples are encoded by each junction level although it is still compressed. This number is the product of the sizes of all local correspondence maps

[1] We have to distinguish sub-trees for tree and junction part in the case of **IndexGraph**.
[2] The last entry is not required for the tuple reconstruction but important in other contexts.

for the lower junction levels. The red boxes surrounding and grouping the respective lower part of the hierarchy in the middle and on the right side in Figure 7.6 indicate: They can be treated as a black box in the sense that only the number of contained tuples is of interest.

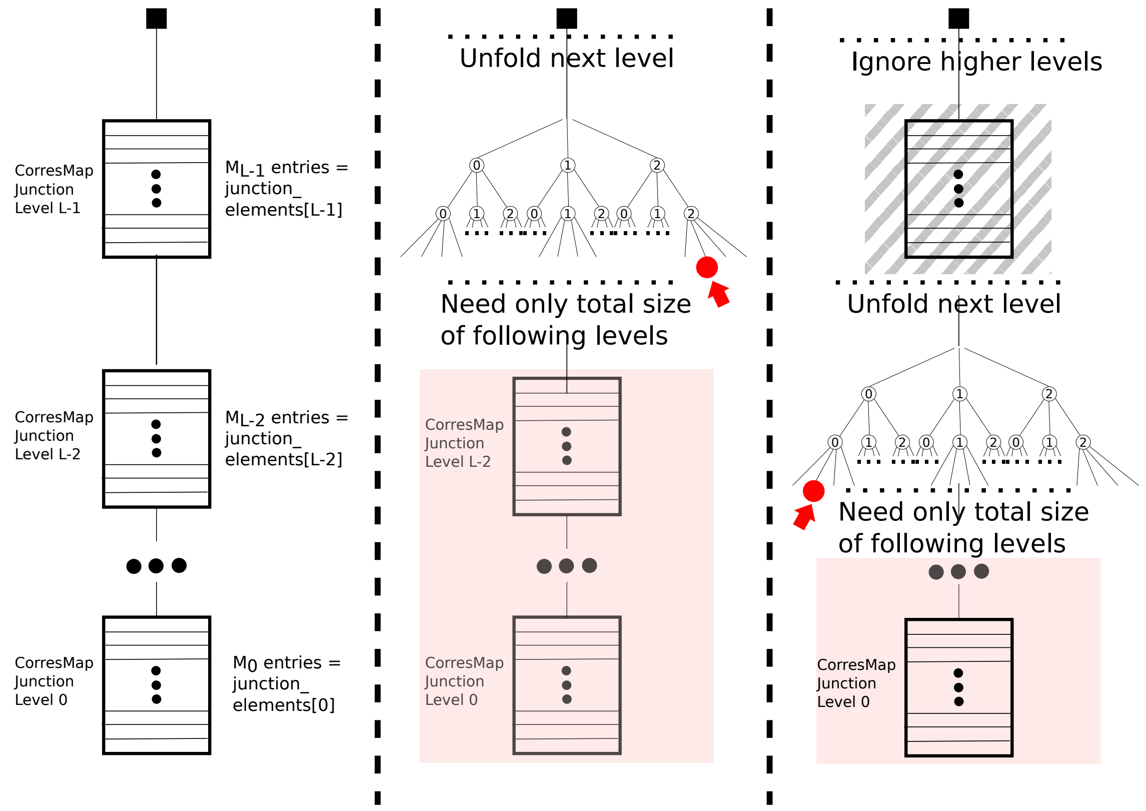

CorresMap Junction Level L-1
M_{L-1} entries = junction_ elements[L-1]
CorresMap Junction Level L-2
M_{L-2} entries = junction_ elements[L-2]
CorresMap Junction Level 0
M_0 entries = junction_ elements[0]
Unfold next level
Need only total size of following levels
Ignore higher levels
Unfold next level
Need only total size of following levels
0
1
2

Figure 7.6.: Schematic of the procedure to unfold tuples.

The number of elements in different junction levels is already available in the array junction_elements that is created during the instantiation of **IndexGraph** (see Section 6.1.2). An integer variable junction_offset stores the product of all entries from junction_elements. Additionally, an array is created once and stored within CSGs which represents the sub-tree sizes for the implicit tree of the whole junction part, i.e., a tree with the order of the tube times eight levels. This array is called junction_subtree_size.

The contribution to the serial index ser_idx of each junction level is extracted with this information and translated to the sub-tuple. To that end, the highest junction level is conceptually unfolded from the compact map to the implicit tree as shown in the middle of Figure 7.6. Remember that in these trees each possible tuple was assigned a local serial index loc_ser_idx from the left to the right and each actually existing tuple additionally received a consecutive index in the same direction.

The leaf, i.e., the local serial index loc_ser_idx corresponding to the respective consecutive index is searched in the unfolded tree. The sub-tuple is recovered from the loc_ser_idx t. This process is indicated by the red leafs and arrows in the drawing. Computationally, this search-step is equivalent to a lookup in the local correspondence map of the current level for the consecutive index that delivers the local serial index. The sub-tuple for loc_ser_idx can directly be calculated in an equivalent fashion to the sub-tuple for the tube part considering this information.

Listing 7.4 summarizes the resolution procedure of the sub-tuple for the junction part. Lines 5 – 6 realize the unfolding of the respective junction level, while lines 10 – 14 realize the actual reconstruction of the sub-tuple in a similar fashion as Listing 7.3. The last lines 17 – 18 ensure that the junction levels of the hierarchy already processed are neglected in the following steps. To that end, the contribution of covered junction levels is removed from the current index act_idx as it is done for the nodes already covered from the total number of nodes in the junction part.

```
 1 // Process the junction levels from high to low
 2 for (int lvl = scnt_level - 1; lvl >= 0; lvl=i-1) {
 3   // Find the consecutive index of the sub-tuple for this junction level, e.g.
        unfold
 4   // the sub-tree corresponding to this junction level
 5   int consecutive_index = act_idx / junction_offset; // remainder can be neglected
         in this line
 6   int loc_ser_idx = findLeafNumberInCorrespondenceMap(consecutive_index, lvl);
 7
 8   // Restore each entry of the sub-tuple starting with the highest entry of this
        junction level
 9   // Equivalent to tube part but the implicit tree has a depth 8 this time
10   for (int t = 7; t >= 0; t=t-1) {
11     int tpl_entry = loc_ser_idx / junction_subtree_size[lvl * 8 + t];
12     tpl.push_back(tpl_entry);
13     ser_idx -= tpl_entry * junction_subtree_size[lvl * 8 + t];
14   }
15
16   // Ignore the upper tree-part in the following
17   act_idx -= consecutive_index * junction_offset;
18   junction_offset /= junction_elements[i];
19 }
```

Listing 7.4: Reconstruction of the junction part of a tuple in CSGs by processing the junction hierarchy level by level.

7.5 Achieving high compression rates

We define the compression rate ρ to determine the quality of the compression achieved by CSGs as the fraction of the number of nodes that are compressed in the corresponding CSG divided by the number of nodes in the original graph which is denoted by n. Hence, its range lies between 0 % if no compression can be achieved at all and 100 % for the impossible case that all nodes are compressed. The compression rate was evaluated for different tubes of order 0 up to 3 and the results are presented in Table 7.2.

Table 7.2.: Compression results for several tubes. The respective tube configuration, order and the resulting number of nodes are shown in the first two columns. The last two columns demonstrate how many nodes of the original tube remain uncompressed and the resulting compression rate ρ.

Tube	No. of Nodes	Nodes uncompressed	Compression rate (ρ)
$(256, 256)$	$1.3 * 10^5$	$2.6 * 10^3$	98 %
$(512, 512)$	$5.4 * 10^5$	$5.1 * 10^3$	99 %
$(1, 4, 12, 18)^1$	$7.6 * 10^4$	$6.1 * 10^3$	92 %
$(2, 6, 12, 18)^1$	$1.6 * 10^5$	$1.2 * 10^4$	93 %
$(1, 4, 8, 355)^1$	$1.0 * 10^6$	$6.5 * 10^3$	99%
$(2, 6, 12, 116)^1$	$1.0 * 10^6$	$1.2 * 10^4$	99%
$(1, 4, 4, 4)^2$	$9.9 * 10^5$	$7.5 * 10^5$	25%
$(1, 4, 8, 4)^2$	$1.9 * 10^6$	$7.5 * 10^5$	61%
$(1, 4, 8, 14)^2$	$6.9 * 10^6$	$7.5 * 10^5$	89%
$(1, 4, 8, 20)^2$	$9.9 * 10^6$	$7.5 * 10^5$	92%
$(1, 2, 4, 4)^3$	$1.6 * 10^7$	$1.2 * 10^7$	25%
$(1, 2, 8, 8)^3$	$6.5 * 10^7$	$1.2 * 10^7$	81%

A very high compression rate of more than 98% can be achieved for the two tubes of order 0. This is possible because of the large values of the parameters d_0 and l_0 which generate a high number of symmetric MZSs and most of them lie in the interior of the tube.

We observe comparable and even higher compression rates for the order 1 tubes. A special case is the SCNT $(1, 4, 8, 355)^1$ which represents a very thin long tube with a lot of inherent translational symmetry. Hence, a compression rate of 99% is attainable. The compression rates for smaller tubes like $(1, 4, 12, 18)^1$ are lower but still remarkably high.

The lowest compression rate of 25% of all tested tubes is attained for the SCNTs $(1, 4, 4, 4)^2$ and $(1, 2, 4, 4)^3$ which are chosen intentionally to demonstrate the worst case scenario. The $(1, 4, 4, 4)^2$ tube is the smallest possible SCNT of order 2. So most MZSs lie at one of both ends or near the zero line. Consequently, the inherent translational and rotational symmetry between the elements is highly limited. However, $(1, 4, 4, 4)^2$ is also a good demonstration to justify the decision of allowing the storage of incomplete MZSs as described in Section 7.4.1: If only complete MZSs would be stored, there would be no compression for such kind of tubes at all.

There is also a second important point to these worst case tubes: $(1, 4, 4, 4)^2$, $(1, 2, 4, 4)^3$ and all other minimal tubes for the respective order are important for the question how CSGs can extend the range of feasible simulations. As already shown, if all non-symmetric and base-symmetry elements of a tube are stored, it is possible to arbitrarily increase the length l_0 without the need to store more structural data. Only the positions of additional nodes need to be stored. We just need to keep those neighborhoods in memory whose reference node lies in the regions where a non-regular ring is adjacent to a regular ring when the diameter d_0 is increased. So, in principle, if a minimum tube of order L $(d_x, l_x, d_0, l_0)^L$ can be stored with IndexGraphs, CSGs make it possible to also store $(d_x, l_x, d_0 + \Delta d, l_0 + \Delta l)$ where the size of Δd and Δl is limited by additional position data. This is demonstrated by the tubes $(1, 4, 8, 4)^2$, $(1, 4, 8, 14)^2$ and $(1, 4, 8, 20)^2$. These SCNTs only differ in their d_0 and l_0 values. This results in a different number of nodes for IndexGraphs, but the number of nodes in the corresponding CSGs is about the same. In contrast, increasing d_0 and l_0 increases the amount of inherent symmetry and hence also the compression rate. The same holds for the tubes $(1, 2, 4, 4)^3$ and $(1, 2, 8, 8)^3$ of order 3. Additionally, the CSGs leave more free memory even if the corresponding IndexGraph can also be stored with the provided memory resources. As the following chapter demonstrates, this additional memory can be used for other important tasks.

8 Solving for Displacements with an Iterative Approach

In this chapter, we first motivate the development of an iterative, matrix-free solver by highlighting the limitations of a direct solver. Afterward, we summarize the iterative conjugate gradient method that is the base of the matrix-free approach and demonstrate its advantages. We discuss the challenge to appropriately preconditioning the equation system in order to reduce the number of steps required for the iterative method in Section 8.1. The following sections describe the different solver implementations for this thesis, which exploit the structure of SCNTs and the properties of the atomic-scale finite element method employed in different ways. A parallelized solver with fully assembled stiffness matrix is discussed in Section 8.2 which is a reliable baseline for performance comparisons. A highly memory-efficient solver, which completely avoids the storage of values of the stiffness matrix, and thus, considerably increases the range of feasible model sizes is presented in Section 8.3. It is also briefly compared to the runtime of the dockSIM framework. Section 8.4 introduces a highly configurable and flexible solver which is able to cache parts of the stiffness matrix but does not require to assemble it completely. Finally, we propose an additional solver for some special scenarios with only small deformations of tubes, resulting in a considerable reduction of the required stiffness data in Section 8.5.

As mentioned in the related work, the dockSIM code of the MISMO group also implements the atomic-scale finite element algorithm presented by Wackerfuß [34] but employs a direct solver for the equation systems. This results in a double memory overhead: On the one hand, the whole stiffness matrix $\mathbf{K}$ needs to be assembled and kept in memory. This includes all bookkeeping data required for the compressed row storage (CRS) format (see e.g. [82] and Section 8.2). On the other hand, a direct solver as realized in the Pardiso library ([83] ,[84], [85]) consumes additional memory since it is based on an LU decomposition of the matrix. This decomposition normally creates more non-zero elements in the decomposed matrix, the so-called *fill elements*, than the number of non-zero elements in the original sparse matrix.

For example, Li and Demmel [86] proposed a new distributed direct solver and tested its scalability and its memory demand at different sparse matrices. They achieve a pretty good scaling for the factorization time. The efficiency is still up to 77% for 128 nodes. But their work also reveals the high memory demand of direct solvers. Table 8.1 lists some of the test matrices and their properties from [86]. There, $|\mathrm{nze}|$ denotes the number of non-zero elements, and $|\mathrm{nze}|$ / row the maximum number of non-zero elements per row. The authors state that the decomposition fails for single node execution for matrices BBMAT and Twotone since 256 MB per node on their test-system are not sufficient. Hence, they only present the memory that is used per node, if a distributed calculation is employed, including some overhead. A comparison of the size of the matrices in CRS and the overall memory required for the execution on four distributed nodes (columns 5 and 6 in Table 8.1) demonstrates that up to 33 times more memory is required during execution than for the storage of the matrix alone. Based on the numbers of the distributed case, we suspect that the values for single node execution lie in the same range, regarding matrix Twotone with less than 10 MB of storage but overcharging a node with 256 MB RAM during LU decomposition.

Table 8.1.: Properties of some example sparse matrices. Values taken from [86]. 32 bit datatypes are assumed.

Name	Order	|nze|	|nze| / row	Size CRS in MB	Size in MB on 2*2
Wang4-Tank	26068	177196	7	1.45	136
Inv-Extrusion-1	30412	1793881	59	13.80	192
Twotone	120750	1224224	10	9.80	320
BBMAT	38744	1771772	45	13.66	456
Mixing-Tank	29957	1995041	67	15.34	224

The increase of the non-zero values is another indicator for the memory overhead of direct solvers. The two main authors of the Pardiso library Schenk and Gärtner proposed a method for an efficient LU decomposition on shared memory multiprocessors [87]. They evaluated their scheme on 22 different sparse matrices. Although they focus on throughput and scalability of their solution, they list the number of non-zeros within the input matrix **A** and in the factorized matrix. In average, the number of non-zeros increases by a factor larger than 10 and by even 26.1 and 26.6 in the two worst cases. Integration of well known reordering and symbolic factorization schemes for symmetric sparse matrices into the algorithm [87, p. 160] cannot prevent this behavior.

In addition to the results of [86] and [87], the vendor COMSOL performed measurements of memory usage of their software, which internally employs Pardiso among other solvers. Run-time and memory required for the solution of a linear equation system grow close to quadratic with the number of degrees of freedom in the system[1].

These facts demonstrate that a direct solver is inherently limited in the problem size that it can cope with, especially on shared-memory systems. Consequently, the first step is to replace the direct solver by an iterative method to allow simulation of larger SCNTs The conjugate gradient (CG) method, originally proposed by Hestenes and Stiefel [88], is chosen. It is based on the idea that solving $\mathbf{A}^{n\times n} * \mathbf{x}^n = \mathbf{b}^n$ is equal to the task to find the $\mathbf{x}^n$ which minimizes the quadratic form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}$ in the case that **A** is symmetric. A series of estimates is generated in m steps to find this **x**, where, in general, in each step one direction of **x** is minimized. In the case of no rounding errors, this method finds the exact solution $\mathbf{x}^n$ in at most n steps. Due to the limited floating point precision of computer systems, the algorithm normally only delivers a good approximation $\mathbf{x}_m$ of the real solution **x**. The CG-algorithm is

[1] `https://www.comsol.com/blogs/much-memory-needed-solve-large-comsol-models/`, accessed 2017-01-04

mainly targeted at symmetric and positive-definite matrices and these two requirements are fulfilled by the underlying stiffness matrix **K** in the simulation of the SCNT (see Section 5). The residual $\mathbf{r}_i = \mathbf{b} - \mathbf{x}_i$ is calculated for each step i. The original formulation in [88] consists of six different calculations. The initial step is given in Equation 8.1 where $\mathbf{x}_0$ represents an arbitrary estimation for the solution:

$$\mathbf{p}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A} * \mathbf{x}_0 \tag{8.1}$$

Afterward, the calculations in Equation 8.2 are executed for every step i.

$$\begin{aligned} a_i &= \frac{\left|\mathbf{r}_i\right|^2}{\mathbf{p}_i^T * \mathbf{A} * \mathbf{p_i}} \\ \mathbf{x}_{i+1} &= \mathbf{x}_i + a_i * \mathbf{p}_i \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - a_i * \mathbf{A} * \mathbf{p_i} \\ b_i &= \frac{\left|\mathbf{r}_{i+1}\right|^2}{\left|\mathbf{r}_i\right|^2} \\ \mathbf{p}_{i+1} &= \mathbf{r}_{i+1} + b_i * \mathbf{p}_i \end{aligned} \tag{8.2}$$

An analysis of this algorithm reveals that it only consists of primitive matrix-vector operations like sum, subtraction, sparse matrix-vector multiplication or scaling which is one of the main advantages of the CG method. Furthermore, a straightforward implementation has a very low memory footprint. It requires only the storage of the n-vectors $\mathbf{r}_i$, $\mathbf{r}_{i+1}$ and $\mathbf{p}_i$ beside the inputs and the output (**A**, **b**, $\mathbf{x}_i$). The values for $\mathbf{x}_{i+1}$ and $\mathbf{p}_{i+1}$ can directly be placed in $\mathbf{x}_i$ and $\mathbf{p}_i$, respectively since, in contrast to the residual, their old value is not required anymore if the update is calculated. Two optimizations can reduce the number of calculations while not requiring more memory. The sparse matrix-vector multiplication (SpMV) $\mathbf{A} * \mathbf{p_i}$ is calculated two times. It makes sense to pre-compute this product and store it in an additional n-vector since this is the most compute intensive calculation within a step. Additionally, the norm $\left|\mathbf{r}_i\right|^2$ is also calculated twice and the result of the first calculation can be stored and reused. In that case, $\mathbf{r}_i$ is not required after the calculation of $\mathbf{r}_{i+1}$ and its value can directly be overwritten, saving the space of one n-vector. So, the CG method may require the storage of the matrix[2] $\mathbf{A}^{n \times n}$ and between four (**x**, **b**, $\mathbf{r}_i$, $\mathbf{p}_i$) and eight n-vectors when additionally $\mathbf{x}_{i+1}$, $\mathbf{r}_{i+1}$ and $\mathbf{p}_{i+1}$ are stored and the result of the SpMV is saved in a temporary vector.

2 Matrix-free versions, as presented in this thesis, can completely avoid it.

An extension of the original CG method is the so-called preconditioned conjugate gradient (PCG) method [89] where an additional symmetric, positive-definite preconditioning matrix $\mathbf{C} = \mathbf{E} * \mathbf{E}^T$ is employed to modify the system $\mathbf{A} * \mathbf{x} = \mathbf{b}$ to an equivalent system $\mathbf{C}^{-1} * \mathbf{A} * \mathbf{x} = \mathbf{C}^{-1} * \mathbf{b}$, where the condition number of **A** is improved [90, p. 39] and **A** has only a few distinct eigenvalues [91, p. 532f]. The number of required steps is reduced with a suitably chosen **C** that approximates **A** in some sense.

The NAG library, which is used within the simulation framework of this thesis, employs a special formulation of the PCG algorithm[3]. The derivation of this alternative formulation can be found in [91, p. 532 - 534]. A summary is also given in [92, p. 13]. The modified algorithm is shown in pseudocode in Listing 8.1 adapted to the notation of this thesis. This algorithm also demonstrates that **C** requires the property that equation systems with **C** need to be solvable at low cost. Furthermore, Listing 8.1 shows that the preconditioning matrix needs to be saved in addition to several vectors. In our case, this results in the additional constraint for C to contain as few non-zero elements as possible to minimize the storage for the preconditioning matrix.

```
 1 $\mathbf{r}_0 = \mathbf{b} - \mathbf{A} * \mathbf{x}_0$
 2 $i = 0$
 3
 4 do {
 5   solve $\mathbf{C} * \mathbf{z}_i = \mathbf{r}_i \Rightarrow \mathbf{z}_i = \mathbf{C}^{-1} * \mathbf{r}_i$
 6   $i = i + 1$
 7   if ($i == 1$)
 8     $\mathbf{p}_i = \mathbf{z}_{i-1}$
 9   else {
10     $b_i = \frac{\mathbf{r}_{i-1}^T * \mathbf{z}_{i-1}}{\mathbf{r}_{i-2}^T * \mathbf{z}_{i-2}}$
11     $\mathbf{p}_i = \mathbf{z}_{i-1} + b_i * \mathbf{p}_{i-1}$
12   }
13   $\mathbf{q}_i = \mathbf{A} * \mathbf{p}_i$
14   $a_i = \frac{\mathbf{r}_{i-1}^T * \mathbf{z}_{i-1}}{\mathbf{p}_i^T * \mathbf{q}_i}$
15   $\mathbf{x}_i = \mathbf{x}_{i-1} + a_i * \mathbf{p}_i$
16   $\mathbf{r}_i = \mathbf{r}_{i-1} - a_i * \mathbf{q}_i$
17 }
18 while (termination criterion not fulfilled)
```

Listing 8.1: Version of PCG as employed by NAG library.

Again, only simple vector or scalar operations are performed in nearly all code lines with only three exceptions. The first one is the occurrence of a sparse matrix-vector multiplication in lines 1 and 13, the second one is the solution of the equation system in line 5 and the third one is the test of the convergence criterion in line 18. The remaining mathematical operations in this

[3] `http://www.nag.co.uk/numeric/CL/nagdoc_cl24/html/F11/f11gdc.html`, accessed 2016-12-019

skeleton can be automated by a library, as realized by NAG, because of the simplicity of the algorithm. Therefore, it is possible to treat it as a black box. The user needs to supply a custom method for sparse matrix-vector multiplication and one routine that is able to solve the equation in line 5 for the given type of preconditioning matrix.

8.1 Preconditioning of the equation system

The framework presented in this thesis offers three types of preconditioners. The first type is the straightforward Jacobi preconditioner [90]. Here, the preconditioning matrix consists of the diagonal of the stiffness matrix. The main advantage of this solver is, of course, its small size since only the diagonal needs to be stored, without further structural information. In addition, the solution of the preconditioning system can be calculated very fast since only N divisions need to be executed for a diagonal of length N. The calculation of the preconditioning matrix can be done in a memory-efficient and parallelized way since the same calculations of potential are required as for the stiffness matrix itself. Consequently, the Jacobi preconditioner fulfills the requirements of efficient solving and resource-conserving instantiation as mentioned by Benzi [93].

The second type of preconditioner is a block-diagonal preconditioning, also being a common technique ([94], [95]). In our case, blocks $\mathbf{B}_i$ of size 3×3 are used. They directly correspond to the stiffness contributions $\mathbf{k}^e_{1,1} = \frac{\delta^2 U^{\text{int,e}}}{\delta \mathbf{x}_1 \delta \mathbf{x}_1}$, i.e., the contribution related to element e on the diagonal. This partitions the original matrix with N rows into $N/3$ independent equation systems with three equations and three unknowns each. The 3×3 symmetric, positive-definite block matrices $\mathbf{B}_i$ are decomposed with a Cholesky decomposition $\mathbf{B}_i = \mathbf{L}_i * \mathbf{L}_i{}^T$ by the dpofa routine of the Linpack [96] library to prepare the repetitive solutions of the precondition system with different vectors. Those can be realized in parallel because of the independence between the block matrices. The resulting lower triangular matrices $\mathbf{L}_i$ are stored, which require $2 * N$ storage places in total. Although Linpack also offers the dposl routine to solve equation systems $\mathbf{A} * \mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ were already decomposed by dpofa, this method is not employed due to performance reasons. Instead, loops within the code of dposl was unrolled since the problem size is known in advance. Additionally, the code of the called routines ddot (dot product) and daxpy (vector plus vector) was inlined and unrolled, too. Afterward, the resulting code fragment was vectorized using AVX vector intrinsics [97] (see Section 8.3.2 for more details about this topic). This results in a very fast solving of the small, independent equation systems that can be efficiently parallelized at near ideal scaling. Altogether, solving the precondition system of the block-diagonal preconditioner is not measurably slower than that of the Jacobi preconditioner.

The last preconditioner that is available is the incomplete Cholesky factorization of the stiffness matrix, which is a lower triangular matrix that approximates the Cholesky factorization and contains less non-zero values than full Cholesky factorization. The number of the non-zero elements depends on the usage of the so-called fill-in. If no fill-in is allowed at all, the resulting factorization contains at maximum as many non-zero elements as the lower triangular of the original matrix does. Increasing the fill-in rate improves the quality of the approximation, but also increases the memory required to store it. Being employed as a preconditioner for PCG, the incomplete Cholesky factorization can drastically improve the convergence rate (see e.g. [98]). Preconditioning with incomplete Cholesky factorization is a field of ongoing research due to its properties and several modified versions for fast or memory-efficient creation exist (see e.g. [99], [100]). The factorization is realized by the NAG library routine nag_sparse_sym_chol_fac within the framework presented. Usage of fill-in is minimized, i.e., setting the integer parameter lfill to 0 or 1. Nevertheless, this type of preconditioner is only applicable for smaller problem sizes because of its memory requirements. Hence, it is mainly used to estimate the quality of the other preconditioners.

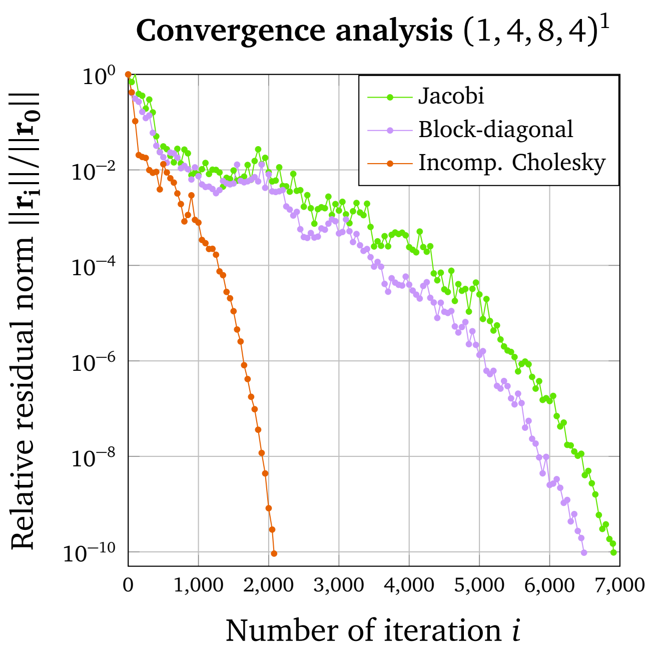
There is a multitude of other different preconditioning methods for CG (for an overview of different techniques refer to [93], [101]). In most cases they are targeted to special mathematical problems like Toeplitz matrices [102] or matrices arising from very specialized application fields like reconstruction of images resulting from positron emission tomography [103] or flow simulations of ground water flow [104]. General preconditioning schemes normally result in matrices with about the same or even more non-zero entries compared to the original matrix. They are not applicable to the SCNT case.

This section is concluded by a short analysis of the convergence behavior of the different preconditioners. To that end, a tension load case with the tubes $(1,4,8,4)^1$ and $(1,4,8,14)^1$ was simulated until NAG reported convergence NAG is generally configured to apply the following termination criterion for the simulation runs of this thesis:

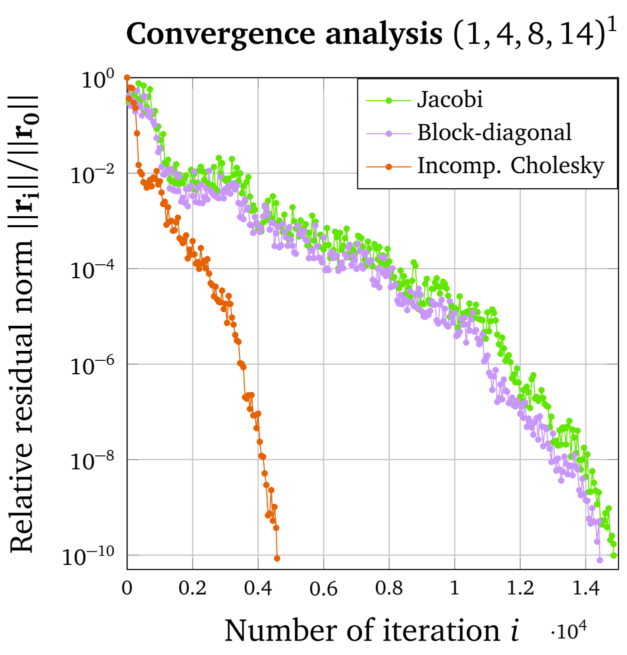
$$\left\|\mathbf{r}_\mathrm{i}\right\|_\infty \leq \tau * \left(\|\mathbf{b}\|_\infty + \|\mathbf{A}\|_\infty \times \left\|\mathbf{x}_i\right\|_\infty\right)$$

with τ being a user-defined tolerance value < 1. In our experiments, we used $\tau = 10^{-10}$. After every fifty steps, the relative residual norm $\left\|\mathbf{r}_\mathrm{rel}\right\|_\infty$ was calculated by the fraction $\frac{\|\mathbf{r}_\mathrm{i}\|_\infty}{\|\mathbf{r}_0\|_\infty}$ of the initial residual norm $\left\|\mathbf{r}_0\right\|_\infty$ and the residual norm at the i'th step $\left\|\mathbf{r}_\mathrm{i}\right\|_\infty$. Development of $\left\|\mathbf{r}_\mathrm{rel}\right\|_\infty$ is plotted against the number of iterations in Figures 8.1a and 8.1b.

The development is similar for both tubes. The incomplete Cholesky factorization requires about one third of the iterations of the others. The number of iterations for the block-diagonal preconditioner is about 6% lower in the case of $(1,4,8,4)^1$. Or in absolute values: 6910 iterations for the Jacobi and 6489 iterations for the block-diagonal preconditioning. The difference is even smaller at 3% (14,851 iterations versus 14,431) for $(1,4,8,14)^1$. While at the beginning the difference in $\left\|\mathbf{r}_\mathrm{rel}\right\|_\infty$ between both preconditioners is higher, it attains about the same values from iteration 13,800 on. So the block-diagonal preconditioner is of considerable advan-



(a) The convergence behavior of the three different preconditioners for a tension calculation on the $(1,4,8,4)^1$ tube. $\left\|\mathbf{r}_0\right\|_\infty = 4.00$.



(b) The convergence behavior of the three different preconditioners for a tension calculation on the $(1,4,8,14)^1$ tube. $\left\|\mathbf{r}_0\right\|_\infty = 4.00$.

Figure 8.1.: Convergence behavior of the three implemented solvers.

tage only if a less accurate solution is acceptable. In general, the simulations finish some percent faster with the block-diagonal preconditioner. It requires some iterations less for convergence and is as fast as the diagonal preconditioner because of its efficient implementation and near ideal scaling. In this way, it moves some computational load from the SpMV to the solution of the solution of the preconditioning system. Consequently, the block-diagonal preconditioner is chosen for the measurements of this thesis.

8.2 Parallelized reference solver

A solver was implemented based on the widespread compressed row storage (CRS) storage format [92, p. 57] to generate a baseline for performance . There are several specialized variants of CRS for vector computers [105], GPUs [106], or FPGAs [107]. In general, the CRS format works as follows: All non-zero values of Matrix **A** are processed row-wise and written in the value vector a_v. Additionally, the index of the corresponding column to each non-zero entry is appended in a vector called column vector icol_v. Finally, a third vector, the row pointer irow_v, contains a pointer to those entries in icol_v where the first non-zero entry of a new row is placed. The CRS implementation in the framework is not targeted to a special hardware. It only exploits the symmetry of the matrix by only storing those entries a_{ij} with $i \geq j$ and has no further assumptions about the structure of the matrix. The straightforward implementation of

```
 1 void matrixVectorMultiply(double* x, double* b) {
 2   for(int i=0; i<this->nodecnt*3; i++)    b[i] = 0.0;
 3
 4 #pragma omp parallel for
 5   for(int row=0; row < irow_v.size()-1; row++) {
 6     double tval = 0.0;
 7     for(int j = irow_v[row]; j < irow_v[row+1]; j++) {
 8       int col = icol_v[j];
 9
10       tval += a_v[j] * x[col];
11
12       if (row != col) {
13         double res = a_v[j] * x[row];
14 #pragma omp atomic
15         b[col] += res;
16       }
17     }
18 #pragma omp atomic
19     b[row] += tval;
20   }
21
22   return 0;
23 }
```

Listing 8.2: OpenMP parallelized implementation of the CRS-based reference solver.

an OpenMP-parallelized[4] sparse matrix-vector multiplication $\mathbf{b} = \mathbf{A} * \mathbf{x}$ using CRS can be found in Listing 8.2. This solver is called *the reference* solver in the following. It is obvious that this solver always requires a fully assembled stiffness matrix.

Exploitation of symmetry occurs in line 12-16 where the value of **A** is reused. The symmetric version of CRS requires two synchronization points between the OpenMP threads (lines 14 and 18) to avoid race conditions. The summations for each contribution of the symmetric parts are synchronized while the values resulting from the actual row can be summed up in a temporary variable t_val to reduce the number of atomics. In a Bachelor's thesis, together with Tristan Wirth [108], two other general data formats, namely blocked CRS [92, p. 58] and SELL-C-σ [109] were investigated for their suitability to perform the sparse matrix vector multiplication (SpMV) with the stiffness matrices arising in this framework on modern CPUs. The results show that SELL-C-σ is not able to deliver a performance gain compared to CRS. This results from the regular structure of the stiffness matrices. Consequently, sorting and permutation operations for SELL-C-σ only cause overhead that neutralize the performance gain through a faster multiplication. In contrast, the blocked CRS speeds up the calculation by about 20 % for block-size 3×3 which directly fits the underlying matrix structure. However, the implementation does not exploit the symmetry of the matrix and was compared against to a CRS version also

[4] http://www.openmp.org/

ignoring symmetry. Hence, the CRS reference solver in Listing 8.2 is used as a baseline for this thesis.

One major drawback of the reference solver, as reported in [23], was its high time for matrix assembly. This issue is solved in the recent implementation by a parallelized assembly with pre-allocated data structures. The knowledge about the SCNTs allows to assume a non-zero count of 90 times the number of nodes as an upper bound, which overestimates the real number by at most 10% (see next section). During the assembly, t threads calculate t consecutive stiffness contribution lines in parallel in each iteration. Then, each thread sorts its row in increasing order of the global index. Afterward one thread after the other writes its newly calculated three matrix lines into the CRS arrays. This speeds up the assembly by more than a factor of 11 when running with 16 threads, compared to the version in [23].

8.3 On-the-fly calculation of intermediate results

The black box nature of the reformulated PCG algorithm makes it possible to completely avoid the storage of the stiffness matrix $\mathbf{K}$ in order to reduce the memory required for solving $\mathbf{K} * \mathbf{x} = \mathbf{r}$. To that end, the required stiffness contributions are recomputed on-the-fly whenever requested and they directly multiplied with the corresponding entries of the vector which the SpMV is calculated with. This procedure was published in [23] and this chapter summarizes the methods of this publication with updated performance results (see also Chapter 9), which reflect the changes the code has undergone since the original publication date. Furthermore, a more exact modeling of the storage required is presented (Section 8.3.1). The solver applying these techniques is called the *on-the-fly solver* in the following.

8.3.1 Memory savings

The on the on-the-fly solver is compared to the reference solver with an assembled matrix in CRS format to demonstrate its reduced memory requirements. Here, the memory for the user-side is modeled, i.e., the data structures that need to be allocated and filled manually, as well as to be passed to the PCG black box as inputs or outputs. The internal variables of the PCG method itself like $\mathbf{z}_i$ or $\mathbf{p}_i$ (see Listing 8.1) are neglected since they are managed by the library and they are the same independent of the solver employed.

The number of degrees of freedom, i.e., the number of rows and columns in $\mathbf{K}$ is defined as $N = 3 * n$ with n being the number of nodes in the SCNT model. It is possible to estimate the number of non-zero values by $60*N$. This results from the up to 20 different nodes in the neighborhood of a reference node for tubes with an order > 0 which determine the regular structure of $\mathbf{K}^{N \times N}$. The actual number is slightly smaller, since three factors reduce the theoretical number. First, nodes at the boundary have incomplete neighborhoods, second, not all nodes are adjacent to octagons within the Y-junctions and last, boundary conditions may set some entries in **K** to 0.

Only the lower (or upper) triangular matrix, including the diagonal, needs to be stored in CRS since **K** is symmetric which we realized in Listing 8.2. Equation 8.3 estimates the number of non-zero values that need to be stored for **K**.

$$size(K) = \underbrace{(60 * N)/2}_{\substack{half\\values}} + \underbrace{N}_{\substack{diagonal\\values}} = 31 * N \tag{8.3}$$

For the number of estimated non-zeroes this means that it can be halved. Comparing the actual number of non-zeros for some configurations with this estimation as shown in Table 8.2 reveals that $60 * N/2 + N$ overestimates the real value by roughly 5% for tubes of order 1 on, where the case occurs that neighborhoods can consist of 20 distinct nodes. This overestimation is, in our view, acceptable.

Table 8.2.: For several tube configurations, the real number of resulting non-zeros in the stiffness matrix as well as the estimated number by the solver and the factor of overestimation.

	Real Number	Estimate	Overestimation
$(256, 256)$	10733867	11796480	1.099
$(512, 512)$	43518536	47185920	1.084
$(1, 4, 8, 14)^1$	3442522	3311616	1.031
$(1, 4, 8, 355)^1$	87459644	89971200	1.029
$(2, 6, 12, 116)^1$	87409543	90201600	1.032
$(1, 2, 8, 8)^2$	72056649	73728000	1.023
$(1, 2, 8, 8)^2$	346253048	356843520	1.031

In addition to these values, the SpMV requires the storage of the actual solution approximation $\mathbf{x}^N$ and the right-hand side $\mathbf{b}^N$. Furthermore, the bookkeeping data for CRS, namely the column and the row pointer, have to be taken into account. There is one entry in the column pointer for each non-zero value and for each row of **K** one value needs to be inserted into the row pointer. Hence, Equation 8.4 gives the total number of values to store using CRS.

$$size(K) = \underbrace{(31 + 1) * N}_{\substack{pure\\data}} + \underbrace{(31 + 2) * N}_{\substack{CRS-\\bookkeeping}} + \underbrace{2 * N}_{\mathbf{x}^N, \mathbf{b}^N} = 67 * N \tag{8.4}$$

The on-the-fly approach also stores $\mathbf{x}^N$ and $\mathbf{b}^N$. Additionally, the preconditioning matrix **C** is required. Assuming the Jacobi preconditioning matrix as preconditioner, it also needs to store

N values while $2*N$ values are required for the block-diagonal preconditioner. Hence, the total memory consumption can be summarized by Equation 8.5 which demonstrates that the one-the-fly approach saves memory by a factor of up to 22. This method increases the range of feasible problem sizes by more than one order of magnitude since memory is the limiting resource in modern HPC systems [110].

$$\text{Total storage} = \begin{cases} 67 * \text{N} & \text{for reference version} \\ 3 * \text{N} & \text{for on-the-fly version, Jacobi preconditioner} \\ 4 * \text{N} & \text{for on-the-fly version, block-diagonal preconditioner} \end{cases} \tag{8.5}$$

8.3.2 Performance optimizations

The recomputation of all stiffness contributions for each iteration of the PCG algorithm is computationally expensive and, of course, slows down the solution of the equation system as demonstrated in [23]. Calculation was parallelized with OpenMP to reduce this overhead. The scaling analysis follows in Chapter 9.

Additionally, a hotspot analysis was performed which demonstrates that the calculation of the dihedral angle and especially its second derivative within the Dreiding potential requires about 80 % of the total runtime. Checking the report of the Intel compiler reveals that it is not able to vectorize the code at all, although the three distinct lines in a stiffness contribution are independent of each other. Augmenting the code with additional vector pragmas and reordering the loops allow the compiler to create auto-vectorized code, resulting in a performance gain for the dihedral calculations of 20 %.

However, examining the assembly code generated by the Intel compiler reveals that this vectorized code is far from being optimal. Hence, the calculation of the second derivative of the dihedral angle was rewritten in AVX (Advanced Vector Extensions)-intrinsics ([111], [97]), an assembly like, close-to-the-hardware level language to perform vectorized computations. AVX allows to directly program the calculations on the SIMD units of modern processors and manually place/move data in the respective registers. The AVX unit has 16 registers which are 256 bit wide and, thus, allow to calculate four double precision values in parallel. Vectorizing hotspots with AVX is a common procedure to optimize scientific codes, see for example Tanikawa [112] for N-body simulations, Fialko [113] for an FE solver or Agulleiro and Fernandez [114] for 3D reconstruction. 70 lines of C++-code were replaced by about 150 lines of AVX intrinsics to employ the AVX units efficiently. A comparison of the old code with the hand-vectorized code shows that the speedup is about 2.5 which is very near to the optimal value of 3. The theoretical maximum is limited to this value since there are only three lines of the result that can be calculated in parallel. The performance of the on-the-fly solver is doubled when combined with a restructuring of the code for the valance angle calculation to enable auto-vectorization as shown in [23].

Another optimization is a stiffness contribution line-based processing of the contributions. Instead of immediately multiplying a stiffness contribution with the input vector, the whole contribution line is assembled first and multiplied afterward. This removes the interleaving of two different tasks, namely to calculate the potential values and to perform the SpMV with the goal to optimize the caching behavior and to facilitate the job of the compiler to generate fast code for both tasks.

8.4 Software-controlled caching of intermediate results

Although the on-the-fly solution exhibits high savings memory, compared to the CRS reference solver, the increased runtime is an issue as shown in [23]. It makes sense to try to reduce the number of recalculations of the Dreiding potential, i.e., stiffness contributions, since most of the runtime, usually around 80 %, is consumed by them.

This can be realized by caching intermediate results and reusing them in other iterations of the PCG method. We decided to use stiffness contribution lines since each of them allows to compute three entries of the result vector during matrix-vector multiplication. Caching single stiffness contributions is also possible, but tests show that the overhead for managing those small pieces of data is very high. As we demonstrate, the percentage of cached stiffness contribution lines is the crucial factor for performance. The number of cached stiffness contribution lines divided by the total number of existing lines is defined as the *caching rate* and the highest rate 1.0 is also called *full caching*. Consequently, the framework tries to maximize the caching rate by saving memory in other parts of the program, even if this may have a negative effect on the performance of the respective parts, since this performance degradation is more than compensated by the faster sparse matrix-vector multiplication. Corresponding stiffness contributions are calculated in the same way as in the on-the-fly solver for all remaining lines of the result vector. This concept of software controlled caching is realized in the *caching solver* that is discussed in the following section which is mainly based on [27].

As already demonstrated, the storage requirements of the reference solver for the raw matrix storage can be estimated by $65 * N$. In contrast, besides the actual stiffness contribution data, no further data structures for bookkeeping are required in the caching solver, if full caching is employed. The stiffness contribution data is accessed by traversing the underlying graph, which needs to be stored anyway. Hence, in the case of full caching only non-zero values of $\mathbf{K}$ need to be stored, which is $31 * N$. So, the caching solver requires only half the memory compared to the reference solver. This means that all problems that can be solved by the reference solver can also be calculated by the caching solver at full caching. The amount of RAM that is used for caching can be either defined by the user at the program start-up or by the framework at runtime. Note, that we can solve the linear equation systems with a storage of $3 * N$ or $4 * N$ if no caching is performed, and any amount of storage up to $32 * N$ or $33 * N$ with our caching solver, taking into account one or two N-vectors for the preconditioner.

8.4.1 Combination with Compressed Symmetric Graphs

The main advantage of the Compressed Symmetric Graphs becomes apparent when they are combined with the caching solver and large tubes. Here, we realize a higher percentage of cached contribution lines compared to the usage of IndexGraphs because of the high compression rate of CSGs. This is illustrated by Figure 8.2. There are mainly three data structures which consume memory when running the caching solver. The first items, shown in red, are the variables, the fields and the private space for the PCG algorithm itself, like the residual or the result vector, the preconditioner, and space consumed by the NAG library, respectively. The blue block shows the second part that represents the graph data structure which needs to be available during the simulation. The third contribution is the cache reserved for stiffness contribution lines in green. Last, there is some unused RAM depicted in orange. If there is enough RAM to cache all existing stiffness contribution lines, this represents the situation of full caching. However, if the graph grows, the green bar also enlarges because of the increasing number of contribution lines. At some point, the memory requirements exceed the memory capacity and the caching solver has to recalculate a certain percentage of the contributions. This situation is shown on the right side of Figure 8.2 where the hatched part stands for recalculated contributions that would be cached if more memory is available. This shows the importance of data compression of the CSGs: CSGs reduce the memory for the graph structure, which is equal to shrinking the blue bar, and the green bar can enlarge while the hatched area shrinks. An increased percentage of cached contributions decreases the overall runtime.

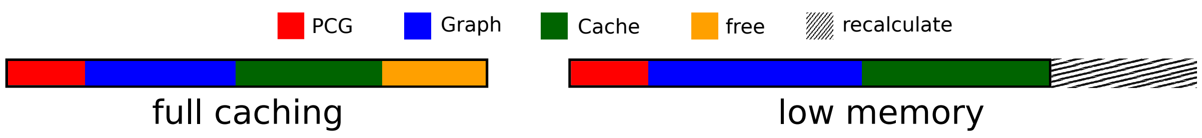


Figure 8.2.: The whole rectangle represents the memory of the system while the colored blocks show the parts of the framework that mainly consume it. The left side depicts the case of full caching and the left the case of low memory, in which stiffness contributions need to be recalculated.

Hence, it is important to estimate how compression of graphs in CSGs correlates to shrinking of the blue bar. This reduction is mainly based on three factors. The first one is the space which is saved by compressing a node, the second one is the storage requirement to cache a contribution line and the third is the number of nodes that can be compressed by CSGs. A node object within the framework contains the following information:

1. Its global and its serial index, both realized as 64 bit integers (in total 16 bytes).
2. Its position composed of three double precision values (in total 24 bytes).
3. Its neighborhood consisting of 20 64 bit values and 23 short integers which relate the distinct neighboring nodes to their position(s) in the neighborhood. So 206 bytes in total.
4. A short integer giving the number of distinct nodes in the neighborhood (2 bytes).
5. A boolean that indicates if the neighborhood is complete (1 bytes).
6. A tuple which is composed of 32 bit integers per entry. In that case, the size in memory depends on the order of the tube and thus, the length of the tuple. As an example, the tuple for a tube of order 2 requires 80 bytes.

In this case, a node requires $16+24+206+2+1+80 = 329$ bytes in total. This value is slightly higher due to memory alignment. The value determined with the **sizeof**-function is 336 bytes.

If such a node is compressed, no storage is required for the neighborhood, additional information about the neighborhood and the tuple. However, both of its indices and position data need to be explicitly stored anyway. Consequently, compressing a node frees $206+2+1+80 = 289$ bytes of memory. The value retrieved with **sizeof**-function is 296 bytes in that case.

The size of the contribution line can be estimated by the number of its individual contributions and their size. Each contribution is a block of 9 doubles and needs $9 * 8 = 72$ bytes, i.e., the storage for the stiffness data corresponding to a node is higher than the storage required for its structural information. The maximum number of contributions per line is 20. Due to the symmetry of the matrix, in average 10 contributions have to be stored per line, resulting in $72 * 10 = 720$ bytes per contribution line. Thus, to cache one additional contribution line $720/296 = 2.5$ nodes of a graph need to be compressed. Now, we assume that for a certain tube and a specific system we have the case of low memory, the caching solver is combined with an IndexGraph, and is only able to cache 90 % of the contribution lines. Then, the CSG corresponding to this IndexGraph needs to compress $2.5 * 10\,\% = 25\,\%$ of the nodes of the original graph to enable full caching. As shown in Table 7.2, this prerequisite is already fulfilled for the worst case tube scenario. Consequently, the higher the compression rate ρ, the higher the *additional* percentage of cached contribution lines μ. In general, it is $\mu = \frac{\rho}{2.5}$ with an upper bound of $\lim\limits_{\rho \to 100\,\%} \frac{\rho}{2.5} = 40\,\%$.

8.4.2 Parallel preparing the cached data

The most important step for the caching solver is the instantiation and storage of the data. This preparation step is separated to a CachingManger class. Its first task is the determination of how many stiffness contribution lines can be cached and which lines need to be recomputed in each iteration. Therefore, the length of each row is required, which varies between 1 and 60 since the symmetry of **K** is exploited and only the lower triangular part of the matrix is taken into account. To that end, the CachingManger iterates over the neighborhood of each node, counts those neighborhood nodes that have a lower or equal index than the reference node and accumulates the result in c_{nnodes}. Each node with this property contributes a 3×3 matrix to **K**, resulting in $c_{\text{nnodes}} * 9 * 8$ bytes of data for the respective line. This value is subtracted from the user-defined memory available for caching in the variable max_stiffness_storage. So, the nodes in G are processed consecutively by their global index until the subtraction of the next node from max_stiffness_storage would return a negative result. If this situation occurs at global index k this means that the first $k * 3$ lines of the stiffness matrix will be cached. Then, the required space is allocated.

The next step are the calculation and insertion of corresponding stiffness contributions. During these calculations, the work for the later following SpMV is also statically distributed to the user-defined number of threads. A consecutive portion of nodes is assigned to each thread while trying to keep the number of stiffness contributions equally distributed. Hence, each thread only needs to know the start and end node of its region and the corresponding memory address of the cached contribution line to perform the SpMV. Afterward, each thread calculates its stiffness contribution lines in parallel.

Initially, the stiffness contributions are stored in row-major order at different locations in memory after their calculation. Then, the CachingManger writes the contribution values column-wise in three different regions of contiguous memory contrib0, contrib1 and contrib2. This procedure is shown in Figure 8.3 for an exemplary stiffness contribution line consisting of three contributions. Colors indicate the relation of entries to their stiffness contribution and numbers give the correspondence of the entry in the 3×3 contribution matrix and its position in one of the regions of contiguous memory.

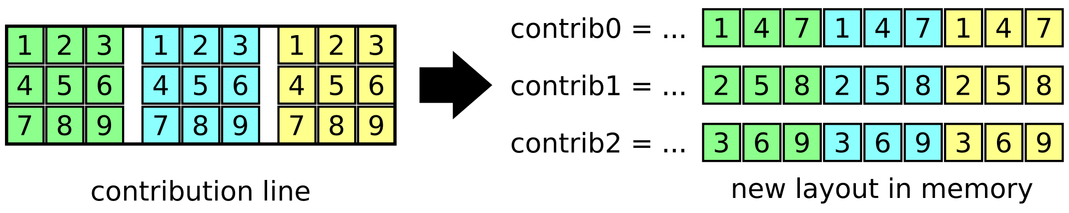

1 2 3 1 2 3 1 2 3
4 5 6 4 5 6 4 5 6
7 8 9 7 8 9 7 8 9
contribution line
contrib0 = ... 1 4 7 1 4 7 1 4 7
contrib1 = ... 2 5 8 2 5 8 2 5 8
contrib2 = ... 3 6 9 3 6 9 3 6 9
new layout in memory

Figure 8.3.: Memory layout for stiffness contributions within the caching solver. The entries in the original stiffness contributions are arranged column-wise in three different consecutive arrays contrib0, contrib1 and contrib2.

8.4.3 Cached sparse matrix-vector multiplication

The cached SpMV $\mathbf{b} = \mathbf{K} * \mathbf{x}$ exploits some knowledge and properties of the underlying stiffness matrices to increase the efficiency of calculation. Each thread iterates over its consecutive fraction of the node list and each iteration covers the stiffness contribution line corresponding to the respective node while taking the symmetry of $\mathbf{K}$ into account. Figure 8.4 demonstrates memory accesses, which are required to calculate the contribution of one line to the SpMV result.

Matrix $\mathbf{K}$ is shown on the left side with the diagonal being represented by the dotted line. Numbers within the different entries indicate the symmetry relations of the entries in the matrix. Only the lower left diagonal of the matrix is cached while the upper triangle results from symmetry. It is assumed that all values resulting from the highlighted row should be calculated. In the row, the values are grouped around bands near the diagonal as shown in Section 5.3, i.e., in most cases, the higher the global index corresponding to the row, the more zeros lie left to the first non-zero element. The grade of distribution depends on the tube configuration. Although the element in pale yellow in $\mathbf{K}$ belongs to the highlighted row it is not processed during this iteration, but is covered later when the row with the pale cyan entry is processed due to the symmetry between both pale elements. We see that several positions in the result vector $\mathbf{b}$ are updated when processing a single row in the matrix because of the exploitation of symmetry, i.e., not only the current row S, but also the rows which correspond to the column index of the entries 1, 2 and 3. However, the region with modified rows in $\mathbf{b}$ is limited by the leftmost entry in the current row and the entry on the diagonal.

In principle, the calculation for the highlighted row requires two different kinds of operations. The first one is to multiply and accumulate the yellow contributions in the corresponding entry of $\mathbf{b}$. The second is to update $\mathbf{b}$ at all cyan positions.

To realize an efficient multiplication, several temporary vectors are allocated for each thread. All of these vectors have $20*3$ entries and thus directly result from the non-zero structure of `K`[5]. Their contribution to the overall memory consumption of the framework can be neglected due to their small size. Two vectors `bn` and `bn_T` are created for the intermediate results. Additionally, the solver allocates four vectors `xn_0`, `xn_1`, `xn_2` and `xn_T` that temporally store small parts of the input vector `x`. All vectors are aligned to enable a vectorized computation. Prior to the start of the actual calculation for a reference node, data is copied from the outside into the four different x-vectors. Assume an input vector `x = {x1_x, x1_y, x1_z, ..., x4_x, x4_y, x4_z, ..., x11_x, x11_y, x11_z, ...}` and that we want to execute the calculation corresponding to the reference node with global index 11 that has the nodes 1 and 4 in its neighborhood. Hence, only the entries in `x` corresponding to the nodes 1, 4 and 11 are of interest. In that case, the temporary vectors look like:

[5] `K`, `b` and `r` are highlighted as code, since we are dealing with the implementation in this chapter and the corresponding variables in the implementation have the same name as in the mathematical notation

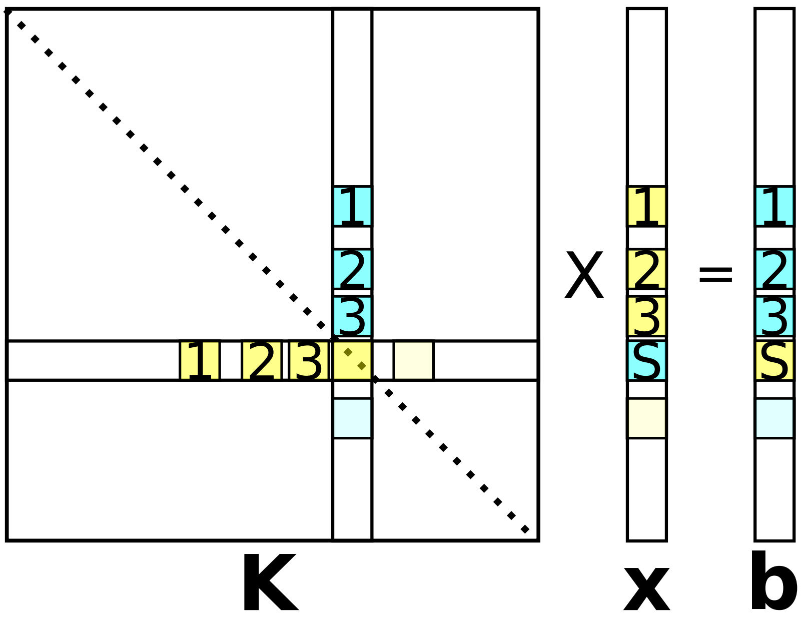


Figure 8.4.: Read and write access in the required data structures during SpMV in caching solver.

- xn_0 = {x1_x, x1_x, x1_x, x4_x, x4_x, x4_x, x11_x, x11_x, x11_x}
- xn_1 = {x1_y, x1_y, x1_y, x4_y, x4_y, x4_y, x11_y, x11_y, x11_y}
- xn_2 = {x1_z, x1_z, x1_z, x4_z, x4_z, x4_z, x11_z, x11_z, x11_z}
- xn_T = {x11_x, x11_y, x11_z, x11_x, x11_y, x11_z, x11_x, x11_y, x11_z}

So, xn_0, xn_1, xn_2 and also xn_T contain some redundant data. Data within these vectors is updated for each reference node.

Now, we consider both temporary b-vectors. On the one hand, bn keeps all the entries for the result vector that are directly calculated from values of the lower triangular matrix per node. Although only three entries of b are influenced by one contribution line, each entry of bn holds the results of the multiplication of a single matrix entry of K with the respective entry of x. Afterward, the single entries are reduced to the entries in b. This procedure allows to realize the vectorized multiplication of four double entries and to perform a packed write of the results to the appropriately aligned memory location bn. On the other hand, bn_T contains the results of the multiplication of the corresponding symmetric parts, so in terms of Figure 8.4 the cyan blocks. The respective multiplications and summations are performed with the same values of K as they are required for bn exploiting the temporal locality of the matrix data.

The number of synchronization points between the different threads can be reduced by employing intermediate vectors for the result vectors. The synchronization overhead in OpenMP or in general parallel programming is substantial and gets worse as soon as the degree of parallelism increases (see e.g. [115], [116] for OpenMP). Hence, the caching solver integrates another method to reduce it. To that end, for each thread a *span* is defined within the result vector b. We know from the previous section that each thread is assigned a consecutive number of global indices. The span starts at the entry with the lowest global index in the result vector that is modified by a distinct thread and ends at the highest one, as depicted in Figure 8.5 in an example scenario with four threads. Colors indicate the region in the vector corresponding to the respective thread. The whole region consisting of a full colored block and a pale part is the span of thread. Please note that within this region, not all entries are accessed by the corresponding thread. The full colored part directly corresponds to rows of K that are assigned to a thread via the global index while the pale remainder are those entries that are modified due to the exploitation of the symmetric matrix. Also reconsider Figure 8.4 which demonstrates that a thread does not modify entries with a global index higher than the end of its span and that there is also a lower bound for the entries modified by one thread. These regions may overlap and within these areas threads need to synchronize when modifying entries. If a thread writes an entry within a non-overlapping area it does not influence other threads and can execute a direct access. The advantage in the case of SCNT simulations is that these spans per thread can easily be determined when preparing the data for the caching solver. The upper bound for a span is always the highest global index, i.e., the last row of K which is assigned to a thread. The lower bound is the lowest index that occurs in all neighborhoods of the nodes that are assigned to the thread. It is sufficient to check only the first half of entries in the neighborhood to identify this node since the neighborhoods are sorted in an ascending order.

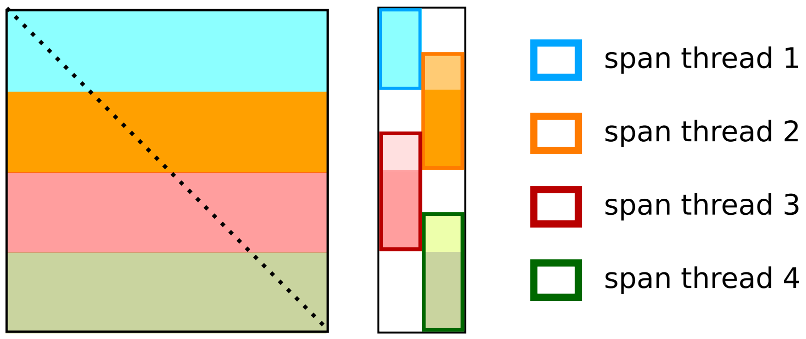


Figure 8.5.: Each thread is assigned a bunch of rows of the stiffness matrix, determining the entries that it needs to modify within the result vector. These regions may overlap.

A second and more heuristic procedure was implemented to demonstrate and estimate the effect of OpenMP synchronization. During the data preparation, the neighborhood with the maximum distance of global indices among its nodes is searched and stored. The distance between the start nodes of adjacent threads is determined when beginning the SpMV and the predetermined maximum distance between indices within all neighborhoods is added to the lower and subtracted from the higher threads' start indices, plus a buffer value. If the difference is high enough, the first thread will already have finished modifying the overlapping entries in the result vector, before the other thread arrives at them. It must also be assumed that the threads proceed through the result vector with about the same speed for this scheme. In that case, it is possible to always execute an unsynchronized write. However, this leads to data races for several runs probably since the chosen buffer was too small. Consequently, the first thread-safe approach is employed for the performance comparisons, but the alternative approach can serve as an indicator which shows to which extent the runtime of the caching solver is limited by the need to synchronize.

8.5 The case of small deformations

In Section 3.3 structural is defined. This section discusses the special case of the so-called *value-symmetry* which is only present in the small deformation case where the simulation is based on the linear theory of elasticity. Here, the static equilibrium of order 0 SCNTs can be used as an estimate for the solution. In those cases, potentials acting on structural symmetric nodes can be deduced from those acting on respective base-symmetry nodes. Put differently, there are the same or very similar contributions to the stiffness matrix for different nodes. We consider small deformations governed by a linearly elastic response to the load conditions. Thus, symmetric boundary conditions result in a symmetric response of the structure. This symmetry can be exploited by the methods presented in this section. Then, it is possible to store all information of the stiffness matrix within only a few megabytes of RAM. These ideas were already presented in [21] and they are extended within this section while new results are also presented. Principles described here are implemented within the *value-symmetric solver*. They are only applicable to order 0 tubes. One main difference to the value-symmetric solver version in [21] is that all required stiffness contributions are always stored since this section demonstrates that memory is not a limiting factor.

Information about value symmetry is encoded in the tuple system in the same way as structural symmetry. Hence, translational and rotational symmetry can be resolved in a similar way as it is realized for the Compressed Symmetric Graphs by determining offsets for jumps in translational and rotational direction. The rotational offset is always 4 since only order 0 tubes are employed. The algorithm is aware of the fact that all non-symmetric nodes lie in the 0-, 1-, $(l-1)$- and l-leading rings due to its knowledge of order 0 SCNTs. This approach is different to the one employed in [21], where parsing of tuples and time-consuming search operations for nodes were required to resolve symmetry. The advantage of faster symmetry identification comes with a small memory overhead since, in that case, the 2-leading ring also needs to be stored completely, because of irregular neighborhoods of some of the 1-leading nodes.

Translational and rotational symmetry are successively discussed in the following, starting with translational symmetry. Here, it is important to know that forces acting on nodes that possess structural translational symmetry are identical, which means that the resulting contribution to $\mathbf{K}$ can be reused directly. The most important question is to which extent the value-symmetry can help to reduce the number of calculations and the required amount of memory. It is possible to estimate the number of stiffness contributions which need to be stored to represent the whole stiffness matrix. The number of base-symmetry nodes for translation, non-symmetric nodes, and symmetric nodes is given by c_{base}, c_{nonsym} and c_{sym} while c_{total} denotes the overall number of nodes in the model that can be calculated by $l_0 * d_0 * 2$ for order 0 SCNTs. c_{base} can easily be determined by $d_0 * 4$ since it is always the number of nodes in the 2-leading ring which consists of d_0 MZSs. c_{nonsym} consists of two contributions. First, we have the 0-leading ring at the left end and second, the l-leading ring at the right end of the tube. Both consist of incomplete MZSs. Unifying both boundary rings results in one complete ring with again $d_0 * 4$ nodes. Additionally, we have to add the 1- and $(l-1)$-leading ring resulting in $c_{\text{nonsym}} = 3*4*d_0$. Obviously, all remaining nodes are symmetric. Consequently, we have to take the whole number of rings given by $\frac{l_0}{2} + 1$ into account and to subtract the five rings already covered to calculate c_{sym}, resulting in $c_{\text{sym}} = (\frac{l_0}{2} - 4) * 4 * d_0$. Now, we are able to estimate the number of stiffness contributions which results from the three different types of nodes:

1. c_{base}: All nodes have complete neighborhoods generating $(4 * d_0 * 19)$ contributions to **K**.
2. c_{nonsym}: The nodes in the incomplete rings have on average 14.2 neighbors[6], while those in the 1- and $(l-1)$-leading rings have complete neighborhoods with 19 different nodes. This results in $(4 * d_0 * 14.2) + (2 * 4 * d_0 * 19)$ stiffness contributions.
3. c_{sym}: The remaining $(\frac{l_0}{2} - 4) * 4 * d_0$ nodes also generate 19 contributions each.

The reduction factor can now be estimated by dividing the overall number of potentials in the tube by the number of potentials that need to be stored in the case of value symmetry. The overall number is equal to the sum of the number of potentials of all non-symmetric, base-symmetry and symmetric nodes which is the sum of all three items in the enumeration above. The number of potentials that needs to be stored in the case of value symmetry is the sum of the potentials of non-symmetric and base-symmetry nodes, hence, the first two items in the enumeration. After calculation and rounding, one arrives at $\frac{38}{284} * l_0 - 0.0068$, which we round to $0.135 * l_0$. To determine the quality of the estimation, Table 8.3 compares the calculated and measured reduction factors for several tubes.

Table 8.3.: Summary of the reduction in the number of stiffness contributions that is achievable by exploiting translational value-symmetry.

CNT configuration	**(64, 64)**	**(128, 128)**	**(256, 256)**	**(256, 512)**	**(512, 512)**	**(1024, 1024)**	**(4096, 4096)**
factor (estimated)	9	17	35	69	69	138	553
factor (observed)	8	17	35	70	70	140	563

We see that both factors are in very good agreement and that the reduction grows with the length of the tube, while the diameter has no influence on it. We note that the attained reduc-

[6] This value was empirically determined.

tion in Table 8.3 is smaller than in [21] as a result of the changed procedure in defining and determining symmetric nodes. Nevertheless, the reduction can be very substantial. This always leads to the situation that a computer system which is capable of constructing a specific tube can also execute the value-symmetric solver on this configuration. The few additional megabytes for the storage of contributions also fit in the system's memory, if a large graph model with several gigabytes fits into it.

All related stiffness contributions are calculated and inserted into two hash maps for all the non-symmetric and base-symmetry nodes, one for each type of nodes. Both maps are indexed by the corresponding global index of the nodes. The only calculation of stiffness contributions occurs during the initialization of the stiffness data, since all contributions are always stored. Those and some single values like the translational offset ($= 4 * d_0$), the global index of the first node in the 2-leading ring and the global index of last node of the $(l-2)$-ring plus the graph is all information that is required. The algorithm iterates over both maps to calculate the SpMV. Contributions are directly multiplied with the corresponding entries in the input vector and written to the result vector for the non-symmetric nodes. Calculations for the global index of the individual base-symmetry nodes are performed, followed by a loop that executes translational jumps as long as the index of the target node is smaller than the start of the $(l-1)$-leading ring. Corresponding calculations for the SpMV are performed for each ring. One optimization in contrast to the version published in [21] is that the data is managed and read line-wise but processed contribution-wise. This exploits the spatial locality of the stiffness contributions.

The next point to discuss is the rotational symmetry. In contrast to translational symmetry, the stiffness contributions need to be transformed when reusing it since rotating the nodes also influences the forces between them. The transformation depends on the angle which a node is rotated around the x-axis of the tube. Fortunately, there is a straightforward transformation of the stiffness contribution $\mathbf{k}^e_{i,j'}$ for three angles. If $\mathbf{k}^e_{i,j'}$ has the form:

$$\mathbf{k}^e_{i,j'} = \begin{pmatrix} xx & xy & xz \\ yx & yy & yz \\ zx & zy & zz \end{pmatrix}$$

then for the angles of 90°, 180° and 270° the following matrices correspond to the contributions for the rotational symmetric nodes:

$$\mathbf{k}^e_{i,j'}(180°) = \begin{pmatrix} xx & -xy & -xz \\ -yx & yy & yz \\ -zx & zy & zz \end{pmatrix}$$

and

$$\mathbf{k}^e_{i,j'}(90^\circ) = \begin{pmatrix} xx & -xz & xy \\ -zx & zz & -zy \\ yx & -yz & yy \end{pmatrix} \qquad \mathbf{k}^e_{i,j'}(270^\circ) = \begin{pmatrix} xx & xz & -xy \\ zx & zz & -zy \\ -yx & -yz & yy \end{pmatrix}$$

This means that it is sufficient to exchange values and to change the sign of several entries for these three angles which can be realized at very low cost. For all other angles it would be required to calculate a respective rotational matrix that needs to be applied to the stiffness contribution, resulting in a much higher computational overhead and most notably introducing inaccuracies. Those can imply a higher number of iterations for PCG or even a non-converging solving process. As a consequence, the exploitation of rotational symmetry is limited to 90°, 180° and 270°. This inherently limits the reduction of the stiffness contributions that need to be stored to a factor of 4. Furthermore, care needs to be taken when determining base-symmetry nodes. Since translational symmetry is also resolved in terms of offset calculations and not directly on the tuples, the zero-line in the tube causes problems if the base-symmetry nodes or symmetric nodes are adjacent to it. Hence, rotational symmetry can only be applied if the diameter of the tube is high enough to distribute the rotational base-symmetry node and its corresponding three symmetric nodes in such a way that none of them is adjacent to the zero line, i.e., $d_0 \geq 12$. Additionally, d_0 needs to be a multiple of four because, otherwise, symmetric nodes at exactly 90°, 180° and 270° of rotation do not exist.

One main improvement of the new value-symmetric solver compared to [21] is that it is capable of combining translational and rotational symmetry. When translational symmetric nodes for a base-symmetry node are processed during the SpMV, the corresponding three rotational symmetric nodes are covered successively for each of them.

However, the combination of translational and rotational symmetry leads to a noticeable deviation in the number of iterations to convergence for a few of the tests . Especially very small entries in the $\mathbf{k}^e_{i,j'}$s with values $< 10^{-10}$ differ between the value that results from directly calculating the stiffness contribution and those resulting from the application of symmetry. These errors seem to sum up for the larger tubes. Consequently, exploitation of rotational symmetry is deactivated for performance measurements to ensure comparable and correct simulations.

9 Results and Evaluation

This thesis presented the structure-tailored graph data structure IndexGraph being employed to assign a unique index to each tuple and to manage the graph data. It is the default data structure for graphs in the framework. Based on a junction of level $L-1$ that is stored as an IndexGraph, the Compressed Symmetric Graphs can create a memory-saving graph representation that is able to dynamically recompute large parts of the structural information instead of storing it. The CSGs can replace the IndexGraphs as underlying data structure during the solution of the equation system. The thesis also presented a novel matrix-free solver which can work in two modes: First, it recomputes the stiffness contributions in each PCG iteration *on-the-fly*, reducing memory requirements by one order of magnitude though implying a runtime overhead. Second, it caches stiffness contributions within the memory available to decrease runtime while claiming only half of the memory if all contributions are cached compared to the reference solver, which requires a fully assembled stiffness matrix. The principles of the caching solver and CSGs become especially important when they are combined, because the CSGs free memory which can be employed by the solver for additional caching of contributions. This leads, in general, to one of two situations: First, the caching solver is able to cache all stiffness contributions when being combined with IndexGraphs and CSGs (full caching). The solution process has a lower memory footprint when employing CSG, but it is slower. Second, the memory is insufficient for full caching and the caching rate for CSGs is higher than for IndexGraphs. In that case, the simulation runs with equal memory footprint for both graph structures, but, in general, the solver combined with CSGs delivers a better performance due to the higher caching rate (case of low memory).

These theoretic aspects are practically validated within this chapter. To that end, the chapter summarizes the test setup (Section 9.1) and it briefly shows the results of the mechanical simulations (Section 9.2). Afterward, as the main part, the performance results when using different solvers (Section 9.3) are presented: We first compare the performance of the three novel solvers, namely the on-the-fly solver (Section 9.3.1), the value-symmetric solver (Section 9.3.2), and the caching solver (Section 9.3.3) with the performance of the parallelized reference solver. IndexGraphs are employed in all these tests. Accordingly, we evaluate the influence of the CSGs on the performance of the caching solver in Section 9.3.4 when they replace the IndexGraphs as the underlying graph data structure. We consider different scenarios with full caching or low memory for order 1 tubes with available relaxed position data. Finally, in Section 9.3.5, we look at the performance of the caching solver with order 2 tubes and with IndexGraphs and CSGs in the cases of full caching and low memory to validate the properties of the solver for higher order tubes, too. For order 2 tubes, we focus on the SpMV due to the lack of relaxed position data.

9.1 Test setup

9.1.1 Test environment

All tests are performed on the *Lichtenberg-Hochleistungsrechner* (HHLR) at the Technische Universität Darmstadt[1]. Two types of nodes are used for the different tests. The first ones are called the *phase one nodes*. They consist of two Intel Sandy Bridge Xeon E5-2680 processors (with $2*8=16$ cores, Hyperthreading turned off) and 20 MB of last-level cache. Each node has 32 GB of main memory. The others are the *phase two nodes* that are equipped with two Intel Haswell Xeon E5-2680v3 processors ($2*12=24$ cores, no Hyperthreading), 30 MB of last-level cache and 64 GB RAM per node. The operating system of the HHLR is CentOS 7.

The code is compiled with the Intel C++ compiler (icpc) in version 17.0.1 and the OpenMP implementation from Intel is employed. The optimization level is set to *full optimization*. All nodes are allocated exclusively to avoid any interference from other calculations, independent of the required number of threads per run.

9.1.2 Different load cases

Three types of load cases are tested, also with multiple steps of the Newton-Raphson method, for tubes of order 0 and 1. Videos that visualize the results can be found online[2].

The first type of load is uniaxial *tension*. In that case, a uniform force in positive x-direction is applied to all nodes on the right boundary, while the nodes on the left boundary of the tube are fixed against movements in axial direction. This is also the main test case that is used for the performance measurements in this thesis, since it can easily be used without the dockSIM framework and thus allows a more direct investigation of the solver routines themselves without the overhead of dockSIM.

The second type of load cases is *torsion* of the SCNT. To model this load, forces are again applied to the boundary nodes. All nodes on the left boundary are forced to uniformly move counterclockwise on the circular path around the x-axis with the radius equal to the tube radius. The nodes on the right boundary are moved analogously but clockwise, which is shown in Figure 9.1.

The last test case employed is *bending* of the tube. Here, the boundary nodes are also moved on a circular path but this time around the z-axis. The radius of the corresponding circle is half

[1] `http://www.hhlr.tu-darmstadt.de/hhlr/index.de.jsp`
[2] `https://www.youtube.com/channel/UCxKMz5tvGWFjMIYdyB0-_9Q`

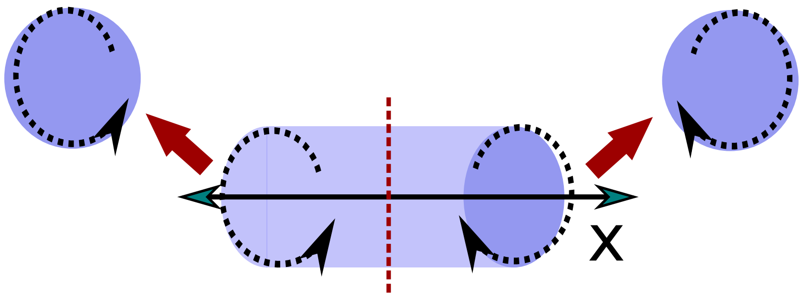


Figure 9.1.: Torsion on a tube. Both ends are wound in opposite direction. (Inspired by on `http://www.ah-engr.com/som/3_stress/text_3-2.htm`, access 2016-08-18)

the length of the tube. The orientation of the circular path on the one end of the tube is again clockwise, while the orientation on the other side is counterclockwise.

9.2 Mechanical simulation results

At the moment, we are in the position to only set up consistent equation systems for order 0 and several order 1 tubes. This is due to the absence of relaxed input data. The relaxation of larger tubes of higher order that would allow to create a consistent stiffness matrix is the subject of ongoing research. Consequently, the mechanical results of this thesis are limited to those which can already be achieved with dockSIM. The resulting displacements of the presented matrix-free solver are congruent to those of dockSIM. This validates at least the correctness of the solver and suggests that the calculations for tubes of order 2 in Section 9.3.5 are also consistent, although the reference solver cannot be employed in all cases to validate the results of the matrix-free approach because of the size of the matrix.

The simulation results of dockSIM and the framework of this thesis deliver the same behavior as those performed by other groups employing different methodologies. Figures 9.2a and 9.2b show the reaction of one junction element in a $(1,4,8,4)^1$ tube if axial tensional load is applied. The tube is stretched then, but the interesting question is where the extension derives from. As depicted, the main reason is the change in the angle between the junction arms, while the length of the arms nearly stays constant for the moment.

We also compared the displacement vectors attained by the dockSIM code and its implementation of the Newton-Raphson method to those calculated by the framework of this thesis in order to verify the correctness and numerical stability of the new implementation. We can compare the entries in the vectors pairwise since we apply the same node numbering resulting from the global indices. In a first test suite, we only perform one single iteration step of the Newton-Raphson method and solve the linear equation system once. Tubes of order 0 and 1 with available relaxed position data are employed including mainly $(128,128)$, $(256,256)$, $(512,512)$, $(1,4,8,4)^1$, $(1,4,8,14)^1$ and $(1,4,8,355)^1$. The comparison demonstrates that the

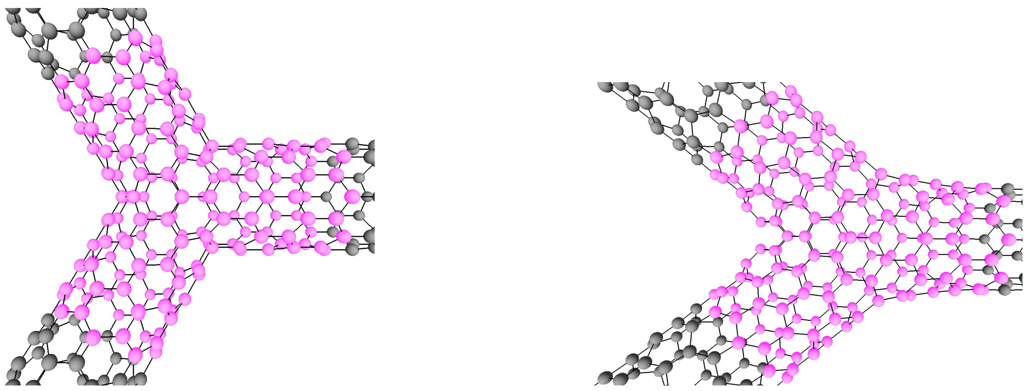

(a) Junction in a $(1,4,8,4)^1$ SCNT before applying load. **(b)** Junction in a $(1,4,8,4)^1$ SCNT after a tension in axial direction was applied.

Figure 9.2.: Effect of tension on SCNT junctions.

entries in both displacement vectors are in very good agreement. The single corresponding entries do not differ by more than $1.0 * 10^6$ % for all tested tube configurations.

Also multi-step simulations are performed and compared to dockSIM. To that end, the code related to the solution of the linear equation systems and the graph data structures of the new code are integrated into dockSIM. We run tension, torsion and bending tests, including eight load steps and a convergence criterion for the inner loop which specifies that the absolute norm of the solution vector needs to be smaller than $1.0 * 10^{-10}$. Convergence is achieved in all cases. Afterward, the solution vectors calculated by the original dockSIM implementation are compared to those achieved by the new code. The maximum difference of pairwise comparison of single entries is only $1.0 * 10^8$ % in the investigated scenarios.

9.3 Performance measurements and comparison

We take the average time for one PCG iteration as unit for comparison to determine and compare the performance of the different solvers. Although this results in only milliseconds for small tubes, at least 5000 iterations are calculated for every run to ensure consistent and reliable measurement values. Each test configuration, consisting of a triple or quadruple (solver, tube, number of threads, [cachesize]) is repeated a suitable number of times. Appendix F summarizes all measurements which are employed for the diagrams in the following sections, including the mean values and the standard deviation. The relative comparison of two test configurations which only differ in the solver employed is based on the cross product of runs. That means if three values exist for the test configuration A and five values are there for test configuration B, then 15 measurement points are created, comparing each result of A with each of B.

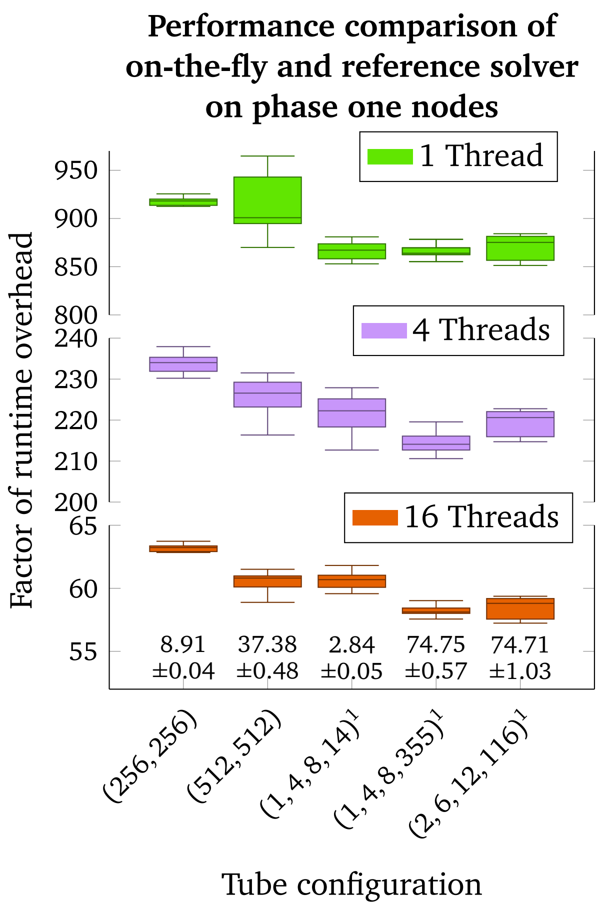
Except Section 9.3.5, all sections are based on those tubes for which relaxed geometric data is available. Not all simulations are run until convergence, but they are terminated after a threshold of up to eight hours, because of the high runtime of some test configurations. This is feasible since measurements until convergence demonstrate that the runtime grows linearly with the number of executed PCG iterations for all configurations investigated.

9.3.1 On-the-fly solver versus reference solver

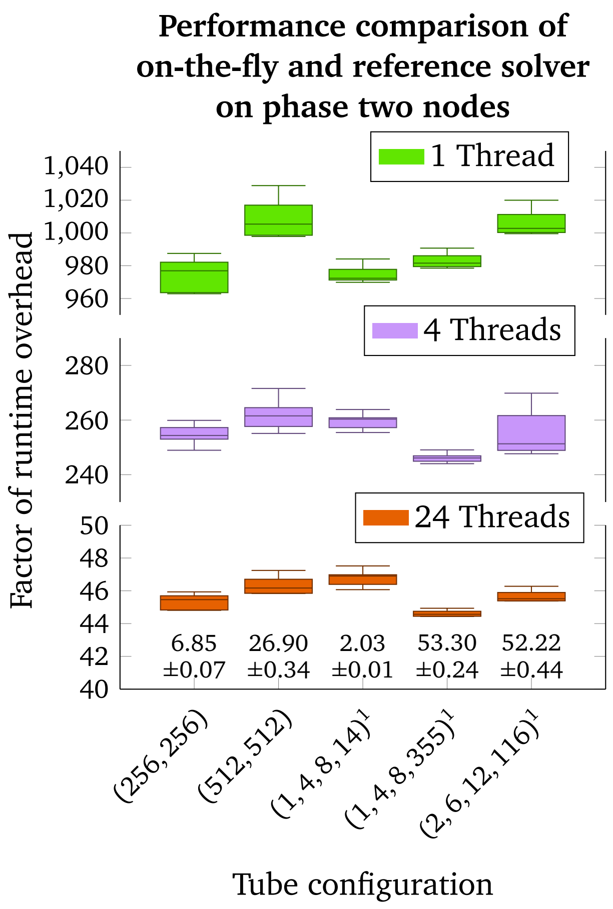
Here, we compare the performance of the on-the-fly solver to the parallelized reference solver beginning on phase one nodes. Figure 9.3a shows the relative performance of both solvers. A complete overview of the overall measurement data is given in Appendix F.1 (Table F.1). In contrast to the tests in [23], also some SCNTs of order 1 are employed, in addition to tubes of order 0. Moreover, a load case that applies uniaxial tension to the right end of the tube is used, while there was no external load in [23].

In [23] a slowdown of a factor lower than 10 is reported when employing the on-the-fly solver running on a compute node with nearly identical configuration as the phase one nodes employed for this thesis. However, Figure 9.3a shows considerably higher values of more than 50 for the fastest on-the-fly configuration running with 16 threads. We have a detailed look at this divergence and explain its origin.

In principle, there are three factors that influence the slowdown. The first one is that there are indeed some parts in the recent on-the-fly solver version which run slower than in the previous version. The data structure for storing the neighborhoods of reference nodes was changed. While it is reported in [26] that there are two fields of size $22 * 8$ bytes for storing the neighborhood information, this scheme is replaced by one field of size $20 * 8$ bytes that stores a sorted list of all distinct appearing neighbors and another field of size $22 * 2$ bytes that maps the sorted list to their local index in the neighborhood. Hence, only 204 bytes per neighborhood are required instead of 352 bytes, saving about 40 % of memory for the neighborhoods. However, the new indirect accessing increases the runtime for calculating the stiffness contributions, since calculating the Dreiding potential is heavily based on accessing neighboring nodes and their spatial positions. Measurements reveal that this slows down the on-the-fly solver by up to 20 %. Hence, this is another sensible trade-off between memory usage and execution speed within the simulation framework. However, the storage scheme for the position data is also adapted, since the positions are no longer part of `Node` objects but are arranged in a vector accessed by the global index of the node. This step is necessary to integrate the Compressed Symmetric Graphs without additional memory overhead, but it simultaneously results in faster access to the data. The new line-based multiplication also speeds up the calculation. This even overcompensates the overhead of the changed neighborhood data structure. As the data in Appendix F.1 (Table F.3) shows, the overall runtime advantage for the new on-the-fly solver is about 10 % when the three changes are combined. The motivation for the first two change is that by reducing the overall storage for the graph data, it is possible to cache more stiffness contribution lines. As the remainder of the thesis demonstrates, the benefit of more cached contributions is much higher



(a) The slowdown when employing the on-the-fly solver for five different tube configurations and different number of threads on phase one nodes. The numbers near the boxes give the mean time per iteration of the reference solver in milliseconds that serves as baseline.



(b) The slowdown when employing the on-the-fly solver for five different tube configurations and different number of threads on phase two nodes. The numbers near the boxes give the mean time per iteration of the reference solver in milliseconds that serves as baseline.

Figure 9.3.: Performance comparison of the on-the-fly solver and the reference solver.

than several percent justifying the decisions that save memory at the cost of several percent of runtime.

A change in the measurement methodology is the second reason for the different slowdown. In [23], the assembly process of the stiffness matrix is included into the average runtime of the reference solver, since it consumes a considerable part of the total runtime. The main goal of the initial assembly process was to avoid unnecessary additional memory usage during the construction of the CRS matrix, but not to optimize its speed. As Section 8.2 shows, this process can now be executed in a parallelized and optimized fashion, resulting in a much lower construction time. Additionally, the setup of the graph-based on-the-fly solver also requires some time, whereas the former on-the-fly solver was able to start immediately. It makes more sense to compare both instantiation times separately independent of the actual time for solving. This increases the value for the slowdown compared to the old methodology by a factor of 1.5.

As a last point, the reference solver is now also parallelized and exploits the matrix symmetry, which has considerable effects on the single-threaded execution time and the scaling behavior compared to the version in [23] (see below). The slower single-threaded execution combined with the performance gain through parallel execution results in an overall speedup of 3.5. Altogether, the product of these three contributions results in the factor 5 which is about the difference of the old and the new on-the-fly solver version and can be observed between Figure 9.3a and the results reported in [23].

Figure 9.3a also demonstrates some jitter in the values that can be noticed for nearly all data points. This is likely a result of the Intel Turbo Boost technology which results in varying clock speed of the CPU depending on its temperature and thermal design power. The fact that the single-threaded runs have higher variation supports this explanation, since in that case the percentage by which the clock speed is reduced can nearly directly be mapped to the increase of the program runtime. Also note that the difference in the time per iteration between the on-the-fly solver and the reference solver is very high. As a consequence, relatively small changes in the runtime of the reference solver have a noticeable effect on the factor for the runtime overhead. The given standard deviation confirms that the different runs deliver a very constant average time per iteration.

The measurements for the on-the-fly solver are repeated on phase two nodes and the results are summarized in Figure 9.3b. The measurement data can be found in Appendix F.1 (Table F.2). One main difference of phase two hardware is the available number of threads which can be set up to 24. The difference for the single-threaded execution and the case of running with four threads is larger than for the phase one nodes. This mainly results from the fact that the reference solver is now operating with 24 threads. Therefore, it achieves a further speedup, and is faster than the reference solver on phase one nodes running with 16 threads. So, e.g., for the $(1, 4, 8, 355)^1$ tube, the reference solver requires on average 74.75 milliseconds per iteration on phase one hardware, while this value for phase two decreases to 53.30 ms. When employing 24 threads, the on-the-fly solver can reduce the slowdown compared to the reference solver to a factor of about 45 for all configurations. We observe, in general, that the on-the-fly solver runs faster on the Haswell processors. Comparing the runs to the same number of threads between both hardware phases reveals that the corresponding runs on phase two are always faster by at least 10 %. At the same time, the reference solver delivers about the same performance with only small advantages on phase two nodes.

The parallel efficiency of the different solvers on phase two has also been investigated. To that end, the median of all times per iteration for all runs with the same number of threads is calculated per tube configuration and solver. Then, the speedup S and the efficiency E when running with M threads are calculated for each tube configuration by the following two formulas (see e.g. [117]) :

$$S(M) = \frac{T_1}{T_M} \qquad\qquad E(M) = \frac{S}{M}$$

Afterward, the resulting $E(M)$'s were averaged for the different tubes per solver. The results are presented in Figure 9.4 which shows the efficiency for the on-the-fly solver, the reference solver exploiting matrix symmetry, and a variation of the reference solver that ignores the symmetry.

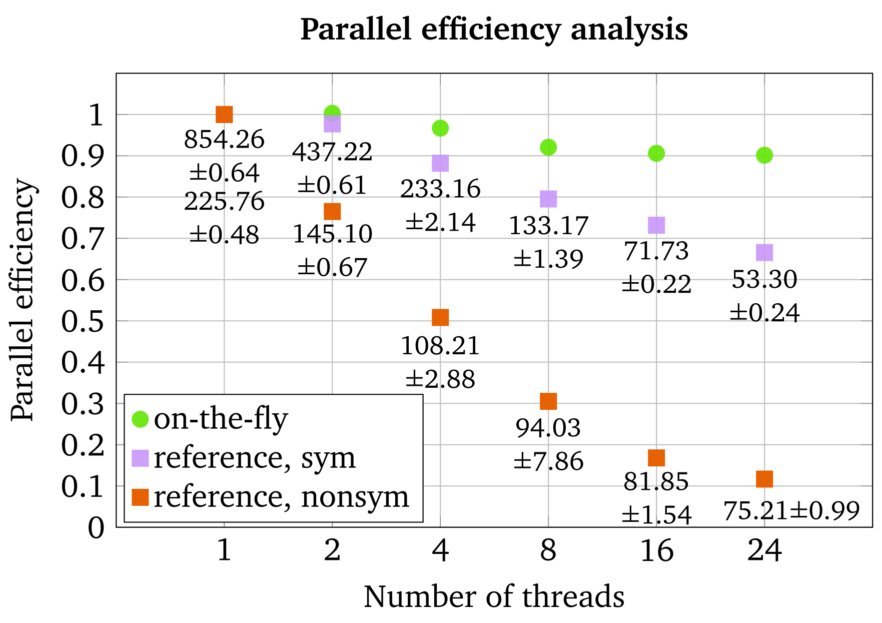


Figure 9.4.: Parallel efficiency for the on-the-fly and the reference solver ignoring and exploiting matrix symmetry on phase two nodes. The numbers give the absolute time per iteration for both CRS based solvers on the $(1,4,8,355)^1$ tube. For 1 thread the efficiency values coincide at 1.0.

The on-the-fly solver delivers very good scalability on the Haswell systems. The efficiency never drops below 0.9 not even for 24 threads. The three biggest tubes $(512,512)$, $(1,4,8,355)^1$ and $(2,6,12,116)^1$ even deliver an efficiency of 0.92 demonstrating good scaling for big problem sizes. This suggests that there will also be an acceptable efficiency on SMP systems with a higher number of available threads.

The reference solver version ignoring the symmetry behaves as expected. As predicted, for example, by the roofline performance model [118] the efficiency of the reference solver drops very quickly already for two threads because of the memory-boundedness of CRS multiplications. For 8 threads the CRS solver achieves a speedup of 2.5 compared to serial execution. This lies in the range of measurements performed by Çatalyürek et al. in 2012 [119], who reported a speedup of 2.5 on a dual socket Intel (Bloomfield, 4 cores per CPU) and of 2.2 on a dual socket AMD (Shanghai, 4 cores per CPU) system. Additionally, Williams et al. [120] investigated the scaling behavior of SpMV on several processors from 2009 and achieved comparable results.

The symmetry-exploiting reference solver behaves differently. Due to the higher OpenMP overhead, its single-threaded runtime is drastically higher than that of the standard CRS multiplication. With increasing number of threads this situation changes and from 16 threads onwards the symmetry-exploiting solver is faster due to better scaling. The numbers under the respective data points in Figure 9.4 show the required time per iteration for the two versions of the reference solver exemplarily for the $(1,4,8,355)^1$ tube. Here, the symmetric version is faster by a factor of 1.4 when 24 threads are employed.

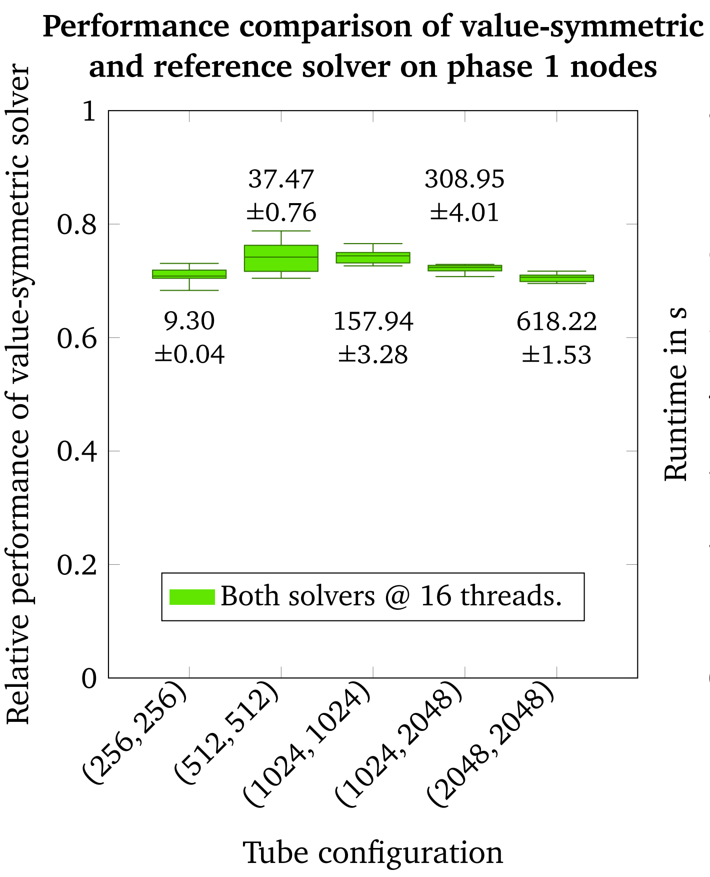
This section demonstrates that the on-the-fly solver is able to reduce the memory consumption by one order of magnitude, while it increases the runtime by up to 50 times when exploiting the shared-memory parallelization on current multicore nodes, compared to the optimized version of the reference solver. The difference to the previous results is mainly caused by a faster reference solver.

9.3.2 Value-symmetric solver versus reference solver

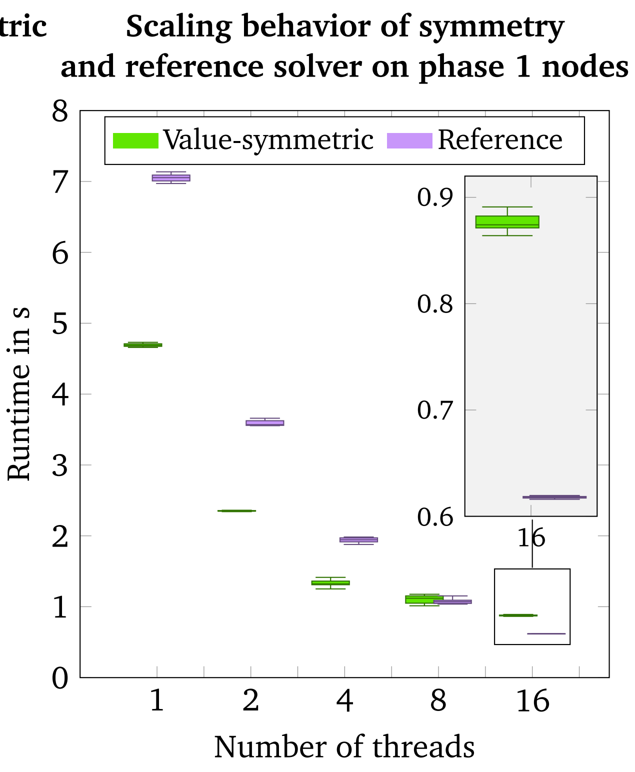
In this section, we compare the runtime of the value-symmetric solver with that of the reference solver on phase one nodes. As described in Section 8.5, the value-symmetric solver can only be applied to tubes of order 0. Hence, five large tubes of order 0 are chosen for the runtime measurements. No external load is applied to the initial CNT. Figure 9.5a summarizes the runtime results (complete data in Appendix F.2, Table F.4).

As can be seen for all configurations, the value-symmetric solver performs worse than the reference solver and achieves only about 75% of the peformance of , but the slowdown lies below 1.4 for the tested configurations with model sizes between $1.3 * 10^5$ and $8.4 * 10^6$ nodes. The main reason for the slowdown is the scaling behavior, which is caused by a straightforward parallelization of the loop over the map that contains the base-symmetry nodes. Additionally, the resolution of symmetry information requires several memory accesses in parallel which may overcharge the memory system. In contrast, during single-threaded execution, the value-symmetric solver is faster by a factor of 2 than the reference for some configurations. This mainly results from the fact that the stiffness contributions for the base-symmetry nodes only need to be fetched once and then can be reused several hundred times in the case of very long tubes. Figure 9.5b visualizes the runtime for both solvers for the $(2048, 2048)$ tube, which is representative for all tested configurations.

This test also reveals a linear dependency of the runtime of the reference solver and the value-symmetric solver on the size of the CNT. Table 9.1 shows the dependence of the reference solver, and Figure 9.5a demonstrates that the value-symmetric solver behaves in the same way.



(a) The slowdown that arises when employing the value-symmetric solver compared to the reference solver when running with 16 threads on different tube configurations. The numbers near the boxes give the mean time per iteration of the reference solver in milliseconds that serves as baseline.



(b) The time per iteration for value-symmetric and reference solver for varying thread number and the tube $(2048, 2048)$.

Figure 9.5.: Performance comparison of the value-symmetric solver and the reference solver.

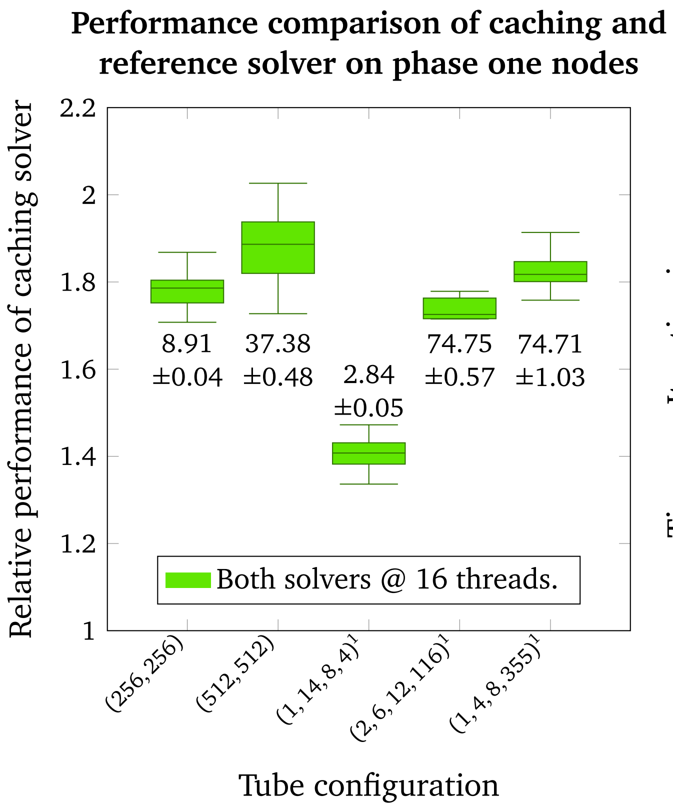
Table 9.1.: Scaling of the runtime of the reference solver with the model size.

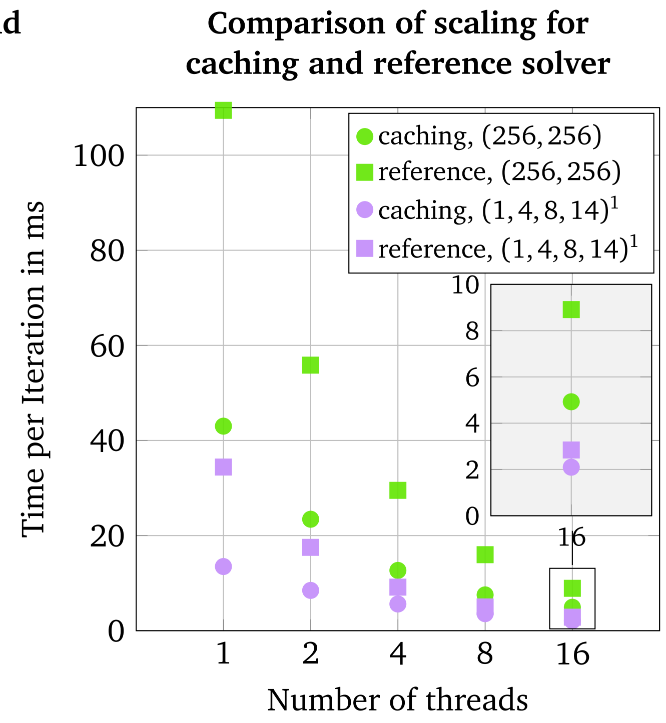
Runtime reference (ms)	**Growth of model size**	**Factor**
9.3		
37.47	4	4.029
157.94	4	4.215
308.95	2	1.956
618.22	4	2.001

So in total, the performance is again somewhat higher than the improved value-symmetric solver presented in [27] while still requiring nearly no additional memory compared to the on-the-fly solver. For example, the $(2048, 2048)$ tube requires 30 MB to cache all required potential data. Additionally, it provides the perspective of further increasing the speed by optimizing the parallelization.

9.3.3 Caching solver versus reference solver

The caching solver is now compared to the reference solver. In this section, it always applies full caching, since this is always possible if there is sufficient memory to execute the reference solver. The fastest configuration of both solvers, that means employing all 16 available threads, is compared in Figure 9.6a. Again, the numbers in the diagram show the absolute mean runtime for the reference solver for the respective tube which is the baseline. Full details about the measurement data can be found in Appendix F.3 (Table F.5).





(a) The speedup which is achieved by the caching solver compared to the reference solver when running with 16 threads on different tube configurations. The numbers near the boxes give the mean time per iteration of the reference solver in milliseconds that serves as baseline.

(b) The time per iteration for caching and reference solver for varying thread number on the tubes $(256, 256)$ and $(1, 4, 8, 14)^1$.

Figure 9.6.: Performance comparison of the caching solver and the reference solver.

The caching solver is able to outperform the reference solver by at least a factor of 1.4 in all cases, but the actual speedup varies. For the smallest tested tube $(1, 4, 8, 14)^1$, the smallest speedup results from the high synchronization overhead of the OpenMP threads. Due to the low node count, the span of nearly every thread completely overlaps with others and thus a safe write is not possible without synchronization. This can be verified by considering Figure 9.6b that shows the development of the time per iteration for the caching and the reference solver when varying the number of threads for two tube configurations. For single-threaded execution, the situation is the same: The caching solver is faster by a factor of 2.5 for both tubes. However, the situation changes with increasing number of threads. While for the larger $(256, 256)$ tube the speedup with four threads is still 1.8 it is only 1.4 for $(1, 4, 8, 14)^1$.

The remaining tubes deliver a comparable speedup of about 1.8. Although $(2, 6, 12, 116)^1$ and $(1, 4, 8, 355)^1$ are of nearly equal size, the caching solver runs somewhat slower on that configuration (43.83 vs 41.25 milliseconds) while the reference solver delivers the same performance (74.75 vs 74.71 seconds). An analysis reveals that the scaling behavior is the main reason for this difference, too. While for both tubes the average runtime of the solver with 1 thread only differs by less than two percent, the difference with 16 threads increases to more than 7%. The nodes within the $(2, 6, 12, 116)^1$ tube are distributed differently since the junctions within $(2, 6, 12, 116)^1$ contain twice as many nodes as those of $(1, 4, 8, 355)^1$. This changes the size of the spans for the different threads and increases the number of synchronized writes, resulting in a worse scaling behavior.

Figure 9.6b also shows that the runtime difference between the caching solver and the reference solver is the highest in the single-threaded case, indicating that the structure-related optimizations in the SpMV have a very positive effect and that the scaling is the limiting factor. The single-threaded runtime for the investigated tubes also grows linearly with the number of nodes in the tube. Calculating the time to process one node during the SpMV results in 328 ns for the $(256, 256)$ tube which is the fastest one, and 354 ns for $(1, 4, 8, 355)^1$ which is the slowest one. To estimate the influence of synchronization in general, the alternative heuristic is applied for test purposes. In that case, the speedup increases to a factor of 2 with 16 threads, which is consequently some kind of upper limit.

But overall, these considerations demonstrate that the caching solver always delivers a better performance than the reference solver, independent of the number of threads employed and the tube configuration, while only requiring half the memory.

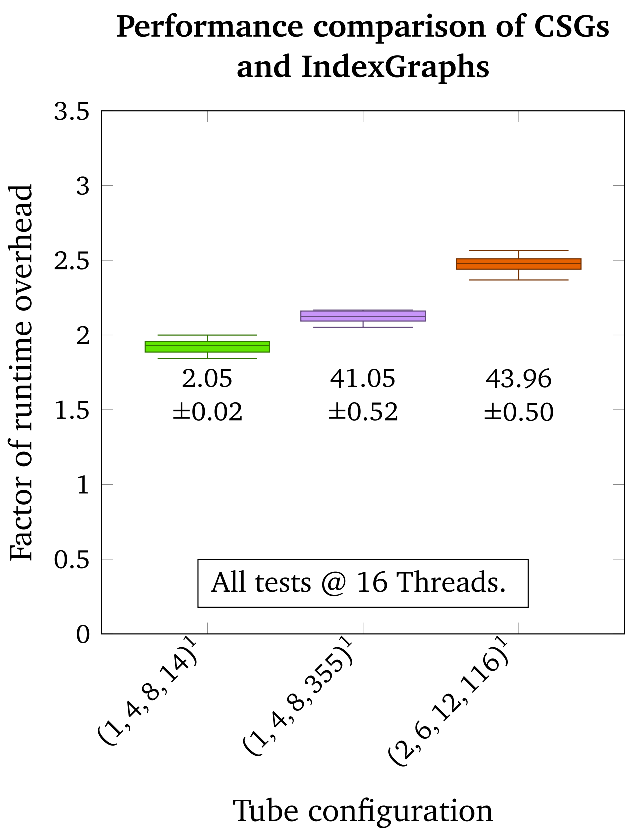
As a short excursion, we want also to briefly compare the runtime of the simulations with the caching solver to the dockSIM framework which represents the state-of-the-art methodology. To that end, a single linear equation system resulting from the tubes $(1, 4, 8, 4)^1$, $(256, 256)$ and $(512, 512)$ is instantiated in a tension load case and solved within the new framework and dockSIM while employing 16 threads. We compare the time for the solution of the system, which includes for the new framework the instantiation of the caching solver and the solution within the PCG method while it includes assembling the stiffness matrix and employing the Pardiso library for the solution of the assembled linear equation system. The average time required for the new framework is 4.9 s for $(1,4,8,4)^1$, 213 s for $(256,256)$ and 1744 s for $(512,512)$. The dockSIM framework is considerably faster with 0.7 s for $(1,4,8,4)^1$, 9 s for $(256,256)$ and 45 s for $(512,512)$. This results from the different approaches of a direct and an iterative solver while the former one outperforms the latter one at the cost of much higher memory requirements for the direct solution (see introduction of Chapter 8). The relative difference for the tubes $(256,256)$ and $(512,512)$ is higher than for $(1,4,8,4)^1$ because the larger tubes also require more PCG iterations to converge. Three points should be additionally kept in mind when considering these runtime values: First, the direct solver exceeds the memory capacity much earlier than the caching solver (or even than the reference solver) and not be applicable for larger models. Second, the equation system was solved to full precision. The runtime difference becomes smaller in the case that a less accurate solution is sufficient. Third, the employed preconditioners deliver a relatively high number of PCG iterations and a structure-tailored variant may be able to considerably reduce this number which would be reflected one-to-one in the total runtime as already discussed in Section 8.1.

9.3.4 Caching solver with IndexGraph and Compressed Symmetric Graphs

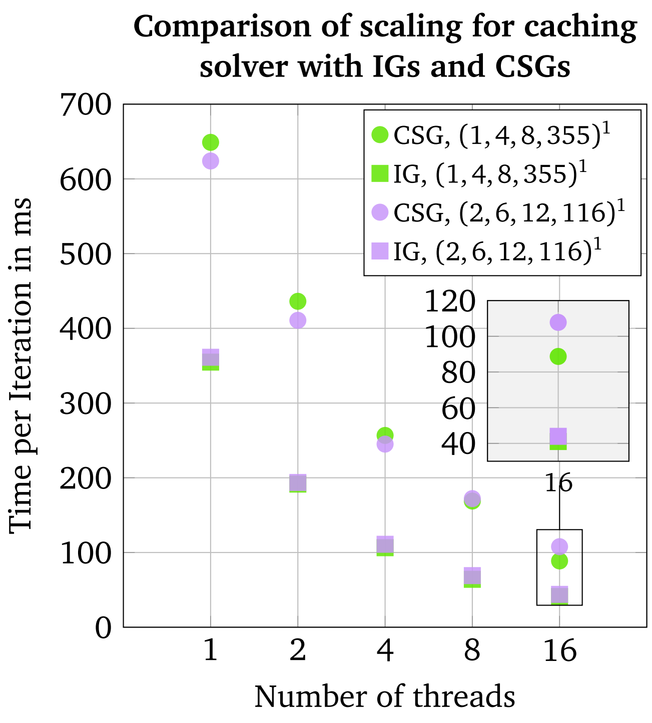
In this section, the influence on the runtime of the Compressed Symmetric Graphs is evaluated. To that end, the caching solver is employed and combined with both the IndexGraphs (IGs) as the default data type of the framework and combined with the CSGs, while the combination with IndexGraphs serves as performance baseline. Two different test suites are performed. In the first one, it is assumed that both graph types leave enough memory to enable the caching solver to fully cache all contribution lines. This allows a direct determination of the overhead that is generated by the CSGs. The second test suite demonstrates the main advantage of the CSGs in case of low memory, i.e., when the caching rate of the caching solver differs when employing IndexGraphs and CSGs. All measurements in this section are performed on phase one nodes.

The results of the first test suite are summarized in Figure 9.7a (detailed measurement data in Appendix F.4 Table F.6). For three different order 1 tubes, the time that is required to calculate one PCG iteration is measured with the two different graph structures. The caching solver always runs with 16 threads because this represents the fastest configuration.

An inspection of Figure 9.7a reveals that the runtime overhead of the CSGs is a factor of about 2 and that the overhead is somewhat higher for the two larger tubes. The slower access to the neighborhood information compared to IndexGraphs, which is required for each reference node when accessing the rows and columns in the cached data during the SpMV, is the main reason for this behavior. For the single-threaded performance this results in the fact that, e.g., one iteration of the $(1,4,8,355)^1$ tube with IndexGraphs requires on average 0.36 s while the solver needs 0.62 s with CSGs. This is already a difference by a factor of 1.7. Another aspect is the



(a) Slowdown when employing CSGs and full caching in the caching solver compared to IndexGraphs. The numbers near the boxes give the mean time per iteration of the caching solver with IndexGraphs in milliseconds that serves as baseline.



(b) The time per iteration for the caching solver employing full caching for varying thread number with IndexGraphs and CSGs and the tubes $(1, 4, 8, 355)^1$ and $(2, 6, 12, 116)^1$.

Figure 9.7.: Comparison of caching solver with IndexGraphs and CSGs and full caching.

scaling behavior of the caching solver, which is worse when combined with the CSGs as with IndexGraphs. The main reason for this is the unequal access time to the node information, resulting in an uneven workload distribution for the different threads and a higher load on the memory interface of the system, since each thread needs to locate and unfold the data requested. Hence, while the caching solver with IndexGraphs is 9.2 times faster when running with 16 threads instead of one the speedup is only 7.0 with CSGs, which is a difference of factor 1.3. Combining 1.3 with the factor 1.7, which was observed for the single-threaded runtime, results in the observed factor of 2.2 for the overall slowdown.

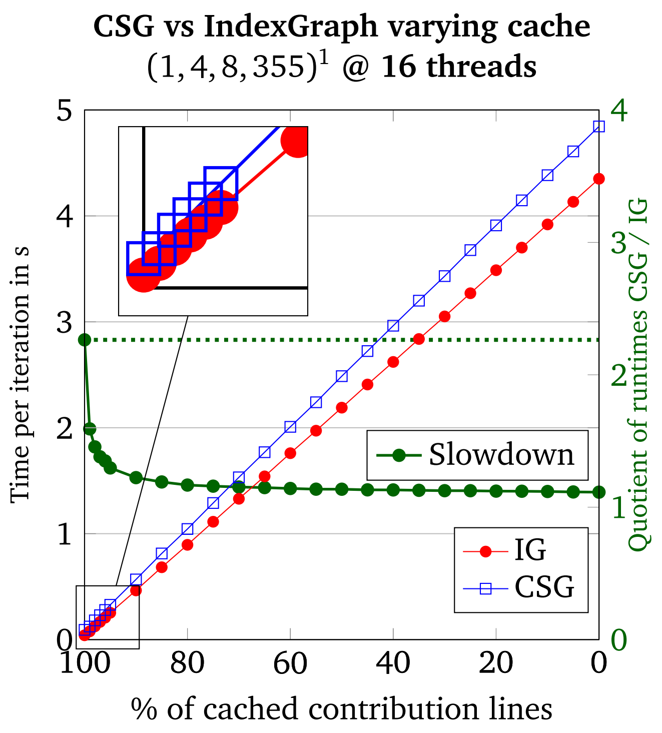
A slightly different behavior can be noticed for the $(2, 6, 12, 116)^1$ tube, despite the fact that both tubes have the same number of nodes. This is highlighted in Figure 9.7b. While the time per iteration for $(2, 6, 12, 116)^1$ for the single-threaded execution with CSGs is even somewhat lower than for $(1, 4, 8, 355)^1$, it is higher by several percent when running with 16 threads. This seems to be an interaction of the worse scaling behavior of the $(2,6,12,116)^1$ tube and the slower and varying access time to the information in CSGs.

However, the behavior with low memory, i.e., after the graph construction and solver initialization there is not enough RAM to fully cache all stiffness contributions, is the more interesting question. The performance results for tests with a predefined percentage of cached contributions are shown in Figures 9.8a and 9.8b for the two largest available tubes (see also Appendix F.5, Table F.7). There, the CSGs have a lower memory footprint for the same caching rate as IndexGraphs. As the diagrams reveal, the required time to calculate one PCG iteration grows linearly with the number of non-cached contribution lines. As can be seen on the zoomed insert in the graph, this is also valid for the first few percent of contributions that cannot be cached. But more important is that the zoom also demonstrates the performance difference between both variants, when comparing the centers of the circles to those of the squares. This is also true for all other investigated tube configurations. Both tubes have a very similar behavior, resulting from their identical size. Figure 9.8a also shows a second graph in green that demonstrates that the relative runtime difference between CSGs and IndexGraphs drops with a decreasing percentage of cached contributions. For full caching the slowdown is 2.26. This value decreases very fast to 1.17 for 80 % and stays relatively constant for the remainder only further reducing to 1.11.

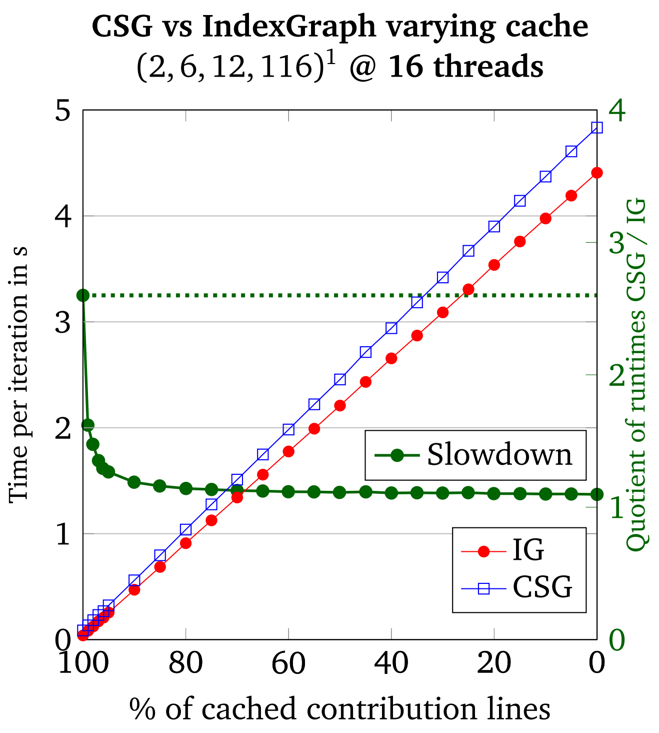
The $(2,6,12,116)^1$ tube behaves in the same way, with a maximum difference of 2.60 for full and a minimum of 1.10 for no caching. This shows that the calculation speed of the caching solver is nearly identical, independent of the employed graph data structure, if the percentage of cached contributions is low.

As Section 7 proofs, the compression rate ρ of the CSG is related to the additional percentage of contribution lines that can be cached μ by $\mu = \frac{\rho}{2.5}$. To demonstrate this main advantage of the CSGs have a look at Table 9.2. We use the slope of the straight lines in Figures 9.8a and 9.8b to interpolate the expected runtime increase if the caching rate between CSGs and IndexGraphs varies, since we regard the dependence of the runtime of the two tubes $(1,4,8,355)^1$ and $(2,6,12,116)^1$ from Figures 9.8a and 9.8b as representative for all higher order SCNTs with comparable configuration. Table 9.2 shows three theoretical scenarios in the case of low memory with different μ, depending on the assumed percentage of nodes the CSGs are able to compress. The second column shows the assumed percentage of cached contribution lines for the caching solver when combined with GSG. The third column shows the resulting caching rate which the solver can achieve if it is combined with IndexGraphs, while the last column gives the speedup for CSGs compared to the configuration with IndexGraphs and the same memory footprint. The speedup results from the higher caching rate with CSGs. Note that within a row the solver has the same memory footprint with both graph types.

The upper part of Table 9.2 summarizes the results for the worst case of $\rho = 25\,\%$ implying $\mu = 10\,\%$. Here, the CSGs have a considerable performance advantage when they are able



(a) Time per iteration when the percentage of cached contribution lines is decreased from full to no caching for the $(1,4,8,355)^1$ tube. Comparing the runtime with same caching rate but lower memory footprint for CSGs.



(b) Time per iteration when the percentage of cached contribution lines is decreased from full to no caching for the $(2,6,12,116)^1$ tube. Comparing the runtime with same caching rate but lower memory footprint for CSGs.

Figure 9.8.: Performance comparison of the caching solver with different graph data structure and varying caching rate.

to cache at least 90 %, but they are faster in all cases while they have the identical memory footprint.

In the middle part, the case of a medium compression is shown. In that case a speedup of up to 13.7 is possible when employing CSGs. The compression rate of 60 % can be achieved for nearly all tubes which are not the smallest possible for a given junction configuration and order. The speed of the calculation can nearly be doubled, if the solver can cache at least 80 % when comined with CSGs.

The lower part of Table 9.2 shows a last scenario for tubes with an l_0-value greater than 22 and at a same time a d_0-value of at least 8. Then, ρ can exceed 95 % enabling a high difference in the cachable nodes compared to IndexGraphs. Considerable performance gains can be attained, if about 60 % of the contribution lines can be cached with CSGs.

9.3.5 Higher order tubes

Also several tests are performed to predict the performance of the simulation of higher order tubes. The code is changed in that way that the PCG algorithm does not execute feasibility and

Table 9.2.: Speedup with CSGs in three scenarios. The underlying performance values are interpolated by the mean values of the tubes $(1,4,8,355)^1$ (Figure 9.8a) and $(2,6,12,116)^1$ (Figure 9.8b). The first column shows the compression rate assumed and the resulting additional percentage of contribution lines which can be cached, resulting in the values shown in columns two and three for CSGs and IndexGraphs. Within a table row, the solver has the same memory footprint.

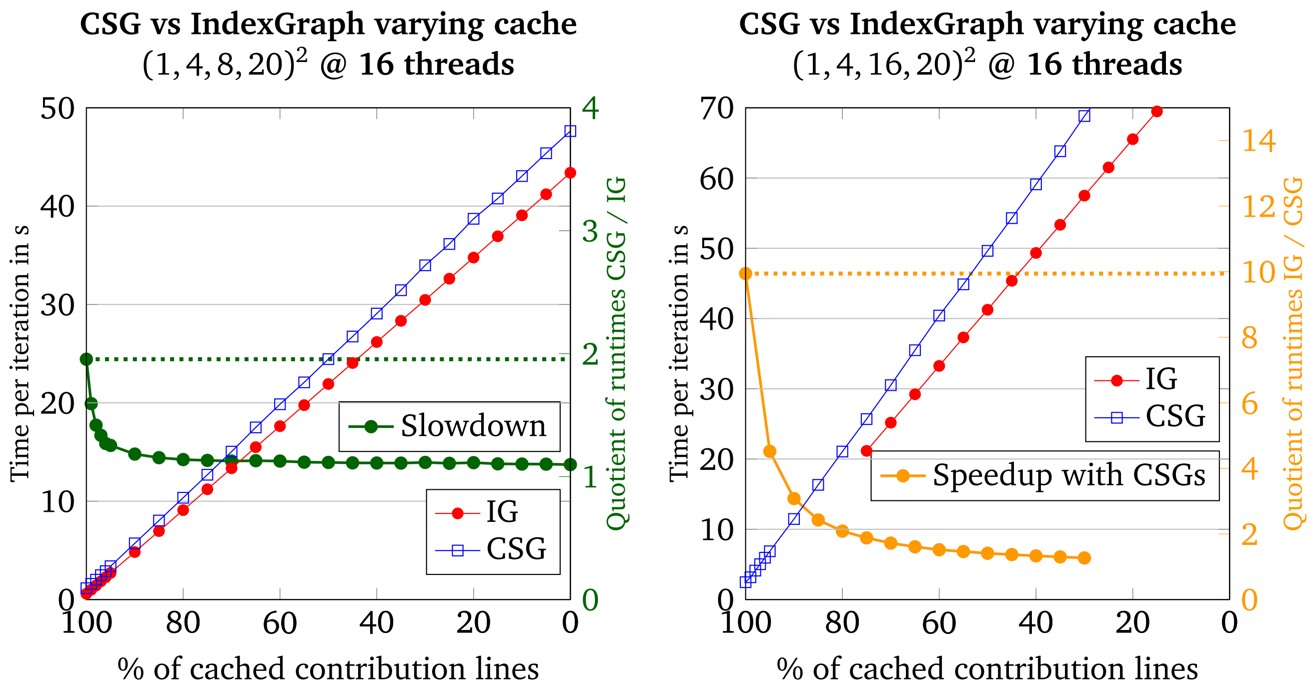
Compression rate ρ	**Cached CSG**	**Cached IG**	**Speedup**
25 % $\Rightarrow \mu = 10\,\%$	100 %	90 %	5.2
	90 %	80 %	1.6
	80 %	70 %	1.2
	70 %	60 %	1.1
	60 %	50 %	1.1
	50 %	40 %	1.1
60 % $\Rightarrow \mu = 25\,\%$	100 %	75 %	13.7
	90 %	65 %	2.2
	80 %	55 %	1.7
	70 %	45 %	1.2
	60 %	35 %	1.1
	50 %	25 %	1.1
96 % $\Rightarrow \mu = 40\,\%$	100 %	60 %	20.8
	90 %	50 %	3.9
	80 %	40 %	2.6
	70 %	30 %	2.0
	60 %	20 %	1.8
	50 %	10 %	1.3

convergence checks but executes a predefined number of iterations, since there is no relaxed position data available for larger tubes. Afterward, the mean value of time per iteration is calculated. The chosen model tube for the first tests is $(1,4,8,20)^2$ since it increases the problem size, compared to the largest two available order 1 tubes, by exactly one order of magnitude and the reference solver is still able to cope with this problem size.

Comparing the performance of the reference solver to that of the on-the-fly solver reveals that the difference is again one order of magnitude. While the reference solver requires 0.76 s per iteration, the on-the-fly solver is slower by a factor of 57, which lies in the range of the results for order 0 and 1 tubes.

When employing the caching solver with full caching, it can outperform the reference solver by a factor of 1.4 (0.56 s versus 0.77 s). Here, the same effect is visible as for the $(1,4,8,14)^1$ tube: The junction elements contain a relatively high number of nodes compared to the length of the tube. This can be confirmed by applying the alternative synchronization heuristic. In that case, the time per iteration is reduced to 0.39 s which increases the performance difference between both solvers to a factor of 2. Hence, this tube of order 2 also confirms the previous results about the caching solver combined with IndexGraphs and full caching.

Like for order 1 tubes, the dependence of the calculation speed on the caching rate for IndexGraphs and CSGs is investigated and the results are shown in Figure 9.9a (see also Appendix F.5, Table F.8). The behavior is very similar to the order 1 tests, since the runtime grows linearly with the number of non-cached contribution lines. We also see that the slowdown when employing CSGs with the same caching rate, i.e., lower memory footprint, is about a factor of 2 when full caching is employed and decreases to less than factor 1.1 when the caching rate is decreased. This is in very good agreement with the order 1 results, too.



(a) Time per iteration when the percentage of cached contribution lines is decreased from full to no caching for the $(1,4,8,20)^2$ tube. Comparing runtime with the same caching rate but lower memory footprint for CSGs.

(b) Time per iteration when the percentage of cached contribution lines is decreased from full to no caching for the $(1,4,16,20)^2$ tube. Comparing runtime of configurations with the same memory footprint but a difference in caching rate of 25 %.

Figure 9.9.: Performance comparison of caching solver with IndexGraphs and CSGs on order 2 tubes.

The last tube that we consider is $(1,4,16,20)^2$, which again doubles the problem size and consists of $1.98 * 10^7$ nodes. This tube demonstrates several of the theoretical predictions made so far. First of all, it shows that the reference solver runs out of memory at some point, while the caching solver is able to deal with larger problem sizes. In that case, the matrix consists of $1.67 * 10^9$ non-zero values, resulting in a memory footprint for the CRS data of nearly 25 GB. This exceeds the capacity of 28 GB of available memory on phase one nodes together with the graph data and additional variables. These values also confirm the estimation of the reference solver's memory consumption by Equation 8.4.

The last point is the comparison between IndexGraphs and CSGs and the attainable caching rate, which is depicted in Figure 9.9b. The caching solver is not able to achieve full caching for tube $(1,4,16,20)^2$ when combined with IndexGraphs due to memory limitations, but full caching is possible when running together with CSGs. Hence, the red line starts at the highest caching rate that is possible using IndexGraphs at the value of 75%. An additional orange line visualizes the performance differences when the solver runs with equal memory footprint with both types of graphs. The left-most data point on the orange line represents the situation when the maximum possible number of contribution lines is cached for both data structures, i.e., 100 % for the CSGs and 75 % for the IndexGraphs. In this case, the simulation with CSGs is faster by one order of magnitude (factor 9.9) while having the same memory footprint. Therefore, this practical case with an additional caching rate of $\mu = 25\,\%$ confirms the results of the second theoretical scenario in Table 9.2 which assumes $\mu = 25\,\%$.

Altogether, the analysis on order 2 tubes confirms the predictions made in our earlier publications and demonstrates that the framework presented can deal with large equation systems with an order up to $5.6 * 10^7$ and highlights the usefulness of CSGs.

10 Conclusion and Outlook

In this final chapter, we summarize the results of this thesis (Section 10.1) and discuss directions for future research (Section 10.2).

10.1 Summary

This thesis was motivated by the work of Schröppel and Wackerfuß, demonstrating that the structure of super carbon nanotubes can be modeled by a graph algebra approach in a very elegant way. As shown in [45], the encoding of structural information within a tuple-based node labeling approach is its main advantage. In this thesis, this description was interpreted in a geometrically constructive fashion and the self-similarity and symmetry were highlighted during the description of the construction process of SCNTs. We demonstrated how exploiting these structural properties of the SCNTs enables very efficient data structures for constructing and storing the tubes.

The issue of indexing the nodes in the graph models was solved by the structure-tailored IndexGraphs that especially deliver fast unique indices in a moderate range. Compared to naive indexing, the procedure in IndexGraph reduces the highest occupied index by five orders of magnitude, thus avoiding overflows during index calculations.

As an alternative, the principles of the perfect hashing algorithm were extended to deal with long tuples with varying range per entry, thus allowing this new generic algorithm to index general multidimensional scientific data. For tubes of up to order 3, i.e., tuples of length 28, we verified that the reduction of the highest occupied index lies in the same range as for the IndexGraphs.

This thesis introduced the concept of Compressed Symmetric Graphs to reduce the necessary storage dedicated to the neighborhood information in the graphs. It is possible to reduce the amount of explicitly stored neighborhood data by up to 99 % by dynamically recalculating neighborhood information from saved parts of the tube. Even in worst case scenarios, 25 % of the nodes can be compressed and the required storage per node is then reduced from 336 to 40 bytes.

The structure of SCNTs was also exploited to develop a novel matrix-free solver for the equations arising during the mechanical simulations, increasing the range of feasible problem sizes. The resulting runtime overhead from recalculating the contributions to the stiffness matrix is managed by an intelligent caching mechanism that fully uses the resources provided by the underlying hardware. Additionally, by walking the graph, it is possible to avoid the storage for the non-zero patterns of the matrix that arises in state-of-the-art sparse matrix storage schemes.

Combining these algorithmic advances with an efficiently vectorized and parallelized implementation, the presented framework is able to solve the problems the reference implementation can cope with twice as fast while requiring only half the memory. With our new approach based on CSGs, problems become feasible that the reference solver cannot address: We can set up the equation systems for tubes of order 2 with 20 million nodes still allowing full caching of stiffness data, whereas the storage of the matrix would require more than 25 GB for the nearly $1.7 * 10^9$ non-zero values.

The algorithms and data structures presented in this thesis were integrated into the dockSIM code of the MISMO group and multi-step simulations for several order 1 tubes were performed. Hence, the work of this thesis also shows the flexibility of the dockSIM framework, since the data structures and the algorithms we employed strongly differ from the procedures that have been integrated into dockSIM so far.

10.2 Future Work

Although the concepts presented in this thesis can considerably reduce the memory requirements for modeling and simulation of SCNTs a single compute node will, at some point, not be capable of coping with all data or will at least have to recompute a lot of stiffness contributions in each PCG iteration, resulting in high runtimes. Hence, a promising avenue for future work are distributed simulations. For the PCG that means that, in particular, the SpMV is a candidate for a distribution between several compute nodes, based on the graph structure and with different rings as base unit. Then, each process can calculate its contribution to the result vector with the caching or the reference solver independently and only requires the rings assigned to it, plus the information of all the nodes in the neighborhood of the assigned ones being a relatively small number compared to the overall nodes. This partitioning and dispatching of the data needs only to be done once at the start of PCG. In the case of the reference solver, each process can instantiate its local CRS data structure with help of the graph while in the case of the caching solver, each process evaluates for itself how many contribution lines can be cached on the compute node it is running on. Each SpMV process needs to receive its portion of the input vector and to send its part of the result vector to a place where it is assembled to the overall result vector. This work distribution scheme also enables a straightforward partitioning of the Jacobi and the block-diagonal preconditioners and allows a distributed solution of the equation system.

These ideas are subject of an ongoing Bachelor's thesis of Taylan Oezden where preliminary results for the reference solver and the Jacobi preconditioner suggest that the partitioning system is working and that the boost-library[1] offers an easy-to-use possibility to transfer the graph objects between processes.

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

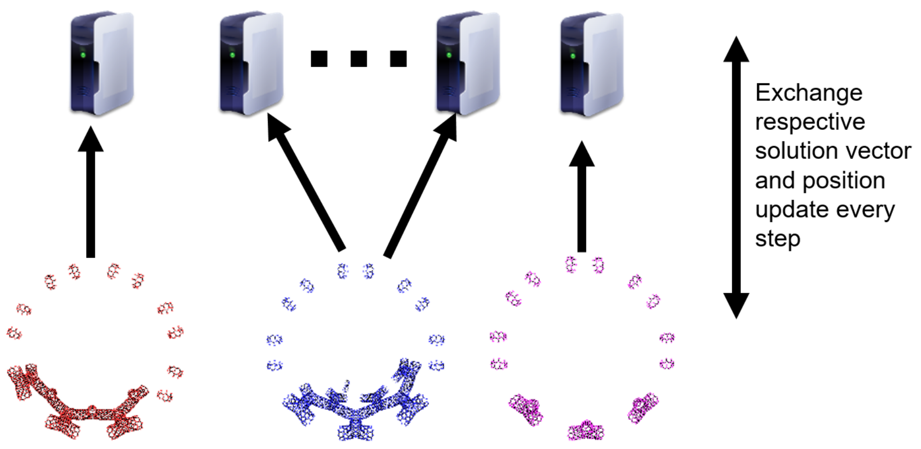

Exchange respective solution vector and position update every step

Figure 10.1.: Distributing CSGs with MPI.

The distributed calculation is another good example for the usefulness of CSGs that is illustrated in Figure 10.1. Instead of dispatching full rings of the tubes, it is sufficient to distribute the respective parts of the CSG representation of the SCNT model to each process which can be categorized in three types: the left and the right boundary of the tube that have to be treated differently and the 1-leading ring, including the very small part of the 2-leading ring. This information is sufficient to reconstruct all symmetric rings.

There are positive effects in two ways: First, the amount of data which needs to be sent around before starting PCG is reduced, while second, each node saves memory for the graph representation that can be spent to increase the caching rate of the solver and thus to speed up the overall simulation.

An algorithmic question is if an appropriate way to propagate the boundary conditions through the tube exists that can be combined with an adapted boundary exchange strategy, so that different processes are able to execute several PCG iterations independently or even solve their local system on their own, while the results are assembled and perhaps smoothed afterward. This offers the main advantage that synchronization is only required after several successive PCG iterations or even after all processes found their local solution to the equation system. However, this may involve several additional iterations until the solution which is achieved in that way converges to the real solution.

In addition, we see further potential for improvement in synchronization and scalability behavior of the caching solver. The alternative synchronization heuristic (see Section 8.4.3) shows that the speedup compared to the reference solver may even be higher if reducing the number of synchronizations between the threads is possible. Here, it is required to identify the compromise between the pessimistic concept of non-overlapping spans and the heuristic one that causes race conditions in some cases because its assumption is too optimistic. Additionally, an evaluation of the synchronization criterion must be possible in an efficient way. Otherwise, the benefit of a good criterion will be masked by the evaluation overhead.

The issue of preconditioning the equation system was covered in Section 8.1. Although the presented Jacobi and block-diagonal preconditioners can be constructed and solved very fast and possess a low memory footprint, their convergence rate is considerably lower than with an incomplete Cholesky factorization preconditioner. Consequently, it makes sense to search for a structure-related preconditioner that can improve the speed of convergence, while keeping the required storage for it as small as possible.

Additionally, further improvement of the speed of fetching the information in the CSGs can be considered. The runtime overhead of factor 1.4 in [26] in 2016 has increased to about 2 now, since on the one hand several modifications led to a speedup for the IndexGraphs and on the other hand the optimizations that were integrated into the caching solver cannot realize their full effect since the CSGs are the limiting factor. Investigating and optimizing the employed data structure may help to reduce this influence. This would further increase the advantages of CSGs over IndexGraphs in the case of low memory.

Altogether, we feel that CSGs are a promising data structure with significant potential even beyond SCNTs and we want to investigate other problem domains in simulation science where CSGs may be of advantage. Other simulations on SCNTs like for electronic properties or the analysis of heat transfer may also profit from CSGs since they also rely on a way to identify neighborhood relations within the graph model and this can be realized in a very memory-efficient fashion with CSGs. A main prerequisite to employ CSGs for another application domain is of course that the underlying structure can be modeled as a graph. This is, for example, the case for mesh models employed in general FEM simulations. Additional prerequisites are that the modeled structure needs to contain inherent self-similarities that are reflected by the meshing method, i.e., self-similar regions must be meshed in the same way and with a regular mesh. Hence, the meshing method must be configured in that way that it sometimes partitions parts of the structure into more polygons than actually required by the computation to generate this regularity. The resulting storage overhead will be compensated by compressing the structure within the modified CSGs.

We also see the possible applicability of the principles behind the caching solver in the generic simulation of materials. In general, the matrix-free solution can be realized by repeating the calculations to assemble the respective stiffness matrix each step of an iterative method. Computationally this requires methods to efficiently calculate contributions to that matrix, in the best case with independent parts. A reordering of the data improving the prefetching and cache behavior, like it is implemented in the caching solver for this thesis, can also be realized by investigating the underlying structure the solver should be adapted to.

Appendices

A Summary of all Tubes

Table A.1.: For each tube employed in this thesis, its configuration $(d_x, l_x, d_0, l_0)^L$, the respective tuple length, followed by the number of nodes and edges in the graph are shown. The next two columns give the size of the junctions, i.e., how many elements of the previous level are contained in them. All levels > 1 have the same number of elements.

Tube	Tuple length	No. of nodes	No. of edges	Elem. in lvl 0 junc.	Elem. in lvl L junc.
$(256, 256)$	4	1.31E+05	3.93E+05	0	0
$(512, 512)$	4	5.24E+05	1.57E+06	0	0
$(1024, 1024)$	4	2.10E+06	6.29E+06	0	0
$(1024, 2048)$	4	4.19E+06	1.26E+07	0	0
$(2048, 2048)$	4	8.39E+06	2.52E+07	0	0
$(4096, 4096)$	4	3.36E+07	1.01E+08	0	0
$(1, 4, 8, 4)^1$	12	1.13E+04	3.37E+04	176	0
$(1, 4, 8, 8)^1$	12	2.25E+04	6.75E+04	176	0
$(2, 6, 4, 8)^1$	12	2.78E+04	8.32E+04	434	0
$(1, 4, 8, 14)^1$	12	3.94E+04	1.18E+05	176	0
$(1, 11, 10, 8)^1$	12	8.19E+04	2.46E+05	512	0
$(1, 4, 16, 16)^1$	12	9.01E+04	2.70E+05	176	0
$(1, 6, 12, 18)^1$	12	1.18E+05	3.52E+05	272	0
$(2, 6, 12, 18)^1$	12	1.56E+05	4.66E+05	360	0
$(1, 4, 8, 355)^1$	12	1.00E+06	3.00E+06	176	0
$(2, 6, 12, 116)^1$	12	1.00E+06	3.01E+06	360	0
$(1, 2, 8, 8)^2$	20	8.19E+05	2.46E+06	80	212
$(1, 3, 6, 6)^2$	20	1.18E+06	3.56E+06	128	260
$(1, 4, 8, 8)^2$	20	3.96E+06	1.19E+07	176	356
$(2, 3, 6, 12)^2$	20	4.77E+06	1.43E+07	182	350
$(1, 4, 8, 20)^2$	20	9.91E+06	2.97E+07	176	356
$(1, 4, 16, 20)^2$	20	1.98E+07	5.95E+07	176	356
$(2, 6, 8, 8)^2$	20	2.41E+07	7.23E+07	434	680
$(1, 2, 4, 4)^3$	28	1.64E+07	4.91E+07	80	212
$(2, 3, 4, 5)^3$	28	2.41E+08	7.23E+08	182	350
$(2, 6, 4, 4)^3$	28	6.42E+09	1.93E+10	434	680

B Some Notes on Terminology of Super Carbon Nanotubes

A CNT can be thought of as a rolled up graphene sheet. This sheet can be wrapped in different ways, e.g. from left to right or from top to down, resulting in tubes of different shape. The way how the sheet is rolled up is called the *chirality* of the CNT. In principle, there are three types of chirality for SWCNTs that are called *zigzag*, *armchair* and *chiral* ([121], [122]). To determine the chirality, a local coordinate system is embedded into the graphene as shown by the two blue vectors a_1 and a_2. The angle between both axes is 60°. Each atom is assigned a 2D coordinate (a_1, a_2) within this coordinate system. The configuration of the chirality is given as the coordinate (m, n) that determines a line within the graphene together with $(0, 0)$. This determines the chirality and diameter of the tube.

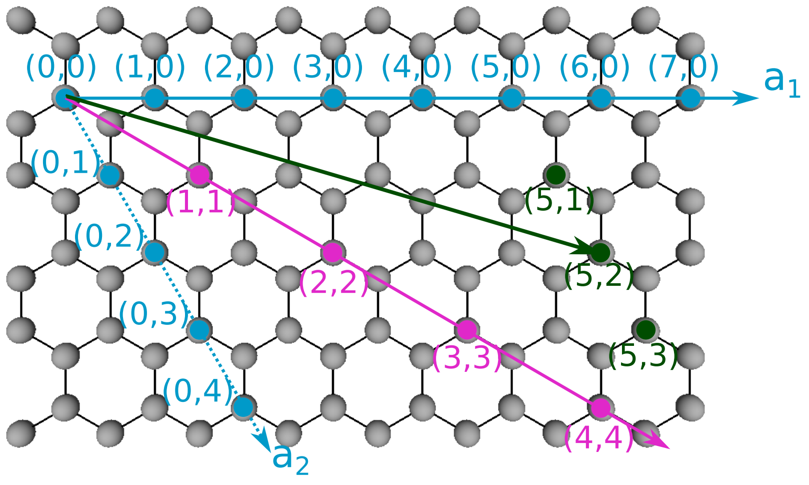


Figure B.1.: The three different chiralities of SWCNTs: zigzag in blue, armchair in magenta and chiral in green.

The three solid vectors indicate the different kinds of chirality. To create a tube of zigzag type, one determines an m-value and vertically cuts the part of the sheet that contains the blue nodes with a higher a_1-value than m. Afterward, the sheet is rolled from the left to the right, leaving zigzag lines of C-atoms on both ends. The zigzag configuration is identified by the tuple $(m, 0)$

The procedure to create an armchair tube is similar. Considering the nodes on the magenta line, again, a value m has to be determined and the sheet is horizontally cut to remove the part that contains the magenta nodes with a_2-values higher than m. In that case, the sheet is rolled from bottom to top and the resulting armchair tube is identified by (m, m), since always $m = n$.

The last option, chiral, is indicated in green. Here, the rolling direction varies, depending on the chosen (m, n) values. Figure B.1 assumes $(m, n) = (5, 2)$. The sheet is rolled along the direction of the green arrow that is called the chiral vector. Cutting occurs in that case on a line perpendicular to the chiral vector through the point (m, n).

The chirality of the tube has some effect on the mechanical behavior of the tube (for a short summary see e.g. [123]) but especially determines if the resulting tube will be conducting or semiconducting [122]. Additionally, the aspect ratio of a SWCNT is defined as $\alpha = l/d$ with l giving the length and d the diameter, respectively.

C Background on the Simulation of Super Carbon Nanotubes

This section employs the common chirality-notation (m, n) and the aspect ratio α within tubes. Both topics are summarized in Appendix C.3.1.

Most studies about SCNTs are limited to tubes of order 1. Hence, unless noted otherwise SCNT stands for a super tube of order 1 in this overview. We start with the results obtained by the group of Coluci in Section C.1 since they proposed SCNTs in 2006. In Section C.2, the work of the group of Li is summarized. Section C.3 focuses on other studies that were mainly concerned with stretching and compressing SCNTs, followed by the consideration of the chirality dependence of the mechanical behavior in section C.3.1. Two crucial points during modeling SCNTs are the Y-junctions and the question how SCNTs of higher order should be treated because of their large size. Hence, section C.3.2 is dedicated to this topic.

C.1 The work of Coluci et al.

Coluci et al. [17], performed the first simulations on SCNTs with a focus on the electronic structure and properties but the publication already contains important information about the mechanical properties as well. In analogy to the chirality notation (m, n) of SWCNTs, they define the configuration of an SCNT by $[M, N]@(m, n)$ to express that the SCNT with chirality $[M, N]$ is formed by (m, n) SWCNTs as arm tubes. They employed the fast tight binding approach from Porezag [124] that is especially suitable for C-C, C-H and H-H bindings. The geometric positions of the nodes were attained by a universal force field including, among others, van der Waals forces. One result is that the energy per atom in SCNTs is higher than in corresponding SWCNTs, caused by the junction elements. SCNTs then should be even more stable than C_{60} fullerene, which can be thought of as a sphere out of graphene. Coluci et al. expected a high flexibility for bending and high tensile strength. For axial tension they predicted a different behavior compared to SWCNTs since the junctions would change their angles before the included SWCNT change their length.

Moreover, highly varying electronic properties are predicted. Depending on the parameters the resulting SCNTs show metallic or semiconducting behavior like it is the case for the SWCNTs that are employed as arm tubes. Both predictions were confirmed by later simulations. Finally, the authors state that the principle of SCNTs is not limited to carbon nanotubes but also is imaginable for other tube structures like boron nitride [32] or other connection types than Y-junctions like X- or T-junctions [33].

In 2007 Coluci et al. [20] presented an extended study on the mechanical properties of SCNTs. With fully atomistic simulations they derived properties as the Young's modulus and the tensile strength for some order 1 tubes. The test case consists of moving the atoms at the SCNT ends along the axis with a speed of $10\frac{m}{s}$. The interactions between atoms were modeled by an empirical bond-order potential from [125] targeted on hydrocarbon systems and thus applicable to graphite. Coluci et al. detected a fishing net like behavior of the SCNTs where, for tensile deformation, the stress is mainly concentrated on the junctions and the angles between the SWCNTs changes. Ruptures normally occur near the ends of junction arms before the SWCNTs are considerably stretched. Coluci et al. also state that the flexibility of the SCNTs can be increased with longer SWCNTs while a higher number of junctions increases the stiffness.

The authors conclude that SCNTs are good candidates for new flexible, high-tensile materials and tough, stiff super-composites.

C.2 The work of Li et al.

In 2008, Li et al. ([18], [30]) published two further studies about the mechanical behavior of SCNT and adapted the $[M,N]@(m,n)$ notation of [17] to denote their configuration. They define the aspect ratio of SCNTs as $\tilde{\alpha} = L/D$ where L and D denote length and diameter of the SCNT and α the aspect ratio of the arm tubes. They generate their models of SCNTs in MATLAB and execute the actual simulation in the ANSYS[1] software. In contrast to Coluci et al., they do not employ molecular dynamics for their simulations but use the molecular structure mechanics method for carbon nanotubes proposed in [126]. The main results of Li et al. are that the in-plane stiffness of the tubes is almost the same as for the unrolled super graphene and that it decreases if α is decreased. In addition, it also only depends on the aspect ratio α. In contrast, Poisson's ratio, which describes the change in the diameter of the tube when it is stretched, is dependent on the value of α and, to some extent, on the chirality. In general, armchair SCNTs have higher Poisson's ratio values than zigzag SCNTs. The higher Poisson's ratio of SCNTs demonstrates their high flexibility and means that SCNTs can be further stretched and compressed than SWCNTs. Finally, the bending rigidity in the tested SCNTs is up to 20 times higher than those of SWCNTs. The authors conclude that the deformation of SCNTs results from a combination of stretching and bending the arm tubes while the Y-junctions coupling stretching and bending. The authors also highlight that SCNTs have a very low mass density, which is only 1 % of graphite and 0.3 % of steel [18]).

C.3 General super carbon nanotube simulations

The conclusions about the behavior of Y-junctions from Coluci and Li are confirmed by three other studies. Wang et al. [28] conducted an analysis based on the finite element method and the thin-shell model. They summarize the behavior of SCNTs under tensional load as follows:

[1] `http://www.ansys.com/`

At the beginning the stiffness is low because only the angles in the Y-junctions change. With increasing load, the arms begin to stretch, resulting in high stiffness. Wang et al. [28] also emphasize that the SCNTs might overcome a large drawback of SWCNTs: The hierarchical construction process results in structures of the macro scale and are not limited to nanoscale, facilitating their usage in material design.

Qin et al. [29] simulated SCNTs under uniaxial tension with the left boundary being fixed against movements in the axial direction and the force being applied to the right boundary. They identify three stages in the deformation process: First, the arm-tubes align themselves into the direction of the tension. Second, the SWCNTs rotate around the centers of the junctions to be parallel to the axial direction, reducing the circumference of the tube. Last, the arm-tubes get stretched resulting in the fact that different forces are required to achieve a deformation (see also [28]). This is the reason for the high flexibility of SCNTs and their different behavior compared to the SWCNTs. In particular, the rupture point stays constant even under 1200 K, while, in that case, SWCNTs rupture earlier. In general, the properties of SCNT that were investigated by Qin et al. [29] are insensitive to the temperature.

Chen et al. [31] presented a study about the nonlinear deformation processes of armchair SCNTs. Their modeling approach employed has the most commonalities in its methodology with the work in this thesis. First of all, for modeling SCNTs they do not distinguish Y-junction and arm tube objects, but only employ Y-junctions whose arm length is half the length of fictive arm tubes. This is also the underlying SCNT construction in this thesis, but here it is extended to higher orders. Another commonality is that they employ the AFEM method of [35] and not molecular dynamics or molecular mechanics for the simulation. The tests reveal that there are, in principle, two different deformation processes for uniaxial tension whose occurrence is strongly dependent on the aspect ratio of the arm tubes. For lower ratios, i.e., $\alpha < 6$, the deformation consists of two different phases (rotation & stretch as unity and rupture), while there are three phases (separated rotations and stretch, rupture) for aspect ratio for $\alpha \geq 6$.

C.3.1 Dependence on chirality

The chirality dependence of the mechanical properties has only been investigated rudimentarily. The simulations of Li et al. [127] demonstrated that the critical compression force and the critical bending force are independent of the chirality of the SCNT. Additionally, the results [18] showed that the in-plane stiffness is independent of the chirality of the arm tubes and of the SCNT, but that in contrast the Poisson's ratio is slightly dependent on the chirality. Qin et al [29] only noticed a weak dependence of the choice of the (M, N, m, n) parameters and the behavior under uniaxial tension.

In general, the situation is similar to the SWCNTs where the chirality has some influence on some parameters, but that the electronic behavior is much more dominated by the chirality. This conclusion is supported by studies of Romo-Herrera et al. [128] who investigated four 2D/3D networks of SWCNT networks: super square, super graphene (both 2D), and super cubic, super diamond (both 3D). Their summary is that the mechanical behavior of the networks is not dependent on the chirality of the employed SWCNTs but only on the type of network. Following Wang et al. [28], it is possible to predict the behavior of SCNTs with super graphene, like it is possible in the order 0 case with graphene and SWCNTs. After their literature review Yin et al. also state that *»the effect of chirality on the physical and mechanical properties of STs [SCNTs] can be neglected* «[42, p. 1328].

C.3.2 Modeling of Y-junctions and scaling laws

Beside the actual simulation there was some theoretical work on SCNTs and their geometry, which is especially important for the creation reliable models.

Lászlò [129] presented a way to create junctions to connect tubes of arbitrary chirality based on Euler's theorem that relates the number of faces, edges and vertices in a polyhedron. They are able to derive the ideal number of defects in the grid (e.g. heptagons, octagons) if only one type of defect is allowed for the junctions.

Yin et al. [51] developed geometric conservation laws for Y-junctions. Those are perfect, meaning that there are no defects (dislocations, misconnected bonds). The junctions are in equilibrium state, have minimal energy (locally and globally), and are symmetric if the angle between arms is 120° as well as all arms have the same radius. The authors state that spontaneous branched tubes are symmetric but not controllable, while forced, templated branching is controllable but not symmetric. They propose a periodic nanochannel template of hexagonal cells for the growth of super sheets. To overcome the issues of rolling up the planar super sheet and to connect it to a seamless cylinder, they propose the idea of a circular cylindrical template.

In later work, Yin et al. ([41], Yin2010) investigated geometric conditions for designing fractal SCNTs with armchair chirality and containing armchair junctions on the one side and zigzag SCNTs with zigzag junctions on the other. They identify two types of self-similarity: Structural and geometric. Structural self-similarity means that the shapes of the different levels are equal. This property is naturally given, since each level has the shape of the tube. Geometric self-similarity means that the geometric size of neighboring levels need to be identical and this is the prerequisite to construct a fractal. The authors identify three conditions for the presence of geometric self-similarity: The number of Y-junctions along the circumference and along the axial direction of each level need to be the same. Additionally, the tube of level i needs to be as long as one side of a hexagon in a tube of level $i + 1$. Yin et al. are not sure whether it will to possible to really construct fractal SCNTs in the future but they highlight the knowledge that can be gained from their theory.

Exploiting the fractal properties of SCNTs, Pugno [40] presented scaling laws in 2006 to estimate the mechanical parameters for larger SCNTs of higher order. In this way, Pugno estimated the behavior of SCNTs up to order 20 with a radius of more than 1 cm and found that the maximum for simultaneously optimizing strength, stiffness, and toughness is reached for order 2 tubes.

D Performance Comparison of Different Container Structures

In Section 6.1.3, the issue of employing vectors data types as keys for map structures arose. Here, we report on studies with different map types, compilers and varying the data types which serve as key to the maps between vectors and scalars.

To evaluate the data structures we developed, several experiments on basic containers were performed to test their suitability for the different scenarios. The main question was how to choose the data type for the key in the maps appropriately. The implementation of [62] suggests to use the tuples, i.e., vector datatypes, as keys for the offset table and the hash table. Thus, for one half of the test maps with std::vectors as keys are used (vector maps), while the other maps employ integers as keys which are calculated from the tuples-vectors before access (vector-integer maps).

The tested containers are the (ordered) map and the unordered_map, one time from the std library and a second time from the boost library.

The tests were performed on two different test systems, the Linux-based Lichtenberg Rechner (see 9.1.1) and a Windows 10 based desktop system with a Xeon 1230v2 (Ivybridge) CPU. For Linux the g++ compiler (version 4.8.1) and the Intel compiler [ICC] (version 17) were tested, while for Windows the Microsoft MSVC [VCC] (version 14.0) and again the Intel icpc compiler (version 17) were used.

A small test program was employed to measure the speed for insertion and of access to elements in the maps. The test program always compares a pair of maps considering their access time. Pair means, keeping the primitive map-type constant and using a vector as key for one instance, while using an integer for the other. The four compared pairs of maps:

1. std::map<**int**, **int**> and std::map<std::vector<**short**>, **int**>
2. std::unordered_map<**int**, **int**> and std::unordered_map<std::vector<**short**>, **int**>
3. boost::containers::map<**int**, **int**> and boost::containers::map<std::vector<**short**>, **int**>
4. boost::unordered_map<**int**, **int**> and boost::unordered_map<std::vector<**short**>, **int**>

Afterward, both maps are filled with nodes. To that end, the test code receives the length of the tuples that should be used and their extent. It iterates over all possible tuples and decides for every tuple if it should be part of the node set. The algorithm is configured to always create a tuple set of about 10 million entries, independent of the chosen dimension. By adaption of the extent, the program achieves sparsely occupied maps as they result from the SCNT construction process.

After the data is initialized, two random query vectors are created that determine the order of the map accesses. To that end, a predefined number of accesses is generated and inserted into a std::vector<vector<**short**>> which is consecutively processed later. To guarantee comparable access schemes for both maps to test, the corresponding serial index for each entry of the query vector is calculated and inserted into a std::vector<**int**> in the same order. Hence, the distance between the elements to access for both maps is equal.

Finally, for both maps and their corresponding vectors, the time to access all the elements consecutively is measured by iterating of the vectors. The results are shown in Table D.1 where 10^6 tuples of length 28 were searched with an extent of 4^{28}.

Several things can be observed: On the Windows machine, independent of the employed container structure and compiler, the vector-integer map is considerably faster than the vector map although the serializing time of the vector is included in the measurements. Another point is that the unordered maps enable a much faster access than their ordered pendants. The compiler seems not to create significant differences. Consequently, for Windows the combination of the vector integer based unordered map is clearly the best choice.

On Linux the situation is different. Although the hardware employed is even faster than on the Windows computer, the performance of ordered and unordered vector-integer maps is drastically lower and in some cases even lower than corresponding vector maps. This behavior could also be replicated on another Linux computer with comparable hardware to the Windows machine. Hence, the reason for the slow vector-integer maps seem to lie in their implementation in the std library which is also used by boost in most parts. But on Linux also the unordered maps are considerably faster than the ordered ones. That is why they should be preferred for this kind of application.

Table D.1.: 10^6 random entries were searched in different map structures. The upper half of the table shows the overall search-times on the Windows system with compilers Visual Studio C++ (VCC) and IntelC++ (ICC), the lower part those of the Linux system with compilers g++ and and again Intel C++. The runtimes are given in seconds.

	VCC_STD_ORDERED_28			VCC_STD_UNORDERED_28		
vecMap:	4.759	4.764	3.672	0.504	0.496	0.503
vecintMap:	2.635	2.890	1.720	0.339	0.333	0.339
Quotient:	1.81	1.65	2.13	1.49	1.49	1.49
	VCC_BOOST_ORDERED_28			**VCC_BOOST_UNORDERED_28**		
vecMap:	3.528	3.784	3.535	0.472	0.470	3.784
vecintMap:	1.256	1.247	1.184	0.320	0.320	1.247
Quotient:	2.81	3.04	2.99	1.47	1.47	3.04
	ICC_STD_ORDERED_28			**ICC_STD_UNORDERED_28**		
vecMap:	4.033	4.201	3.934	0.483	0.514	0.506
vecintMap:	2.333	2.172	1.966	0.331	0.336	0.331
Quotient:	1.73	1.93	2.00	1.46	1.53	1.53
	ICC_BOOST_ORDERED_28			**_ICC_BOOST_UNORDERED_28**		
vecMap:	3.724	3.523	3.929	0.461	0.465	0.458
vecintMap:	1.343	1.192	1.261	0.321	0.316	0.316
Quotient:	2.77	2.96	3.12	1.43	1.47	1.45

	g++_STD_ORDERED_28			g++_STD_UNORDERED_28		
vecMap:	3.489	3.403	3.539	0.535	0.539	0.575
vecintMap:	2.418	2.300	2.822	0.484	0.483	0.512
Quotient:	1.44	1.48	1.25	1.11	1.12	1.12
	g++_BOOST_ORDERED_28			**g++_BOOST_UNORDERED_28**		
vecMap:	3.843	3.684	3.637	0.624	0.592	0.622
vecintMap:	4.312	3.779	3.850	0.500	0.483	0.493
Quotient:	0.89	0.98	0.94	1.25	1.22	1.26
	ICC_STD_ORDERED_28			**ICC_STD_UNORDERED_28**		
vecMap:	3.367	3.561	3.501	0.586	0.545	0.595
vecintMap:	2.501	2.667	2.587	0.512	0.487	0.533
Quotient:	1.35	1.34	1.35	1.14	1.12	1.12
	ICC_BOOST_ORDERED_28			**ICC_BOOST_UNORDERED_28**		
vecMap:	3.658	3.818	3.947	0.601	0.590	0.596
vecintMap:	4.156	3.694	3.646	0.478	0.468	0.472
Quotient:	0.88	1.03	1.08	1.26	1.26	1.26

E Heap Consumption of the new EdgeMap

This appendix provides more details about the comparison of the old bimap based **EdgeMap** and the structure-related **EdgeMap** based on EdgeEntries that was presented in section 6.1.4. The first part is the investigation of the memory demand during the actual graph construction process. The example is done with the $(1,4,8,8)^2$ tube, like in section 6.1.4. In Figure E.1 we see a screenshot of the massif-visualizer [1] tool that automatically creates plots analyzing the total heap-memory consumption of the graph construction process on the base of massif profile files. The y-axis gives the heap consumption of the total program in kilobytes and the areas under the differently colored lines visualize the contributions of different data structures. Please note, that due to a problem of the massif-visualizer the lines for memory consumption are drawn with somewhat higher y-values as they as they are reported in profile file. The values used in the following text are taken from the massif output file and the textual output of massif-visualizer. The x-axis depicts the time steps from program start to the finishing of the graph construction phase, thus the right side shows the memory that is required to keep the final tube which are about 1.6 gigabytes in this case. The large red part of the area marked reflects the memory for the edge map itself and the underlying boost::bimap. With over 1 gigabytes it consumes nearly two third of the overall space.

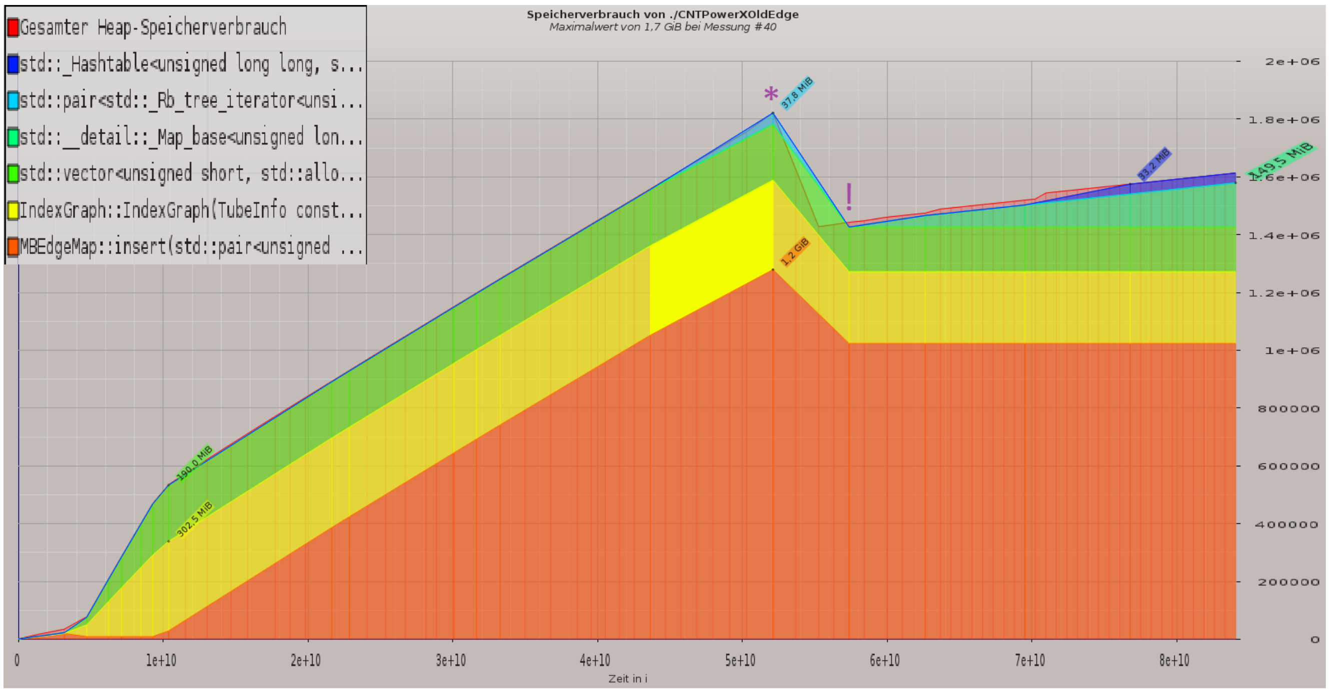


Figure E.1.: Edited screenshot from the heap analysis with the tool massif for the construction of the $(1,4,8,8)^2$ tube and the old edge map.

[1] http://milianw.de/tag/massif-visualizer

Also two other things can be noticed in this plot. The first thing is that the peak memory consumption occurs during the construction process at the point marked by the purple *. There, the uncut graph is constructed that contains several nodes that will later be removed. In that case, the **EdgeMap** requires more than 1.2 GB. The decrease of the curve marks the begin of the cutting operation which particularly reduces the memory for the edges from 1.2 to 1.0 GB, i.e., down to 83 % of the peak value. The yellow and dark green areas depict the memory for the tuple- and index-information. Together they are reduced to 80 % of the peak value (492 to 393 MB) during the cutting, which corresponds nearly exactly to the reduction of nodes in the model (from 4956160 to 3964928, i.e., 80 %). The two lines in light green and purple at the top of the diagram correspond to the position data of the point. The geometry construction starts after the end of the cutting process (marked by an **!**) and is done in the last part of the construction. The second thing to note is the additional memory demand caused by the composition graph algebra operation which only is required during the construction of the uncut tube. It is the cyan line in the diagram.

Figure E.2 shows the heap consumption of the new edge storing scheme for comparison. On the right side the overall allocated memory is below 800 megabytes and hence only half of the old scheme. The colors of the lines now belong in some cases to different data structures. The red and dark orange (number 1 and 2) this time represent space for the node storage which is, like for the measurement with the old **EdgeMap**, 393 MB. The contribution of the **EdgeMap** itself is split in that case in the light yellow (3), dark yellow (4), green (5) and blue (6) lines which together consume 190 MB of the heap. The main part of these contributions are required by the **EdgeEntry** objects which are 136 MB (numbers 3 and 4), followed by the structural information of the internal hash map (33 MB, number 5) and the additional space for reallocating additional edge pointers in the case that an **EdgeEntry** has more than two of those (18 MB, number 6). The light orange (7) and cyan (8) curves belong two the position data.

In contrast to the bimap approach, the memory consumption for the uncut tube without geometry (*) is about the same as for the final tube with calculated positions. The difference is caused by the new **EdgeEntry**. Not only its total size is smaller but also the slope of the curve is lower. This results from two facts: The overhead for the structure itself is lower, especially for a large number of edges, and the exploitation of the fact that each edge has an opposite edge in the final graph comes into play.

After the analysis of the memory behavior, Table E.1 opposes the runtime of the construction for both **EdgeMap** types for various tubes. In most cases, for order 2 and 3 the speedup is about a factor of 1.9. For the lower order tubes, the speedup with the new map is even higher. The reason is that in those cases only junctions of level 0 need to be constructed which is a very time-consuming operation for higher levels. Hence, the main part of the runtime is spent on insert operations of edges during the final tube construction. The performance of insertions

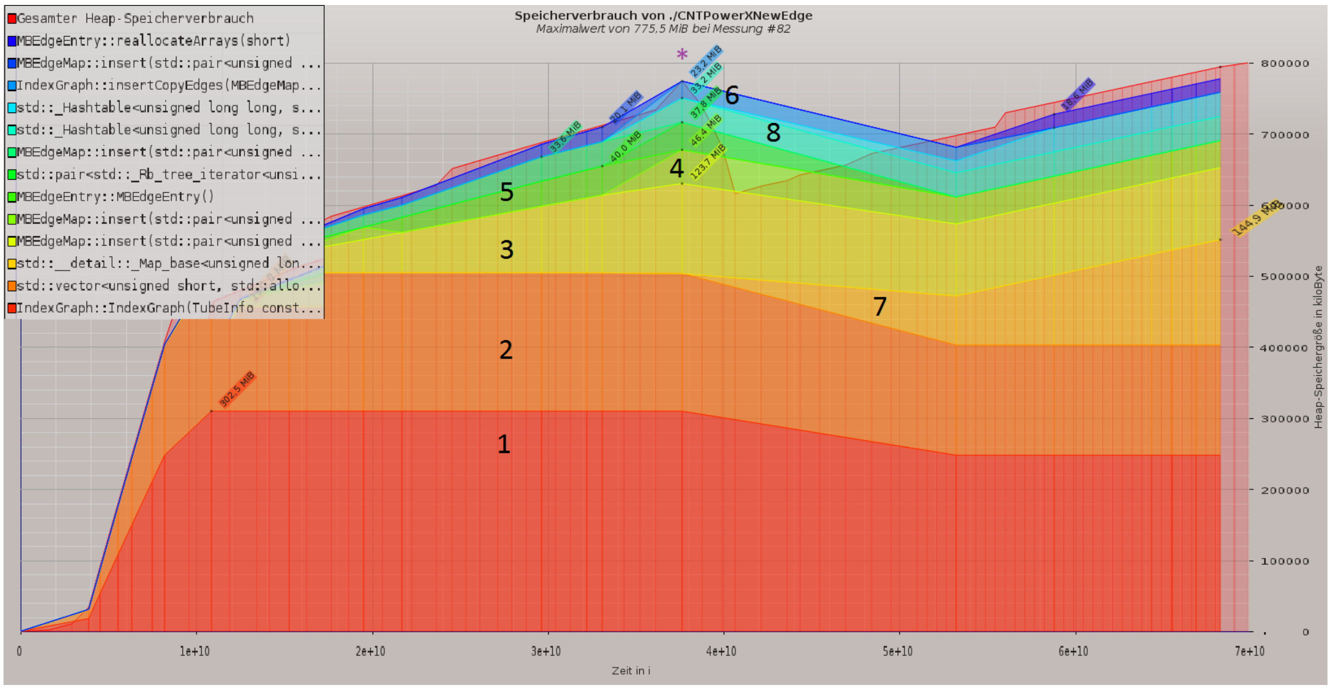


Figure E.2.: Edited screenshot from the heap analysis with the tool massif for the construction of the $(1,4,8,8)^2$ tube and the new edge map.

in the new **EdgeMap** is superior to that of the old version, resulting in the significant overall speedup.

Table E.1.: Comparison of the runtime for the construction of various tubes for both **EdgeMap** variants. The first two columns show the tube configuration, the next two the runtime in seconds, and the last row the fraction of the runtime for the old and new **EdgeMap**.

Tube config	No. of nodes	Time in s (old)	Time in s (new)	Factor
$(6,10,64,64)^1$	$1.3*10^7$	190	47	4.0
$(4,8,128,128)^1$	$3.2*10^7$	404	105	3.8
$(1,2,4,4)^2$	$3.0*10^5$	2.2	1.4	1.6
$(1,4,8,8)^2$	$4.0*10^6)$	48	26	1.9
$(2,4,8,8)^2$	$9.0*10^6)$	120	61	2.0
$(4,8,8,16)^2$	$2.4*10^8)$	4404	2187	2.0
$(1,2,4,4)^3$	$2.5*10^7$	321	174	1.8
$(1,4,4,4)^3$	$1.7*10^8$	4447	2373	1.9
$(1,4,4,6)^3$	$2.6*10^8$	6344	2995	2.1
$(1,4,6,6)^3$	$3.9*10^8$	9209	4917	1.9

F Performance Results

F.1 On-the-fly versus reference solver

Table F.1.: Comparison of the on-the-fly and reference solver on phase one nodes, grouped by the tube configuration. Samples are grouped by the no. of threads per row. Runtime in ms. μ and σ give median and standard deviation for samples with same solver, tube configuration and no. of threads.

		On-the-fly Solver					**Reference Solver**										
No of Sample		**1**	**2**	**3**	μ	σ	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	μ	σ
(256, 256)	1T	8176.06	8179.14	8176.30	8177.17	1.40	109.24	109.43	109.77	109.35	109.39	109.46	109.39	109.29	109.49	109.42	0.14
	2T	4086.03	4097.17	4096.49	4093.23	5.10	55.51	56.09	56.03	55.59	56.27	55.47	56.06	55.48	56.07	55.84	0.30
	4T	2102.54	2087.74	2062.58	2084.29	16.50	29.42	29.61	29.42	29.41	29.45	29.46	30.00	29.64	29.21	29.51	0.21
	8T	1071.10	1089.90	1054.36	563.07	0.08	15.76	16.20	15.71	15.80	16.56	15.71	15.77	15.82	16.51	15.98	0.33
	16T	563.03	563.18	562.99	1071.79	14.52	8.90	8.94	8.90	8.95	8.84	8.96	8.89	8.95	8.87	8.91	0.04
(512, 512)	1T	35528.60	33453.10	33462.30	34148.00	976.24	439.02	438.95	439.13	439.20	439.16	438.98	439.22	439.01	439.14	439.09	0.10
	2T	16570.80	16567.30	16599.50	16579.20	14.43	225.28	224.09	224.10	223.69	223.67	224.11	223.75	224.19	223.15	224.00	0.55
	4T	8319.72	8515.64	8525.84	8453.73	94.85	117.01	121.21	121.41	117.10	117.29	121.24	120.06	119.46	117.61	119.15	1.80
	8T	4351.29	4417.03	4328.06	4365.46	37.68	65.15	63.30	65.38	67.35	65.38	63.30	66.87	62.74	64.07	64.84	1.52
	16T	2264.28	2265.07	2264.92	2264.76	0.34	36.83	37.31	37.15	37.24	36.95	38.45	37.47	37.88	37.14	37.38	0.48
(1,4, 8,14)	1T	2452.14	2452.68	2454.75	2453.19	1.12	34.06	34.07	34.09	34.05	34.06	34.08	35.05	35.08	35.14	34.41	0.48
	2T	1231.66	1234.06	1234.78	1233.50	1.33	17.58	17.55	17.53	17.54	17.54	17.53	17.55	17.35	17.54	17.52	0.06
	4T	618.02	635.01	633.19	628.74	7.62	9.19	9.16	9.13	9.14	9.12	9.15	9.12	9.14	9.27	9.16	0.05
	8T	319.53	327.60	337.37	328.16	7.29	5.08	4.91	4.92	4.95	5.06	5.11	4.91	5.15	4.89	5.00	0.10
	16T	171.32	172.23	171.26	171.60	0.44	2.79	2.81	2.82	2.87	2.81	2.84	2.98	2.83	2.83	2.84	0.05
(1,4,8, 355)	1T	64801.20	64815.10	64695.90	64770.73	53.22	883.19	882.42	882.43	882.89	882.74	882.43	882.61	883.08	882.19	882.66	0.32
	2T	32180.60	32104.70	32201.60	32162.30	41.62	448.59	449.51	448.94	454.48	449.37	451.58	450.90	450.71	451.21	450.59	1.70
	4T	16202.40	15929.30	16020.20	16050.63	113.55	241.08	241.55	237.37	236.74	241.25	241.20	242.16	235.12	237.50	239.33	2.47
	8T	8405.89	8324.38	8349.08	8359.78	34.13	139.72	136.32	132.06	131.19	131.11	127.34	136.94	132.13	133.70	133.39	3.52
	16T	4353.95	4354.89	4355.41	4354.75	0.60	74.86	75.09	75.04	75.04	73.83	74.53	75.64	73.79	74.92	74.75	0.57
(2,6,12,116)	1T	64955.60	65037.70	65058.00	65017.10	44.27	883.02	882.72	883.20	882.93	883.54	883.52	882.73	883.26	884.40	883.26	0.49
	2T	32154.00	32302.30	32223.50	32226.60	60.58	451.85	450.93	449.17	449.45	450.16	449.27	450.63	449.63	450.73	450.20	0.85
	4T	16384.50	16383.60	16391.00	16386.37	3.30	241.89	237.50	244.30	242.38	241.76	242.70	234.32	235.77	233.41	239.34	3.86
	8T	8196.26	8407.21	8629.47	8410.98	176.88	130.71	135.88	131.95	131.47	131.04	130.36	128.24	131.58	129.61	131.20	1.97
	16T	4367.19	4367.19	4368.23	4367.54	0.49	74.27	76.30	76.14	74.03	75.64	73.58	73.61	73.80	75.05	74.71	1.03

Table F.2.: Comparison of the on-the-fly and reference solver on phase two hardware, grouped by the tube configuration. The runtime of samples is given in milliseconds. They are grouped by the number of employed threads for the runs per row. μ and σ give median and standard deviation for all samples with same solver, tube configuration and number of threads.

		On-the-fly Solver					Reference Solver				
No of Sample		**1**	**2**	**3**	μ	σ	**1**	**2**	**3**	μ	σ
(256,256)	1T	6679.72	6681.57	6687.09	6682.79	3.13	105.59	106.10	106.09	105.93	0.24
	2T	3353.23	3351.39	3356.90	3353.84	2.29	54.09	53.83	54.34	54.08	0.20
	4T	1758.63	1726.95	1759.84	1748.47	15.23	30.05	30.32	30.18	30.18	0.11
	8T	926.59	910.18	925.34	920.70	7.46	16.65	16.63	16.71	16.66	0.03
	16T	464.94	465.60	464.72	465.08	0.37	9.26	9.21	9.30	9.26	0.04
	24T	310.92	311.05	310.84	310.94	0.09	6.94	6.77	6.84	6.85	0.07
(512,512)	1T	27177.30	27196.80	27199.00	27191.03	9.75	424.82	426.20	425.54	425.52	0.56
	2T	13559.00	13575.80	13659.60	13598.13	44.00	217.23	217.80	217.65	217.56	0.24
	4T	7037.22	6947.96	7179.09	7054.76	95.17	123.53	119.04	121.17	121.25	1.84
	8T	3682.08	3705.94	3695.10	3694.37	9.75	68.67	66.66	68.68	68.00	0.95
	16T	1869.92	1868.83	1870.50	1869.75	0.69	36.58	37.68	36.44	36.90	0.56
	24T	1248.87	1248.62	1248.37	1248.62	0.20	26.44	27.24	27.04	26.90	0.34
(1,4,8,14)	1T	1981.61	1984.18	1985.99	1983.93	1.80	32.94	32.98	32.91	32.94	0.03
	2T	997.70	995.33	995.53	996.18	1.07	16.81	16.98	17.05	16.95	0.10
	4T	521.91	531.49	532.49	528.63	4.77	9.31	9.44	9.37	9.37	0.05
	8T	271.60	271.60	271.60	271.60	0.00	5.19	5.19	5.26	5.21	0.03
	16T	140.27	140.09	140.15	140.17	0.07	2.82	2.80	2.79	2.80	0.01
	24T	95.76	94.13	95.89	95.26	0.80	2.04	2.04	2.02	2.03	0.01
(1,4,8,355)	1T	52478.00	52392.10	52409.70	52426.60	37.05	855.17	853.84	853.78	854.26	0.64
	2T	25846.70	25965.30	25987.50	25933.17	61.81	436.53	438.02	437.11	437.22	0.61
	4T	13123.80	13066.20	13193.70	13127.90	52.13	230.15	234.32	235.00	233.16	2.14
	8T	7024.52	7080.62	7030.21	7045.12	25.21	135.08	132.61	131.81	133.17	1.39
	16T	3560.49	3565.13	3569.48	3565.03	3.67	71.52	72.04	71.63	71.73	0.22
	24T	2379.06	2380.13	2379.35	2379.51	0.45	53.38	53.54	52.97	53.30	0.24
(2,6,12,116)	1T	52587.50	52610.80	52640.00	52612.77	21.48	855.17	853.84	853.78	854.26	0.64
	2T	25987.50	25977.60	26042.30	26002.47	28.45	431.84	437.95	436.43	435.41	2.60
	4T	13121.20	14154.40	13030.40	13435.33	509.81	240.35	239.54	264.30	248.06	11.49
	8T	7034.59	7056.89	7060.43	7050.64	11.44	132.17	131.99	132.43	132.20	0.18
	16T	3574.43	3575.09	3579.39	3576.30	2.20	71.17	72.15	71.80	71.71	0.40
	24T	2388.26	2387.49	2387.48	2387.74	0.37	51.61	52.45	52.61	52.22	0.44

Table F.3.: Comparison of the runtimes of the old solver version presented in [23] with the recent version. The performance values for the old version are taken from the database of [23], while the values for the new solvers result from the average values for the respective columns from Appendix F.1. The total factor for the runtime difference in the last column is the product of the relative performance difference of the old and new on-the-fly (Diff. Matrix-free Old/New) solver, that of the old and new reference-solver (Diff. Reference Old/New), and the performance increase of the reference solver when employing 16 instead of 1 thread (Scaling Reference). The slowdown particularly results from the improvements of the new reference solver.

(64,64)	**Old Version**		**New Version**		**Diff. Matrix-free Old/New**	**Diff. Reference Old/New**	**Scaling Reference**	**Total Difference**
	Matrix free	**Reference**	**Matrix free**	**Reference**				
	Per Step (ms)	**Per Step (ms)**	**Per Step (ms)**	**Per Step (ms)**				
1T		2.89	430.86	6.34	0.92	0.46	7.23	3.0
16T	33.37		30.71	0.88				
(128, 128)	**Old Version**		**New Version**		**Diff. Matrix-free Old/New**	**Diff. Reference Old/New**	**Scaling Reference**	**Total Difference**
	Matrix free	**Reference**	**Matrix free**	**Reference**				
	Per Step (ms)	**Per Step (ms)**	**Per Step (ms)**	**Per Step (ms)**				
1T		12.86	1702.90	25.46	0.91	0.50	10.69	4.9
16T	133.40		121.15	2.38				
(256, 256)	**Old Version**		**New Version**		**Diff. Matrix-free Old/New**	**Diff. Reference Old/New**	**Scaling Reference**	**Total Difference**
	Matrix free	**Reference**	**Matrix free**	**Reference**				
	Per Step (ms)	**Per Step (ms)**	**Per Step (ms)**	**Per Step (ms)**				
1T		51.84	6842.92	107.91	0.92	0.48	10.50	4.7
16T	533.80		492.92	10.28				
(256, 512)	**Old Version**		**New Version**		**Diff. Matrix-free Old/New**	**Diff. Reference Old/New**	**Scaling Reference**	**Total Difference**
	Matrix free	**Reference**	**Matrix free**	**Reference**				
	Per Step (ms)	**Per Step (ms)**	**Per Step (ms)**	**Per Step (ms)**				
1T		105.23	13832.40	215.04	0.90	0.49	10.62	4.7
16T	1087.32		979.74	20.25				

F.2 Symmetry solver versus reference solver

Table F.4.: Comparison of the value-symmetric and reference solver on phase one hardware, grouped by the tube configuration. The runtime of samples is given in ms. They are grouped by the number of threads employed for the runs per row. μ and σ give median and standard deviation for all samples with same solver, tube configuration and number of threads.

		Value-symmetric Solver											**Reference Solver**										
No of Sample		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	μ	σ	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	μ	σ
(512, 512)	1T	184.47	184.89	183.84	190.12	192.62	184.94	191.05	185.95	192.00	187.76	3.39	438.98	439.00	438.97	439.10	438.84	439.06	438.96	439.29	439.32	439.06	0.15
	2T	130.39	147.08	106.48	137.76	141.35	141.73	110.23	111.52	112.38	126.55	15.30	223.92	222.99	224.24	223.08	223.39	223.59	223.70	223.76	224.14	223.65	0.41
	4T	80.85	83.03	82.25	69.40	85.20	81.18	64.25	82.45	81.14	78.86	6.66	121.23	118.84	118.60	120.73	120.36	120.82	121.04	120.70	120.49	120.31	0.89
	8T	58.05	59.17	57.21	56.62	59.51	58.38	58.04	58.50	58.20	58.19	0.84	66.93	69.71	65.29	65.54	65.66	63.83	65.46	65.13	64.22	65.75	1.63
	16T	48.95	50.71	54.51	49.58	52.80	51.22	50.46	54.75	53.15	51.79	1.98	38.58	37.23	37.90	37.42	38.09	36.40	38.22	37.21	36.21	37.47	0.76
(1024, 1024)	1T	836.76	820.43	814.39	816.07	841.83	827.59	829.57	828.21	777.03	823.86	17.80	1751.03	1760.56	1772.58	1753.12	1750.57	1754.04	1753.23	1751.43	1750.84	1761.39	6.77
	2T	473.26	459.21	470.08	451.73	564.91	474.26	509.04	493.39	485.84	486.86	32.06	898.15	897.99	899.68	896.77	899.68	895.90	901.39	897.90	897.06	898.28	1.61
	4T	270.46	317.68	294.00	344.60	264.08	318.06	353.43	358.51	276.68	310.83	34.31	489.88	480.18	485.42	488.49	487.64	488.94	479.55	479.59	488.28	485.33	4.09
	8T	209.82	249.05	252.92	259.77	255.83	250.25	251.90	246.24	249.78	247.28	13.76	261.68	269.14	289.26	288.05	265.25	269.19	269.36	260.48	274.97	271.93	9.84
	16T	214.17	212.76	209.08	219.97	215.35	220.38	214.92	212.20	217.66	215.16	3.48	160.07	165.21	160.25	154.94	158.08	155.01	157.29	154.30	156.33	157.94	3.28
(1024, 2048)	1T	2291.51	2294.33	2268.60	2278.98	2281.95	2267.24	2282.19	2277.65	2278.09	2280.06	8.49	3595.75	3593.33	3482.57	3482.22	3481.22	3516.03	3479.71	3486.76	3518.61	3515.13	44.70
	2T	1145.34	1223.70	1149.66	1153.10	1193.10	1141.04	1206.12	1148.13	1141.70	1166.88	29.92	1773.79	1801.13	1783.93	1776.16	1788.19	1782.04	1785.28	1792.13	1781.00	1784.85	7.82
	4T	657.67	623.65	659.95	828.06	675.31	650.92	837.52	701.77	809.86	716.08	79.79	972.94	969.79	970.38	950.01	955.45	963.75	963.70	948.72	962.96	961.96	8.30
	8T	430.02	516.16	556.50	530.99	384.21	545.44	421.41	532.64	544.26	495.74	61.30	544.84	522.68	539.00	549.39	568.83	528.95	528.40	532.07	544.74	539.88	13.27
	16T	438.52	434.11	434.27	436.00	447.26	434.47	440.88	441.05	435.78	438.04	4.14	316.45	316.20	306.12	306.31	307.02	306.86	308.67	307.13	305.82	308.95	4.01
(2048, 2048)	1T	4730.38	4712.60	4670.67	4690.30	4706.51	4719.96	4658.29	4688.34	4677.27	4694.92	22.70	6980.50	7112.03	7034.84	7068.06	7135.98	6971.04	7222.66	7055.07	7052.61	7070.31	73.78
	2T	2352.18	2349.80	2388.33	2349.19	2361.28	2347.83	2360.87	2341.47	2348.90	2355.54	13.01	3570.04	3589.23	3554.24	3726.73	3556.85	3659.21	3567.72	3560.31	3752.28	3615.18	73.23
	4T	1414.31	1321.24	1368.40	1299.83	1251.40	1354.58	1321.45	1323.82	1374.78	1336.65	44.81	1975.02	1973.11	1924.34	1906.88	1953.59	1970.60	1983.52	1936.83	1878.28	1944.69	33.83
	8T	1163.29	1148.62	1156.58	1093.97	1175.98	1013.35	888.99	1145.44	1086.99	1097.02	87.92	1152.66	1044.70	1092.49	1049.74	1037.04	1069.28	1091.23	1145.73	1073.69	1084.06	39.29
	16T	874.75	871.09	885.35	871.48	873.59	887.80	891.00	879.50	863.99	877.62	8.41	619.60	617.65	617.16	621.50	617.21	618.96	618.31	615.98	617.61	618.22	1.53
(256, 256)	1T	45.42	44.50	44.82	44.85	44.54	44.79				44.82	0.30	112.60	112.53	112.60	112.60	112.56	112.67				112.59	0.04
	16T	13.14	13.14	13.22	13.63	12.77	12.95				13.14	0.26	9.31	9.33	9.34	9.22	9.29	9.29				9.30	0.04

F.3 Caching solver versus reference solver

Table F.5.: Comparison of the caching and reference solver on phase one hardware, grouped by the tube configuration. The runtime of samples is given in ms. They are grouped by the number of threads employed for the runs per row. μ and σ give median and standard deviation for all samples with same solver, tube configuration and number of threads.

		Caching Solver											**Reference Solver**										
No of Sample		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	μ	σ	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	μ	σ
(256, 256)	1T	42.90	42.3508	44.5601	42.4328	42.2957	42.2102	43.238	42.618	44.4851	43.01	0.86	109.24	109.427	109.77	109.349	109.391	109.458	109.391	109.286	109.487	109.42	0.14
	2T	23.37	23.1012	23.1893	23.1156	24.1559	23.3054	23.3826	23.2244	24.076	23.44	0.38	55.51	56.0874	56.0346	55.5923	56.267	55.4706	56.0577	55.4805	56.0655	55.84	0.30
	4T	12.62	12.6678	12.6823	12.6045	12.5788	12.8657	12.8041	12.7662	12.4803	12.67	0.11	29.42	29.61	29.4184	29.4124	29.4472	29.4564	30.0036	29.6379	29.2082	29.51	0.21
	8T	7.44	7.51832	7.43409	7.41573	7.75338	7.55134	7.57891	7.54115	7.53096	7.53	0.10	15.76	16.2016	15.7141	15.7954	16.5554	15.7123	15.7703	15.8161	16.5141	15.98	0.33
	16T	4.87	4.9068	4.86793	4.93471	5.08875	5.13779	4.80412	4.8512	4.87334	4.93	0.11	8.90	8.93794	8.90454	8.95119	8.83715	8.95973	8.88764	8.94908	8.86905	8.91	0.04
(512, 512)	1T	182.00	178.966	176.974	195.633	177.329	176.734	176.909	176.586	184.053	180.58	5.88	439.02	438.945	439.132	439.201	439.161	438.982	439.22	439.006	439.141	439.09	0.10
	2T	98.27	99.4159	93.685	99.0492	96.0553	94.5152	94.6945	93.681	97.547	96.32	2.17	225.28	224.094	224.101	223.692	223.672	224.112	223.749	224.19	223.147	224.00	0.55
	4T	52.86	52.6976	53.6903	53.2608	55.0552	53.4004	55.097	52.6099	56.3504	53.89	1.23	117.01	121.209	121.406	117.102	117.287	121.238	120.055	119.46	117.612	119.15	1.80
	8T	31.48	30.9976	30.8613	29.492	29.2982	30.9832	32.1954	29.9114	30.9678	30.69	0.89	65.15	63.3045	65.3751	67.3485	65.3794	63.3039	66.868	62.741	64.0658	64.84	1.52
	16T	19.88	19.9908	20.0353	20.1602	19.9808	21.4847	20.2189	20.9945	20.2862	20.34	0.51	36.83	37.3125	37.1507	37.2402	36.9451	38.4538	37.4747	37.8834	37.1439	37.38	0.48
(1,4,8,14)	1T	13.75	13.983	13.7487	13.4361	13.6113	13.191	13.3553	13.1714	13.1776	13.49	0.28	34.06	34.0674	34.0939	34.0501	34.0605	34.0792	35.0543	35.0773	35.1446	34.41	0.48
	2T	8.49	8.79347	8.36036	8.45026	8.37136	8.45957	8.41539	8.48524	8.40652	8.47	0.12	17.58	17.5457	17.5327	17.5403	17.5448	17.5259	17.5522	17.3454	17.536	17.52	0.06
	4T	5.62	5.55272	5.53921	5.57441	5.6272	5.64156	5.8661	5.5647	5.5317	5.61	0.10	9.19	9.16068	9.13281	9.13639	9.12171	9.14676	9.11929	9.1432	9.27297	9.16	0.05
	8T	3.61	3.57664	3.55846	3.6176	3.58996	3.49149	3.55791	3.60691	3.55946	3.57	0.04	5.08	4.9072	4.9199	4.94921	5.05722	5.11177	4.90904	5.15462	4.89363	5.00	0.10
	16T	2.07	2.04334	2.03835	2.19517	2.05424	2.15135	2.09897	2.12995	2.08395	2.10	0.05	2.79	2.80698	2.816	2.87482	2.80692	2.8429	2.97714	2.83352	2.82814	2.84	0.05
(1,4,8,355)	1T	349.13	351.898	350.044	353.675	359.221	359.046	350.112	350.421	356.82	353.37	3.79	883.19	882.423	882.431	882.885	882.739	882.429	882.612	883.08	882.192	882.66	0.32
	2T	188.80	191.15	196.999	188.062	191.06	197.822	196.982	191.228	188.578	192.30	3.69	448.59	449.51	448.942	454.484	449.372	451.575	450.904	450.706	451.208	450.59	1.70
	4T	104.58	109.243	104.398	108.208	107.681	104.709	103.398	104.202	106.537	105.88	1.96	241.08	241.547	237.366	236.739	241.245	241.198	242.159	235.115	237.504	239.33	2.47
	8T	61.77	63.9636	65.6373	69.5257	66.7223	63.0329	69.8826	64.3552	61.3722	65.14	2.92	139.72	136.318	132.063	131.194	131.11	127.338	136.938	132.131	133.704	133.39	3.52
	16T	42.00	41.8632	41.506	41.0653	40.4931	41.3858	40.4095	41.2451	41.3024	41.25	0.51	74.86	75.093	75.0429	75.0391	73.8296	74.5321	75.6397	73.7942	74.9226	74.75	0.57
(2,6,12,116)	1T	350.37	350.834	371.981	348.691	351.502	355.655	349.016	349.359	355.65	353.67	6.93	883.02	882.724	883.202	882.931	883.54	883.519	882.734	883.263	884.402	883.26	0.49
	2T	194.28	191.182	193.046	190.86	196.845	190.675	193.59	190.928	188.898	192.26	2.27	451.85	450.929	449.167	449.454	450.164	449.27	450.628	449.627	450.725	450.20	0.85
	4T	111.75	112.368	110.955	114.55	111.023	112.57	112.031	112.315	112.198	112.20	0.99	241.89	237.497	244.303	242.38	241.756	242.703	234.322	235.766	233.411	239.34	3.86
	8T	66.94	64.9906	72.7936	67.8109	65.381	67.2841	65.3419	65.5038	68.3009	67.15	2.30	130.71	135.878	131.949	131.467	131.042	130.355	128.243	131.578	129.613	131.20	1.97
	16T	42.90	45.0925	43.9505	43.5263	42.939	42.8954	45.338	44.2235	43.6077	43.83	0.86	74.27	76.3025	76.1378	74.0261	75.6378	73.5789	73.6081	73.7984	75.0465	74.71	1.03

F.4 Caching solver with CSGs and IndexGraphs

Table F.6.: Comparison of caching solver with two different graph types, grouped by tube configuration. Samples are grouped by the no. of threads employed per row pair. Runtime of in ms. μ and σ give median and standard deviation for samples with same graph, tube and no. of threads.

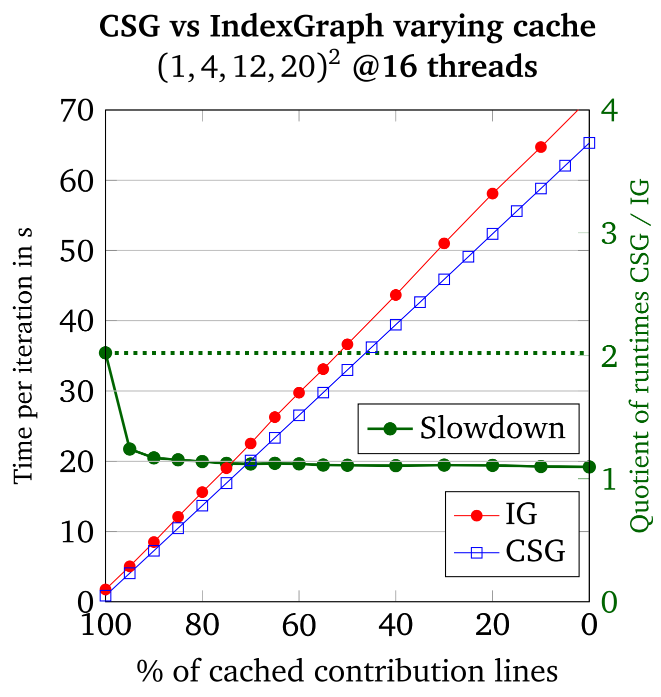
		Caching Solver with CSGs							Caching Solver with IGs						
No of Sample		**1 / 6**	**2 / 7**	**3 / 8**	**4 / 9**	**5**	μ	σ	**1 / 6**	**2 / 7**	**3 / 8**	**4 / 9**	**5**	μ	σ
$(1,4,8,14)$[1]	1T	21.03	20.97	21.18	21.21	20.99			13.21	13.82	13.53	13.13	13.12		
		21.10	21.14	20.96	21.20		21.09	0.09	13.25	13.31	13.77	13.29		13.38	0.25
	2T	16.55	16.42	13.37	16.58	16.43			8.48	8.58	8.35	8.24	8.45		
		16.45	16.52	13.66	16.39		15.82	1.24	8.38	8.38	8.29	8.59		8.42	0.11
	4T	9.23	9.90	9.85	9.90	9.42			5.63	5.64	5.63	5.55	5.56		
		9.72	8.76	9.38	10.19		9.59	0.41	5.64	5.62	5.54	5.63		5.60	0.04
	8T	6.99	6.78	7.42	7.34	7.36			3.59	3.57	3.59	3.61	3.59		
		7.08	7.44	8.01	6.87		7.26	0.35	3.55	3.52	3.56	3.66		3.58	0.04
	16T	4.04	4.00	3.95	3.88	4.03			2.06	2.03	2.05	2.07	2.05		
		3.95	3.59	4.00	3.82		3.92	0.13	2.02	2.05	2.06	2.04		2.05	0.02
$(1,4,8,355)$	1T	645.55	649.77	648.67	651.97	657.38			349.55	373.61	360.18	349.98	348.99		
		647.61	647.28	646.91	644.88		648.89	3.62	357.30	349.20	354.14	350.21		354.80	7.65
	2T	466.80	430.62	395.92	462.97	423.66			192.40	191.21	192.65	194.84	187.87		
		473.60	474.58	404.37	392.91		436.16	32.01	191.27	191.16	191.09	191.43		191.55	1.73
	4T	269.75	260.06	269.61	265.80	278.97			103.01	106.52	104.45	104.31	107.52		
		243.17	236.14	221.12	265.50		256.68	17.88	110.20	108.30	108.77	104.59		106.41	2.31
	8T	170.32	177.78	167.98	165.06	162.80			63.52	64.13	64.22	62.67	63.53		
		164.15	172.64	166.10	172.91		168.86	4.64	64.88	69.58	63.21	61.31		64.12	2.16
	16T	87.02	86.61	96.50	86.11	87.26			41.21	40.40	40.82	41.87	40.27		
		95.06	86.88	87.02	85.91		88.71	3.82	41.68	40.68	41.37	41.18		41.05	0.52
$(2,6,12,116)$	1T	621.74	621.66	625.45	647.81	622.79			348.91	351.94	373.66	349.20	360.49		
		618.87	617.64	618.01	621.69		623.96	8.75	381.00	356.39	373.63	356.38		361.29	11.20
	2T	452.54	371.08	449.70	384.60	401.56			202.84	190.11	193.74	190.96	192.06		
		374.88	384.31	454.48	423.28		410.71	32.76	193.65	190.61	198.12	191.53		193.73	3.95
	4T	214.31	268.65	268.16	213.36	252.38			110.61	109.30	113.55	112.12	109.44		
		271.39	260.00	252.69	204.52		245.05	25.19	112.28	110.29	108.98	112.40		111.00	1.54
	8T	188.26	172.49	168.30	169.03	163.01			67.71	72.67	68.63	68.82	74.13		
		179.13	171.35	165.14	172.44		172.13	7.20	65.13	66.50	65.74	72.38		69.08	3.07
	16T	109.29	106.02	109.65	97.83	110.13			44.00	44.62	44.30	44.77	43.27		
		109.48	109.23	110.99	107.53		107.79	3.78	43.38	43.99	43.75	43.54		43.96	0.50

Table F.7.: Comparison of caching running with two different graph type and varying amount of caching, grouped by the tube configuration of order 1. The runtime of samples is given in milliseconds. They are grouped by the number of threads employed and the caching rate. μ and σ give median and standard deviation for all samples with same graph, tube configuration and number of threads.

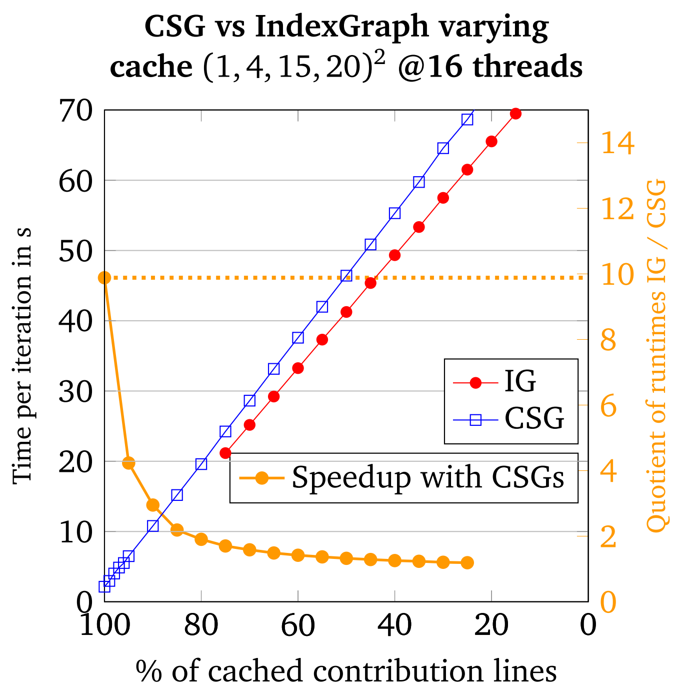
	Caching Solver with CSGs												Caching Solver with IGs											
	%	Sample			μ	σ	%	Sample			μ	σ	%	Sample			μ	σ	%	Sample			μ	σ
		1	2	3				1	2	3				1	2	3				1	2	3		
	100	92.20	87.53	87.37	89.03	2.24	**99**	144.26	141.78	134.62	140.22	4.09	**100**	41.50	40.71	40.99	41.07	0.33	**99**	86.61	86.10	83.87	85.53	1.19
	98	183.56	184.39	183.44	183.80	0.42	**97**	232.72	232.05	240.55	235.11	3.86	**98**	128.61	127.61	127.31	127.84	0.55	**97**	170.69	171.92	171.51	171.37	0.51
	96	288.53	283.31	290.97	287.60	3.20	**95**	336.17	341.02	337.62	338.27	2.03	**96**	215.76	214.46	214.22	214.81	0.68	**95**	257.09	257.13	257.27	257.16	0.08
	90	579.79	583.23	582.02	581.68	1.43	**85**	827.18	831.08	827.70	828.66	1.73	**90**	470.64	472.28	472.36	471.76	0.79	**85**	687.25	686.52	686.59	686.79	0.33
	80	1072.11	902.35	1062.72	1012.39	77.91	**75**	1305.54	1314.31	1304.76	1308.20	4.33	**80**	902.35	1850.42	902.38	1218.38	446.92	**75**	1116.11	1116.74	1115.87	1116.24	0.37
(1,4,8, 355)	**70**	1558.64	1556.62	1548.39	1554.55	4.43	**65**	1793.16	1791.08	1796.67	1793.64	2.31	**70**	1331.36	1331.98	1331.35	1331.56	0.29	**65**	1544.88	1544.96	1544.61	1544.82	0.15
	60	2038.26	2041.03	2041.54	2040.28	1.44	**55**	2238.71	2242.56		2240.64	1.92	**60**	1759.27	1760.27	1761.17	1760.24	0.78	**55**	1970.25	1972.25	1973.49	1972.00	1.33
	50	2487.75	2486.43		2487.09	0.66	**45**	2721.43	2726.43		2723.93	2.50	**50**	2188.83	2189.73	2189.97	2189.51	0.49	**45**	2408.36	2408.31	2408.90	2408.52	0.27
	40	2961.08	2965.57		2963.33	2.25	**35**	3197.69	3201.67		3199.68	1.99	**40**	2618.85	2621.38	2622.92	2621.05	1.68	**35**	2837.06	2837.97	2838.69	2837.91	0.67
	30	3434.62	3428.64		3431.63	2.99	**25**	3675.92	3678.14		3677.03	1.11	**30**	3051.05	3051.05	3051.82	3051.31	0.36	**25**	3269.59	3269.95	3269.86	3269.80	0.15
	20	3909.31	3911.76		3910.54	1.23	**15**	4141.03	4152.54		4146.79	5.76	**20**	3485.73	3487.26	3487.88	3486.96	0.90	**15**	3700.49	3700.78	3701.37	3700.88	0.37
	10	4386.00	4383.01		4384.51	1.50	**5**	4605.61	4612.59		4609.10	3.49	**10**	3919.28	3919.67	3919.51	3919.49	0.16	**5**	4133.73	4133.14	4132.11	4132.99	0.67
	0	4844.94	4844.34		4844.64	0.30							**0**	4349.40	4350.71	4351.39	4350.50	0.83						
	100	97.83	107.53	106.02	103.78	0.42	**99**	134.56	135.04	134.51	134.70	0.24	**100**	39.29	39.94	38.76	39.33	0.48	**99**	82.57	83.32	83.56	83.15	0.42
	98	179.67	184.67	183.44	182.59	2.13	**97**	232.03	230.88	233.86	232.26	1.23	**98**	123.20	122.49	125.66	123.78	1.36	**97**	171.18	170.90	173.32	171.80	1.08
	96	271.83	262.31	271.27	268.47	4.36	**95**	322.86	323.69	323.21	323.26	0.34	**96**	207.25	207.66	207.74	207.55	0.21	**95**	257.15	254.42	254.07	255.21	1.38
	90	557.85	561.07	559.65	559.52	1.32	**85**	795.15	797.20	795.18	795.84	0.96	**90**	469.95	470.87	471.13	470.65	0.51	**85**	685.78	685.82	686.20	685.93	0.19
	80	1040.01	1037.71	1038.06	1038.59	1.01	**75**	1277.84	1275.20	1275.94	1276.33	1.11	**80**	910.25	909.80	909.98	910.01	0.18	**75**	1125.52	1127.43	1124.27	1125.74	1.30
	70	1506.31	1513.23	1510.97	1510.17	2.88	**65**	1748.79	1748.07	1746.62	1747.83	0.90	**70**	1340.93	1340.48	1343.09	1341.50	1.14	**65**	1556.37	1559.32	1556.81	1557.50	1.30
(2,6,12, 116)	**60**	1983.51	1983.87	1982.39	1983.26	0.63	**55**	2221.36	2219.70	2222.71	2221.26	1.23	**60**	1776.22	1777.67	1773.38	1775.76	1.78	**55**	1991.46	1992.19	1989.63	1991.09	1.08
	50	2456.33	2456.66	2454.76	2455.92	0.83	**45**	2734.64	2705.97	2703.36	2714.66	14.17	**50**	2209.76	2208.05	2209.88	2209.23	0.84	**45**	2432.41	2433.98	2431.94	2432.78	0.87
	40	2940.56	2940.15	2938.78	2939.83	0.76	**35**	3197.09	3180.64	3173.48	3183.74	9.88	**40**	2653.52	2655.56	2656.39	2655.16	1.21	**35**	2870.19	2870.04	2871.07	2870.43	0.45
	30	3418.66	3419.33	3419.09	3419.03	0.28	**25**	3663.24	3688.85	3659.32	3670.47	13.09	**30**	3089.81	3089.86	3087.82	3089.16	0.95	**25**	3305.37	3306.67	3308.66	3306.90	1.35
	20	3905.26	3897.63	3897.13	3900.01	3.72	**15**	4137.02	4137.26	4153.95	4142.74	7.92	**20**	3538.90	3536.35	3536.33	3537.19	1.21	**15**	3758.41	3757.87	3758.85	3758.38	0.40
	10	4371.04	4371.84	4371.25	4371.38	0.34	**5**	4611.76	4607.46	4608.63	4609.28	1.82	**10**	3976.31	3978.49	3973.12	3975.97	2.21	**5**	4191.46	4191.63	4191.68	4191.59	0.09
	0	4836.33	4834.06	4830.62	4833.67	2.35							**0**	4406.91	4408.80	4406.86	4407.52	0.90						

Table F.8.: Comparison of caching running with two different graph type and varying amount of caching, grouped by the tube configuration of order 2. The runtime of samples is given in milliseconds. They are grouped by the number of threads employed and the caching rate. μ and σ give median and standard deviation for all samples with same graph, tube configuration and number of threads.

	Caching Solver with CSGs												Caching Solver with IGs											
	%	Sample			μ	σ	%	Sample			μ	σ	%	Sample			μ	σ	%	Sample			μ	σ
		1	2	3				1	2	3				1	2	3				1	2	3		
	100	1.20	1.07	1.20	1.15	0.0606	**99**	1.64	1.62	1.52	1.59	0.0506	**100**	0.62	0.58	0.58	0.59	0.0199	**99**	1.00	1.00	1.00	1.00	0.0009
	98	2.09	2.01	1.97	2.02	0.0486	**97**	2.42	2.47	2.54	2.48	0.0499	**98**	1.42	1.42	1.43	1.42	0.0022	**97**	1.85	1.85	1.87	1.86	0.0115
	96	2.87	2.90	2.89	2.89	0.0143	**95**	3.48	3.35	3.33	3.39	0.0653	**96**	2.27	2.27	2.27	2.27	0.0013	**95**	2.70	2.70	2.71	2.70	0.0066
	90	5.67	5.77	5.70	5.71	0.0419	**85**	8.00	8.09	8.01	8.03	0.0425	**90**	4.83	4.83	4.83	4.83	0.0020	**85**	6.95	6.95	6.98	6.96	0.0117
	80	10.36	10.34	10.35	10.35	0.0071	**75**	12.68	12.67	12.72	12.69	0.0240	**80**	9.09	9.09	9.09	9.09	0.0030	**75**	11.22	11.21	11.21	11.21	0.0016
(1,4, 8,20)	**70**	15.09	15.02	15.01	15.04	0.0350	**65**	17.54	17.51	17.45	17.50	0.0379	**70**	13.36	13.35	13.35	13.35	0.0045	**65**	15.49	15.50	15.50	15.49	0.0045
	60	19.85	19.89	19.83	19.86	0.0253	**55**	22.15	22.06	22.04	22.08	0.0475	**60**	17.62	17.64	17.63	17.63	0.0064	**55**	19.77	19.76	19.77	19.77	0.0021
	50	24.44	24.43	24.46	24.44	0.0105	**45**	26.74	26.76	26.73	26.74	0.0149	**50**	21.93	21.91	21.91	21.92	0.0079	**45**	24.05	24.05	24.06	24.05	0.0030
	40	29.09	29.07	29.08	29.08	0.0099	**35**	31.46	31.45	31.45	31.45	0.0045	**40**	26.19	26.18	26.18	26.18	0.0026	**35**	28.35	28.33	28.34	28.34	0.0072
	30	33.97	33.97	34.00	33.98	0.0148	**25**	36.18	36.15	36.13	36.15	0.0207	**30**	30.46	30.48	30.46	30.47	0.0080	**25**	32.63	32.62	32.61	32.62	0.0073
	20	38.68	38.73	38.73	38.71	0.0244	**15**	40.78	40.78	40.77	40.77	0.0055	**20**	34.76	34.76	34.78	34.77	0.0119	**15**	36.93	36.94	36.93	36.93	0.0042
	10	43.04	43.06	43.06	43.05	0.0112	**5**	45.41	45.37	45.35	45.38	0.0241	**10**	39.07	39.06	39.07	39.07	0.0024	**5**	41.22	41.21	41.19	41.21	0.0120
	0	47.65	47.63	47.61	47.63	0.0133							**0**	43.40	43.39	43.38	43.39	0.0113						
	100	2.49	2.50	2.42	2.47	0.0349	**99**	3.34	3.13	3.04	3.17	0.1262	**100**						**99**					
	98	4.25	3.95	4.16	4.12	0.1284	**97**	4.96	5.14	4.99	5.03	0.0775	**98**						**97**					
	96	6.09	5.83	5.94	5.95	0.1065	**95**	6.73	6.99	6.89	6.87	0.1093	**96**						**95**					
	90	11.48	11.45	11.46	11.46	0.0139	**85**	16.39	16.26		16.33	0.0683	**90**						**85**					
	80	21.08	21.06		21.07	0.0081	**75**	25.77	25.60		25.69	0.0827	**80**						**75**					
	70	30.50	30.46	30.58	30.51	0.0518	**65**	35.48	35.33	35.65	35.49	0.1281	**70**	26.79	26.77	26.76	26.77	0.0094	**65**	31.07	31.08	31.06	31.07	0.0094
(1,4,16, 20)	**60**	40.37	40.48		40.43	0.0593	**55**	44.92	44.82		44.87	0.0509	**60**	35.37	35.14	35.47	35.33	0.1413	**55**	39.66	39.64	39.63	39.64	0.0138
	50	49.66	49.61		49.64	0.0207	**45**	54.32	54.27		54.29	0.0227	**50**	43.99	43.95	43.96	43.97	0.0150	**45**	48.32	48.31	48.30	48.31	0.0091
	40	59.09	59.09	59.13	59.10	0.0186	**35**	63.84	63.80		63.82	0.0198	**40**	52.57	52.50	52.56	52.54	0.0288	**35**	57.07	57.10	0.00	38.06	26.9100
	30	68.82	68.82		68.82	0.0002	**25**	73.37	73.24		73.31	0.0650	**30**	61.47	61.39	0.00	40.95	28.9594	**25**	65.71	65.65	0.00	43.78	30.9598
	20	78.53	78.36		78.44	0.0891	**15**	83.09	82.73		82.91	0.1772	**20**	69.98	70.07	0.00	46.68	33.0110	**15**	74.46	74.46	0.00	49.64	35.1018
	10	87.21	87.52		87.37	0.1516	**5**	92.06	91.66		91.86	0.1982	**10**	78.71	78.80	0.00	52.50	37.1252	**5**	83.01	83.01	0.00	55.34	39.1325
	0	96.68	96.58		96.63	0.0516							**0**	87.29	87.29	0.00	58.19	41.1480						



(a) Time per iteration when the percentage of cached contribution lines is decreased from full to no caching for the $(1-4-12-20)^2$ tube.



(b) Time per iteration when the percentage of cached contribution lines is decreased from full to no caching for the $(1-4-15-20)^2$ tube.

Figure F.1.: Performance comparison of caching solver with IndexGraphs and CSGs on two order 2 tubes.

List of Figures

1.1. Heuristical construction of SCNTs 2

3.1. Four examples of categorical product and graph composition. 17
3.2. Mirroring graph G with the concept of transfer graphs. 20
3.3. Junctions as basic elements of abstraction. 21
3.4. The first stages steps S_1 - S_3 for constructing a sheet of order 0. 23
3.5. Stage S_4 during construction of an order 0 sheet. 25
3.6. A base element of order zero. 26
3.7. A half and a complete junction arm of level 0. 28
3.8. Details about level 0 junctions. 29
3.9. Distance between two junctions. 30
3.10. G_{connect} within the first step of construction S_1^t. 31
3.11. First three steps for the construction of an order 1 sheet. 32
3.12. Sheet of order 1. 33
3.13. The four basic geometric transformations to roll up a sheet around the x-axis. . . . 34
3.14. Wrapping a line around a point on a circular path. 35
3.15. Rings in an order 1 tube. 38
3.16. Orientation of hexagons. 38
3.17. Translational symmetry for order 0 and 1 tubes. 39
3.18. Rotational symmetry for order 0 and 1 tubes. 40
3.19. Different levels of hierarchy in an order 2 tube. 43

4.1. Comparison of two different renderer quality levels. 47

5.1. Correct applications of bearings to immobilize a plane against rigid body movements. 53
5.2. Appropriately mounted SCNT with six bearings to prevent rigid body movements. 54
5.3. Details of the Dreiding potential. 56
5.4. Multiple local indices 58
5.5. Sparsitiy pattern of the stiffness matrix for tubes of orders 0 - 3. 59

6.1. Implicit tree to index nodes. 63
6.2. Principle of IndexGraphs. 66
6.3. Fetching data in PSH. 70
6.4. Reduction factor for the different indexing schemes. 77

7.1. Details for general Mirrored Z-like Structures. 85
7.2. Resolving combined symmetry with jumps in translational and rotational direction. Connection of top and bottom of the former graphene in blue. 86
7.3. Elements a CSG needs to store explicitly 88
7.4. Adjacent MZSs in tubes of order 0 and 1. 89
7.5. Zero line within a $(16, 16)$ tube shown in blue with the bottom of the graphene in magenta. 90
7.6. Schematic of the procedure to unfold tuples. 97

8.1. Convergence behavior of the three implemented solvers. 107
8.2. Memory consumption during simulation 113
8.3. Memory layout for stiffness contributions within the caching solver. 115
8.4. Read and write access in the required data structures during SpMV in caching solver. 117
8.5. Spans of different threads in the result vector 118

9.1. Torsion on a tube. 125
9.2. Effect of tension on SCNT junctions. 126
9.3. Performance comparison of the on-the-fly solver and the reference solver. 128
9.4. Parallel efficiency for two solvers. 130
9.5. Performance comparison of the value-symmetric solver and the reference solver. . 132
9.6. Performance comparison of the caching solver and the reference solver. 133
9.7. Scaling IndexGraphs versus CSGs and full caching 136
9.8. Performance comparison of the caching solver with different graph data structure and varying caching rate. 138
9.9. Performance IndexGraphs versus CSGs 140

10.1. Distributing CSGs with MPI. 145

B.1. The three different chiralities of SWCNTs. 150

E.1. Edited screenshot from the heap analysis with the tool massif for the construction of the $(1, 4, 8, 8)^2$ tube and the old edge map. 160
E.2. Edited screenshot from the heap analysis with the tool massif for the construction of the $(1, 4, 8, 8)^2$ tube and the new edge map. 162

F.1. Performance IndexGraphs versus CSGs Order 2 Part II 171

Bibliography

[1] Sumio Iijima. Helical microtubules of graphitic carbon. *Nature*, 354(6348):56, 1991.

[2] L.V. Radushkevich and V.M. Lukyanovich. About the structure of carbon formed by thermal decomposition of carbon monoxide on iron substrate. *Journal of Physical Chemistry (Moscow)*, 26:88–95, 1952. In Russian.

[3] M. Endo. Grow carbon fibers in the vapor phase. *Chemtech*, pages 568–576, 1988.

[4] A. Oberlin, M. Endo, and T. Koyama. Filamentous growth of carbon through benzene decomposition. *Journal of Crystal Growth*, 32(3):335–349, 1976.

[5] S. Iijima and T. Ichihashi. Single-shell carbon nanotubes of 1-nm diameter. *Nature*, 363(6430):603–605, Jun 1993.

[6] R. S. Ruoff and D. C. Lorents. Mechanical and thermal properties of carbon nanotubes. *Carbon*, 33(7):925–930, 1995.

[7] R. H. Baughman, A. A. Zakhidov, and W. A. de Heer. Carbon nanotubes–the route toward applications. *Science*, 297(5582):787–792, 2002.

[8] D. Qian, G. J. Wagner, W. K. Liu, M.-F. Yu, and R. S. Ruoff. Mechanics of carbon nanotubes. *Applied Mechanics Reviews*, 55(6):495–533, Oct 2002.

[9] A. E. Aliev and R. H. Baughman. Carbon nanotubes: An explosive thrust for nanotubes. *Nature Materials*, 9(5):385–386, 2010.

[10] H. Park, A. Afzali, S.-J. Han, G. S. Tulevski, A. D. Franklin, J. Tersoff, J. B. Hannon, and W. Haensch. High-density integration of carbon nanotubes via chemical self-assembly. *Nature Nanotechnology*, 7(12):787–791, Dec 2012.

[11] L. Chico, V. H. Crespi, L. X. Benedict, S. G. Louie, and M. L. Cohen. Pure carbon nanoscale devices: Nanotube heterojunctions. *Physical Review Letters*, 76:971–974, Feb 1996.

[12] P. Nagy, R. Ehlich, L.P. Biró, and J. Gyulai. Y-branching of single walled carbon nanotubes. *Applied Physics A*, 70(4):481–483, 2000.

[13] W. Z. Li, J. G. Wen, and Z. F. Ren. Straight carbon nanotube Y junctions. *Applied Physics Letters*, 79(12):1879–1881, 2001.

[14] Y. C. Choi and W. Choi. Synthesis of Y-junction single-wall carbon nanotubes. *Carbon*, 43(13):2737–2741, 2005.

[15] L. P. Biró, R. Ehlich, Z. Osváth, A. Koós, Z. E. Horváth, J. Gyulai, and J. B. Nagy. From straight carbon nanotubes to Y-branched and coiled carbon nanotubes. *Diamond and Related Materials*, 11(3-6):1081–1085, 2002.

[16] G. E. Scuseria. Negative curvature and hyperfullerenes. *Chemical Physics Letters*, 195(5-6):534–536, 1992.

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

[18] Y. Li, X. Qiu, F. Yang, X.-S. Wang, Y. Yin, Q. Fan, and X. Qiu. A comprehensive study on the mechanical properties of super carbon nanotubes. *Journal of Physics D: Applied Physics*, 41(15):155423, 2008.

[19] X. Liu, Q.-S. Yang, X.-Q. He, and Y.-W. Mai. Molecular mechanics modeling of deformation and failure of super carbon nanotube networks. *Nanotechnology*, 22(47):475701, 2011.

[20] V. R. Coluci, N. M. Pugno, S. O. Dantas, D. S. Galvão, and A. Jorio. Atomistic simulations of the mechanical properties of 'super' carbon nanotubes. *Nanotechnology*, 18(33):335702, 2007.

[21] M. Burger, C. Bischof, C. Schröppel, and J. Wackerfuß. Exploiting structural properties during carbon nanotube simulation. In O. Gervasi, B. Murgante, S. Misra, M. L. Gavrilova, A. M. Rocha, C. Torre, D. Taniar, and B. O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2015*, volume 9156 of *Lecture Notes in Computer Science*, pages 339–354. Springer International Publishing, 2015.

[22] M. Burger and C. Bischof. Using instancing to efficiently render carbon nanotubes. In M. Mehl, editor, *Proc. 3rd International Workshop on Computational Engineering*, volume 3, pages 206–210, Stuttgart, Germany, Oct 2014.

[23] M. Burger, C. Bischof, C. Schröppel, and J. Wackerfuß. A unified and memory efficient framework for simulating mechanical behavior of carbon nanotubes. *Procedia Computer Science*, 51:413–422, 2015.

[24] M. Burger, C. Bischof, C. Schröppel, and J. Wackerfuß. Methods to model and simulate super carbon nanotubes of higher order. *Concurrency and Computation: Practice and Experience*, page cpe3872, 2016.

[25] M. Burger, G. N. Nguyen, and C. Bischof. Extending perfect spatial hashing to index tuple-based graphs representing super carbon nanotubes. *Procedia Computer Science*, 2017. In print.

[26] M. Burger, C. Bischof, and J.Wackerfuß. Compressed symmetric graphs for the simulation of super carbon nanotubes. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 286–293, Jul 2016.

[27] M. Burger, C. Bischof, C. Schröppel, and J. Wackerfuß. An improved algorithm for simulating the mechanical behavior of super carbon nanotubes. In *IEEE 18th International Conference on Computational Science and Engineering (CSE)*, pages 286–293, Oct 2015.

[28] M. Wang, X. Qiu, and X. Zhang. Mechanical properties of super honeycomb structures based on carbon nanotubes. *Nanotechnology*, 18(7):075711, 2007.

[29] Z. Qin, X.-Q. Feng, J. Zou, Y. Yin, and S.-W. Yu. Superior flexibility of super carbon nanotubes: Molecular dynamics simulations. *Applied Physics Letters*, 91(4):043108, 2007.

[30] Y. Li, X. Qiu, F. Yang, X.-S. Wang, and Y. Yin. The effective modulus of super carbon nanotubes predicted by molecular structure mechanics. *Nanotechnology*, 19(22):225701, 2008.

[31] Y.-L. Chen, B. Liu, Y. Yin, Y.-G. Huang, and K.-C. Hwuang. Nonlinear deformation processes and damage modes of super carbon nanotubes with armchair-armchair topology. *Chinese Physics Letters*, 25(7):2577, 2008.

[32] N. G. Chopra, R.J. Luyken, K. Cherrey, V. H. Crespi, et al. Boron nitride nanotubes. *Science*, 269(5226):966, 1995.

[33] D. Zhou and S. Seraphin. Complex branching phenomena in the growth of carbon nanotubes. *Chemical Physics Letters*, 238(4-6):286–289, 1995.

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

[35] 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*, 72:035435, Jul 2005.

[36] B. J. Alder and T. E. Wainwright. Studies in Molecular Dynamics. I. General Method. *The Journal of Chemical Physics*, 31(2):459–466, 1959.

[37] D. W. Brenner. Empirical potential for hydrocarbons for use in simulating the chemical vapor deposition of diamond films. *Physical Review B*, 42:9458–9471, Nov 1990.

[38] S. L. Mayo, B. D. Olafson, and W. A. Goddard. DREIDING: A generic force field for molecular simulations. *Journal of Physical Chemistry*, 94:8897–8909, 1990.

[39] M. Malakouti and A. Montazeri. Nanomechanics analysis of perfect and defected graphene sheets via a novel atomic-scale finite element method. *Superlattices and Microstructures*, 94:1–12, 2016.

[40] N. M. Pugno. Mimicking nacre with super-nanotubes for producing optimized supercomposites. *Nanotechnology*, 17(21):5480–5484, 2006.

[41] Y. Yin, T. Zhang, F. Yang, and X. Qiu. Geometric conditions for fractal super carbon nanotubes with strict self-similarities. *Chaos, Solitons & Fractals*, 37(5):1257–1266, 2008.

[42] Y. J. Yin, Q. S. Fan, F. Yang, and Y. Li. Super carbon nanotubes, fractal super tubes and fractal super fibres. *Materials Science and Technology*, 26(11):1327–1331, 2010.

[43] R. D. Kangwai, S. D. Guest, and S. Pellegrino. An introduction to the analysis of symmetric structures. *Computers & Structures*, 71(6):671–688, 1999.

[44] K. Koohestani. Exploitation of symmetry in graphs with applications to finite and boundary elements analysis. *International Journal for Numerical Methods in Engineering*, 90(2):152–176, 2012.

[45] C. Schröppel and J. Wackerfuß. Meshing highly regular structures: The case of super carbon nanotubes of arbitrary order. *Journal of Nanomaterials*, (Article ID 736943), 2015.

[46] Jonas Marczona. Erweiterung des Forschungscodes MISMO hinsichtlich einer neuartigen Benutzerschnittstelle zur Implementierung beliebiger numerischer Methoden im Kontext allgemeiner Diskretisierungsverfahren. Master's thesis, Technische Universität Darmstadt, 2015.

[47] J. Marczona. Konzept für eine flexible und parallelisierte Programmplattform zum Lösen von Randwert- und Anfangswertproblemen. Bachelor's thesis, Technische Universität Darmstadt, Germany, 2013.

[48] J. Marczona. Optimierung des FEM Berechnungsprogrammes von MISMO in Hinblick auf eine Portierung auf den XeonPhi Coprozessor. Studienarbeit, Technische Universität Darmstadt, Germany, 2013.

[49] J. A. Bondy. *Graph Theory With Applications*. Elsevier Science Ltd, 1976.

[50] R. Diestel. *Graphentheorie*. Springer-Lehrbuch. Springer, 1996.

[51] Y. Yin, Y. Chen, J. Yin, and K. Huang. Geometric conservation laws for perfect Y-branched carbon nanotubes. *Nanotechnology*, 17(19):4941, 2006.

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

[53] L. Pauling. *The nature of the chemical bond and the structure of molecules and crystals: An introduction to modern structural chemistry*, volume 18. Cornell University Press, 1960.

[54] T. L. Brown, H. R. Lemay, B. E. Burstein, C. J. Murphy, P. M Woodward, and Stoltzfus M. W. *Chemistry: The Central Science*. Pearson Prentice Hall, 1997.

[55] E. Clementi, D. L. Raimondi, and W. P. Reinhardt. Atomic Screening Constants from SCF Functions. II. Atoms with 37 to 86 Electrons. *The Journal of Chemical Physics*, 47(4):1300–1307, 1967.

[56] J. C. Slater. Atomic radii in crystals. *The Journal of Chemical Physics*, 41(10):3199–3204, 1964.

[57] D. Shreiner, G. Sellers, John M. Kessenich, and B. M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[58] R. J. Rost, B. Licea-Kane, D. Ginsburg, J. Kessenich, B. Lichtenbelt, H. Malan, and M. Weiblen. *OpenGL Shading Language*. Pearson Education, 2009.

[59] B.-T. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[60] D. Gross, W. Hauger, J. Schröder, and W. A. Wall. *Technische Mechanik: Band 1: Statik*. Springer-Verlag, Berlin Heidelberg, 2009.

[61] W.-K. Ng and C. V. Ravishankar. Block-oriented compression techniques for large statistical databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):314–328, Mar 1997.

[62] S. Lefebvre and H. Hoppe. Perfect spatial hashing. In J. Finnegan and J. Dorsey, editors, *ACM SIGGRAPH 2006 Papers*, pages 579–588, 2006.

[63] G. N. Nguyen. Efficient indexing method for tuple-based graphs. Bachelor's thesis, Technische Universität Darmstadt, Germany, Mar 2016.

[64] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, SA '11, pages 161:1–161:8, New York, NY, USA, 2011.

[65] C. T. Pozzer, C. A. de Lara Pahins, and I. Heldal. A hash table construction algorithm for spatial hashing based on linear memory. In *Proceedings of the 11th Conference on Advances in Computer Entertainment Technology*, ACE '14, pages 35:1–35:4, New York, NY, USA, 2014.

[66] M. Buckland. *Programming Game AI by Example*. Wordware Pub Co, first edition, Oct 2004.

[67] E. J. Hastings, J. Mesit, and R. K. Guha. Optimization of large-scale, real-time simulations by spatial hashing. In *Proc. 2005 Summer Computer Simulation Conference*, volume 37, pages 9–17, 2005.

[68] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta. Real-time parallel hashing on the GPU. In *ACM SIGGRAPH Asia 2009 Papers*, SIGGRAPH Asia '09, pages 154:1–154:9, New York, NY, USA, 2009.

[69] B. Schling. *The Boost C++ Libraries*. XML Press, 2011.

[70] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51(2):261–272, 1995.

[71] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, Jun 1962.

[72] S. Chen and J.H. Reif. Efficient lossless compression of trees and graphs. In *Data Compression Conference, 1996. DCC '96. Proceedings*, pages 428–437, Mar 1996.

[73] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[74] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA, 1973.

[75] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, Apr 1976.

[76] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 419–432, New York, NY, USA, 2008.

[77] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[78] A. C. Gilbert and K. Levchenko. Compressing network graphs. In *Proceedings of the LinkKDD workshop at the 10th ACM Conference on KDD*, volume 124, 2004.

[79] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009.

[80] P. Boldi and S. Vigna. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004.

[81] I. Dhillon, Y. Guan, and B. Kulis. A fast kernel-based multilevel algorithm for graph clustering. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 629–634, New York, NY, USA, 2005.

[82] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[83] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[84] O. Schenk, M. Bollhöfer, and R. A. Römer. On large-scale diagonalization techniques for the Anderson Model of localization. *SIAM Journal on Scientific Computing*, 28(3):963–983, 2006.

[85] A. Kuzmin, M. Luisier, and O. Schenk. *Fast Methods for Computing Selected Elements of the Green's Function in Massively Parallel Nanoelectronic Device Simulations*, pages 533–544. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[86] X. S. Li and J. W. Demmel. SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, Jun 2003.

[87] O. Schenk, K. Gärtner, and W. Fichtner. Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared Memory Multiprocessors. *BIT Numerical Mathematics*, 40(1):158–176, 2000.

[88] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, Dec 1952.

[89] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148, 1977.

[90] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[91] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[92] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[93] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[94] O. Axelsson and G. Lindskog. On the rate of convergence of the preconditioned conjugate gradient method. *Numerische Mathematik*, 48(5):499–523, 1986.

[95] G. Meurant. The block preconditioned conjugate gradient method on vector computers. *BIT Numerical Mathematics*, 24(4):623–633, 1984.

[96] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G.W. Stewart. *LINPACK Users' Guide*. 1979.

[97] C. Lomont. Introduction to Intel Advanced Vector Extensions. *Intel White Paper*, 2011.

[98] D. S. Kershaw. The incomplete Cholesky-conjugate gradient method for the iterative solution of systems of linear equations. *Journal of Computational Physics*, 26(1):43–65, 1978.

[99] C.-J. Lin and J. J. Moré. Incomplete Cholesky Factorizations with Limited Memory. *SIAM Journal on Scientific Computing*, 21(1):24–45, 1999.

[100] M. T. Jones and P. E. Plassmann. An Improved Incomplete Cholesky Factorization. *ACM Transactions on Mathematical Software*, 21(1):5–17, Mar 1995.

[101] M. Benzi, C. D. Meyer, and M. Tůma. A sparse approximate inverse preconditioner for the conjugate gradient method. *SIAM Journal on Scientific Computing*, 17(5):1135–1149, 1996.

[102] R. H. Chan and M. K. Ng. Conjugate Gradient Methods for Toeplitz Systems. *SIAM Review*, 38(3):427–482, 1996.

[103] J. A. Fessler and S. D. Booth. Conjugate-gradient preconditioning methods for shift-variant pet image reconstruction. *IEEE Transactions on Image Processing*, 8(5):688–699, May 1999.

[104] S. F. Ashby and R. D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, 1996.

[105] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. *Vectorized Sparse Matrix Multiply for Compressed Row Storage Format*, pages 99–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[106] J. L. Greathouse and M. Daga. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 769–780, Nov 2014.

[107] J. Sun, G. Peterson, and O. Storaasli. Sparse Matrix-Vector Multiplication Design on FPGAs. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 349–352, Apr 2007.

[108] T. Wirth. Optimized Multiplication of Vectors and Sparse Matrices of certain structure on Multicore-CPUs and GPUs. Bachelor's thesis, Technische Universität Darmstadt, Germany, Jun 2017.

[109] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing*, 36(5):C401–C423, 2014.

[110] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, Mar 2001.

[111] N. Firasta, P. Buxton, M.and Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[112] A. Tanikawa, K. Yoshikawa, T. Okamoto, and K. Nitadori. N-body simulation for self-gravitating collisional systems with a new SIMD instruction set extension to the x86 architecture, Advanced Vector eXtensions. *New Astronomy*, 17(2):82–92, 2012.

[113] S. Fialko. Application of AVX (Advanced Vector Extensions) for improved performance of the PARFES - finite element Parallel Direct Solver. In *2013 Federated Conference on Computer Science and Information Systems*, pages 447–454, Sept 2013.

[114] J.-I. Agulleiro and J.J. Fernandez. Tomo3D 2.0 - Exploitation of Advanced Vector eXtensions (AVX) for 3D reconstruction. *Journal of Structural Biology*, 189(2):147–152, 2015.

[115] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, pages 99–105, 1999.

[116] J. M. Bull and D. O'Neill. A microbenchmark suite for OpenMP 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, Dec 2001.

[117] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, Mar 1989.

[118] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, Apr 2009.

[119] Ü. V. Çatalyürek, K. Kaya, and B. Uçar. On shared-memory parallelization of a sparse matrix scaling algorithm. In *2012 41st International Conference on Parallel Processing*, pages 68–77. IEEE, 2012.

[120] S. Williams, L. Oliker, R.Vuduc, J. Shalf, K.Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[121] R. Saito, M. Fujita, G. Dresselhaus, and M. S. Dresselhaus. Electronic structure of chiral graphene tubules. *Applied Physics Letters*, 60(18):2204–2206, 1992.

[122] M.S. Dresselhaus, G. Dresselhaus, and R. Saito. Physics of carbon nanotubes. *Carbon*, 33(7):883–891, 1995.

[123] M. Meo and M. Rossi. Prediction of young's modulus of single wall carbon nanotubes by molecular-mechanics based finite element modeling. *Composites Science and Technology*, 66(11-12):1597–1605, 2006.

[124] D. Porezag, T. Frauenheim, T. Köhler, G. Seifert, and R. Kaschner. Construction of tight-binding-like potentials on the basis of density-functional theory: Application to carbon. *Physical Review B*, 51:12947–12957, May 1995.

[125] S. J. Stuart, A. B. Tutein, and J. A. Harrison. A reactive potential for hydrocarbons with intermolecular interactions. *The Journal of Chemical Physics*, 112(14):6472–6486, 2000.

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

[127] Y. Li, X. Qiu, F. Yang, X.-S. Wang, Y. Yin, and Q. Fan. Chirality independence in critical buckling forces of super carbon nanotubes. *Solid State Communications*, 148(1-2):63–68, 2008.

[128] J. M. Romo-Herrera, M. Terrones, H. Terrones, S. Dag, and V. Meunier. Covalent 2D and 3D networks from 1D nanostructures: Designing new materials. *Nano letters*, 7(3):570–576, 2007.

[129] I. László. Topological description and construction of single wall carbon nanotube junctions. *Croatica chemica acta*, 78(2):217–221, 2005.

Academic Curriculum Vitae

Michael Burger studied computer science at Technische Universität Darmstadt. He got a bachelor's degree in 2007 and graduated in 2011 with a master's degree in computer science.

During his studies, he focused on computer graphics, programming of different platforms (GPUs, FPGAs, microprocessors) and software engineering. In 2012, he started working as a scientific assistant at the Scientific Computing group at TU Darmstadt. Additionally, he was a full scholarship holder of the Graduate School of Computational Engineering in Darmstadt from 2013-2016.

In 2017, he was awarded with a doctoral degree in natural sciences from Technische Universität Darmstadt.

His research interests include parallel programming and high performance computing techniques with a focus on the efficient simulation of super carbon nanotubes by exploiting their inherent structural properties.