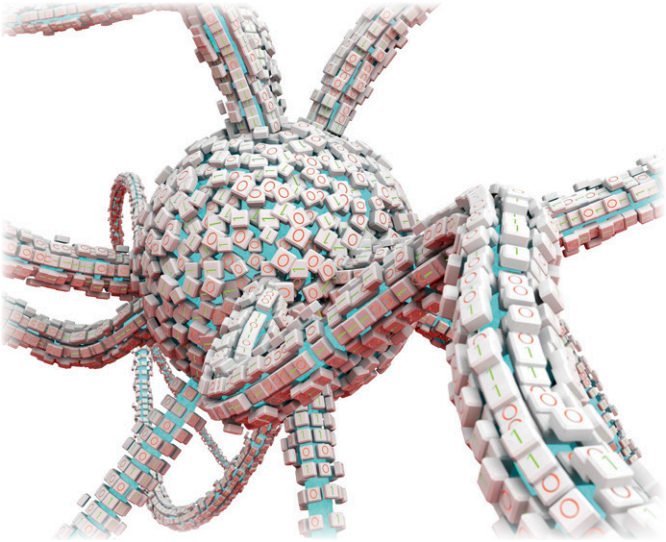
Turbo für Graphdatenbanken

Graphpartitionierung mit KaHIP

Heiko Papenfuß, Peter Sanders, Christian Schulz

In einer zunehmend vernetzten, digitalen Welt erfährt ein neuartiger Typ Datenbank großen Zulauf: die Graphdatenbank. Mit ihr lassen sich Datenpunkte und ihre Verbindungen untereinander besser abbilden als mit klassischen relationalen Datenbanken. Damit die ihnen zugrunde liegende Komplexität für Software beherrschbar bleibt und effizient verarbeitet werden kann, braucht es Algorithmen, die Graphen partitionieren. Dieser Beitrag erläutert, wie diese funktionieren und welche Anwendungsmöglichkeiten daraus entstehen.



Graphen: Knoten und Kanten

Graphen sind Strukturen aus „Knoten" und sie untereinander verbindenden „Kanten". Sie eignen sich zur Modellierung von Netzwerken aller Art, in denen Objekte zueinander in gerichteten oder ungerichteten Beziehungen stehen. Ein besonders anschauliches Beispiel ist wohl der Graph sozialer Netzwerke, der einerseits Personen (Knoten) samt ihrer Eigenschaften beschreibt, aber eben auch die Beziehungen dieser Personen zueinander. Die Kanten zwischen den Knoten beschreiben in diesem Fall, welche Personen miteinander als „Freund" oder „Follower" verbunden sind. Bei Twitter beispielsweise handelt es sich aufgrund der Möglichkeit, jemandem einseitig zu folgen, um gerichtete Graphen. Die Freundschaftsbeziehungen bei Facebook mit seiner reziproken Struktur sind hingegen ein ungerichteter Graph.

Nun lässt sich eine Fülle von Beziehungen mit Graphen darstellen. So ist ein Straßennetz nichts anderes als ein Graph, der für Navigationssoftware in einer Datenbank abgebildet werden muss. Auch Schaltkreise in Computerchips lassen sich als Graph beschreiben, genauso wie Sensornetzwerke. Ebenfalls augenfällig sind die Einsatzmöglichkeiten bei Kundendaten, die für Marketing und Vertrieb genutzt werden.

Beziehungsspezialisten: Graphdatenbanken

Zur digitalen Speicherung von Graphen haben spezielle NoSQL-Graphdatenbanken in jüngster Zeit an Bedeutung gewonnen. Sie sind klassischen relationalen Datenbanken überlegen, weil sie eben nicht nur die Knoten in einzelnen Datensätzen abbilden, sondern auch die Beziehungen. Im Management von Kundenstammdaten (auch Customer Master Data Management, kurz Customer MDM, genannt, [Uniserv]) beispielsweise kommen gleich mehrere Vorzüge von Graphdatenbanken zum Tragen.

Mit ihnen ist es einfach, bestehende Datensätze mit weiteren Daten anzureichern. Insbesondere auch dann, wenn diese Daten beim Anlegen der Kundendatenbank noch gar nicht vorgesehen waren. Das kann beispielsweise beim Anschließen einer neuen Datenquelle vorkommen, einem gerade bei MDM-Lösungen häufigen Vorgang. Zudem ist die Verarbeitung der Beziehungen und ihrer Eigenschaften deutlich flexibler und effizienter möglich als mit relationalen und den anderen NoSQL-Datenbanken. Durch ihre Optimierung auf Graphdaten ist die Verarbeitung von Abfragen, Datenänderungen und Analysen, die der inneren Graphen-Logik der Daten folgen, performanter. Für den Endanwender bedeutet das mehr Komfort und Tempo und für die IT-Abteilung geringere Kosten, wenn der Durchsatz eines MDM-Systems gesteigert wird. Für Entwickler einer MDM-Software ermöglichen die auf Graphen optimierten Abfragesprachen von Graphdatenbanken die Implementierung der Anwendung mit erheblich vereinfachten Datenbankabfragen. So wird nicht nur der Entwicklungsaufwand reduziert, weil keine komplizierten JOIN-Statements zur Auswertung der Graphdaten entwickelt werden müssen, sondern auch die Aufwände für Wartung und Weiterentwicklung sinken.

Nicht zuletzt gibt es für die Entwicklung von Anwendungen, die Graphdaten verarbeiten, beispielsweise mit TinkerPop ein Java-Framework zur Nutzung von Graphdatenbanken. Es bietet eine einheitliche Programmierschnittstelle zur Modellierung und Abfrage von Graphdaten und ermöglicht so den Einsatz verschiedener Graphdatenbanken im Backend, ohne die eigentliche Programmlogik ändern zu müssen [TinkerPop].

Im Java-Umfeld sind die Open-Source-Graphdatenbanken Neo4j (GPLv3 CE sowie AGPLv3 Advanced und Enterprise Edition) und OrientDB (Apache-Lizenz 2.0) verbreitet. Ebenfalls unter der attraktiven Apache-Lizenz 2.0 verfügbar ist die Open-Source-Graphdatenbank Titan, die wegen ihrer hohen Skalierbarkeit (Clustering) und der engen Integration mit Apache Hadoop und dem TinkerPop-Framework hervorzuheben ist. Weitere, in C++ geschriebene Optionen sind ArangoDB und DEX/Sparksee, die letztere allerdings als Closed Source. Daneben gibt es weitere, kommerzielle Graphdatenbanken; bitte beachten Sie dazu die Links unter [GraphDB].

Die technische Infrastruktur für die Speicherung von Graphen ist also auch im Java-Umfeld problemlos aufzubauen. Um diese möglichst performant in Anwendungen einsetzen zu können, bedarf es aber eines Verfahrens, das komplexe Graphenstrukturen so aufteilt, dass Anfragen an die Datenbank verteilt effizient bearbeitet werden können.

Graphpartitionierung – Der Schlüssel zur Performance

Wenn wir von Graphen sprechen, sind wir in praktischen Anwendungen wie Navigation, Kundendatenmanagement, Sozialer-Netzwerk-Analyse, aber auch wissenschaftlich-techni-

schen Kontexten wie Strömungssimulation, biochemischen Analysen oder Sensornetzwerken schnell in Größenordnungen von mehreren Hundert Gigabyte oder gar im Terabyte-Bereich unterwegs. Da heutige Hardware noch nicht in der Lage ist, Graphdatenbanken in diesen Größenordnungen im Arbeitsspeicher zu halten, ist Parallelverarbeitung auf mehreren Prozessorkernen und/oder Prozessoren mit jeweils eigenem Arbeitsspeicher notwendig. Eine entsprechend große Rolle spielt die möglichst performante Gestaltung der Software, die aus den Graphdaten Mehrwert ziehen soll.

Wie häufig in der Parallelverarbeitung ist der leistungsbegrenzende Faktor die Kommunikation zwischen den Prozessoren beziehungsweise Prozessorkernen. Wenn ein Prozessor auf Daten eines anderen warten muss, vergeht wertvolle Zeit. Auf Graphdatenbanken übertragen bedeutet das, mit zwei Beschränkungen möglichst effizient zurecht zu kommen:

- Erstens müssen die Knoten des Graphen möglichst gleichmäßig auf die verfügbaren Prozessoren verteilt werden, um Imbalancen zu vermeiden.
- Zweitens muss die Kommunikation zwischen Prozessoren minimiert werden. Da Kanten in diesem Kontext häufig Kommunikation modellieren, bedeutet dies bei der Aufteilung des Graphen – der Graphpartitionierung –, dass so wenig wie möglich Kanten „durchschnitten" werden.

Nun ist die Aufteilung des Graphen etwas, das kein Java-Programmierer selbst in seiner Software umsetzen muss. Es gibt fertige Algorithmenbibliotheken zur Graphpartitionierung, die als Service aufgerufen werden und so in ein Programm integriert werden können. Worauf es allerdings ankommt, ist die Leistung des Algorithmus.

Da das allgemeine Graphpartitionierungsproblem NP-schwer ist, werden in der Praxis häufig Heuristiken verwendet, um Partitionen von Graphen zu erzeugen. Eine sehr erfolgreiche Heuristik ist das Mehrschichtverfahren, welches ausgehend vom Eingabegraphen zunächst durch Kontraktion von zum Beispiel Matchings eine Hierarchie von immer kleineren Graphen erzeugt. Ein Matching ist dabei eine Teilmenge der Verbindungen des Graphen, die sich keinen Endpunkt teilen. Ein Beispiel so einer Kontraktion finden Sie in Abbildung 1.

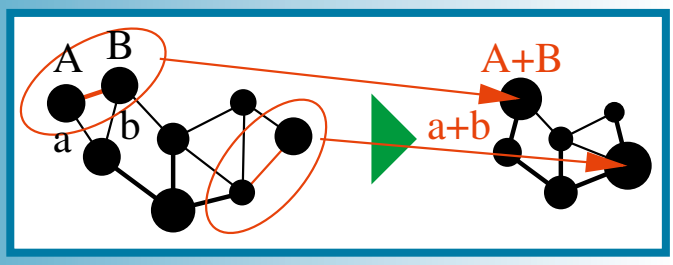


Abb. 1: Die Kontraktion von Matchings im Zuge des Mehrschichtverfahrens hilft bei Schaffung einer Hierarchie kleinerer Graphen

Die kleineren Graphen der Hierarchie haben in der Regel ähnliche Eigenschaften wie der Eingabegraph. Der kleinste Graph in dieser Folge wird dann mithilfe einer Heuristik initial partitioniert, das heißt, die Knoten des Graphen werden Blöcken zugewiesen. Anschließend wird die Kontraktion schrittweise rückgängig gemacht, indem eine Partitionierung auf die nächste, feinere Ebene in der Hierarchie übertragen wird. Dies ist der Haupttrick des Verfahrens: Der Wert der Zielfunktion des Optimierungsproblems auf der gröberen Ebene entspricht genau dem Wert der übertragenen Lösung auf der feineren Ebene.

Zusätzlich versucht ein lokaler Verfeinerungsalgorithmus jeweils nach dem Lösungstransfer auf einer Ebene, Knoten zwischen den Blöcken zu bewegen, um die Zielfunktion, also zum Beispiel den Kantenschnitt, zu verbessern (s. Abb. 2).

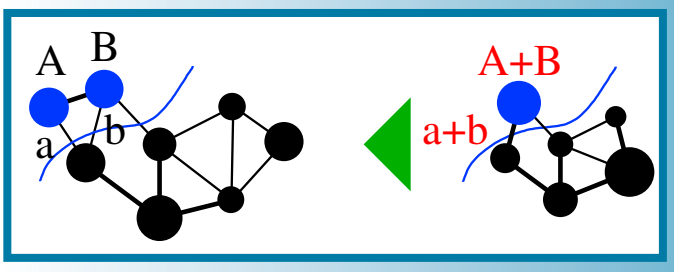


Abb. 2: Der per Kontraktion verkleinerte Graph lässt sich leichter partitionieren und danach wieder in seinen Ursprungszustand zurückversetzen

Im Rahmen eines Forschungsprojekts am Karlsruher Institut für Technologie (KIT) wurden bekannte Mehrschichtverfahren auf den Prüfstand gestellt und mit dem Ziel, eine höhere Partitionierungsqualität zu erreichen, signifikant verbessert. Daraus entstanden eine Reihe von Verfahren, die zu verbesserter Lösungsqualität führen. Dies beinhaltet verschiedene Kontraktionsheuristiken, flussbasierte Methoden, verbesserte lokale Suchen, wiederholte Versuche ähnlich zu solchen Verfahren, die in Mehrgitterlösern verwendet werden, einen verteilt evolutionären Algorithmus und einen neuen Algorithmus für den Fall, dass stark balancierte Partitionen benötigt werden. Eine grobe Übersicht über die verschiedenen Beiträge des Projekts liefert Abbildung 3.

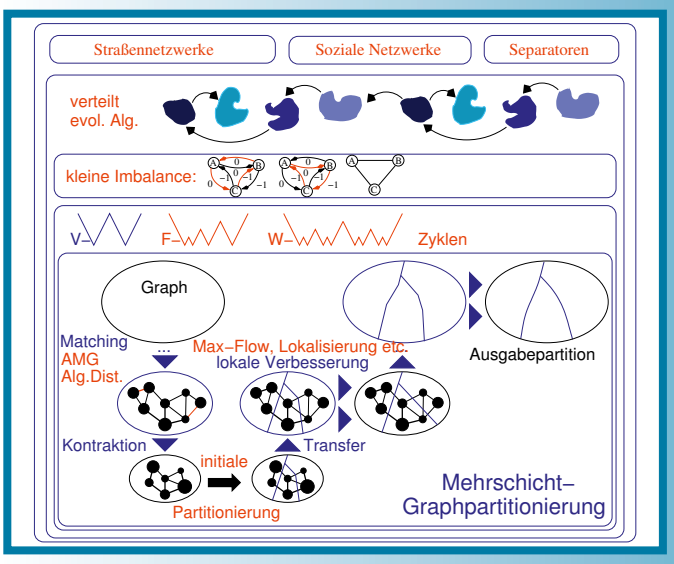


Abb. 3: Der Hochleistungs-Graphpartitionierer KaHIP nutzt eine Reihe optimierter Algorithmen, um die Qualität der Ausgabepartitionen zu verbessern

KaHIP – Hochleistungs-Graphpartitionierung made in Germany

Das Ergebnis des von Uniserv mit einem Forschungspreis ausgezeichneten Projekts ist der Open-Source-Graphpartitionierer KaHIP (für Karlsruhe High Quality Partitioning). Das KaHIP-Framework wurde Benchmark-getrieben entwickelt, das heißt,

jede Komponente des Algorithmus wurde mit dem Ziel entwickelt, in gängigen Leistungstests besser abzuschneiden als die bisherigen Spitzenreiter.

Die dazu angewandte Methodik nennt sich „Algorithm Engineering“, die im Wesentlichen einen Kreislauf aus Algorithmendesign, Analyse, Implementierung und experimenteller Untersuchung beinhaltet. Dadurch wurden Verbesserungen in den Bereichen der Vergröberungs- und Verfeinerungsmethoden, global gesteuerter Suchen und im Bereich der Meta-Heuristiken erzielt. Anders als man zunächst vermuten würde, kamen bei der Arbeit zahlreiche „einfachere“ Graphalgorithmen zur Anwendung, obwohl das Problem NP-schwer ist. Dazu gehörten zum Beispiel gewichtete Matchings, Breitensuchen, dominierende und unabhängige Mengen, maximale Flüsse, starke Zusammenhangskomponenten oder kürzeste Wege sowie die Detektion von negativen Kreisen in gerichteten Graphen.

Insgesamt ist so ein System entstanden, das fast alle Einträge im Walshaw-Benchmark verbessern oder reproduzieren konnte. Dabei handelt es sich um einen anerkannten Performancetest, der alle weltweit verfügbaren Graphpartitionierer standardisiert miteinander vergleicht. Im Teilwettbewerb Graphpartitionierung des zehnten DIMACS-Implementierungswettbewerbs 2012 zum Clustern und Partitionieren von Graphen lieferte KaHIP ebenfalls die besten Ergebnisse.

KaHIP wurde Anfang 2013 als Open-Source-Algorithmus zur allgemeinen Nutzung freigegeben [KaHIP]. Er ist in C++ geschrieben, lässt sich aber per JNI über seine Programmierschnittstelle in Java-Projekte einbinden (siehe Kasten). Die freie Verfügbarkeit unter GPL 3.0-Lizenz hat binnen kurzer Zeit dafür gesorgt, dass KaHIP in kommerziellen wie wissenschaftlichen Projekten zum Einsatz kommt, so etwa in der Planung von Verkaufsterritorien, in der Routenplanung und der Kapazitätsmodellierung von parallel laufenden Maschinen in der Halbleiterherstellung. Uniserv als Stifter des Forschungspreises am KIT sieht, wie eingangs geschildert, Einsatzmöglichkeiten für die hoch-performante Analyse von Kundendaten im „Big Data“-Maßstab.

Kurzanleitung zur Einbindung von KaHIP in Java-Projekte

KaHIP ist Gegenstand aktueller Forschung und Weiterentwicklung und als solches zumindest bisher nicht Teil eines Java-APIs, das sich direkt in Java-Projekten nutzen ließe. Das heißt aber nicht, dass es mit einigen Vorbereitungen nicht ginge.

KaHIP ist eine C++-Bibliothek und liegt als Source-Code-Paket vor [GitHub]. Daher verlangt die Einbindung des Graphpartitionierers in Java-Projekte die Nutzung des Java Native Interface (JNI), mit den bekannten Vor- und Nachteilen (hohe Performance des nativen Codes vs. Plattformabhängigkeit). Die im Paket enthaltenen Build-Skripte sind zwar prinzipiell auf Windows übertragbar, aber auf Linux als Plattform ausgelegt. Wir setzen im Folgenden daher Linux als Plattform voraus.

Konkret besteht die Integration von KaHIP in ein Java-Projekt dann aus den folgenden Schritten:

1. Zunächst muss die KaHIP-Bibliothek auf der Zielplattform gemäß der Anleitung im Source-Code-Paket mitübersetzt werden. Dazu müssen folgende Programme beziehungsweise Bibliotheken auf dem System bereits vorhanden sein: SCons (`http://www.scons.org/`), argtable (`http://argtable.sourceforge.net/`) und OpenMPI (`http://www.open-mpi.org/`). Als Compiler wird der C++-Compiler „g++“ der GNU Compiler Suite mindestens in Version 4.5 vorausgesetzt. Danach liegen zum Beispiel unter Linux im Unterverzeichnis „deploy“ unter anderem das Programm „kaffpa“, die Header-Datei „kaHIP_interface.h“ und die dazugehörige dynamische Bibliothek „libkahip.so“. Diese stellt den Graphpartitionierer KaFFPa zur Verfügung.
2. Mit den Kommandozeilenprogrammen im Unterordner „deploy“ des Build-Verzeichnisses steht die gesamte Funktionalität von KaHIP zur Verfügung. Input/Output der Graphen und der Partitionierung ist bei diesen jedoch über Dateien realisiert, was bei einer Nutzung aus einem Java-Projekt heraus sehr ineffizient ist. Besser ist es, die dynamische Bibliothek libkahip.so per JNI in das Java-Projekt zu integrieren. Dazu wird ein Wrapper benötigt, der vor allem die Übersetzung von Java-Arrays in C-Arrays für den Aufruf des Partitionierers über die Bibliotheksfunktion `kaffpa()` sowie die Rückgabe des Ergebnisses in geeigneter Form bewerkstelligt. Dieser Wrapper wird dann per JNI eingebunden. Ein Beispiel für einen solchen Wrapper ist im Source-Code-Paket enthalten. Im Beispiel besteht der Java-Teil des Wrappers aus zwei Klassen: `KaHIPWrapper`, dem eigentlichen Wrapper, und `KaHIPWrapperResult`, die nach Aufruf von KaFFPa das Ergebnis der Partitionierung enthält.
3. Der C++-Anteil des Wrappers selbst muss wie bei JNI üblich zunächst zu einer dynamischen ladbaren Bibliothek übersetzt werden (siehe „makeWrapper.sh“ im Quellcode des Beispiel-Wrappers). Danach kann KaFFPa direkt aus dem Java-Projekt heraus genutzt werden, wie in Listing 1 dargestellt. Die Argumente haben dieselbe Bedeutung wie beim nativen `kaffpa`-Aufruf, wie er in der Dokumentation des KaHIP-Quellcodes beschrieben ist. Das zurückgegebene Objekt vom Typ `KaHIPWrapperResult` enthält dann das Ergebnis der Partitionierung in einem Integer-Array, das für jeden Knotenindex die Nummer des Blocks angibt, dem dieser Knoten zugeordnet wurde (also die eigentliche Partitionierung), sowie die Größe des gesamten Kantenschnitts. Achtung: Damit der JNI-Aufruf erfolgreich ist, müssen alle benötigten Bibliotheken (siehe Punkt 1) inklusive libkahip.so im Suchpfad (z. B. durch Definition von `LD_LIBRARY_PATH`) zu finden sein.

```
/* ... Initialisierung der Variablen, die den Graphen beschreiben:
   n, vwgt, xadj, adjcwgt, adjncy ... */

/* ... Initialisierung der Variablen, die die Parameter der
   Partitionierung bestimmen: nparts, imbalance, suppress_output,
   seed, mode ... */

KaHIPWrapperResult result = KaHIPWrapper.kaffpa(
  n, vwgt, xadj, adjcwgt, adjncy, nparts, imbalance,
  suppress_output, seed, mode);
int edgecut = result.getEdgeCut();
int[] part = result.getPart();

/* ... weiterer Code, der dann Gebrauch von
   der Partitionierung macht ... */
```

Listing 1: Nutzung von KaFFPa direkt aus dem Java-Projekt heraus

Fazit

Durch ihre breiten Einsatzmöglichkeiten und ihr großes Leistungsvermögen sind Graphdatenbanken schon heute eine der wichtigsten Unterkategorien von NoSQL-Datenbanken. Die Verfügbarkeit besonders leistungsfähiger Graphpartitionierer wie KaHIP, die zudem noch Open Source sind, macht die Nutzung von Graphdaten außerdem für eine größere Bandbreite von Anwendungen attraktiv. Aus Sicht der Autoren dieses Beitrags spricht auch für Java-Entwickler nichts mehr dagegen, sich mit der Materie zu befassen und mit den frei verfügbaren Tools eigene Anwendungen zu projektieren.

Links

[DIMACS] P. Sanders, Ch. Schulz, High Quality Graph Partitioning, 2012, Center for Discrete Mathematics and Theoretical Computer Science, Rutgers, New Jersey, `http://www.cc.gatech.edu/dimacs10/papers/[01]-high_quality_graph_partitioning_final.pdf`

[GitHub] KaHIP bei GitHub, `https://github.com/schulzchristian/KaHIP`

[GraphDB] Übersichten zu Graphdatenbanken bei Wikipedia und DB-Engines von solid IT, `http://en.wikipedia.org/wiki/Graph_database` und `http://db-engines.com/de/ranking/graph+dbms`

[JNI] Java Native Interface Specification bei Oracle, `http://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html`

[KaHIP] Karlsruhe High Quality Partitioning, mit zahlreichen weiterführenden Literaturhinweisen, `http://algo2.iti.kit.edu/documents/kahip/`

[TinkerPop] Homepage, An Open Source Graph Computing Framework, `www.tinkerpop.com`

[Uniserv] Homepage zu Smart Customer MDM, `http://www.uniserv.com/loesungen/smart-customer-mdm/`

[Walshaw] `http://staffweb.cms.gre.ac.uk/~wc06/partition`

Dr. Heiko Papenfuß hat sich in seinem Physikstudium und seiner Promotion mit Softwareentwicklung in verschiedenen Programmiersprachen und der Systemadministration von Unix/Linux-Systemen beschäftigt. Das Management und die Integration heterogener Entwicklungsumgebungen und der Betrieb hochverfügbarer SaaS-Lösungen bilden die Schwerpunkte seiner Arbeit bei der Uniserv GmbH. Uniserv entwickelt Lösungen für Smart Customer MDM und vereint Datenqualitätssicherung und Datenintegration zu einem ganzheitlichen Ansatz.
E-Mail: heiko.papenfuss@uniserv.com

Prof. Dr. Peter Sanders studierte und promovierte in Karlsruhe und war dann sieben Jahre am Max-Planck-Institut für Informatik in Saarbrücken. Seit 2004 hat er einen Lehrstuhl für Algorithmen an der Universität Karlsruhe (jetzt KIT). Er beschäftigt sich mit Algorithmen für große Datenmengen in Theorie und Praxis, insbesondere auch mit Graphenalgorithmen wie Routenplanung oder Graphpartitionierung. Für seine Arbeiten erhielt er zahlreiche Auszeichnungen, darunter den Leibnizpreis der DFG 2012.
E-Mail: sanders@kit.edu

Dr. Christian Schulz absolvierte ein Parallelstudium der Mathematik und Informatik an der Universität Karlsruhe und promovierte anschließend in der Gruppe von Professor Sanders auf dem Gebiet der Graphpartitionierung. Er erhielt unter anderem den Preis für den besten Informatikabschluss des akademischen Jahres 2009/2010 am KIT sowie den Uniserv-Forschungspreis für seine Dissertation.
E-Mail: christian.schulz@kit.edu