View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Institute of Transport Research:Publications

Vergleich von Partitionierungsmethoden für dünnbesetzte Matrizen, die den Hamiltonoperator von Spinketten repräsentieren

Rebekka-Sarah Hennig

Geboren am 14. August 1994 in Bergisch-Gladbach

13. Oktober 2017

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Marc Alexander Schweitzer

Zweitgutachter: Dr. Jonas Thies

INS und DLR

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Inhaltsverzeichnis

1 Einleitung **2**

2 Vorbereitung **4**

2.1 Paralleles Rechnen . . . 4

2.2 Graphpartitionierungsproblem . . . 5

2.3 Spinketten-Matrizen . . . 6

2.3.1 Spin-Spin-Wechselwirkung . . . 6

2.3.2 äußeres Magnetfeld . . . 8

3 Vorpartitionierung **9**

3.1 Implementierung . . . 9

3.2 Sortierungen . . . 11

3.2.1 arithmetisch . . . 13

3.2.2 evbit . . . 14

3.2.3 bitcount . . . 15

3.2.4 evbitcount . . . 16

3.2.5 resabit . . . 17

3.3 Tests . . . 18

4 Vergleich von Standardverfahren **24**

4.1 Multilevel-Graphpartitionierung . . . 25

4.2 Parallelisierung . . . 26

4.2.1 ParMETIS . . . 27

4.2.2 ParHIP . . . 28

4.3 PT-Scotch . . . 28

4.3.1 zweifache rekursive Bipartition . . . 29

4.4 Tests . . . 30

5 Fazit **34**

6 Anhang **35**

1 Einleitung

Beim Graphpartitionierungsproblem geht es darum, die Knoten eines Graphen möglichst gleichmäßig auf k Partitionen aufzuteilen, wobei die Menge der Kanten zwischen verschiedenen Partitionen minimiert werden soll. Dieses Problem findet Anwendung in der parallelen Berechnung von Programmen. Um Matrix-Vektor-Produkte dünnbesetzter Matrizen parallel zu berechnen, werden die Zeilen der Matrix und der Vektoren über die Prozessoren verteilt und die Prozessoren müssen zum Informationsaustausch miteinander kommunizieren, was wertvolle Zeit kostet. Die benötigte Kommunikation wird verringert, indem die Verteilung einer guten Partition des durch die Matrix induzierten Graphen entspricht. Hier wird dann das Graphpartitionierungsproblem betrachtet. Da es NP vollständig ist, suchen die Algorithmen nur nach einer näherungsweisen Lösung des Problems.

Die dünnbesetzten Matrizen, die wir betrachten, kommen aus der Quantenmechanik. Sie stellen den Hamiltonoperator des Spin-$\frac{1}{2}$ Heisenberg-Modells dar und beschreiben Übergänge zwischen Spinketten-Zuständen, die zur Betrachtung von magnetischen Phänomenen in Festkörpern (besonders in Isolatoren) dienen. Sie gehören zur Gruppe der skalenfreien Graphen. Diese kommen oft in Netzwerkanwendungen vor und sind nicht leicht zu partitionieren. Die Anzahl der Kanten zu jedem Knoten variiert sehr stark, was sie von sehr strukturierten Graphen unterscheidet, die beispielsweise bei der Lösung von partiellen Differentialgleichungen auftreten.

Bei den Spinketten-Matrizen können wir einige für die Partitionierung gute Strukturen wie unabhängige Subgraphen erkennen. Anhand dieser wollen wir einige Sortierungen der Graphen bzw. der Matrizen zur Vorpartitionierung finden. Diese sollen den Graphpartitionierern Arbeit abnehmen, sodass wir die Laufzeit dieser Programme verkürzen oder zur Verbesserung der Partitionen beitragen. Setzen wir unsere Spinketten zusätzlich einem äußerem Magnetfeld aus, so kommt es zu Verbindungen zwischen den Subgraphen. Auch für diesen Fall wollen wir eine gute Vorpartitionierung finden.

Für die einzelnen Subgraphen erweist sich die Vorpartitionierung als schwierig. Hier wollen wir verschiedene Algorithmen für die Graphpartitionierung vergleichen, um das Verfahren ausfindig zu machen, dass für unsere Graphen möglichst schnell gute Ergebnisse liefert. Für den Vergleich betrachten wir ParMETIS und PT-Scotch, zwei Standardverfahren für parallele Graphpartitionierung, und ParHIP, ein neueres vielversprechendes Programm, dessen Schwerpunkt in der Partitionierung komplexer Netzwerke liegt, was es für uns so interessant macht.

Die Arbeit ist wie folgt aufgebaut: In Kapitel 2 werden wir genauer auf unsere Problemstellung mit dem Graphpartitionierungsproblem und den Hintergrund unserer Matrizen eingehen. In Kapitel 3 werden wir uns günstige Sortierungen für die Matrizen aus deren Struktur herleiten und deren Auswirkung die Vorpartitionierung auf ParMETIS betrachten. Den

Vergleich der verschiedenen Graphpartitionierern werden wir in Kapitel 4 durchführen.

2 Vorbereitung

Da es in dieser Arbeit um Partitionierungsmethoden für dünnbesetzte Matrizen aus der Quantenmechanik geht, schauen wir uns zunächst genauer die Hintergründe und die Definition des Graphpartitionierungsproblem an. Zusätzlich wollen wir auch auf den physikalischen Hintergrund der Spinketten-Matrizen eingehen.

2.1 Paralleles Rechnen

Beim parallelen Rechnen wollen wir Programme beschleunigen, indem wir gleichzeitig auf verschiedenen Prozessoren im Computer rechnen. Die Idee ist einfach: Dadurch, dass wir Rechenschritte gleichzeitig ausführen anstatt nacheinander, verkürzt sich die Laufzeit des Programms. Zu beachten ist jedoch, dass die Rechenschritte aufeinander aufbauen können oder Informationen, die auf dem einen Prozessor liegen, bei einem anderen gebraucht werden können. Somit müssen die Prozessoren miteinander kommunizieren, was wertvolle Zeit kostet. Um eine optimale Laufzeit zu erreichen, muss man also die Arbeit möglichst gleichmäßig auf die Prozessoren verteilen und gleichzeitig die benötigte Kommunikation zwischen den Prozessoren minimieren.

Betrachten wir iterative Verfahren für lineare Gleichungssysteme oder Eigenwertlöser, so kommt es zu Matrix-Vektor-Produkten $Ax \to y$, wobei wir dünnbesetzten Matrix A betrachten wollen. Dies sind Matrizen mit einer so geringen Anzahl an nicht-null Einträgen, dass man diese Eigenschaft ausnutzen möchte, um Speicher zu sparen oder Algorithmen wie beispielsweise iterative Verfahren effizienter zu machen. Bei der Parallelisierung werden x, y und die Zeilen von A auf gleiche Art auf die Prozessoren verteilt. Das heißt, dass wir für die Berechnung der Einträge von y die Einträge von x zwischen den Prozessoren kommunizieren müssen. Genauer: Liegen die Zeilen der Matrix und die entsprechenden Einträge i und j der Vektoren auf verschiedenen Prozessoren p_i und p_j und die Matrix A hat in der i-ten Zeile an j-ter Stelle einen nicht-null Eintrag, so brauchen wir für die Berechnung der i-ten Stelle des y-Vektors die j-te Stelle des x-Vektors, den wir dann von Prozessor p_j nach p_i kommunizieren müssen. Hierbei kommt uns die Dünnbesetztheit unserer Matrix zu gute.

Ist die Besetzungsstruktur der betrachteten Matrix symmetrisch, dass bedeutet, dass der Matrixeintrag $A_{ij} \neq 0$ genau dann wenn $A_{ji} \neq 0$, so können wir die Matrix ziemlich einfach als Graph darstellen. Ansonsten verwenden wir die Matrix $|A|+|A|^T$ für die Darstellung durch den Graphen. Die Knoten des induzierten Graphen stehen in jedem Fall für die Zeilen der Matrix und die nicht-null Einträge der Matrix werden durch eine Kante zwischen den entsprechenden Knoten dargestellt, wobei Diagonaleinträge nicht betrachtet werden. Verteilen wir den Graphen nun entsprechend der

Matrix über die Prozessoren, so bedeutet eine Kante, die zwischen zwei Knoten i und j von verschiedenen Prozessoren verläuft, dass wir hier einmal den i-ten und den j-ten x-Eintrag kommunizieren müssen. Das heißt, die Anzahl der zu kommunizierenden x-Einträge entspricht 2-mal der Anzahl der bei der Partitionierung „durchtrennten“ Kanten. So kommen wir zum Graphpartitionierungsproblem.

2.2 Graphpartitionierungsproblem

Zunächst einige graphentheoretischen Definitionen, die wir im weiteren noch gebrauchen werden:

G = (V, E) sei ein ungerichteter Graph mit V Menge der Knoten und E $\subset V \times V$ Menge der Kanten. Für eine ungerichtete Kante e $\in$ E zwischen v und w $\in$ V schreiben wir e = $\{v, w\}$. Bei einem gewichteten Graphen haben wir die zwei Kostenfunktionen c: V $\rightarrow \mathbb{R}_{>0}$ Funktion der Knotengewichte und w: E $\rightarrow \mathbb{R}$ Funktion der Kantengewichte gegeben. Wir definieren c(W)=$\sum_{w \in W}$ c(w) für ein W $\subseteq$ V und w(F)=$\sum_{e \in F}$ w(e) für ein F $\subseteq$ E. Der Knotengrad eines Knotens v ist die Anzahl der Kanten e $\in$ E, die v mit einem anderen Knoten verbinden. Eine Adjazenzliste für einen Knoten v ist eine Liste der Nachbarn von v, das heißt, eine Liste der Knoten, die über eine Kante e $\in$ E mit v verbunden sind.

Beim Graphpartitionierungsproblem haben wir einen ungerichteten (gewichteten) Graphen G und ein k $\in \mathbb{N}_{>1}$ gegeben. Gesucht wird eine Partition der Knotenmenge V in k Teilmengen, dh wir suchen $V_1, ..., V_k \subset$ V mit V_i $\neq \varnothing$ für alle i, $V_i \cap V_j = \varnothing$ für alle i $\neq$ j und V = $\cup_{i=1}^{k}$ V_i mit i,j $\in \{1, ..., k\}$.

Um die Knoten gleichmäßig auf die Teilmengen zu verteilen, fordern wir eine Gleichgewichtsbedingung. Bei gewichteten Graphen berücksichtigen wir hierbei die Knotengewichte: $\forall i \in \{1, ..., k\}$: c(V_i) $\leq (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$. $\epsilon \in \mathbb{R}_{\geq 0}$ nennen wir den Ungleichgewichtsparameter. Im Fall des ungewichteten Graphen betrachten wir die Anzahl der Knoten in den Teilmengen: $\forall i \in \{1, ..., k\} : |V_i| \leq (1 + \epsilon)\lceil \frac{|V|}{k} \rceil$. Das maximale Ungleichgewicht berechnen wir als $\frac{max_i |V_i|}{\lceil |V|/k \rceil}$.

Wir wollen zusätzlich, dass unsere Partition eine so genannte Zielfunktion g minimiert bzw. maximiert. Wir erhalten ein Optimierungsproblem. Als Beispiel für eine Zielfunktion können wir uns den Schnitt S=$\{\{v, w\} \in E | v \in V_i, w \in V_j, i \neq j\}$ anschauen und die Menge der Schnittkanten $|S|$ (Edge-Cut) bzw. bei gewichteten Graphen das Gewicht der Schnittkanten w(S)=$\sum_{e \in S} w(e)$ minimieren.

Dies würde auch unserem Problem beim parallelen Rechnen entsprechen: Wir teilen die Knoten und somit die Zeilen der Matrix möglichst gleichmäßig in k Teilmengen auf, die wir auf einzelne Prozessoren legen, und minimieren durch die Zielfunktion den Edge-Cut, der proportional zur benötigten Kommunikation ist.

Eine Verbesserung der Partition bezeichnet sowohl eine Annäherung an das Minimum bzw. Maximum der Zielfunktion (bei uns das Verkleinern des Edge-Cuts) als auch die Minderung des maximalen Ungleichgewichts, wobei wir unser Augenmerk im folgenden vor allem auf ersteres legen.

Wir haben unser Problem also auf das Graphpartitionierungsproblem zurückgeführt. Nun stellt sich noch die Frage, wie genau die Matrizen aussehen, die wir betrachten wollen.

2.3 Spinketten-Matrizen

Wir werden hier nun kurz auf die physikalischen Hintergründe unserer Matrizen eingehen. Für genauere Details siehe [2] [13] [8] [9] [4].

Unsere Matrizen repräsentieren den Hamiltonoperator von Spinketten. Dieser Operator kommt vom Spin-1/2 Heisenberg-Modell, welches ein Spezialfall des n-Vektor-Modells der statistischen Physik ist und bei magnetischen Isolatoren zum Einsatz kommt. Eine mögliche praktische Anwendung der Matrizen liegt in der Simulation von Eigenschaften eines Quantencomputers auf herkömmlichen Rechnern.

Das n-Vektor-Modell dient zur Beschreibung von Phasenübergängen und Magnetismus. Hier werden L n-dimensionale Vektoren auf Gitterpunkte eines d-dimensionalen Gitters gesetzt. In unserem Spezialfall ist n=3 und da wir Spinketten betrachten ist d=1. Dabei wollen wir auch noch periodische Randbedingungen annehmen, sodass die Ketten zu einem Ring geschlossen sind.

Der Spin S_i ist der Eigendrehimpuls des i-ten Teilchens. Mit den Pauli'schen Spinmatrizen $\sigma = (\sigma_x, \sigma_y, \sigma_z), \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ können wir die Spinoperatoren schreiben als $\hat{S}^j = \frac{\hbar}{2}\sigma_j$ mit $\hbar$= planck'sches Wirkungsquantum und j $\in \{x, y, z\}$. Beim Spin-1/2 Heisenberg-Modell können unsere Spins zwei Spinzustände haben. Dies sind die Eigenzustände des $\hat{S}_z$-Operators $\begin{pmatrix} 1 \\ 0 \end{pmatrix} =\uparrow$ und $\begin{pmatrix} 0 \\ 1 \end{pmatrix} =\downarrow$. Hieraus ergibt sich, dass die Spinkette 2^L verschiedene Zustände annehmen kann. Diese können wir auch als Bitpattern der Länge L darstellen, indem wir $\uparrow$ mit 1 und $\downarrow$ mit 0 identifizieren.

2.3.1 Spin-Spin-Wechselwirkung

Der Hamiltonoperator soll die Übergänge zwischen den Zuständen der Spinkette beschreiben, die durch magnetische Wechselwirkungen zwischen den Spins der Kette hervorgerufen werden.

Die magnetischen Wechselwirkungen führen dazu, dass benachbarte Teilchen mit unterschiedlichen Spinzuständen ihre Spins vertauschen können.

Dies geschieht, indem beide Spins auf eins der Teilchen übergehen, sich dort vertauschen und wieder trennen. Wir erhalten also zwei Austauschpfade $|\uparrow,\downarrow\rangle \rightarrow |\uparrow\downarrow,0\rangle \rightarrow |\downarrow,\uparrow\rangle$ und $|\uparrow,\downarrow\rangle \rightarrow |0,\uparrow\downarrow\rangle \rightarrow |\downarrow,\uparrow\rangle$.

Da wir Ferromagnetismus betrachten, ist es für die Zustände außerdem günstig, wenn benachbarte Teilchen gleichgerichtete Spins haben. Das bedeutet, dass sich ein Spinketten-Zustand mit sehr vielen parallel ausgerichteten Nachbarpaaren eher nicht ändert, während einer mit vielen antiparallelen Nachbarpaaren dazu tendiert, sich zu ändern.

Der Hamiltonoperator stellt diese Überlegungen durch die Spinoperatoren da: $\mathcal{H} = -\sum_{i,j} J_{ij}\hat{S}_i\hat{S}_j$. Hierbei ist J_{ij} die Kopplungskonstante zwischen dem i-ten und dem j-ten Spin, die positiv ist für Ferromagnetismus, und $\hat{S}_i$ ist der Spinoperator angewendet auf den Zustand des i-ten Spins.

Mit den Spinflip-Operatoren $\hat{S}^+ = \hat{S}^x + i\hat{S}^y = \hbar\begin{pmatrix}0 & 1\\ 0 & 0\end{pmatrix}$ und $\hat{S}^- = \hat{S}^x - i\hat{S}^y = \hbar\begin{pmatrix}0 & 0\\ 1 & 0\end{pmatrix}$ können wir das Produkt der Spinoperatoren aufschlüsseln zu $\hat{S}_i\hat{S}_j = \hat{S}^z_i\hat{S}^z_j + \hat{S}^x_i\hat{S}^x_j + \hat{S}^y_i\hat{S}^y_j = \hat{S}^z_i\hat{S}^z_j + \frac{1}{2}(\hat{S}^+_i\hat{S}^-_j + \hat{S}^-_i\hat{S}^+_j)$. Die Terme mit den Spinflip-Operatoren stellen dann die Austauschpfade dar, während der $\hat{S}^z$-Operator das Verhältnis der parallelen zu den antiparallelen Nachbarpaaren zählt und damit ein Maß dafür ist, wie sehr die Spinkette dazu tendiert, ihren Zustand nicht zu ändern.

Die Kopplungskonstante J_{ij} ist symmetrisch ($J_{ij} = J_{ji}$), $J_{ii} = 0$ und da wir nur die Wechselwirkungen zwischen nächsten Nachbarn betrachten, ist $J_{ij} = 0$ für $j \neq i+1$. Es folgt für den Hamiltonoperator:

$\mathcal{H} = -\sum_{i=0}^{N} J^z_{i,i+1}\hat{S}^z_i\hat{S}^z_{i+1} + \frac{1}{2}J^{xy}_{i,i+1}(\hat{S}^+_i\hat{S}^-_{i+1} + \hat{S}^-_i\hat{S}^+_{i+1})$, wobei der L-te Spin der 0-te Spin ist auf Grund der periodischen Randbedingung.

Die Matrix, die den Hamiltonoperator repräsentiert, erhalten wir nun durch die Gleichung:

$H_{ij} = \langle Z^i|\mathcal{H}|Z^j\rangle = \sum_{m,n} J_{m,n}\langle Z^i_m|\hat{S}_m|Z^j_m\rangle\langle Z^i_n|\hat{S}_n|Z^j_n\rangle \frac{\prod_{a=0}^{N-1}\langle Z^i_a|Z^j_a\rangle}{\langle Z^i_n|Z^j_n\rangle\langle Z^i_m|Z^j_m\rangle}$, wobei Z^i der i-te mögliche Spinketten-Zustand ist (in der arithmetischen Reihenfolge der Bitpattern gezählt) und wir die Konstanten vernachlässigen, das heißt $J_{i,i+1} = 1$ und wir vernachlässigen das $\frac{\hbar}{2}$ der Spinoperatoren. Der hintere Bruch garantiert uns, dass der $\hat{S}^z$-Operator nur bei den Diagonalelementen einen Beitrag leisten und die Spinflip-Operatoren nur das Vertauschen von höchstens einem Nachbarpaar gleichzeitig erlauben.

Dies ergibt dann genau unsere Spinketten-Matrizen, die wir $spin\langle L\rangle$ nennen werden, wobei L die Anzahl der Spins angibt.

Anstatt die volle Matrix zu betrachten, können wir uns auch auf die Bitpattern mit einer festen Anzahl $0 \leq NUp \leq$ L an 1, das heißt einem festen Bitcount, beschränken. Diese erfassen $\binom{N}{NUp}$ Zustände und sind voneinander unabhängig, da die Spinflip-Operatoren nur die Positionen der 1 verändert, aber nicht ihre Anzahl. Die Matrizen mit einer symmetrischen Anzahl an 1

und 0 nennen wir $spinSZ\langle L\rangle$.

2.3.2 äußeres Magnetfeld

Wir können nun auch noch ein äußeres Magnetfeld betrachten, in dem sich unsere Spinkette befindet. Diese wird häufig in z-Richtung betrachtet. Bei unseren Matrizen haben wir ein B-Feld in z- und in x-Richtung. Wir erhalten für den Hamiltonoperator:

$\mathcal{H} = -\sum_{i=0}^{N} J_{ii+1}\hat{S}_i^z\hat{S}_{i+1}^z + \frac{1}{2}(\hat{S}_i^+\hat{S}_{i+1}^- + \hat{S}_i^-\hat{S}_{i+1}^+) + \sum_{i=0}^{N-1} B_z\hat{S}_i^z + B_x\hat{S}_i^x$

Da wir noch immer die Spin-Spin-Wechselwirkungen haben, kommt es wieder zur ersten Summe. Die zweite Summe entsteht durch die Wechselwirkung der Spins mit dem Magnetfeld. Das Magnetfeld in z-Richtung gibt uns keinen Übergang zwischen verschiedenen Zuständen, sondern beschreibt wieder, wie günstig der betrachtete Zustand ist. Bei der Wechselwirkung mit dem Magnetfeld kommt es darauf an, dass die Spins in die selbe Richtung zeigen wie das Magnetfeld. Wir berechnen also die Differenz zwischen Spins, die parallel und antiparallel zum Magnetfeld ausgerichtet sind.

Das Magnetfeld in x-Richtung führt wieder zu Zustandsänderungen, da sich die Spins über diesen Weg umdrehen können. Dadurch kommt es zu spontanen Spin-Umkehrungen, die sich auch in unserer Matrix bemerkbar machen. Wir können so zwischen Spinketten-Zuständen, die sich nur durch einen Spin unterscheiden, wechseln.

Auch hier werden bei der Berechnung wieder die Konstanten vernachlässigt, in dem Sinne, dass $B_x = B_z = 1$ und wir das $\frac{\hbar}{2}$ wieder nicht berücksichtigen. Unsere entstehenden Matrizen nennen wir $spinB\langle L\rangle$.

3 Vorpartitionierung

Um die Graphen zu partitionieren, wird ParMETIS verwendet. Dies ist ein Standardverfahren für Graphpartitionierung, das in Kapitel 4 näher beschrieben wird. Um die Partitionierung unserer Graphen zu verbessern, wollen wir mit einer Vorpartitionierung der Graphen arbeiten. Diese soll dem Graphpartitionierer einiges an Rechenarbeit ersparen und eine gute Vorlage geben, sodass der Algorithmus schneller zu einem Ergebnis kommt bzw. eine bessere Partition liefert.

Die Vorpartitionierung werden wir realisieren, indem wir die Zeilen unserer Matrizen umsortieren, bevor wir die induzierten Graphen an den Algorithmus übergeben. Geben wir einen Graphen an ParMETIS, so werden die Knoten zu Beginn so auf die Prozessoren aufgeteilt, dass Prozessor p die Knoten $p\frac{nnodes}{P}, ..., (p+1)\frac{nnodes}{P}$ erhält, mit $p \in \{0, ..., P\}$, P Anzahl der Prozessoren und nnodes Anzahl der Knoten. Diese Aufteilung der Knoten auf die Prozessoren nennen wir die Inputpartition. Durch die Umsortierung der Zeilen und der damit verbundenen Umnummerierung der Knoten des induzierten Graphen, kommt es automatisch zu einer Umverteilung der Knoten auf den Prozessoren. Das bedeutet, dass eine Umsortierung der Matrixzeilen zu einer Vorpartitionierung der Knoten auf die Prozessoren führt.

Die Frage ist nun noch, wie wir die Umsortierung der Matrix im Computer umsetzen wollen und was erfolgversprechende Sortierungen für unsere Matrizen sind.

3.1 Implementierung

Wir haben die zwei Zeilenfunktionen SpinChainSZ und SpinChain gegeben, die uns die Zeilen der Matrizen erzeugen. Die SpinChain-Funktion ist hierbei für die Matrizen zu verwenden, bei denen wir alle 2^L Zustände betrachten, und wir können hier ein externes Magnetfeld in die Berechnung mit einbeziehen. Bei der SpinChainSZ-Funktion betrachten wir nur die Zustände mit einer bestimmten Anzahl NUp an $\uparrow$ und kümmern uns nur um die Spin-Spin-Wechselwirkung.

Wollen wir nun eine dieser Matrizen beispielsweise für eine Matrix-Vektor-Multiplikation verwenden, so wird die entsprechende Matrix durch diese Zeilenfunktionen im Algorithmus erzeugt. Dafür wird die Funktion zunächst mit dem Übergabeparameter row=-2 aufgerufen, um Voreinstellungen zu tätigen. Diese sind die Anzahl der betrachteten Spins L, die periodischen Randbedingungen und das Magnetfeld Bx und Bz bzw. die Anzahl der $\uparrow$ NUp. Mit row=-1 werden einige Informationen über die entstehende Matrix gespeichert, wie beispielsweise die Anzahl der Zeilen und Spalten. Anschließend wird über die Zeilen der Matrix iteriert.

Übergeben wir unseren Zeilenfunktionen row=i mit i$\in \{0, ..., Ns-1\}$ mit Ns Anzahl der betrachteten Zustände, so geben sie uns die Arrays cols und

vals zurück. In dem vals-Array stehen die Einträge der i-ten Zeile und in dem cols-Array die Spalte, in die die Einträge gehören, sodass vals[j] in die i-te Zeile und cols[j]-te Spalte der Matrix gehört. Die einzelnen Einträge speichern wir dann in einer Matrix ab. Da die Zeilen voneinander unabhängig berechnet werden, funktioniert dies auch parallel sehr gut.

Nun wollen wir aber die Matrix umsortieren. Um nicht erst die originale Matrix zu erstellen und dann in einem zweiten Schritt die Zeilen und Spalten umzusortieren, setzen wir direkt beim Erzeugen der Matrix an, um direkt zur umsortierten Matrix zu gelangen. Hierbei ist es hilfreich, dass wir uns die einzelnen Zeilen der Matrix erstellen können.

Bei der Umsortierung der Matrix A zur Matrix B überlegen wir uns, welche Zeile aus B aus welcher Zeile in A resultieren soll. Nummerieren wir die Zeilen von oben nach unten von 0 bis Ns-1 durch für beide Matrizen, so erhalten wir eine bijektive Abbildung $f : \{0, ..., Ns - 1\} \to \{0, ..., Ns - 1\}$, die uns die Nummerierung der Matrix B in die Nummerierung der Matrix A übersetzt. Damit es bei einer Umsortierung bleibt und wir nicht die Bedeutung der Matrix ändern, müssen die Spalten auf gleiche Weise vertauscht werden wie die Zeilen, sodass $A_{f(i),f(j)} = B_{i,j}$. Betrachten wir die Umkehrabbildung f^{-1} so gilt: $A_{i,j} = B_{f^{-1}(i),f^{-1}(j)}$. Diese beiden Funktionen benötigen wir für unser Programm und haben sie hier als die beiden Arrays sort (f) und sortback (f^{-1}) gespeichert.

Die umsortierte Matrix B erzeugen wir nun aus der ursprünglichen Matrix A, indem wir die Abbildungen nutzen. Wollen wir die i-te Zeile von B berechnen, so sagt uns sort, welcher Zeilennummer das in der Matrix A entspricht und die Zeilen-Funktion berechnet dann diese Zeile der ursprünglichen Matrix. An dem Werte-Array ist nichts mehr zu tun, aber die Spaltennummern entsprechen denen in A, weshalb wir an dieser Stelle sortback benötigen, um die Spaltennummer der umsortieren Matrix zu erhalten. So können wir direkt unsere umsortierte Matrix erzeugen.

Auf eine parallele Implementation verzichten wir an dieser Stelle und konzentrieren uns stattdessen darauf, verschiedene Sortierungen an verhältnismäßig kleinen Matrizen zu erproben.

Wir benutzen den Code $spin_pre - partition_graph.cpp.in$, welcher uns unsere Matrix direkt als Graph in ein Textfile im .graph-Format schreibt, das heißt, wir speichern die Adjazenzliste des Graphen. Hierbei brauchen wir die Einträge auf der Diagonalen nicht zu betrachten und auch die Werte aus vals interessieren uns nicht, da bei den Graphen die Kanten keine Gewichte haben sollen und es keine Schleife geben soll, das bedeutet, keine Kante, die einen Knoten mit sich selbst verbindet. Diese Dateien können wir dann über die Kommandozeile an ParMETIS übergeben und können so vergleichen für welche Sortierung Parmetis besseren Edge-Cut liefert bzw. eine kürzere Laufzeit aufweist.

Die Stelle im Code, mit der wir unsere Textdatei erstellen, ist folgende:

```
// iterieren ueber Zeilen von B
for (phist_lidx i = 0; i < nlocal; i++)
{
  fprintf(fp,"\n");
  // umwandeln in Zeilennummer von A
  phist_gidx row = sort[ilower+i];
  phist_lidx row_nnz = 0;
  // Zeile von A berechnen
  SpinChain(row,&row_nnz,cols.data(),vals.data(), NULL);
  for(phist_lidx j = 0; j < row_nnz; j++) {
    // im Graphen betrachten wir keine Diagonaleintraege
    if(cols[j]!=row){
      // umwandeln in Spaltennummer von B
      fprintf(fp, "     %d", sortback[cols[j]]+1);
      count++;
    }
  }
}
```

Es bleibt noch zu bestimmen, wie die Funktionen f und f^{-1} aussehen sollen.

3.2 Sortierungen

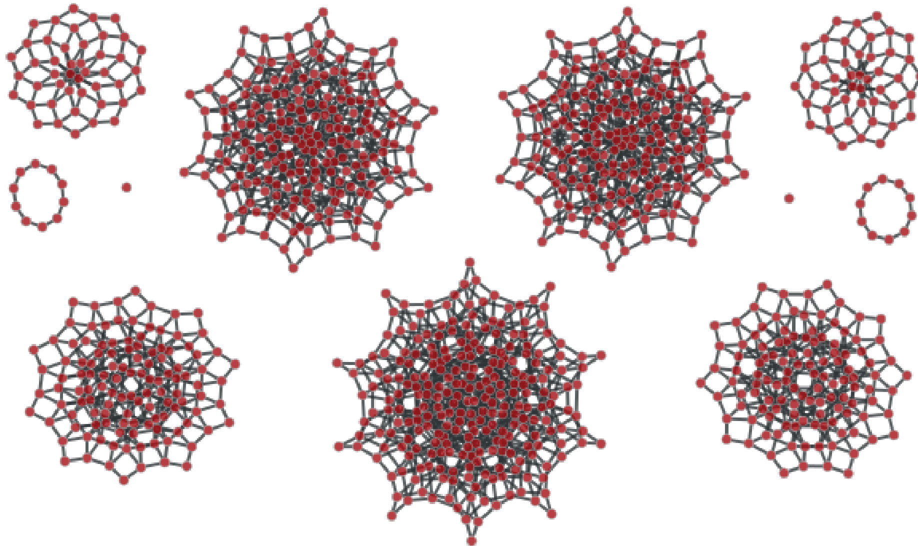

Abbildung 1: Visualisierung des Graphen der Matrix spin10. Die unabhängigen Subgraphen bestehen aus Knoten des selben Bitcounts.

Um uns gute Sortierungen für die Matrizen zu überlegen, müssen wir uns zunächst mit dem Aussehen der induzierten Graphen beschäftigen, um so auffällige Strukturen zu erkennen, die wir für die Vorpartitionierung ausnutzen können.

Betrachten wir die Graphen ohne Magnetfeld, so fällt auf, dass es nur Kanten zwischen Knoten mit dem selben Bitcount, also mit der selben Anzahl an 1 in der Darstellung als Bitpattern, gibt. Dies kommt daher, dass wir bei den Zustandsänderungen durch die Spin-Spin-Wechselwirkungen Spins nur vertauschen können und somit die Anzahl der $\uparrow$ bzw. $\downarrow$ nicht verändert wird. Da die Bitpattern Länge L haben und alle Stellen unabhängig voneinander entweder den Wert 0 oder 1 annehmen können, kann unser Bitcount die Werte 0 bis L annehmen. Wir erhalten also L+1 unabhängige Subgraphen. Für die Größe der Subgraphen gilt die Formel $\binom{L}{NUp}$, mit NUp = Bitcount der Knoten im Subgraphen. Die Anzahl ihrer Knoten variiert also sehr stark. In Abbildung 1 sehen wir einen solchen Graphen für L=10.

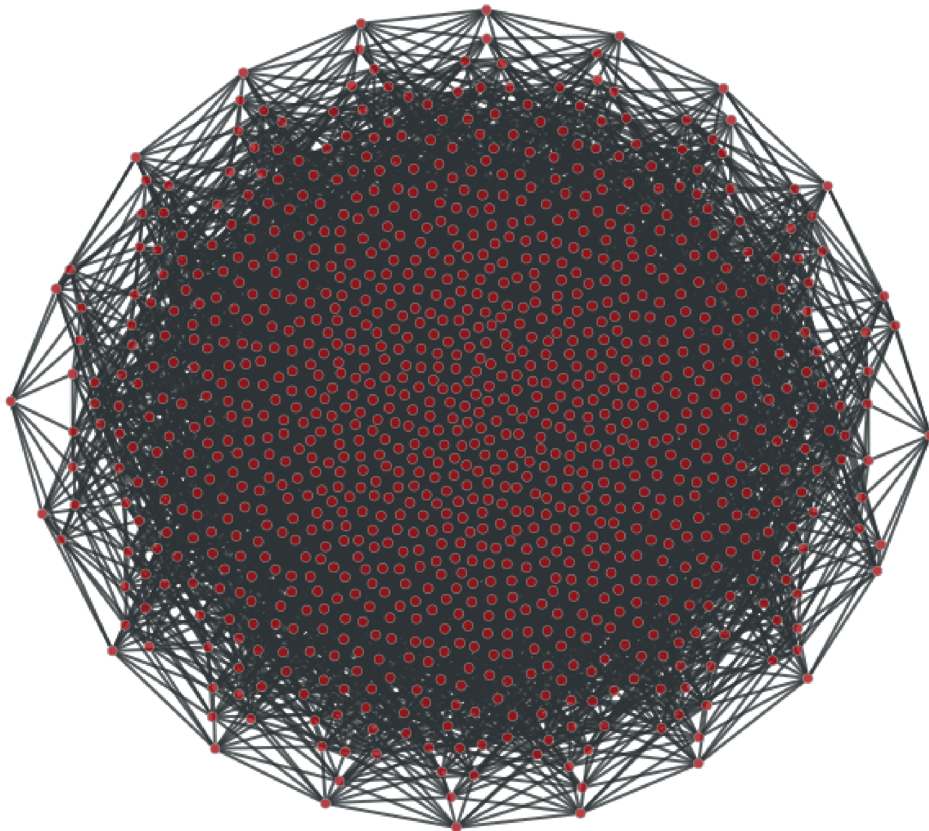

Abbildung 2: Visualisierung des Graphen der Matrix spinB10

Nehmen wir nun das Magnetfeld hinzu, so kommt es zu spontanen Spin-Umkehrungen, die Kanten zwischen Knoten unterschiedlichem Bitcounts mit

sich ziehen. Da sich jeder Spin umkehren kann, aber immer nur einer zur selben Zeit, bekommt jeder Knoten noch L weitere Kanten hinzu. Diese verlaufen zwischen den Subgraphen, deren Bitcounts sich um den Wert 1 unterscheiden, und führen zu einem dichten Geflecht der Kanten (siehe Abbildung 2).

Nun betrachten wir die Sortierungen der Matrix. Auf den Bildern zu den Sortierungen sehen wir die Input-Partition für k=4 des Graphen mit L=10 Spins ohne Magnetfeld, die durch die jeweilige Sortierung bestimmt ist. Hierbei befinden sich die Knoten einer Farbe auf der selben Partition. Die jeweiligen Farben entsprechen folgender Maßen den 4 Partitionen: blau= Partition 0, grau=Partition 1, rosa=Partition 2, gelb=Partition 3.

3.2.1 arithmetisch

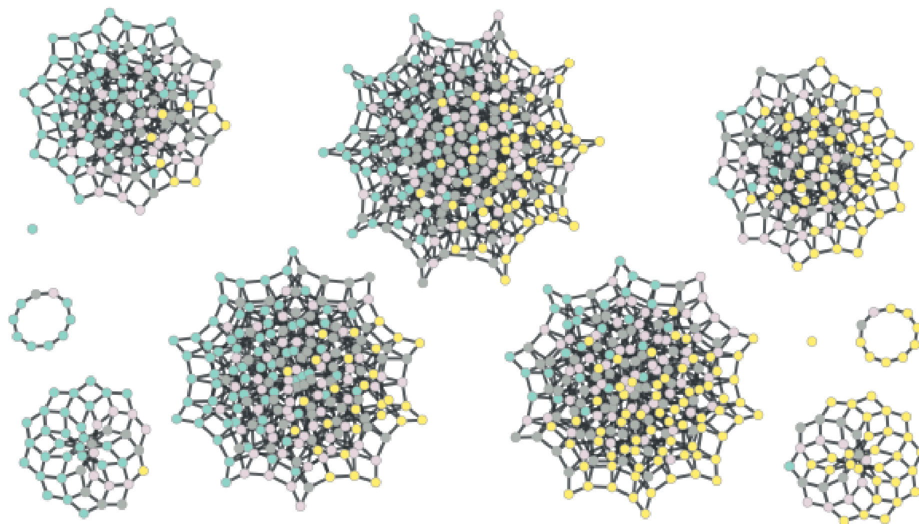

Abbildung 3: Inputpartition von spin10 mit originaler (arithmetischer) Sortierung bei k=4. Die Farben stehen für die verschiedenen Partitionen.

Die ursprünglichen Matrizen sind nach der Darstellung der Spinketten-Zustände als Bitpattern arithmetisch sortiert. Betrachten wir beispielsweise den Fall L=4, so ergibt sich die Reihenfolge: 0000, 0001, 0010, 0011, ..., 1100, 1101, 1110, 1111. ParMETIS verteilt die Knoten somit zu Beginn so auf die k=P=2^j Prozessoren, dass für alle Knoten eines Prozessors die ersten j Stellen der Bitpattern die selben sind.

Diese Sortierung macht in soweit Sinn, dass sich das Aussehen unseres Bitpatterns sowohl durch das Vertauschen zweier benachbarter Spins

als auch durch die Spin-Umkehrung eines einzelnen Spins nicht sonderlich verändert (die meisten Stellen bleiben gleich). Es kommt zwar auch zu Kanten zwischen „weit entfernten“ Knoten, wie beispielsweise 1000 und 0001, aber wenn j< $\frac{L}{2}$ ist, so ist für die meisten Knoten die Anzahl der Kanten zu Knoten mit den selben j Anfangszahlen, das heißt, zu Knoten auf dem selben Prozessor, größer bis gleich zu der Anzahl der Kanten zu Knoten auf anderen Prozessoren. Dieser Unterschied ist gerade bei eingeschaltetem Magnetfeld sehr groß, da dieses für jeden Knoten zu L-j Kanten auf dem Prozessor und j Kanten zwischen den verschiedenen Prozessoren führt, aber auch ohne Magnetfeld erhalten wir diesen Effekt.

Die arithmetische Sortierung beachtet die Überlegungen zu den unabhängigen Subgraphen nicht und so sehen wir in Abbildung 3, dass die Knoten der Subgraphen sehr stark über die Prozessoren verteilt sind. Dies wollen wir mit unseren Umsortierungen verbessern, um so die Laufzeit und die Ergebnisse von ParMETIS zu beeinflussen.

3.2.2 evbit

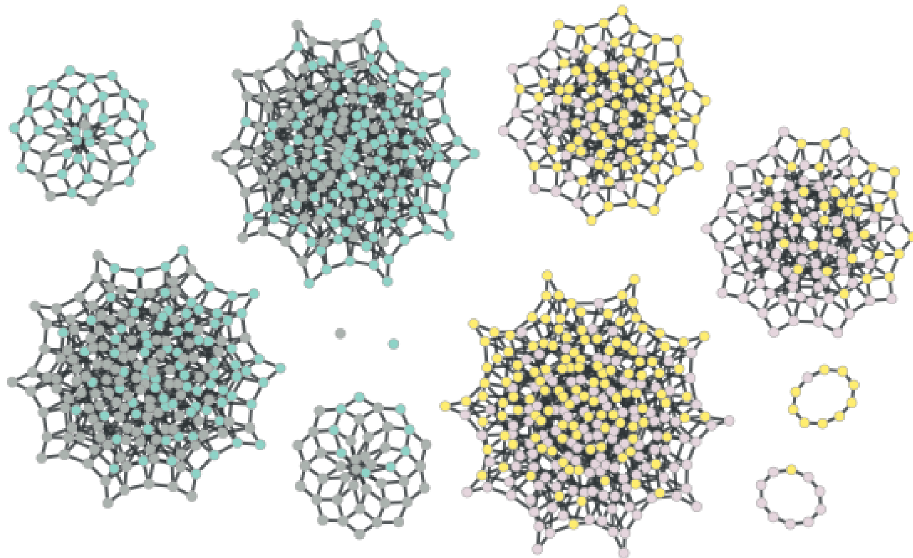

Abbildung 4: Inputpartition von spin10 mit evbit Sortierung bei k=4. Die Farben stehen für die verschiedenen Partitionen.

Bei der Sortierung evbit sortieren wir den Graphen nach geradem und ungeradem Bitcount der Bitpattern. Innerhalb dieser Unterteilung bleiben die Knoten arithmetisch sortiert. Für das Beispiel L=4 heißt das: 0000, 0011, 0101, 0110,..., 0001, 0010, 0100, 0111,... .

Aus dem binomischen Lehrsatz folgt mit x=1 und y=-1, dass $\sum_{l\,gerade} \binom{L}{l} = \sum_{l\,ungerade} \binom{L}{l}$. Da die Anzahl der Knoten mit Bitcount=NUp gleich $\binom{L}{NUp}$

ist, bedeutet dies, dass wir im Fall ohne Magnetfeld durch diese Sortierung eine perfekte Bipartition (Edge-Cut=0, beide Partitionen gleich groß) geschaffen haben. Bei mehr Prozessoren haben wir keine weitere Verbesserung zur arithmetischen Sortierung. Dies sehen wir auch in Abbildung 4 . Die gelben und rosa Knoten und die blauen und grauen Knoten ergeben die beiden Bipartitionen. Unterteilen wir diese Gruppen nochmal, also so, dass wir die einzelnen Farben betrachten, so haben alle Subgraphen Knoten, die auf zwei unterschiedlichen Partitionen liegen und somit ist hier noch einiges zu tun für ParMETIS.

Im Fall mit Magnetfeld zerschneiden wir bei dieser Sortierung schon im Fall k=2 alle Kanten, die durch das Magnetfeld entstehen, da dieses den Bitcount immer um genau eins erhöht oder verringert, ein Endpunkt der Kanten also geraden und der andere ungeraden Bitcount hat. Bei der Bipartition zerschneiden wir zwar keine Kanten der Subgraphen, aber da für jeden Knoten gilt, dass die Anzahl der Kanten, die durch die Spin-Spin-Wechselwirkung entstehen, höchstens L ist (gleich L ist sie bei den Spinketten mit immer abwechselnd $\uparrow$ und $\downarrow$ und alle anderen haben weniger Kanten), also höchstens so groß ist wie die Anzahl der Knoten, die das Magnetfeld erzeugt, besteht der Edge-Cut so aus mehr als der Hälfte aller Kanten. Dies scheint nicht gerade günstig zu sein.

3.2.3 bitcount

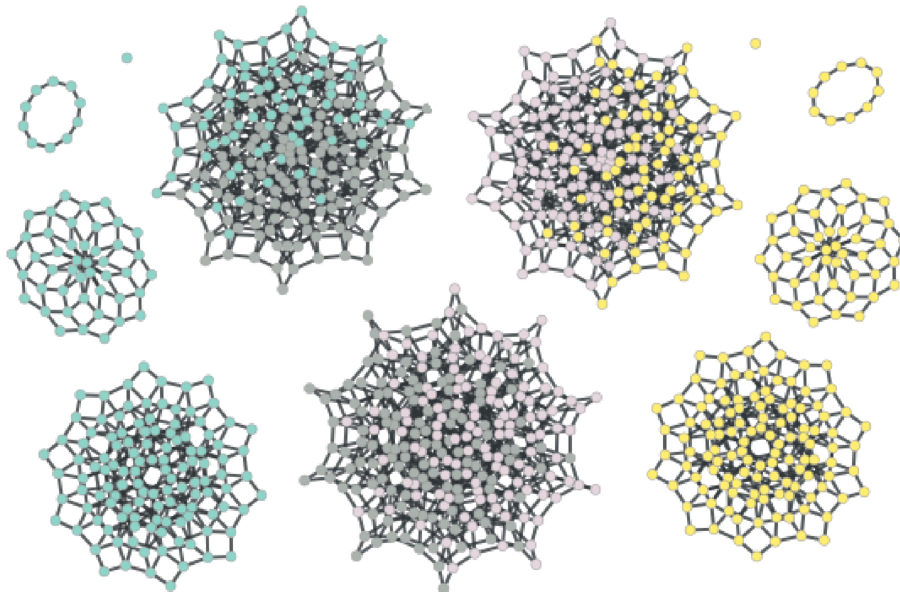

Abbildung 5: Inputpartition von spin10 mit bitcount Sortierung bei k=4. Die Farben stehen für die verschiedenen Partitionen.

Bei der Sortierung bitcount sortieren wir den Graphen nach dem Bitcount, das heißt erst die Knoten mit Bitcount 0, dann 1, 2, 3,... . Dabei bleiben die Knoten innerhalb dieser Unterteilung wieder arithmetisch sortiert.

Im Fall ohne Magnetfeld erhalten wir so eine Blockmatrix, da wir die Knoten der Subgraphen alle nebeneinander angeordnet haben. Würden wir diese Blöcke einfach auf entsprechend viele Prozessoren aufteilen, so hätten wir zwar keine Kommunikation zwischen den Prozessoren, allerdings wäre die Balance auf Grund des großen Größenunterschieds der Subgraphen äußerst schlecht. Auf der anderen Seite haben wir genau L+1 dieser Subgraphen, wobei L gerade ist. Das heißt, teilen wir den Graphen in k=2^j gleich große Stücke, so zerschneiden wir immer den Subgraph zum Bitcount $\frac{L}{2}$, der der Subgraphen mit den meisten Knoten und Kanten ist und zum größten Edge-Cut führt. Im Gegensatz zu der arithmetischen und der evbit Sortierung zerteilen wir hier aber nicht mehr unbedingt alle Subgraphen bei größerem k. In Abbildung 5 sehen wir, dass nur die Knoten der größten Subgraphen durch die Sortierung über verschiedenen Prozessoren verteilt sind und die kleineren nur Knoten einer Farbe beinhalten.

Schalten wir das Magnetfeld ein, so bekommen wir noch Kanten zwischen Subgraphen, deren Bitcount sich um eins unterscheidet. Aus diesem Grund scheint die bitcount Sortierung in diesem Fall sinnvoll, da der Graph hier so sortiert ist, dass die Zerteilung an einer beliebigen Stelle nur zu der Zerteilung eines Subgraphen und der Kanten von diesem Subgraphen zu den zwei benachbarten Subgraphen führt. Dies ist unter den Sortierungen einzigartig. Alle anderen Sortierungen zerschneiden auch noch Kanten zwischen anderen Subgraphen.

3.2.4 evbitcount

Die evbitcount Sortierung ist eine Kombination aus der bitcount und der evbit Sortierung. Wir sortieren die Matrix wieder anhand des Bitcounts, wobei wir erst die geraden und dann die ungeraden Bitcounts betrachten.

So erhalten wir im Fall ohne Magnetfeld wieder eine perfekte Bipartition bei k=2 (in der Abbildung 6 wieder die Kombinationen blau mit grau und rosa mit gelb) und da die Knoten wieder nach den Subgraphen sortiert ist, wirkt die Sortierung auch bei größeren k günstig, da die meisten Subgraphen nur Knoten einer Farbe haben. Zusätzlich wird die Bipartition, die nicht den größten Subgraph enthält (bei uns hier die Bipartition mit den blauen und grauen Knoten), wieder perfekt bipartitioniert. Dies hängt damit zusammen, dass $\binom{L}{l} = \binom{L}{L-l}$ für belibiges $l \in \{0, ..., L\}$. Für die andere Hälfte des Graphen (die rosa und gelben Knoten) wird nur ein Subgraph auf zwei Prozessoren verteilt. Dies ist zwar wieder der größte Subgraph, aber die anderen Subgraphen sind nicht verteilt.

Für den Fall mit Magnetfeld verliert die Sortierung die Besonderheit der

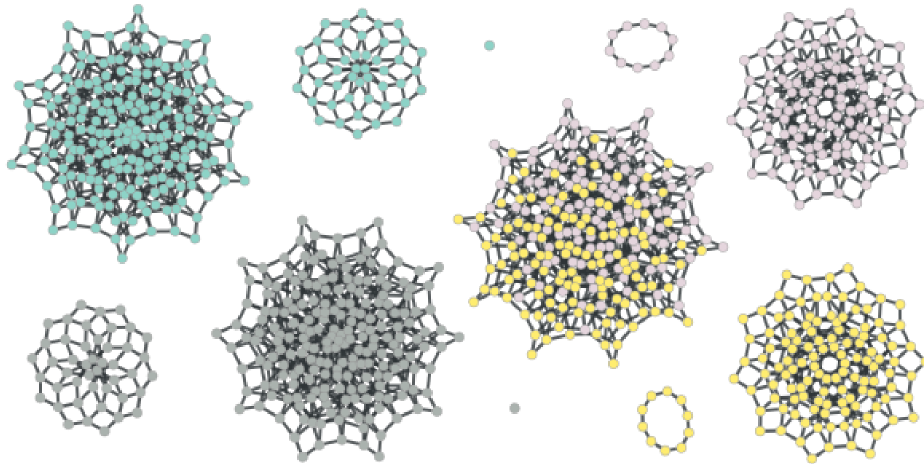

Abbildung 6: Inputpartition von spin10 mit evbitcount Sortierung bei k=4. Die Farben stehen für die verschiedenen Partitionen.

bitcount Sortierung leider und führt zu den gleichen Problemen wie die evbit Sortierung.

3.2.5 resabit

Bei der resabit Sortierung wollen wir die evbitcount Sortierung so verändern, dass bei größerem k große Subgraphen möglichst wenig zerschnitten werden. Bei k=4 hat evbitcount das Problem, dass der größte Subgraph zu gleichen Teilen auf zwei Prozessoren aufgeteilt wird. Da es nicht möglich ist, eine perfekte Partition in 4 Teile zu erzielen, wollen wir erreichen, dass nicht der größte sondern einer der kleineren Subgraphen zerteilt wird und somit der Edge-Cut der Inputpartition weiter reduziert wird. Eine allgemeine Formulierung dafür zu finden ist schwer, da eine solche Sortierung für jedes L etwas anders aussieht. Im Allgemeinen scheint es aber gut zu sein, wenn wir die größten Subgraphen der geraden bzw. ungeraden Subgraphen an die Ränder der Bipartitionen legen und neben den größten Subgraphen von allen noch die Subgraphen mit 4 $\uparrow$ Spins weniger und mehr setzen. Diese bilden dann eine Gruppe, die in etwa so groß ist, wie ein Viertel aller Zustände. Für unseren Graphen mit 10 Spins heißt das beispielsweise, dass wir die Knoten nach folgender Reihenfolge ihres Bitcounts (in unserem Code wird diese Reihenfolge über das helpsort Array festgelegt) sortieren: 5, 1, 9, 3, 7, 4, 0, 2, 8, 10, 6.

Wir erhalten also im Fall ohne Magnetfeld eine perfekte Bipartition, deren eine Partition wieder eine perfekte Bipartition ist und bei der anderen Partition wird im Fall k=4 ein kleinerer Subgraph zerschnitten und nicht der größte (siehe Abbildung 7).

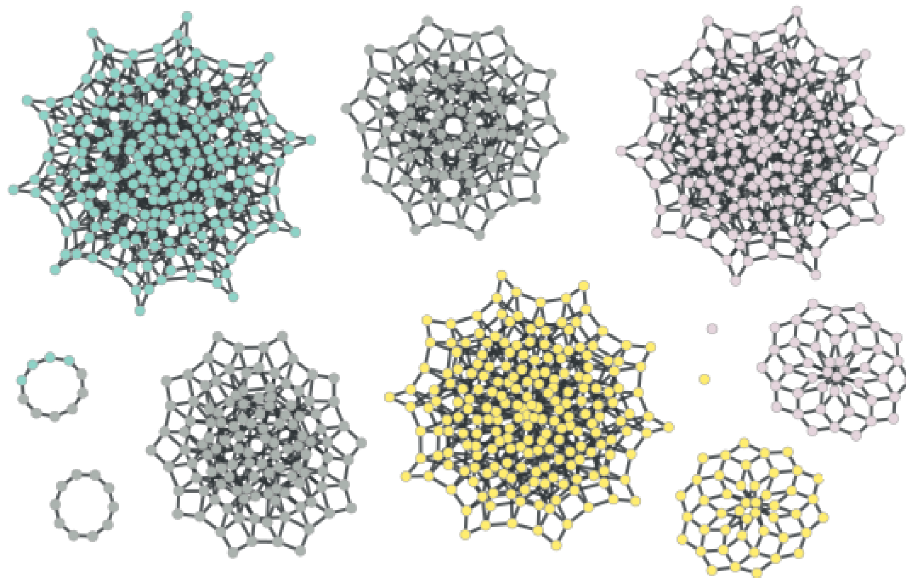

Abbildung 7: Inputpartition von spin10 mit resabit Sortierung bei k=4. Die Farben stehen für die verschiedenen Partitionen.

Für den Fall mit Magnetfeld ändert sich nicht viel zu der evbitcount Sortierung und wir haben somit wieder das Problem, dass wir wieder alle Magnetfeldkanten zerschneiden.

3.3 Tests

In unseren Tests übergeben wir unsere Graphen mit den verschiedenen Sortierungen an Parmetis. Auf der Maschine, auf der wir rechnen, können wir mit bis zu 64 Intel® Xeon Phi „Knights Landing“ Prozessoren arbeiten. Da sich unsere Überlegungen zu den Sortierungen für die Vorpartitionierung auf eine Verbesserung bei einer kleine Anzahl an Partitionen beschränkt, sollte diese Prozessoranzahl ausreichen, um das asymptotische Verhalten abschätzen zu können. Wir lassen ParMETIS für jede Sortierung und jedes k$\in \{2,4,8,16,32,64\}$ 10 Durchgänge laufen, wobei wir jedes Mal den seed für den Pseudozufallszahlengenerator ändern. Wir bekommen für jeden Durchgang die benötigte Zeit, den Edgecut und die Balance zurück und können uns so die durchschnittliche Zeit (avg time) und den durchschnittlichen Edge-Cut (avg cut) berechnen.

Betrachten wir zunächst die Ergebnisse für die Matrizen ohne Magnetfeld.

Hier haben wir einmal die Spinanzahl L=22 und L=24 für die Tests in

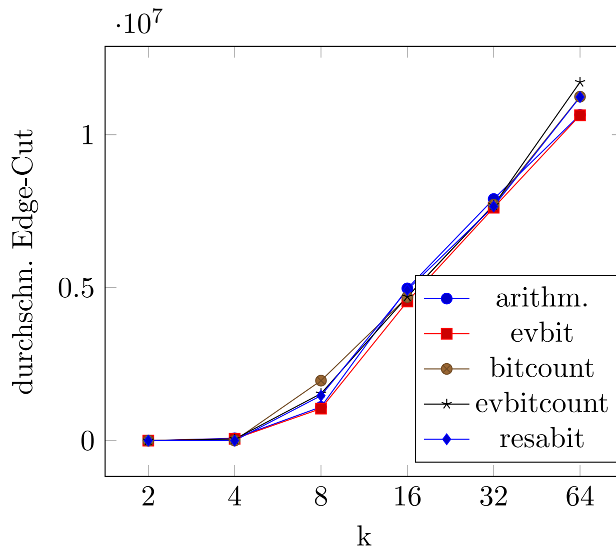

Abbildung 8: Schnittkurve, die ParMETIS für die verschiedenen Sortierungen des Graphen der spin24 Matrix erzeugt

den Tabellen 3 und 4 verwendet. Diese gehören nicht zu den größten Matrizen, sind aber so groß, dass wir unsere Ergebnisse bei diesen Matrizen auf die größeren übertragen können, sie aber noch eine akzeptable Laufzeit für ParMETIS hervorbringen und eine akzeptable Menge an Speicher für die .graph-Datei (Adjazenzliste des Graphen) verbrauchen. Betrachten wir die Fälle für k, in denen wir keine Partition mit Edge-Cut=0 finden, das heißt k>4, so unterscheidet sich der durchschnittliche Edge-Cut bei den verschiedenen Sortierungen nur um 1-2% (siehe Abbildung 8). Für k=2 und k=4 findet Parmetis bei allen Sortierungen eine Partition ohne Edge-Cut, wobei dies bei k=2 immer der Fall ist und bei k=4 auch schon mal Partitionen mit Edge-Cut auftreten, wobei dies nicht von der Sortierung abhängig erscheint. Die verschiedenen Sortierungen der Matrix scheinen also keine besondere Rolle beim Edge-Cut unserer Partition zu spielen.

Anders sieht es aus, wenn wir uns die durchschnittlichen Zeiten anschauen, die ParMETIS für die Berechnung braucht. Hier verlaufen die Zeitkurven aller Umsortierungen unterhalb der Zeitkurve unseres originalen Graphen (siehe Abbildung 9). Durch die Vorsortierung der Matrix sparen wir bis zu 49% der Zeit, die ohne benötigt wurde. Allerdings nimmt die Zeitersparnis mit der Erhöhung der Prozessoranzahl (fast immer) ab.

Bei der Berechnung einer Bipartition (k=2) braucht ParMETIS für alle Umsortierungen etwa nur $\frac{2}{3}$ der Zeit, die es für eine Bipartition des originalen Graphen braucht. Das Einbeziehen des Bitcounts in die Sortierung führt also wirklich zu einem positiven Effekt auf die Laufzeit, wobei der Unterschied zwischen den verschiedenen Umsortierungen klein ist.

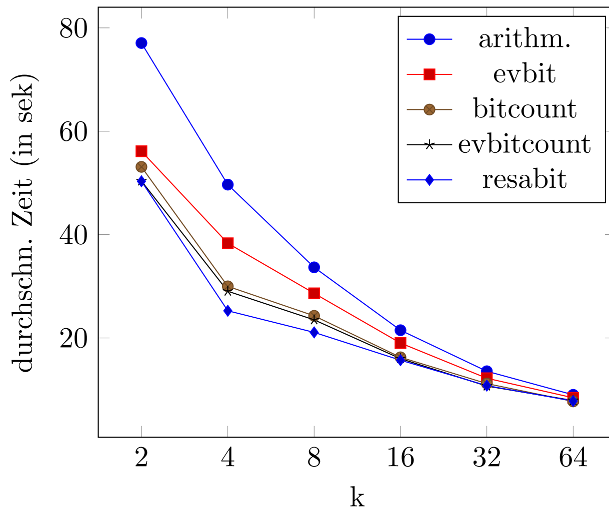

Abbildung 9: Zeitkurven von ParMETIS für die verschiedenen Sortierungen des Graphen der spin24 Matrix

Erhöhen wir nun die Prozessoranzahl, so trennen sich die Verläufe der verschiedenen Sortierungen bei der durchschnittlichen Zeit. Die Kurve der evbit-Sortierung nähert sich der Kurve des originalen Graphen stetig an. Bei den anderen Sortierungen tritt bei k=4 noch eine weitere Verbesserung der Zeitersparnis auf. Bei der bitcount Sortierung und der evbitcount Sortierung haben wir eine Zeitersparnis von etwa 40%, während wir bei der resabit Sortierung nur 51% der Laufzeit von ParMETIS bei der originalen Matrix brauchen. Ab hier nähern sich die Kurven stetig an, wobei ab k=16 die drei Sortierungen bitcount, evbitcount und resabit nicht mehr zu unterscheiden sind, aber immer noch ein wenig besser sind als die evbit Sortierung.

Dieses Zeitverhalten entspricht unseren Überlegungen zu den einzelnen Sortierungen. Da die Sortierung evbit eine Bipartition ist und ansonsten sehr dem originalen Graphen ähnelt, kommt es zu einem ähnlichen Verlauf der Zeitkurve, wobei wir eine Zeitersparnis durch die Bipartition erhalten, die sich für größere k immer mehr abbaut. Die weitere Sortierung in die Subgraphen bei den Sortierungen bitcount, evbitcount und resabit liefert noch eine weitere Verbesserung der Zeit, wobei wir durch die Überlegung, dass wir nicht die größten Subgraphen zerteilen wollen, bei der resabit Sortierung die größte Zeitersparnis erhalten.

Wir erhalten also im Fall ohne Magnetfeld durch unsere Umsortierungen bessere Laufzeiten für ParMETIS. Hierbei ist zwar die Verbesserung durch die resabit Sortierung am größten, aber um eine optimale Vorpartitionierung mit der dahintersteckenden Idee zu erhalten, müsste sie für jede Matrix individuell angepasst werden, weshalb die Sortierungen bitcount und evbitcount, deren Ergebnisse nur ein wenig schlechter sind, zu bevorzugen

sind, wenn wir sie an größeren Matrizen anwenden. Es scheint aber insgesamt gut zu sein, die Subgraphen über die Prozessoren zu verteilen und dann die einzelnen Subgraphen weiter zu partitionieren.

B	L	arithm	evbit	bitcount	evbitcount	resabit
0	22	10.655	11.129	15.181	15.619	15.446
	24	47.291	48.629	67.786	70.643	69.232
1	20	5.317	5.521	6.388	6.522	6.506
	22	23.379	24.491	28.993	29.341	29.243

Tabelle 1: Zeit (in sek) die *spin_pre − partition_graph.cpp.in* zum Sortieren und Erstellen der Matrix für die verschiedenen Sortierungen braucht mit L Anzahl der betrachteten Spins und B=0 bzw. 1, wenn Magnetfeld aus- bzw. eingeschaltet ist.

Was wir bei der Betrachtung des gesamten Vorgehens mit der Vorpartitionierung auch noch berücksichtigen müssen, ist, dass die Umsortierung der Matrix uns auch Zeit kostet. Schauen wir uns die Werte in Tabelle 1 an, so fällt auf, dass die Sortierung der Matrix in etwa die Zeit länger dauert, die ParMETIS für kleine k einspart.

Die Erstellung der umsortierten Matrix können wir momentan nur seriell ausführen. Das bedeutet, dass wir auf die Zeiten für die umsortierten Matrizen noch eine Konstante drauf addieren müssen und so schon bei 8 Prozessoren die Berechnung mit der ursprünglichen Matrix doch schneller ist, da sich der Abstand der Zeitkurven immer weiter verringert. Nur für die evbit Sortierung ist diese Konstante so klein, dass noch für k=16 eine minimal bessere Zeit zustande kommt.

Eine wichtige Aufgabe, wenn wir die Verbesserung der Laufzeit durch ParMETIS nutzen wollen, wäre es darum, das Programm für die Umsortierung der Matrizen zeitlich zu verbessern. Da das ganze Verfahren mit der Vorpartitionierung und der anschließenden Partition durch ParMETIS parallel laufen soll, müsste man den Sortieralgorithmus parallelisieren. Es bleibt zu betrachten, ob man einen parallelen Sortieralgorithmus findet, dessen Laufzeitunterschied für die Sortierungen zu der originale Sortierung nicht die Verbesserung der Laufzeit bei ParMETIS übersteigt.

Nun schauen wir uns noch die Ergebnisse für die Matrizen mit Magnetfeld an.

Hier haben wir einen sehr deutlichen Unterschied sowohl beim durchschnittlichen Edge-Cut als auch bei der durchschnittlichen Laufzeit zwischen der originalen, arithmetischen Sortierung und den anderen, wobei die ursprüngliche Sortierung bei beidem am besten ist (siehe Abbildung 11, 10).

Beim Edge-Cut beginnen die Verfahren noch mit einem relativ gleichen

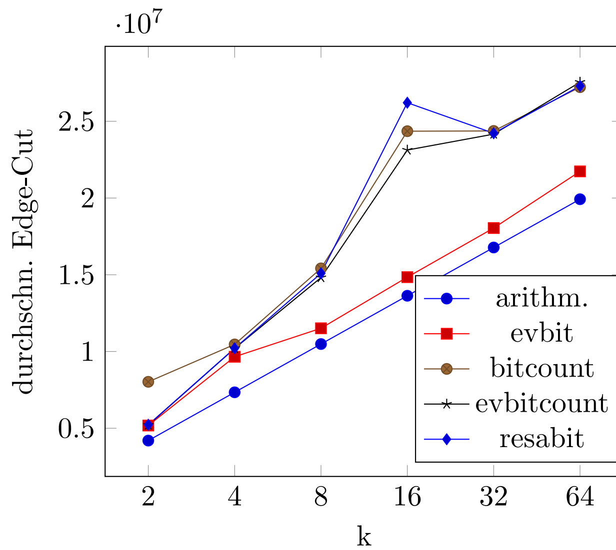

Abbildung 10: Schnittkurve, die ParMETIS für die verschiedenen Sortierungen des Graphen der spinB22 Matrix erzeugt

Wert, bevor die Sortierungen bitcount, evbitcount und resabit eine große Distanz zum Edge-Cut der originalen Sortierung aufbauen. An der extremsten Stelle führt resabit zum 1,7-1,8 fachen des arithmetischen Edge-Cuts. Nur die evbit Sortierung bleibt mit etwa 1% mehr Edge-Cut als die arithmetische Sortierung in der Nähe der besten Kurve. Betrachten wir die besten Schnitte der Verfahren, so sehen wir, dass für keine Sortierung der beste Schnitt besser ist als der Durchschnitt der ursprünglichen Sortierung und das bei dieser beide Werte gleich sind (siehe Tabelle 6 und 5). Wenn wir uns das Partitionsarray, welches uns ParMETIS ausgibt, für den originalen Graphen anschauen, so ist dieses dasselbe wie die Inputpartition. Das bedeutet, ParMETIS hat die anfängliche Sortierung nicht verändert. Verbinden wir dies mit dem Ergebnis, dass wir keine Partition mit niedrigerem Edge-Cut gefunden haben, so schließen wir darauf, dass die Inputpartition, die die arithmetische Sortierung induziert, schon sehr nahe an der best möglichen Partition ist. Dies würde dann erklären, warum ParMETIS Schwierigkeiten hat, eine bessere Partition zu finden.

Da ParMETIS die Input-Partition der arithmetischen Sortierung nicht verändert, führt diese Sortierung auch zu der niedrigsten Laufzeit. Der Algorithmus muss hier viel weniger Arbeit verrichten als bei den anderen Sortierungen, wo er die Knoten hin und her verschiebt. So kommt es zu den wesentlich höheren Laufzeiten der anderen Sortierungen, die etwa 2 bis 3 mal so lang sind, wie die Laufzeit des ursprünglichen Graphen, und alle sehr ähnlich verlaufen. Die einzige Ausnahme ist die bitcount Sortierung bei k=2. Diese braucht hier nur die Hälfte der Zeit länger, die die anderen Umsor-

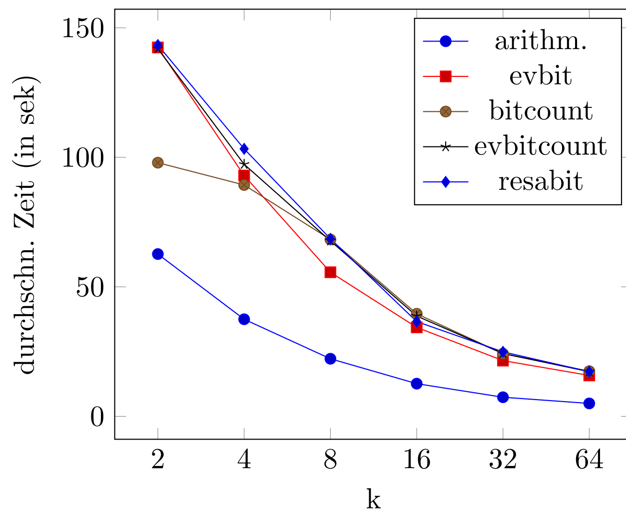

Abbildung 11: Zeitkurven von ParMETIS für die verschiedenen Sortierungen des Graphen der spinB22 Matrix

tierungen länger brauchen als die arithmetische Sortierung. Dies könnte die Auswirkung der Überlegungen zu den Verbindungen zwischen den Subgraphen sein, die wir im Abschnitt über die bitcount Sortierung getätigt haben, wodurch wir nur einen Teil der Magnetfeldkanten zerschneiden und nicht wie bei den anderen Verfahren alle. Dieser Zeiteffekt nimmt für größere k aber schnell.

Zusätzlich müssen wir wieder die Zeiten betrachten, die durch die Umsortierung hinzukommen, wodurch die arithmetische Sortierung weiter an Zeit gewinnt.

Für die Matrizen mit Magnetfeld haben die Tests also klar gezeigt, dass die arithmetische Sortierung die beste unserer Sortierung ist und das es schwer ist, eine Partition zu finden, die besser ist als die hierbei entstehende Inputpartition.

4 Vergleich von Standardverfahren

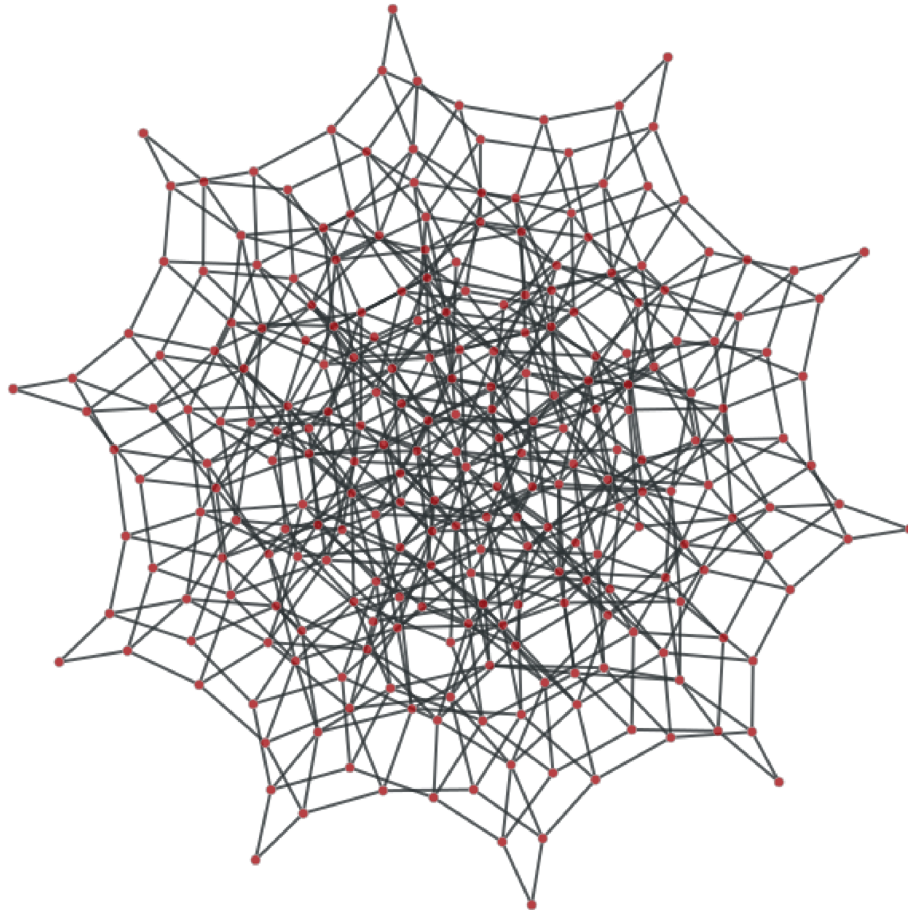

Abbildung 12: Visualisierung des Graphen der spinSZ10 Matrix. Dies ist der größte Subgraph des Graphen der Matrix spin10

Um unsere Sortierungen der Matrizen ohne äußeres Magnetfeld noch weiter zu verbessern, müssen wir uns die einzelnen Subgraphen anschauen und herausfinden, wie wir diese bestmöglich über mehrere Prozessoren verteilen. Hierbei betrachten wir vor allem den Subgraphen mit dem Bitcount $\frac{L}{2}$, also unsere SpinSZ$\langle L \rangle$ Matrizen, da diese die größten Subgraphen sind und die meisten Kanten haben. Da wir hier anhand der Struktur der Matrizen keine Anhaltspunkte für die Sortierung haben, versuchen wir hier, einige Standardverfahren für Graphpartitionierung auf die Matrizen anzuwenden, um herauszufinden, ob es eine Alternative zu ParMETIS gibt, die schneller ist oder bessere Ergebnisse liefert.

In dem Paper [7] wird ParMETIS mit ParHIP, einem Graphpartitionie-

rer der Karlsruher Universität, verglichen. Hier wird davon gesprochen, dass ParHIP zwar eine höhere Laufzeit hat als ParMETIS, aber zu deutlich besserem Edge-Cut führt, insbesondere bei Webgraphen und Sozialen Netzwerken. Da unsere Graphen auch zu dieser Gruppe von komplexen Netzwerken gehören, erhoffen wir uns, dass auch für unsere Graphen eine Verbesserung der Partitionierung auftritt. Aus diesem Grund wollen wir ParHIP als Vergleichspartner verwenden.

Außerdem qualifiziert sich auch PT-Scotch als weiteres Standardverfahren für einen Vergleich.

4.1 Multilevel-Graphpartitionierung

Bei ParMETIS und ParHIP handelt es sich um Multilevel-Graphpartitionierer. Die Idee hinter der Multilevel-Graphpartitionierung (MLGP) ist, dass wir uns allgemeinere Versionen unseres Graphen betrachten, die allgemeinste Version leichter partitionieren können und die entstandene Partition auch ein gutes Ergebnis im ursprünglichen Graphen liefert.

Die MLGP besteht also grundsätzlich aus 3 Schritten:

1) Die Kontraktion des Graphen

2) Partitionierung des allgemeinsten Graphen

3) Aufhebung der Kontraktion und lokale Verbesserung

Im ersten Schritt erzeugen wir eine Reihenfolge von Graphen $G_i = (V_i, E_i)$ mit steigendem Level der Verallgemeinerung, dh $|V_i| < |V_{i-1}| \forall i$. Dies geschieht indem wir Teilmengen V_{i-1}^v von V_{i-1} durch einzelne Elemente v in V_i ersetzen. Wir setzen c(v)=$\sum_{u \in V_{i-1}^v}$ c(u). Die Kanten in G_i entstehen, indem wir alle Kanten e=$\{u, w\} \in E_{i-1}$ mit u $\in V_{i-1}^v$ durch eine Kante $\tilde{e}$=$\{v, w\} \in E_i$ ersetzen mit Kantengewicht w($\tilde{e}$)=$\sum_{\substack{e=\{u,w\} \in E_{i-1}, \\ u \in V_{i-1}^v}} w(e)$

Wenn unser Graph klein genug ist, zerlegen wir ihn in Schritt zwei mit einem beliebigen Partitionierungsalgorithmus (beispielsweise mit rekursiver Bisektion) in k Teile.

In Schritt 3 heben wir die Kontraktion nach und nach wieder auf, wobei die Knoten V_{i-1}^v zur Partition p $\in \{0, ..., k-1\}$ gehören, wenn v der Partition p zugeordnet ist. Hierdurch und durch die Art der Kontraktion des Graphen gelingt es uns, dass der Edge-Cut und die Balance gleich bleiben, wenn wir eine Partition eines Graphen allgemeineren Levels auf einen feineren Graphen übertragen. Die gute Lösung des verallgemeinerten Graphen soll so zu eine gute Lösung des feineren Graphen führen. Um die Partition noch zu verbessern, wenden wir in jedem Schritt lokal einen Algorithmus an, der durch Verschiebung der Knoten zwischen den Partitionen den Edge-Cut

bzw. die Balance verbessert. Diese Algorithmen basieren meist auf Verfahren der lokalen Suche.

Wie genau wir nun die Teilmengen V_{i-1}^v bestimmen, ist der Hauptunterschied zwischen den verschiedenen Multilevel-Graphverfahren.

ParMETIS nutzt hier Matchings.

Ein Matching M $\subseteq$ E ist eine Teilmenge der Kanten vom Graph, sodass für alle Kanten in M gilt, dass sie sich keinen Knoten teilen. Ein Matching M heißt nicht erweiterbar, falls für jede Kante $e \in E \setminus M$ gilt, dass $\{e\} \cup M$ kein Matching ist.

Als V_{i-1}^v werden dann jeweils die beiden Endknoten der einzelnen Kanten in M gewählt, sodass der Graph entlang dieser Kanten zusammengezogen wird. Dabei soll ein nicht erweiterbares Matching gefunden werden, um die Größe des Graphen möglichst schnell zu verkleinern, dass heißt, möglichst viele Knoten in einem Schritt zu kontrahieren.

Bei ParHIP wird die sogenannte Label Propagation mit Größenbeschränkung verwendet.

Hierbei bekommt jeder Knoten zunächst ein eigenes Label. Dann wird in Phasen gearbeitet. In jeder Phase geht der Algorithmus die Knoten in beliebiger Reihenfolge durch und der gerade betrachtete Knoten u wird dem Cluster C_j zugeordnet, bei dem $w(\{\{u,v\}|v \in N(u) \cap C_j\})$ maximiert wird, wobei N(u) die Nachbarschaft von u ist, das heißt, es ist die Menge aller Knoten, die über eine Kante mit u verbunden sind. Der Knoten u wird also dem Cluster zugeordnet, welches die stärkste Verbindung zu u hat.

Um die Balancebedingung zu erfüllen, wird eine Größenbeschränkung einführen, dh eine obere Grenze U für die Größe der Cluster C_j. Es werden dann nur die Cluster betrachtet, die U nicht überschreiten, wenn sie u aufnehmen und u wird zu dem Cluster bewegt, welches davon die stärkste Verbindung zu u hat.

Um den Algorithmus zu beschleunigen, werden die Knoten anstatt beliebig in aufsteigender Reihenfolge ihres Knotengrades betrachtet, da so bei Knoten höheren Grades schon eine Grundstruktur der anderen Knoten zu erkennen ist. Die C_j, die in der letzten Phase berechnet werden, sind dann die gesuchten V_{i-1}^v.

4.2 Parallelisierung

Serielle Algorithmen für die Graphpartitionierung wie METIS produzieren gute Partitionen in wenig Zeit, es kommt allerdings bei großen Graphen zu Schwierigkeiten, da nicht genug Speicherplatz vorhanden ist [6]. Aus diesem Grund schauen wir uns parallele Algorithmen an, die mit größeren Datenmengen arbeiten können.

Bei parallelen Programmen mit verteiltem Speicher bekommt jeder Prozessor einen Teilgraphen, sodass sie gemeinsam den ganzen Graphen ergeben. Diese Teilgraphen bestehen aus lokalen Knoten, deren IDs im Intervall I=a,...,b liegen, den dazugehörigen Kanten und den Endknoten der Kanten, deren IDs nicht im Intervall I liegen und die wir ghost Knoten nennen. Die Kanten zu den ghost Knoten führen dazu, dass wir uns mit der Kommunikation zwischen den Prozessoren auseinandersetzen müssen, damit uns diese nicht zu viel Zeit kostet.

4.2.1 ParMETIS

Bei ParMETIS wird mit einer Färbung der Knoten gearbeitet. Das heißt, dass den Knoten Farben zugewiesen werden, sodass je zwei benachbarte Knoten verschiedene Farben haben, wobei möglichst wenig Farben verwendet werden sollen. Dies hat den Hintergrund, dass einerseits Berechnungen gleichzeitig getätigt werden und dass andererseits, um die Kommunikation zwischen den Prozessoren zu vermindern, die zu kommunizierenden Informationen gesammelt werden und dann gemeinsam gesendet werden. In der Kontraktionsphase könnten so verschiedene Prozessoren versuchen, den selben Knoten unterschiedlich zuzuordnen und während der lokalen Verbesserung im 3. Schritt könnten zwei Verbesserungen auf verschiedenen Prozessoren, gemeinsam ausgeführt, zu einer Verschlechterung der Partition führen. Um dies zu vermeiden, wird die Färbung ausgenutzt, indem nur für Knoten mit gleicher Farbe gleichzeitig Matchingpartner gesucht werden bzw nur diese Knoten gleichzeitig verschoben werden. Berechnet wird die Färbung, indem eine maximale unabhängige Menge der ungefärbten Knoten durch Luby's Algorithmus berechnet wird. Diese Knoten bekommen alle die selbe Farbe und werden dann bei der nächsten Iteration nicht mehr betrachtet.

Bei der Kontraktion des Graphen wird nun über die verschiedenen Farben iteriert. In der i-ten Iteration wählen die noch nicht zugeordneten Knoten der Farbe i den Nachbarn mit dem größten Kantengewicht als Partner aus. Falls zwei Knoten den selben Nachbarn ausgesucht haben, wird der Nachbar befragt, wen er sich als Partner gespeichert hat. Dieser wird dann diesem Nachbarn zugeordnet und der andere Knoten bleibt vorerst ohne Partner. Wenn durch das gesammelte Kommunizieren bei mehrere Prozessoren der selbe Knoten gewählt wurde, wird der mit der höchst gewichteten Kante zugeordnet. So wird das globales Matching ohne Probleme parallel berechnet.

Beim Aufheben der Kontraktion werden ebenso in der i-ten Iteration nur die Knoten der i-ten Farbe für die lokale Verbesserung betrachtet. Die Knoten, deren Bewegung zu einer Verbesserung führen, werden dann gleichzeitig verschoben und da diese Knoten unabhängig sind, bleibt es bei einer Reduktion des Edge-Cuts. Die Gleichgewichtsbedingung wird hierbei lokal aktualisiert, aber zu jeder neuen Phase global berechnet und an alle Prozes-

soren weitergegeben. Um die Kommunikation weiter zu verringern, werden die Knoten in diesem Schritt nicht zwischen den Prozessoren verschickt, sondern nur ihre Partitionsnummer geändert, da sonst auch alle Informationen über die Knoten verschickt werden müssten, die in dem Knoten kontrahiert sind. Die Knoten werden dann erst am Ende des Algorithmus physisch versendet.

4.2.2 ParHIP

Auch bei ParHIP werden Informationen wieder phasenweise kommuniziert, um so Zeit zu sparen. Dies lässt sich gut bei der Label Propagation umsetzen, indem jeder Prozessor die Label Propagation auf seinem Teilgraphen ausführt und die Änderungen an den Randknoten, das sind die lokalen Knoten, die einen ghost Knoten als Nachbarn haben, phasenweise an die entsprechenden Prozessoren schickt. So erhält jeder Prozessor für seine nächste Phase die Veränderungen, die die anderen Prozessoren in der letzten Phase an seinen ghost-Knoten getätigt haben. Bei der Bestimmung über die Reihenfolge, in der die Knoten bei der Label Propagation betrachtet werden, werden nun einfach die Knotengrade der lokalen Knoten verwendet.

Schwierigkeiten bei der Parallelisierung liegen hier in der Einhaltung des Gleichgewichts bzw. der Größenbeschränkung, da die Cluster sehr unterschiedliche Größen aufweisen können. Bei der Kontraktion wird die Größenbeschränkung anhand der Gewichte der Cluster der lokalen und ghost Knoten errechnet. Die Gewichte werden dann lokal geändert. So kann es immer noch zu einem deutlichen Größenunterschied kommen, wenn mehrere Prozessoren den selben Clustern eher mehr oder weniger Knoten zuordnen. Da bei der Kontraktion aber die Größenbeschränkung nicht so strikt einzuhalten ist wie die Gleichgewichtsbedingung bei der Verbesserung in Schritt 3, liegt das im Rahmen. Bei der Verbesserung in Schritt 3 wird wie bei ParMETIS auch die Gleichgewichtsbedingung lokal aktualisiert und zu jeder neuen Phase global berechnet, um hier strikter die Balancebedingung einhalten zu können.

4.3 PT-Scotch

Im Abschnitt 2.1 haben wir das Problem, ein paralleles Programm auf einen parallelen Rechner zu verteilen, auf das Graphpartitionierungsproblem zurück-geführt, indem wir das Programm als Graph dargestellt haben und gesehen haben, dass die Anzahl der zu kommunizierenden Informationen dem zweifachen Edge-Cut entspricht. Im Gegensatz zu ParMETIS und ParHIP setzt Scotch bzw. PT-Scotch nicht erst beim Partitionieren des Graphen ein, sondern löst das Problem mit dem parallelen Programm auf dem parallelen Rechner direkt. Hier werden nun das parallele Programm und die parallele Maschine beide als Graph dargestellt und es wird nach einer Abbil-

dung zwischen diesen Graphen gesucht, die die gesamte Laufzeit minimiert.

Der Grund dafür, dass Scotch nicht erst beim Partitionieren des Programm-Graphen einsetzt, ist, dass die Maschine nicht homogen aufgebaut sein muss. Die Prozessoren können unterschiedliche Rechenleistungen haben, es kann Prozessoren ohne gemeinsamen Kommunikationsweg geben oder die Kommunikationswege führen zu verschiedene Kosten bei der Informationsübermittlung. Einfaches minimieren des Edge-Cuts kann so zeitlich teuer sein, wenn hierbei vor allem lange Kommunikationswege benutzt werden. Es wäre also besser, stark verknüpfte Knoten auf benachbarte Prozessoren zu verteilen. Dies versucht Scotch zu beachten. Unser Graphpartitionierungsproblem erhalten wir genau dann, wenn wir unsere parallele Maschine als vollständigen homogenen Graphen darstellen können, was bedeutet, dass alle Prozessoren miteinander kommunizieren können und diese Kommunikation und die Rechenleistung jedes Prozessors gleich sind. In dem Fall führt der minimale Edge-Cut wirklich zur minimalen Laufzeit.

Formal können das Problem folgender Maßen beschreiben:

Wir stellen unser paralleles Programm als Graphen S dar, indem die Knoten die Prozesse und die Kanten die Kommunikationskanäle repräsentieren, wobei die Knotengewichte $c_S(v_S)$ die Berechnungskosten und die Kantengewichte $w_S(e_S)$ die Menge der benötigten Kommunikation angibt. Die parallele Maschine stellen wir als Graph T dar mit Knoten für die Prozessoren, Kanten für die Kommunikationswege, und den Gewichten $c_T(v_T)$ für die Rechenleistung der Prozessoren und $w_T(e_T)$ für die Kommunikationskosten. Gesucht werden zwei Abbildungen $\tau_{S,T} : V(S) \to V(T)$ und $\rho_{S,T} : E(S) \to P(E(T))$ sodass die Kostenfunktion $f_C(\tau_{S,T}, \rho_{S,T}) = \sum_{e_S \in E(S)} w_S(e_S)|\rho_{S,T}(e_S)|$ minimiert wird, wobei die Balance der Prozesse pro Prozessor innerhalb einer Toleranz liegen soll. Hierbei ist P(E(T)) die Menge aller kreisfreier Wege in T und die Kostenfunktion sorgt dafür, dass stark kommunizierende Prozesse auf nahegelegenen Prozessoren landen.

4.3.1 zweifache rekursive Bipartition

Scotch löst dieses Problem mit zweifacher rekursiver Bipartition. Hierbei werden die Graphen S und T beide rekursiv in zwei geteilt, wobei wir die Partitionen von S rekursiv den Partitionen von T zuordnen.

Der hierbei verwendete Teile-und-Herrsche Ansatz bipartitioniert in jedem Schritt die betrachten Gruppen von Prozessoren auch Domäne genannt, wobei die Domäne zu Beginn aus allen Prozessoren besteht. Die betrachteten Prozesse werden über einen Graphbipartitionierer in zwei geteilt und dann wird jeder entstandenen Subdomäne eine der zwei Partitionen zugeordnet. Auf diese Zuordnungen wird dann rekursiv wieder der Algorithmus angewandt, solange, bis die Domäne nur noch aus einem Prozessor besteht, dem dann die Partition der Prozesse zugeordnet wird oder es keine Prozesse mehr in der Partition gibt und somit nichts mehr zugeordnet werden muss.

Die Rekursion wird mit Breitensuche durchlaufen, da so die Informationen darüber, welche Prozesse zu welchen Subdomänen im vorherigen Level der Rekursion zugeordnet wurden, bei den Kommunikationskosten miteinbezogen werden können, wobei nun partielle Kostenfunktionen betrachtet werden. Die partielle Kostenfunktion entsteht aus der globalen Kostenfunktion, indem diese auf Kanten beschränkt wird, die mindestens einen Endknoten in dem betrachteten Subgraphen haben. Dadurch werden Knoten, deren gemeinsame Kante im Schnitt liegt, auf möglichst nahegelegene Prozessoren gelegt.

Der Teile-und-Herrsche Ansatz macht den Algorithmus gut geeignet, um ihn zu parallelisieren zu PT-Scotch. Dabei wird wieder eine Kommunikationsphase eingeführt um die Breitensuche zu synchronisieren und so die partiellen Kostenfunktionen zu bestimmen. Zusätzlich werden parallele Graphbipartitionierer genutzt, da sonst gerade bei den ersten Rekursionslevels große Bipartitionen auf sehr wenigen Prozessoren berechnet werden würden, was viel Zeit kosten würde.

4.4 Tests

Unsere Tests führen wir auf dem Rechencluster Emmy aus. Dieses besitzt 560 Rechenknoten mit je zwei Xeon 2660v2„Ivy Bridge“ Prozessoren und ist über ein Fat Tree Verbindungsnetzwerk verbunden, welches eine Bandbreite von 40 GBit/s pro Link und Richtung hat [1]. Damit über das Netzwerk kommuniziert werden muss, nutzen wir außer im Fall L=28, bei dem wir 4 MPI Prozessoren für jeden der 32 Knoten verwenden, 32 Koten mit je einem MPI (IntelMPI 5.1.3) Prozess. Wir verwenden den Intel 2016/03 Compiler, wobei wir ohne Unterstützung für Threads kompilieren, sodass wir nur über MPI parallelisieren.

Bei den Partitionierungsverfahren nutzen wir die Versionen ParMETIS 4.0.3, KaHIP 2.0 und Scotch 6.0.3. Hierbei betrachten wir bei ParHIP die beiden Konfigurationen fastsocial und ultrafastsocial und bei Scotch die Einstellungen default, cs (speed) und cx (scalability), während wir bei ParMETIS nur die Variante mit $ParMETIS_V3_PartKway$ nutzen.

Da zurzeit noch keine ParHIP API (Anwendungsprogrammierinterface) zur Verfügung steht, mit dem wir ParMETIS bei der Anwendung in anderen Programmen, wie beispielsweise bei der Berechnung der Matrix-Vektor-Multiplikation, ersetzen könnten, rufen wir die Programme von der Kommandozeile auf. Diese Aufrufe geben uns die Laufzeit, den Edge-Cut und die Balance der gefundenen Partition zurück.

Wir betrachten wieder die Matrizen, mit denen wir noch gut arbeiten können ($L \in \{22, 24, 26, 28\}$). Für die Anzahl der betrachteten Prozessoren überlegen wir uns folgendes:

Wir wollen durch die Verteilung der Knoten auf möglichst viele Pro-

zessoren die Rechenlast pro Prozessor minimieren, um die Laufzeit unseres Programms zu verringern, allerdings kommt es bei Erhöhung der Prozessoranzahl auch zu einem Anstieg der benötigten Kommunikationsmenge, was unser Programm wiederum verlangsamt. Um bei kleineren Graphen die Kommunikation nicht unnötig in die Höhe zu treiben, betrachten wir eine kleinere Anzahl an Prozessoren (Bei L=22 betrachten wir k=2). Für unsere Matrizen gilt, dass die nächst größere Matrix mit 2 Spins mehr aus etwa 4 mal so viele Knoten besteht. Indem wir auch die Prozessoranzahl um den Faktor 4 erhöhen, sorgen wir dafür, dass die Prozessoren für alle Matrizen in etwa die selbe Rechenlast erhalten. Erzeugen wir die Partition des gesamten Graphen, indem wir Partitionen der einzelnen Subgraphen berechnen, so ist dies wichtig, damit die Prozessoren ähnlich ausgelastet sind.

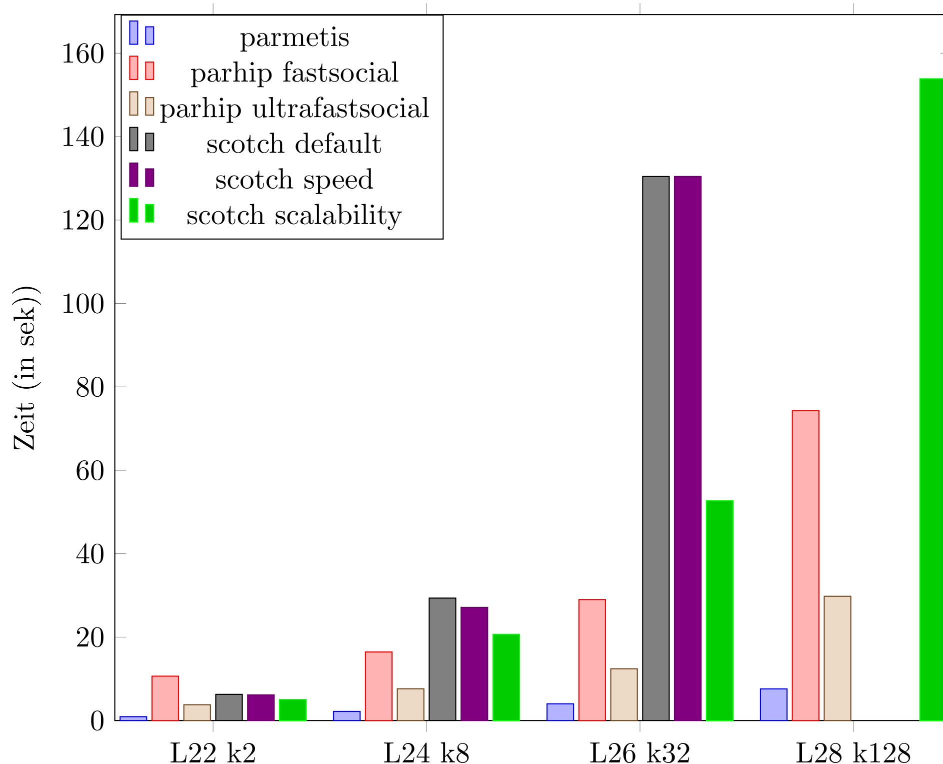

Abbildung 13: Zeitvergleich der verschiedenen Graphpartitionierer für unterschiedliche spinSZ$\langle L \rangle$ Matrizen und k.

Bei den Ergebnissen 2 fällt uns direkt auf, dass wir für den Fall L=28 keine Ergebnisse mit Scotch erhalten. Dies liegt daran, dass es vor allem für einen parallelen direkten Löser entwickelt wurde und somit ab einer gewissen Matrixgröße nicht mehr mit ihr arbeiten kann. Durch die scalability Einstellung von Scotch lässt sich noch ein Ergebnis erzielen. Hierfür müssen

wir aber 16 Prozesse pro Knoten verwendet statt wie für die anderen Programme 4 pro Knoten, da es sonst zu einem Laufzeitfehler kommt.

Betrachten wir die Abbildung 13, so sehen wir, dass es einen großen Unterschied zwischen den Laufzeiten der Verfahren gibt. Die Einstellung ultrafastsocial von ParHIP spart zwar mehr als die Hälfte der Zeit ein, die ParHIP mit der Einstellung fastsocial braucht, aber die Laufzeit ist immer noch um einen Faktor ≥ 3 langsamer als ParMETIS. Die beiden Konfigurationen default und cs von Scotch unterscheiden sich nur kaum merklich in ihrer Laufzeit, was bedeutet, dass die speed Variante nicht zu einer Reduktion der Laufzeit führt. Scotch cx (scalability) hingegen schafft es, die Zeit, die Scotch benötigt, zu verkürzen, allerdings liegen alle drei Varianten deutlich über der Zeit von ParMETIS und ParHIP. ParMETIS hat also unter allen Verfahren die beste Laufzeit.

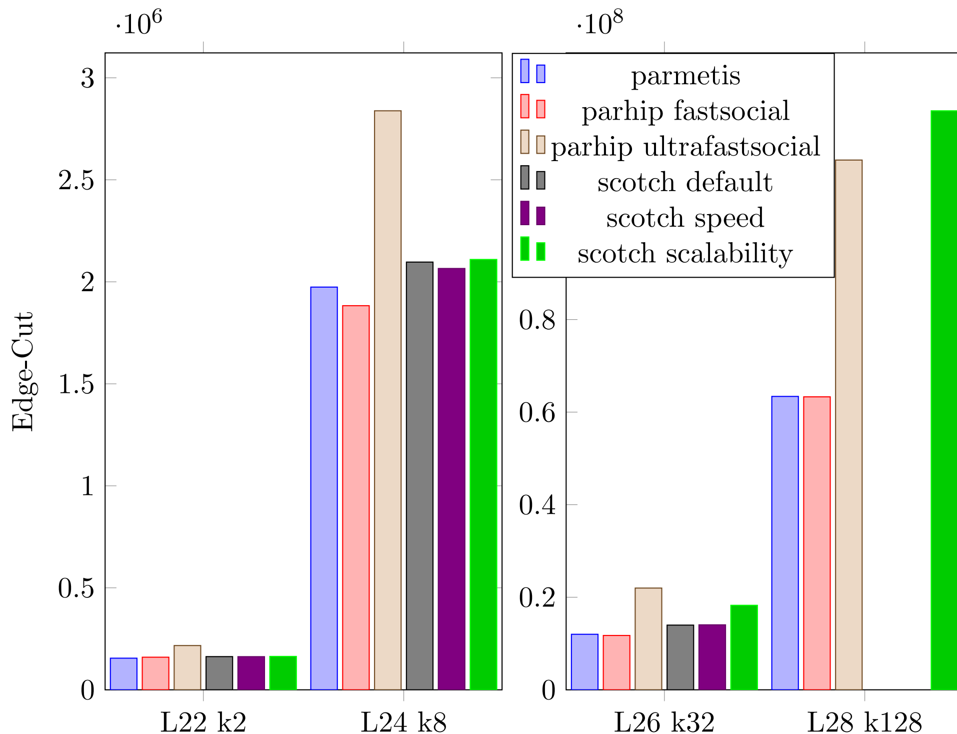

Abbildung 14: Vergleich des Edge-Cuts der verschiedenen Graphpartitionierer für unterschiedliche spinSZ$\langle L \rangle$ Matrizen und k. Um auch die Ergebnisse der kleineren Graphen betrachten zu können, nutzen wir zwei verschiedene Maßstäbe.

Der Edge-Cut unterscheidet sich kaum zwischen ParMETIS und ParHIP mit fastsocial (Abbildung 14). Wir erkennen allerdings, dass wir etwas besseren Edge-Cut erreichen können als den von ParMETIS. Die Einstellung ultrafastsocial von ParHIP führt jedoch zu einem deutlich erhöhten Edge-

Cut. Die eingesparte Zeit zeigt sich also in einem erhöhten Edge-Cut. Der Edge-Cut von den Scotch Varianten bis L=26 ist erhöht gegenüber ParMETIS, liegt aber unter dem Ede-Cut der ParHIP ultrafastsocial Variante. Untereinander unterscheiden sie sich auch nur wenig, wobei Scotch cx die größten Edge-Cuts liefert. Im Fall L=28, indem wir nur die scalability Variante von Scotch betrachten können, erzeugt diese den größten Edge-Cut aller Verfahren.

Fassen wir noch einmal alle Ergebnisse zusammen, so sehen wir, dass ParMETIS am besten für unsere Berechnungen geeignet ist. Bei der Verwendung von Scotch ist sowohl der Edge-Cut wie auch die Laufzeit gegenüber ParMETIS erhöht und da wir darauf hin arbeiten, dass wir größere Matrizen partitionieren wollen, bekommen wir hier auch Probleme, da Scotch mit diesen Matrizen nicht mehr arbeiten kann. Durch ParMETIS fastsocial schaffen wir es zwar, einen niedrieren Edge-Cut zu erzeugen als ParMETIS, da dieser allerdings nur minimal ist und wir dafür große Einbuße bei den Laufzeiten hinnehmen müssen, ist ParMETIS die bessere Wahl. Hierbei ist zu beachten, dass ParHIP auch noch ein vergleichsweise junges Softwarepaket für die Graphpartitionierung ist und hier in Zukunft sicher noch einiges an der Verbesserungen der Performance getan wird, weshalb man in einiger Zeit die Tests nochmal wiederholen sollte.

5 Fazit

Wir haben in dieser Arbeit anhand der Substrukturen unserer Graphen Sortierungen zur Vorpartitionierung entwickelt, die darauf zielen, dem Graphpartitionierer möglichst viel Arbeit abzunehmen, indem sie dem Algorithmus schon eine gute Partition mitgeben.

Für unsere Spinketten-Matrizen mit Magnetfeld haben die Ergebnisse eindeutig gezeigt, dass die Sortierung, die wir bisher verwendet haben, die beste ist. ParMETIS gibt in diesem Fall am Ende die Inputpartition unverändert aus und der Edge-Cut sowie die Laufzeit sind deutlich geringer als bei den anderen Sortierungen, was uns darauf schließen lässt, das wir eine Lösung des Problems gefunden haben, die sehr nah an der optimalen liegt.

Für unsere Matrizen ohne Magnetfeld konnten wir zeigen, dass die Vorpartitionierung der Matrizen zwar keine Auswirkungen auf den Edge-Cut hat, den ParMETIS erzeugt, aber dass sich die Laufzeit des Programms durch die Sortierungen anhand der Subgraphen für eine kleine Anzahl an Partitionen um einiges verkürzt. Es scheint also eine gute Idee zu sein, die Subgraphen über die Prozessoren zu verteilen und sich diese dann einzeln zu betrachten. Hierfür muss aber noch eine parallele Formulierung der Sortierungen gefunden werden, sodass die benötigte Zeit für die Sortierung unsere eingesparte Zeit bei der Partitionierung nicht übersteigt.

Zusätzlich haben wir, auf der Suche nach einem guten Verfahren für die einzelnen Substrukturen der Graphen im Hinblick auf Schnelligkeit und Güte der Partition, die Graphpartitionierer ParMETIS, ParHIP und PT-Scotch miteinander verglichen, wobei wir hierbei immer die größten Subgraphen verwendet haben.

Dieser Vergleich zeigt, dass ParMETIS am besten für unsere Graphen geeignet ist. Scotch führt zu schlechterem Edge-Cut und einer längeren Laufzeit als ParMETIS und da es vor allem für einen parallelen direkten Löser entwickelt wurde, bekommen wir für größere Matrizen, auf deren Partitionierung wir gerade hinarbeiten wollen, keine Ergebnisse mehr. Deshalb ist Scotch keine Option für uns. Mit ParHIP schaffen wir es, etwas besseren Edge-Cut zu erzielen als mit ParMETIS, haben aber eine deutlich erhöhte Laufzeit, weshalb wir das andere Verfahren bevorzugen. Da ParHIP aber noch ein recht junger Graphpartitionierer ist, wird sich hier sicher noch einiges an der Performance des Algorithmus ändern, weshalb man hier nochmal nach einiger Zeit neue Tests machen sollte.

6 Anhang

		parmetis	parhip fs	parhip ufs	scotch	scotch cs	scotch cx
L22	time	0.935	10.644	3.795	6.281	6.155	5.05942
k2	cut	155072	159856	217002	162760	162510	163688
	bal.	1.003	1.028	1.028	1.02	1.02	1.02
L24	time	2.19	16.437	7.618	29.359	27.145	20.6847
k8	cut	1974123	1882950	2838402	2096770	2065385	2109579
	bal.	1.048	1.03	1.03	1.039	1.035	1.04996
L26	time	4.017	29.03	12.415	130.434	130.44	52.702
k32	cut	11996772	11741605	21982940	13978877	14038311	18260074
	bal.	1.049	1.03	1.03	1.039	1.039	1.04999
L28	time	7.595	74.299	29.799	NaN	NaN	153.87*
k128	cut	63393628	63303442	114462631	NaN	NaN	125107255*
	bal.	1.05	1.03	1.03	NaN	NaN	1.05*

Tabelle 2: Ergebnisse der Tests der verschiedenen Graphpartitionierer für unterschiedliche spinSZ$\langle L \rangle$ Matrizen und k. * Hier benutzen wir 16 Prozessoren pro Rechenknoten statt wie für die anderen Verfahren 4.

k		arithm	evbit	bitcount	evbitcount	resabit
2	avg time	16.494	11.749	11.554	10.982	10.869
	avg cut	0	0	0	0	0
	avg balance	1.026	1.019	1.019	1.021	1.028
	best time	16.219	11.675	11.476	10.784	10.789
	best cut	0	0	0	0	0
4	avg time	10.243	7.938	6.875	6.19	5.562
	avg cut	0	12707	26473	4669	8642
	avg balance	1.07	1.074	1.077	1.093	1.088
	best time	10.168	7.7	6.324	5.95	5.289
	best cut	0	0	0	0	0
8	avg time	7.402	6.303	5.497	5.273	4.503
	avg cut	408728	366768	372854	451669	429170
	avg balance	1.051	1.054	1.064	1.067	1.057
	best time	7.197	6.051	5.2	5.019	4.399
	best cut	341634	288378	288066	341300	415790
16	avg time	4.831	4.265	3.667	3.571	3.591
	avg cut	1144872	1170087	1254962	1173947	1159165
	avg balance	1.05	1.05	1.076	1.05	1.05
	best time	4.679	4.082	3.338	3.463	3.507
	best cut	1010771	1050684	1114830	1086301	1062577
32	avg time	2.988	2.72	2.416	2.438	2.477
	avg cut	2022474	1894716	1981766	1974313	1945999
	avg balance	1.056	1.051	1.052	1.051	1.051
	best time	2.834	2.601	2.25	2.368	2.372
	best cut	1845490	1802844	1874648	1903661	1898847
64	avg time	2.349	2.139	1.941	1.987	1.997
	avg cut	2711221	2852482	2833865	2856277	2831908
	avg balance	1.053	1.06	1.051	1.052	1.051
	best time	2.197	1.963	1.797	1.878	1.912
	best cut	2646565	2701849	2793144	2809196	2787505

Tabelle 3: durchschnittliche und beste Ergebnisse von ParMETIS angewandt auf verschiedene Sortierungen des Graphen der spin22 Matrizen

k		arithm	evbit	bitcount	evbitcount	resabit
2	avg time	77.065	56.129	53.108	50.368	50.321
	avg cut	0	0	0	0	0
	avg balance	1.02	1.017	1.019	1.035	1.019
	best time	76.01	55.838	52.894	50.216	50.083
	best cut	0	0	0	0	0
4	avg time	49.668	38.329	29.992	29.045	25.263
	avg cut	76112	58491	0	49478	0
	avg balance	1.06	1.059	1.07	1.068	1.067
	best time	47.716	36.233	29.842	27.887	25.169
	best cut	0	0	0	0	0
8	avg time	33.671	28.639	24.287	23.508	21.081
	avg cut	1101565	1046483	1960658	1537214	1469640
	avg balance	1.133	1.144	1.086	1.13	1.069
	best time	32.228	27.476	23.735	19.737	19.849
	best cut	395918	648006	689404	727956	1024678
16	avg time	21.502	19.022	16.262	16.002	15.702
	avg cut	4976941	4547921	4696380	4706209	4927263
	avg balance	1.05	1.05	1.05	1.053	1.05
	best time	20.56	18.545	15.643	15.285	14.681
	best cut	3987202	3795423	3806291	3864208	3918452
32	avg time	13.563	12.21	11.176	10.694	10.781
	avg cut	7899283	7615317	7712166	7695928	7657865
	avg balance	1.054	1.052	1.05	1.051	1.051
	best time	12.523	11.45	10.306	9.925	10.061
	best cut	7002912	6986871	7451652	7436757	7381127
64	avg time	9.004	8.429	7.723	7.899	7.804
	avg cut	10653836	10636546	11246301	11719711	11232625
	avg balance	1.055	1.056	1.054	1.053	1.052
	best time	8.374	7.674	7.041	7.032	7.4
	best cut	10129233	10244284	10984686	11126703	10787882

Tabelle 4: durchschnittliche und beste Ergebnisse von ParMETIS angewandt auf verschiedene Sortierungen des Graphen der spin24 Matrizen

k		arithm	evbit	bitcount	evbitcount	resabit
2	avg time	11.429	25.242	17.81	25.514	25.629
	avg cut	1048576	1310125	1911507	1303290	1303547
	avg balance	1	1.019	1.024	1.024	1.019
	best time	11.239	24.871	17.383	25.041	25.203
	best cut	1048576	1196950	1891494	1267584	1196032
4	avg time	7.01	17.001	16.749	17.476	18.782
	avg cut	1835008	2487796	2503956	2502826	2562453
	avg balance	1	1.046	1.045	1.048	1.044
	best time	6.958	16.656	16.326	16.81	18.228
	best cut	1835008	2244881	2356165	2343955	2364039
8	avg time	4.386	10.63	12.612	12.777	12.962
	avg cut	2621440	2940409	3690742	3656773	3741602
	avg balance	1	1.042	1.05	1.05	1.05
	best time	4.235	9.626	12.075	12.363	12.634
	best cut	2621440	2738368	3602782	3516374	3615052
16	avg time	2.603	6.902	7.476	7.35	7.314
	avg cut	3407872	3901977	5722182	6289931	5930078
	avg balance	1	1.054	1.075	1.084	1.076
	best time	2.522	6.431	6.582	6.742	6.96
	best cut	3407872	3617314	4589858	5448965	4709814
32	avg time	1.694	4.513	5.184	5.117	5.07
	avg cut	4194304	4659373	6228691	5759908	5753512
	avg balance	1	1.053	1.075	1.067	1.066
	best time	1.593	4.163	4.869	4.822	4.753
	best cut	4194304	4432503	5498998	5331755	5340369
64	avg time	1.566	3.525	3.86	3.928	3.868
	avg cut	4980736	5436065	6494265	6495003	6463989
	avg balance	1	1.055	1.073	1.079	1.073
	best time	1.509	3.366	3.516	3.856	3.628
	best cut	4980736	5251149	6115839	6216797	6114810

Tabelle 5: durchschnittliche und beste Ergebnisse von ParMETIS angewandt auf verschiedene Sortierungen des Graphen der spinB20 Matrizen

k		arithm	evbit	bitcount	evbitcount	resabit
2	avg time	62.657	142.434	97.934	141.931	143.359
	avg cut	4194304	5175709	8023892	5204546	5225821
	avg balance	1	1.012	1.023	1.012	1.015
	best time	62.201	139.897	96.299	140.247	141.167
	best cut	4194304	5011834	8003600	5016292	5042096
4	avg time	37.483	92.971	89.335	97.311	103.279
	avg cut	7340032	9656534	10460638	10200990	10209294
	avg balance	1	1.049	1.046	1.043	1.045
	best time	37.3	84.017	86.726	95.934	102.138
	best cut	7340032	8040448	9981128	9814064	9469236
8	avg time	22.228	55.631	68.288	67.737	68.534
	avg cut	10485760	11512917	15416896	14813736	15094091
	avg balance	1	1.043	1.05	1.05	1.05
	best time	21.599	51.842	66.362	65.656	66.357
	best cut	10485760	10889670	14543516	14267750	14424528
16	avg time	12.613	34.35	39.602	38.73	36.586
	avg cut	13631488	14844787	24355980	23127680	26211716
	avg balance	1	1.053	1.072	1.066	1.079
	best time	12.261	32.181	37.655	37.041	29.642
	best cut	13631488	14263181	21030756	19353416	19653136
32	avg time	7.375	21.52	24.212	24.251	24.899
	avg cut	16777216	18045114	24371380	24164316	24222118
	avg balance	1	1.053	1.066	1.07	1.07
	best time	7.118	19.803	20.792	22.774	23.678
	best cut	16777216	17633840	21268184	21553148	21459502
64	avg time	4.985	15.791	17.43	17.243	17.249
	avg cut	19922944	21737494	27224874	27549158	27299626
	avg balance	1	1.056	1.081	1.086	1.085
	best time	4.587	14.541	17.001	16.649	16.308
	best cut	19922944	20775350	24999024	25989228	24260702

Tabelle 6: durchschnittliche und beste Ergebnisse von ParMETIS angewandt auf verschiedene Sortierungen des Graphen der spinB22 Matrizen

Literatur

[1] Emmy compute-cluster. https://www.anleitungen.rrze.fau.de/hpc/emmy-cluster/. Accessed: 2017-10-11.

[2] Assa Auerbach. *Interacting electrons and quantum magnetism*. Springer Science & Business Media, 2012.

[3] Charles-Edmond Bichot and Patrick Siarry. *Graph partitioning*. John Wiley & Sons, 2013.

[4] Frank Hesmer. Approximation des Heisenberg-Hamilton-Operators in der Basis der spin-kohärenten Zustände. Diplomarbeit, Universität Osnabrück, 2003. http://obelix.physik.uni-bielefeld.de/~schnack/examina/diplom-hesmer.pdf.

[5] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[6] George Karypis and Vipin Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[7] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[8] Wolfgang Nolting. *Quantentheorie des Magnetismus 1*. Teubner Verlag, 1986.

[9] Wolfgang Nolting. *Quantentheorie des Magnetismus 2*. Teubner Verlag, 1986.

[10] François Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 486–493. IEEE, 1994.

[11] François Pellegrini. Pt-scotch and libscotch 5.1 user's guide. 2010.

[12] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

[13] Ralph Skomski. *Simple models of magnetism*. Oxford University Press on Demand, 2008.