Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Bachelorarbeit

im Studiengang
angewandte Mathematik

über das Thema

Überlappung von Kommunikation und Rechnung am Beispiel der Methode der minimalen Residuen

Autor: Patrick Aulbach
Patrick.Aulbach@gmx.net

Prüfer: Prof. Dr. Edeltraud Gehrig
Dr. Jonas Thies

Abgabedatum: 30. Juli 2015

II Inhaltsverzeichnis

1 Einleitung

1.1 Einrichtung

Das deutsche Zentrum für Luft und Raumfahrt e.V.[1] ist die staatliche Forschungseinrichtung der Bundesrepublik Deutschland und ist in der Erforschung der Bereiche Luft- und Raumfahrt, Verkehr, Energie und Sicherheit tätig. An den 16 über Deutschland verteilten Standorten sind 8000 Mitarbeiter beschäftigt. Die Einrichtung Simulations- und Softwaretechnik beschäftigt sich hier mit der Softwareentwicklung für verteilte und mobile Systeme, Software für eingebettete Systeme, Visualisierung und High Performance Computing. Dabei beschäftigt sich die Gruppe High Performance Computing des DLR mit der Entwicklung hoch paralleler Algorithmen, welche höchste Leistung auf heutigen und zukünftigen Supercomputern erreichen sollen. Ein Supercomputer besteht aus einer Vielzahl von Prozessoren welche über ein Netzwerk miteinander verbunden sind. Die derzeit schnellsten Supercomputer sind in der Lage über 10^{15} Gleitkommaoperationen pro Sekunde zu berechnen. Dabei sollen zukünftige Systeme den Bereich der ExaFLOP/s (10^{18} Gleitkommaoperationen pro Sekunde) erreichen.

1.2 Motivation

Gleichungssysteme heutiger Eigenwertprobleme können aus 10^9 und mehr Unbekannten bestehen. Im ESSEX-Projekt[2], welches von der DFG im Rahmen des SPPEXA-Programmes[3] gefördert wird, wird das Softwarepaket PHIST (Pipelined Hybrid-parallel Iterative Solver Toolkit) entwickelt. Dieses beinhaltet iterative Löser für solch große lineare Gleichungssysteme und Eigenwertprobleme. Ein Eigenwertlöser dieses Pakets besteht aus einer Block-Jacobi-Davidson-Methode zur Berechnung einiger Eigenwerte von großen dünnbesetzten Matrizen. Dabei muss in jeder Iteration ein Gleichungssystem näherungsweise gelöst werden.
Um mit solchen Lösern die Architektur heutiger und zukünftiger Supercomputer effizient zu nutzen, verwendet man Programmiermodelle, mit welchen man Teile des Programms parallel auf mehreren Prozessoren ausführen kann. Das Netzwerk ist hierbei häufig der Flaschenhals, d. h. dass einzelne Prozesse auf die Nachrichten der anderen Prozesse warten und die CPU's während dieser Wartezeit untätig sind. In vielen Fällen ist es nicht möglich diese Wartezeit für andere sinnvolle Operationen zu nutzen, da z. B. Datenabhängigkeiten vorhanden sind. Um sinnvolle Berechnungen während der Kommunikation durchführen zu können, ist es nötig den Algorithmus in kleinere Teile zu zerlegen, ohne signifikante

[1] http://www.dlr.de/
[2] http://blogs.fau.de/essex/
[3] http://www.sppexa.de/

Änderungen am Algorithmus selbst durchzuführen. Das Ziel solcher Überlappungen von Kommunikation und Rechnung ist eine Steigerung der Performanz des Algorithmus im parallelen Falle.

1.3 Ziel der Arbeit

In dieser Arbeit soll eine Variante des MINRES-Verfahren in das Softwarepaket PHIST integriert werden. Insbesondere soll untersucht werden, inwiefern Überlappung von Kommunikation und Rechnung in diesem Algorithmus möglich ist. Anschließend sollen diese Überlappungen in den Algorithmus integriert werden. Dabei soll ein Tasking-Framework verwendet werden, welches von den Entwicklern des ESSEX-Projekts bereitgestellt wird und welches es erlaubt, gezielt einzelne Funktionen des Algorithmus mit anderen Funktionen zu überlappen.

1.4 Aufbau der Arbeit

In dieser Arbeit wird zuerst das MINRES-Verfahren vorgestellt. Dabei wird der Algorithmus detailliert hergeleitet und anschließend wird an Hand von Pseudocode die Variante gezeigt, welche letztlich in PHIST implementiert wurde. Im nächsten Kapitel wird ein Überblick über die parallelen Programmiermodelle MPI und OpenMP gegeben und deren Verhalten auf einem Cluster von Shared-Memory-Nodes diskutiert. Dies wird anhand einer dünnbesetzten Matrix-Vektor Multiplikation und eines Skalarprodukts im parallelen Falle demonstriert, da Matrix-Vektor-Operationen ein wesentlicher Bestandteil von iterativen Lösern sind. Anschließend wird gezeigt, inwiefern Überlappung von Kommunikation und Rechnung im MINRES-Algorithmus möglich ist und wie dies in der PHIST-Variante mithilfe des Tasking-Modells realisiert werden kann. Zum Schluss dieser Arbeit werden die Tests, welche mit verschiedenen Matrizen auf einem Rechencluster gemacht wurden, präsentiert. Dabei wird ein Vergleich des MINRES mit und ohne Tasks dargestellt. Anschließend werden die einzelnen Funktion, welche sich überlappen sollen, genauer untersucht.

2 MINRES Verfahren

Im folgenden Kapitel betrachten wir das MINRES-Verfahren (MINimal RESidual), welches 1975 von C. C. Paige und M. A. Saunders [**?**] vorgestellt wurde. Dieses Verfahren gehört zu den Krylov-Unterraum-Verfahren [**?**] und dient zur Lösung von symmetrisch indefiniten Gleichungssystemen.

2.1 Herleitung

Wir betrachten das Gleichungssystem

$$Ax = b \tag{1}$$

mit $A \in \mathbb{R}^{n\times n}$ und $x, b \in \mathbb{R}^n$.
Im k-ten Schritt wird nun die Norm des Residuums $\|b-Ax\|_2$ über den Krylov-Unterraum $x_0 + \mathcal{K}_K(r^{(0)}, A) = \text{span}(r^{(0)}, Ar^{(0)}, \ldots, A^{k-1}r^{(0)})$ minimiert.

Definition 2.1 *Sei $r^{(0)} \in \mathbb{R}^n$, das Startresiduum $r^{(0)} = b - Ax_0$ und $A \in \mathbb{R}^{n\times n}$, so ist $\mathcal{K}_k(r^{(0)}, A) :=$span$(r^{(0)}, Ar^{(0)}, \ldots, A^{k-1}r^{(0)})$ der k-te von $r^{(0)}$ und A aufgespannte Krylovraum.*

Wir suchen also ein x, aus welchem $\|b - Ax\|_2 = \min$ folgt.
Zuerst ist es hilfreich, eine Basis des Krylovraums zu berechnen, da sich unsere Lösung als Linearkombination von Basisvektoren in dem um x_0 verschobenen Krylov-Unterraum darstellen lässt,

$$x = x_0 + Q_k y \tag{2}$$

wobei $Q_k = (q_1, \ldots, q_k) \in \mathbb{R}^{n\times k}$ die Matrix der Basisvektoren und $y = (y_1, \ldots, y_k) \in \mathbb{R}^k$ ist.

2.1.1 Berechnung der Basisvektoren

Da unsere Matrix nach Voraussetzung symmetrisch ist, können wir mit dem Lanczos Verfahren [**?**] eine Orthonormalbasis $Q_k = (q_1, q_2, \ldots, q_k)$ des Krylovraums bestimmen. Damit lässt sich das Minimierungsproblem in der Form

$$\min_x \|b - Ax\|_2 = \min_y \|b - A(x_0 + Q_k y)\|_2 = \min_y \|r_0 - AQ_k y\|_2 \tag{3}$$

schreiben. Als ersten Basisvektor wählen wir

$$q_1 = \frac{r_0}{\|r_0\|_2} \tag{4}$$

Im Lanczos Verfahren werden Skalare α_k und β_k berechnet, welche die Tridiagonale $(k+1) \times k$ Hessenbergmatrix bilden.

$$H_k = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & 0 & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ 0 & 0 & \beta_3 & \alpha_4 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \beta_{k-1} \\ 0 & 0 & \dots & 0 & \beta_{k-1} & \alpha_k \\ 0 & 0 & \dots & 0 & 0 & \beta_k \end{pmatrix} \in \mathbb{R}^{(k+1)\times k}$$

Spannen nun die Vektoren $(q_1, q_2, \dots, q_k)$ den Krylovraum $\mathcal{K}_k$ auf, so spannen die Vektoren $(q_1, q_2, \dots, q_n, Aq_k)$ den Krylovraum $\mathcal{K}_{k+1}$ auf. Nun müssen wir den Vektor Aq_k gegenüber den vorherigen orthonormalisieren. Damit ergibt sich die Relation

$$AQ_k = Q_{k+1}H_k \tag{5}$$

welche aus dem Lanczos Verfahren folgt. Somit lässt sich das Minimierungsproblem (3) mithilfe von (4) und (5) schreiben als

$$\begin{gathered} \min_x \|b - Ax\|_2 = \min_y \|r_0 - AQ_k y\|_2 = \min_y \|\|r_0\|_2 v_1 - Q_{k+1}H_k y\|_2 \quad = \\ \min_y \|Q_{k+1}(\|r_0\|_2 e_1 - H_k y)\|_2 = \min_y \|\|r_0\|_2 e_1 - H_k y\|_2 \end{gathered} \tag{6}$$

Somit wird das Minimierungsproblem zu

$$\min_y \|||r_0||_2 e_1 - H_k y\|_2 \tag{7}$$

Dieses Problem kann mithilfe einer QR-Zerlegung gelöst werden.

2.1.2 QR-Zerlegung der Hessenberg-Matrix

Durch die Struktur von H_k müssen nur die Elemente der unteren Nebendiagonalen eliminiert werden. Dazu verwendet man am einfachsten Givens-Rotationen [**?**]

$$G_{i,j} = \begin{pmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & c & 0 & \dots & 0 & s & & \\ & & & 0 & 1 & & 0 & 0 & & \\ & & & \vdots & \vdots & \ddots & \vdots & \vdots & & \\ & & & 0 & 0 & & 1 & 0 & & \\ & & & -s & 0 & \dots & 0 & c & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{pmatrix} \text{ mit } c^2 + s^2 = 1$$

Eine Givens Matrix $G_{i,j}$ bewirkt eine Rotation in der durch e_i und e_j aufgespannten Ebene. Somit lässt sich die Matrix H_k durch Anwendung von k Givens Rotationen auf obere Dreiecksgestalt transformieren. Da die Matrix H_k Tridiagonalgestalt hat erhalten wir

$$G_{k+1,k} \dots G_{32} G_{21} H_k = \begin{pmatrix} * & * & * & & \dots & 0 \\ 0 & * & * & * & & 0 \\ & & & \ddots & \ddots & \\ & & & & & * \\ & & & & & * \\ & & & & & 0 \end{pmatrix} = \hat{H}_k$$

Wendet man die Rotationen auch auf die rechte Seite des Ausgleichsproblem (7) an, so erhält man

$$\hat{b}_k = \begin{pmatrix} b_k \\ \|r_k\|_2 \end{pmatrix} \in \mathbb{R}^{k+1} \tag{8}$$

was aus sich aus Multiplikation mit Givens-Rotationen von $||r_0||_2 e_1$ ergibt und sich rekursiv aus

$$\hat{b}_k = G_{k+1,k} \begin{pmatrix} \hat{b}_{k-1} \\ 0 \end{pmatrix} \in \mathbb{R}^{k+1} \tag{9}$$

berechnen lässt. Die Givens-Rotationen wirken sich nur auf die letzten beiden Einträge von $\hat{b}_k$ aus und somit nur auf den letzten Eintrag von b_k. Damit erhalten wir

$$\hat{b}_k = G_{k+1,k} \begin{pmatrix} b_{k-1} \\ \|r_{k-1}\|_2 \\ 0 \end{pmatrix} = \begin{pmatrix} b_{k-1} \\ c_k \|r_{k-1}\|_2 \\ -s_k \|r_{k-1}\|_2 \end{pmatrix} \in \mathbb{R}^{k+1} \tag{10}$$

Somit lässt sich (7) nach k Givens-Rotationen schreiben als

$$\hat{H}_k y = b_k \tag{11}$$

Damit lässt sich die Lösung y einfach durch Rückwärtssubstitution berechnen und wir können somit die Lösung von (2) bestimmen.

2.1.3 Berechnung des Lösungsvektors

Im MINRES-Verfahren ist möglich, mit Hilfe einer Rekursion in jeder Iteration eine Approximation der Lösung x_k zu bestimmen. Dazu setzen wir

$$W_k = (\omega_1, \omega_2, \ldots, \omega_n) := Q_k \hat{H}_k^{-1} \Leftrightarrow W_k \hat{H}_k = Q_k \tag{12}$$

d.h. die Spalten von W_k sind eine Linearkombination der Spalten von Q_k. Damit lässt sich der Vektor ω_k durch die spezielle Gestalt von $\hat{H}_k$ darstellen als

$$q_k = \hat{h}_{k,k}\omega_k + \hat{h}_{k-1,k}\omega_{k-1} + \hat{h}_{k-2,k}\omega_{k-2} \tag{13}$$

Da wir den Vektor ω_k noch nicht kennen, liefert uns (11) eine Rekursionsvorschrift um den Vektor ω_k zu bestimmen

$$\omega_k = \frac{q_k - \hat{h}_{k-1,k}\omega_{k-1} - \hat{h}_{k-2,k}\omega_{k-2}}{\hat{h}_{k,k}} \tag{14}$$

Damit lässt sich der Lösungsvektor x_k mit (2), (9), (10) und (12) folgendermaßen darstellen:

$$\begin{aligned} x_k &= x_0 + W_k b_k \\ &= x_0 + \begin{pmatrix} W_{k-1} & \omega_k \end{pmatrix} \begin{pmatrix} b_{k-1} \\ c_k \|r_{k-1}\|_2 \end{pmatrix} \\ &= x_0 + W_{k-1} b_{k-1} + \omega_k c_k \|r_{k-1}\|_2 \end{aligned} \tag{15}$$

Somit lässt sich der Lösungsvektor rekursiv durch

$$x_k = x_{k-1} + \omega_k c_k \|r_{k-1}\|_2 \tag{16}$$

bestimmen.

2.2 Algorithmus

Letztlich erhalten wir den folgenden Algorithmus der in dieser Form implemetiert werden kann.

Algorithm 1 MINRES

Wähle $x^0 \in \mathbb{R}^n$ beliebig
$v_0 \leftarrow 0$
$v_1 \leftarrow b - Ax_0$
$c_0 \leftarrow c_1 \leftarrow 1$
$s_0 \leftarrow s_{-1} \leftarrow 0$
$\beta_1 \leftarrow \eta \leftarrow \|v_1\|_2$
for $i = 1 \dots maxit$ **do**
    Lanczos
    $v_i \leftarrow \frac{1}{\beta_i} v_i$
    $\alpha_i \leftarrow v_i^T A v_i$
    $v_{i+1} \leftarrow Av_i - \alpha_i v_i - \beta_i v_{i-1}$
    $\beta_{i+1} \leftarrow \|v_{i+1}\|_2$
    QR-Zerlegung mittels Givens-Rotationen
    $\delta \leftarrow c_i \alpha_i - c_{i-1} s_i \beta_i$
    $\rho_1 \leftarrow \sqrt{\delta^2 + \beta_{i+1}^2}$
    $\rho_2 \leftarrow s_i \alpha_i + c_{i-1} c_i \beta_i$
    $\rho_3 \leftarrow s_{i-1} \beta_i$
    $s_{i+1} \leftarrow \frac{\beta_{i+1}}{\rho_1}$
    $c_{i+1} \leftarrow \frac{\delta}{\rho_1}$
    Berechnung des Lösungsvektors
    $\omega_i \leftarrow \frac{1}{\rho_1}(v_i - \rho_2 \omega_{i-1} - \rho_3 \omega_{i-2})$
    $x_i \leftarrow x_{i-1} + c_{i+1} \eta \omega_i$
    $\eta \leftarrow -s_{i+1} \eta$
end

2.3 MINRES in PHIST

Der Vorteil der Standard-Variante des MINRES-Verfahrens, welche in Algorithmus 1 gezeigt wird, sind die speicherschonenden Rekursionen. Dabei ist es nicht nötig, dauerhaft die Basis Q_k des Krylovraums sowie die Hessenbergmatrix H_k im Speicher zu behalten. Speicher wird hier nur für die Vektoren v_{i-1}, v_i, v_{i+1}, ω_{i-2}, ω_{i-1} und ω_i benötigt.
Die Phist-Variante des MINRES-Verfahrens dient als innerer Löser des Jacobi-Davidson-Algorithmus. Dazu ist es nicht nötig das lineare Gleichungssystem exakt zu lösen, sondern es genügt eine kleine feste Anzahl an Iterationen. Da hierbei der Speicherplatz keine Rolle spielt, wird in der PHIST-Variante die Basis Q_k sowie die Hessenbergmatrix H_k komplett gespeichert. Dadurch kann in der PHIST-Variante auf die Rekursionen, welche die Vektoren ω_i und x_i in jeder Iteration aktualisieren, verzichtet werden. Der Vorteil, welcher sich dadurch ergibt, ist, dass während einer Iteration zwei Vektoren weniger aktualisiert werden müssen. Weiterhin muss am Ende der Laufzeit nur noch ein Vektor aktualisiert werden, wodurch eine Vektor-Aktualisierung pro Iteration komplett wegfällt. In der PHIST-Variante wird die Gleichung (9) am Ende der Laufzeit durch Rückwärtseinsetzen gelöst und in Gleichung (2) der approximierte Lösungsvektor berechnet. Bedingt dadurch, dass wir den MINRES als inneren Löser des Jacobi-Davidson-Algorithmus verwenden, wird hier nicht das Gleichungssystem $Ax = b$ sondern die Korrekturgleichung $(\mathbb{1} - qq^T)(A - \sigma\mathbb{1})x = -r$ [**?**] gelöst. Dabei ist q ein Satz von orthogonalen Vektoren aus dem Jacobi-Davidson-Algorithmus mit $q \in \mathbb{R}^{n\times m}$ und $m \lll n$, was in den folgenden Kapiteln als Multivektor bezeichnet wird. $\mathbb{1} \in \mathbb{R}^{n\times n}$ bezeichnet die Einheitsmatrix und $\sigma \in \mathbb{R}$ einen reellen Wert. Die PHIST-Variante des Jacobi-Davidson-Algorithmus ist eine Blockvariante. Dabei werden mehrere Gleichungssysteme der Form $(\mathbb{1} - qq^T)(A - \sigma_i\mathbb{1})x_i = -r_i, i = 1, \ldots, 4$ gelöst. Die Blockgröße spielt für die Überlappung von Kommunikation und Rechnung allerdings keine Rolle. Für die Betrachtungen in den nächsten Kapiteln wurde eine Blockgröße $nb = 4$ und $m = 20$ gewählt.
Somit erhalten wir folgenden Algorithmus:

Algorithm 2 PHIST Implementation MINRES

1: Wähle $x^0 \in \mathbb{R}^n$ beliebig
2: $v_0 \leftarrow 0$
3: $v_1 \leftarrow b - Ax_0$
4: $c_0 \leftarrow c_1 \leftarrow 1$
5: $s_0 \leftarrow s_{-1} \leftarrow 0$
6: $\beta_1 \leftarrow \eta \leftarrow \|v_1\|_2$
7: **for** $i = 1 \dots maxit$ **do**
8: Lanczos
9: $v_i \leftarrow \frac{1}{\beta_i} v_i$
10: $v_{i+1} \leftarrow (\mathbb{1} - qq^T)(A - \sigma\mathbb{1})v_i$
11: $\alpha_i \leftarrow v_i^T v_{i+1}$
12: $v_{i+1} \leftarrow v_{i+1} - \alpha_i v_i - \beta_i v_{i-1}$
13: $\beta_{i+1} \leftarrow \|v_{i+1}\|_2$
14: QR-Zerlegung mittels Givens-Rotationen
15: $\delta \leftarrow c_i\alpha_i - c_{i-1}s_i\beta_i$
16: $\rho_1 \leftarrow \sqrt{\delta^2 + \beta_{i+1}^2}$
17: $\rho_2 \leftarrow s_i\alpha_i + c_{i-1}c_i\beta_i$
18: $\rho_3 \leftarrow s_{i-1}\beta_i$
19: $s_{i+1} \leftarrow \frac{\beta_{i+1}}{\rho_1}$
20: $c_{i+1} \leftarrow \frac{\delta}{\rho_1}$
21: $b_{k+1} \leftarrow -s_{i+1}b_k$
22: $b_k \leftarrow c_{i+1}b_k$
23: **end**
24: Berechne Lösungsvektor
25: $y_k \leftarrow \frac{b_k}{\rho_{kk}}$
26: **for** $i = k - 1 \dots 1$ **do**
27: $y_i = \frac{b_i - \sum_{j=i+1}^{k} \rho_{ij} y_j}{\rho_{ii}}$
28: **end**
29: $x \leftarrow x_0 + \sum_{i=1}^{k} y_i v_i$

In diesem Algorithmus entsteht mit ρ_1, ρ_2 und ρ_3 in jeder Iteration eine neue Zeile der oberen Dreiecksmatrix $\hat{H}_k$.

3 Programmiermodelle auf Parallelrechnern

Das folgende Kapitel gibt einen kurzen Einblick über den Aufbau eines HPC-Clusters, sowie einen Einblick in das Kommunikationsmodell von MPI und über die Funktionsweise von OpenMP.

3.1 Aufbau eines HPC-Clusters

Ein High-Performance-Computing-Cluster besteht im wesentlichen aus einer Anzahl von Knoten, auf welchen Rechnungen durchgeführt werden können. Ein Knoten besteht wiederum aus einer Anzahl von CPU's, welche mehrere Kerne besitzen. Jeder Knoten besitzt einen eigenen Speicher, auf den die CPU's zugreifen können. Die einzelnen Knoten sind über ein schnelles Netzwerk miteinander verbunden über welches sie mittels MPI kommunizieren können. Dabei laufen im einfachsten Fall auf jedem Prozessor des Knotens ein MPI-Prozess. Zusätzlich besitzt jeder Prozessor eine bestimmte Anzahl an Kernen auf denen während der Laufzeit bei Bedarf mit OpenMP weiter parallelisiert wird. Dabei läuft im einfachsten Fall auf jedem Kern ein OpenMP-Thread. Abbildung 1 zeigt einen einfach Aufbau eines Clusters, welches aus zwei Knoten und vier CPU's besteht.

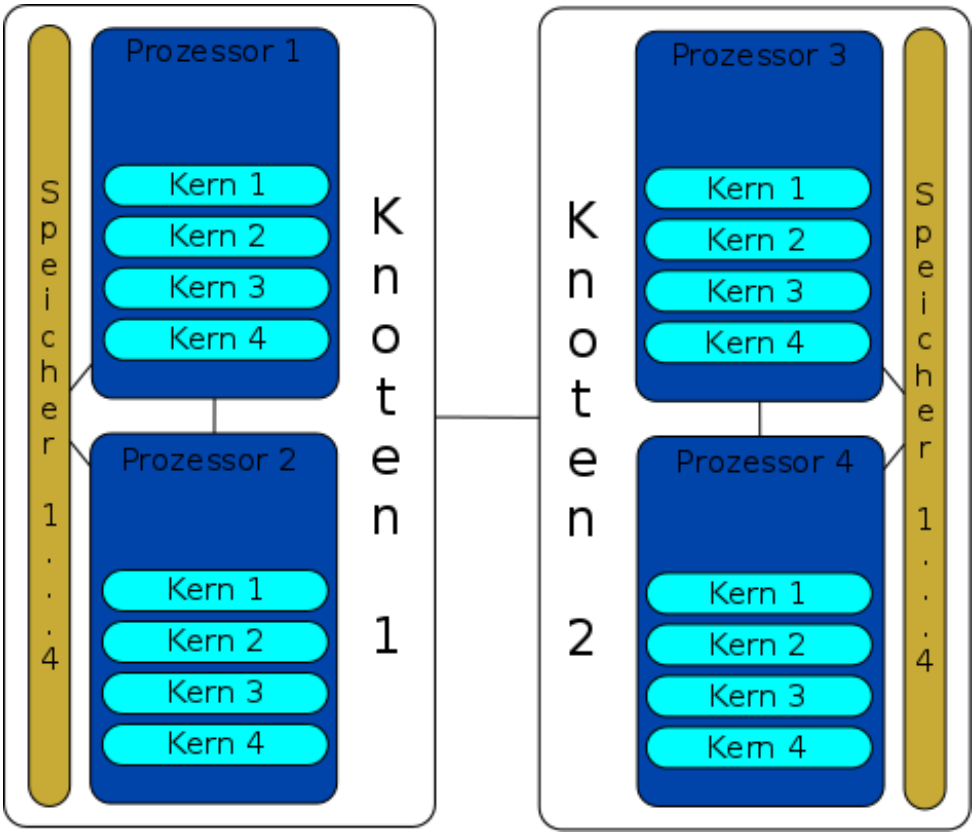

Abbildung 1: Beispiel eines einfachen Clusters, bestehend aus zwei Knoten mit jeweils zwei Prozessoren

In den folgenden Kapiteln wird gezeigt, wie Parallelisierung mit MPI und OpenMP funktioniert.

3.2 Verteilte-Speicher-Parallelisierung mit MPI

Mittels MPI [**?**] ist es möglich, Prozesse über das Netzwerk miteinander kommunizieren zu lassen und somit effektiv Daten zwischen den Prozessen auszutauschen. Dabei unterscheidet man zwischen einem synchronen und einem asynchronen Kommunikationsmodell.
Bei der synchronen Kommunikation wird mittels MPI_Send ein Datenpaket an einen anderen Prozess geschickt. Hier wird solange gewartet, bis der übergebene Buffer wieder verwendet werden kann. Gleichzeitig steckt der Prozess, der die Daten empfangen soll, in MPI_Recv fest, bis die Daten angekommen sind. Diese Form der Kommunikation nennt man blockierende Kommunikation. Bei der nichtblockierenden Kommunikation muss der Prozess nicht in den Funktionsaufrufen warten. Dies kann mittels MPI_Isend und MPI_Irecv realisiert werden. Hierbei muss der Prozess welcher die Daten sendet, nicht auf den Empfang der Daten warten und der Prozess welcher die Daten empfängt, muss erst dann warten, wenn er die Daten benötigt. Dies lässt Spielraum für Überlappungen, da die Prozesse während sie Daten empfangen/senden lokal weitere Rechnungen durchführen können und es somit idealerweise zu keiner Wartezeit kommt, in denen die Prozesse nichts machen.
Bei der asynchronen Kommunikation kann es sehr leicht zu falschen Berechnungen oder Speicherzugriffsverletzungen kommen. Hier muss sehr genau darauf geachtet werden, was die Prozesse während der Kommunikation lokal rechnen, damit kein Prozess versucht Daten, welche sich gerade im Sende- oder Empfangsvorgang befinden zu verändern. Sonst kann es passieren, dass falsche Daten gesendet werden und somit falsche Ergebnisse entstehen.

3.3 Gemeinsame-Speicher-Parallelisierung mit OpenMP

Open Multi-Processing (OpenMP) [**?**] ist eine Programmierschnittstelle zur Shared-Memory-Programmierung. Im Code werden mittels Präcompiler-Direktiven Regionen festgelegt, welche später parallel laufen sollen. Dabei startet ein OpenMP-Programm als einzelner Thread. OpenMP erzeugt neue Threads, wenn es zu einer parallelen Region kommt. Danach wird das Programm wieder seriell ausgeführt. Dies kann während der Laufzeit beliebig oft geschehen. Die Threads können auf gemeinsame Variablen zugreifen. Dabei muss darauf geachtet werden, dass die Threads nicht gleichzeitig Variablen ändern, oder eine Variable gleichzeitig geändert und gelesen wird. Dies kann zu fehlerhaften Berechnungen führen.

4 Überlappung von Kommunikation und Rechnung

Krylov-Unterraum-Verfahren zeichnen sich dadurch aus, dass im wesentlichen nur Matrix-Vektor-Multiplikationen sowie Skalarprodukte in jeder Iteration benötigt werden. Dieses Kapitel gibt einen Überblick über die Kommunikation von wesentlichen Matrix-Vektor-Operationen, sowie einen Einblick in die Kernel-Operationen, welche in PHIST Verwendung finden und zur Implementierung des MINRES-Verfahrens verwendet wurden. Einen Ansatz zur Überlappung von Kommunikation weiterer Krylov-Unterraum-Verfahren, liefern die Arbeiten von P.Ghysels und W.Vanroose [?] [?]

4.1 Kommunikation in Matrix-Vektor-Operationen

Im Folgenden schauen wir uns an, inwiefern bei diesen Kernfunktionen im Algorithmus Kommunikation auftritt. Dazu betrachten wir das Matrix-Vektor-Produkt aus (1). Diese Operation kann parallel abgearbeitet werden. Dazu werden die Matrix A und der Vektor x folgendermaßen auf die einzelnen Prozesse verteilt: Jeder Prozess kennt einen Block aus Zeilen der Matrix A, sowie den dazugehörigen Block des Vektors x. Kommunikation ist hier insofern nötig, da die einzelnen Prozesse die Einträge des Vektors x untereinander tauschen müssen. Da die Matrizen dünnbesetzt sind, müssen die Prozesse nicht alle miteinander kommunizieren, sondern die Prozesse benötigen im Idealfall nur Daten der jeweiligen Nachbarn. In der folgenden Abbildung wird solch eine Datenverteilung einer dünnbesetzten Matrix gezeigt.

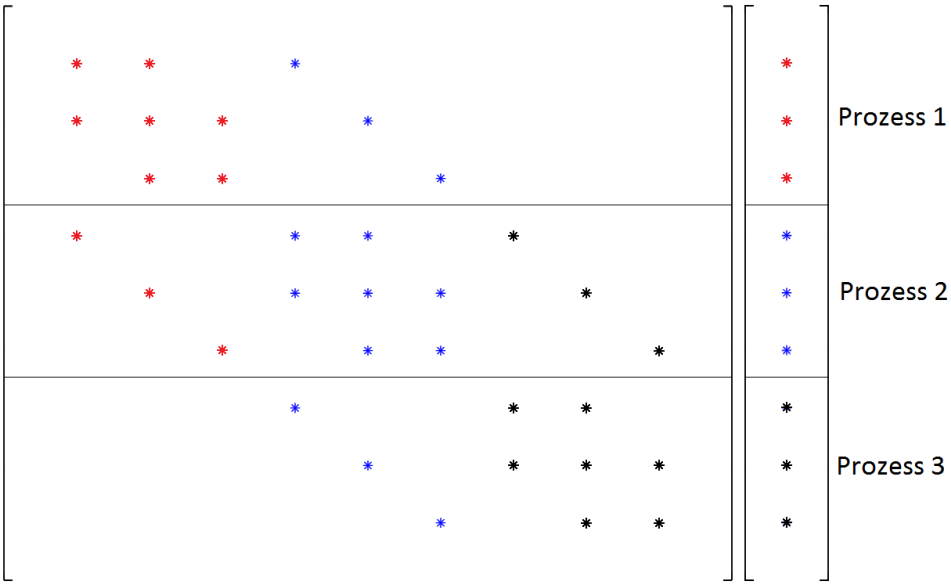

Abbildung 2: Beispiel der Datenverteilung bei einer dünnbesetzten Matrix-Vektor-Multiplikation. Die Farben der Matrixeinträge zeigen auf welchen Prozessen die Daten der Vektoreinträge liegen.

Weiterhin betrachten wir das Skalarprodukt

$$\alpha = x^T y \tag{17}$$

Jeder Prozess speichert wie bei der dünnbesetzten Matrix-Vektor-Multiplikation einen Satz von Zeilen der Vektoren x und y und kann damit lokal seinen Teil berechnen. Kommunikation wird hierbei benötigt, um diese lokalen Ergebnisse zu addieren. Dies wird in der Implementation des MINRES über ein MPI_Allreduce realisiert. Ein vereinfachtes Modell dieser Operation zeigt Abbildung 3. Dies funktioniert folgendermaßen: Ein Prozess sammelt die lokalen Ergebnisse der anderen Prozesse ein, addiert diese und schickt das erhaltene Ergebnis an die anderen Prozesse zurück.

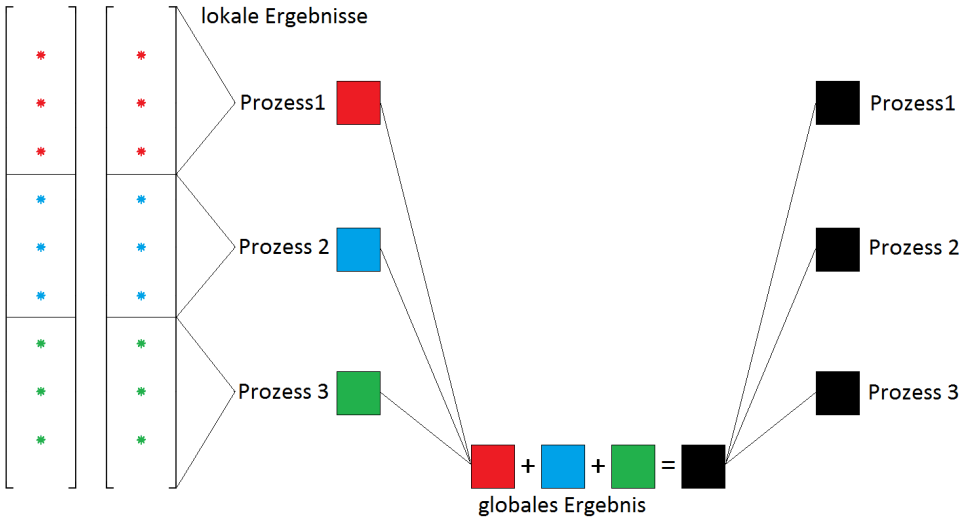

Abbildung 3: Beispiel eines Skalarproduktes mittels MPI_Allreduce

4.1.1 Beispiel von Überlappungen innerhalb eines Matrix-Vektor-Produkts

Während einer Matrix-Vektor-Multiplikation ist es bereits möglich, Kommunikation und Rechnung zu überlappen. Diese Möglichkeit bietet eine in PHIST bereits vorhandene Funktion. Dabei wird zu Beginn die Kommunikation gestartet und währenddessen mit den lokal vorhandenen Einträgen gerechnet. Wenn die Kommunikation abgeschlossen ist, werden im Anschluss die restlichen Werte berechnet. Als Beispiel wurde das in PHIST vorhandene Matrix-Vektor-Produkt einmal mit und einmal ohne Überlappungen getestet. Der folgende Plot zeigt die Auswertungen, wobei zu erkennen ist, dass das überlappende Matrix-Vektor-Produkt schneller ist. In der späteren PHIST-Implementation wird darauf allerdings verzichtet, da der Ergebnisvektor zwei mal aktualisiert werden muss und es somit hier vermutlich sinnvoller ist, nur die vollständigen Operationen mit anderen Operationen zu überlappen.

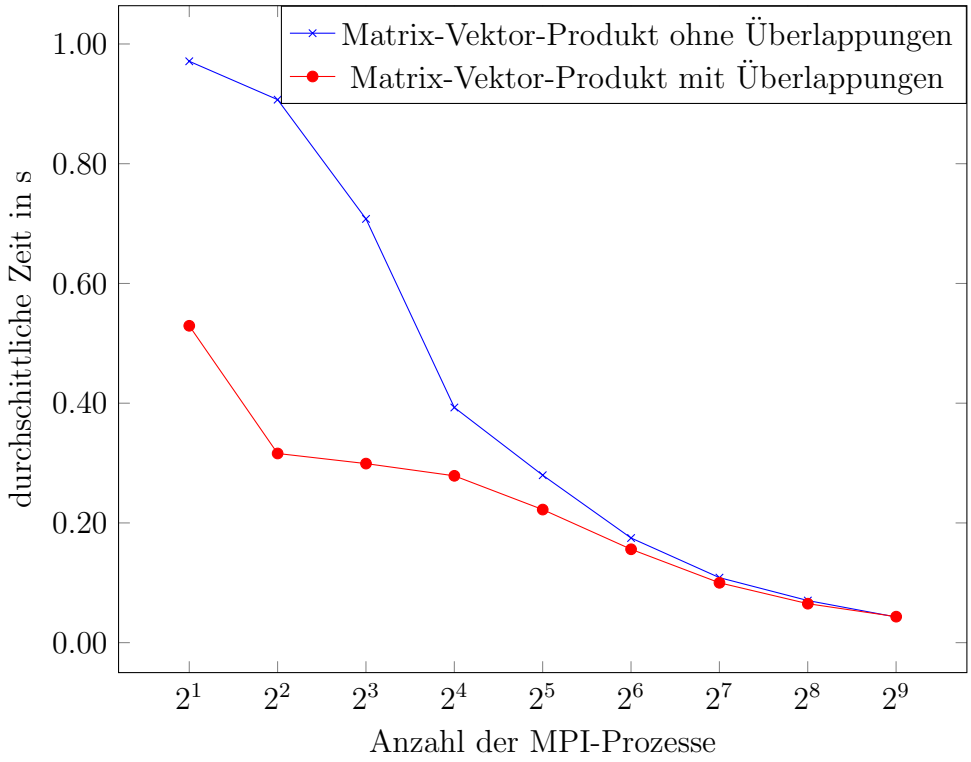

Abbildung 4: Vergleich einer Matrix-Vektor-Multiplikation mit und ohne Überlappungen für die SpinSZ28-Matrix, welche in Kapitel 5 beschrieben wird

4.2 Kerneloperationen

Um einen Einblick in die PHIST-Syntax zu erhalten, werden im Folgenden die relevanten PHIST-Kerneloperationen, welche in der Implementation des MINRES-Verfahrens verwendet wurden, in einer Tabelle vorgestellt.

PHIST-Kerneloperation	**führt folgende Berechnung durch**
mvec_vscale(v,α);	$v \leftarrow v\alpha$
mvec_add_mvec(α,v,β,w);	$w \leftarrow \alpha v + \beta w$
mvec_dot_mvec(v,w,α);	$\alpha_i \leftarrow v_i^T w_i$
mvecT_times_mvec(α,v,w,β,y);	$y \leftarrow \alpha v^T w + \beta y$
sparseMat_times_mvec(α,A,v,β,y);	$w \leftarrow \alpha Av + \beta w$
mvec_times_sdMat(v,w,M);	$v \leftarrow v - wM$

Die vorgestellten Kerneloperation sind in GHOST (General Hybrid Optimized Sparse Toolkit) [**?**] verfügbar, welches mit PHIST genutzt wird. Da die MINRES-Variante in PHIST für mehrere Gleichungssysteme gleichzeitig aufgerufen wird, wird hier zwischen einem Skalarprodukt mvec_dot_mvec und einem dichten Matrix-Matrix-Produkt

mvecT_times_mvec unterschieden. Zum genaueren Verständnis betrachten wir für beide Funktionen ein Beispiel mit konkreten Dimensionen:
mvec_dot_mvec(v,w,α) mit $v, w \in \mathbb{R}^{n\times 4}$ und $\alpha \in \mathbb{R}^4$. Dabei wird α mittels MPI_Allreduce auf allen Prozessen redundant gespeichert.
mvecT_times_mvec(M,w,v) mit $w \in \mathbb{R}^{n\times 20}$, $v \in \mathbb{R}^{n\times 4}$ und $M \in \mathbb{R}^{20\times 4}$. M wird auch hier redundant auf allen Prozessen gespeichert.
Zusätzlich wird die Funktion mvec_times_sdMat(v,w,M) mit $v \in \mathbb{R}^{n\times 4}$, $w \in \mathbb{R}^{n\times 20}$ und $M \in \mathbb{R}^{20\times 4}$ verwendet. Da diese allerdings nur lokale Rechnungen ausführt, wird diese hier nicht weiter diskutiert. In den folgenden Abschnitten wird nun erläutert, inwiefern Überlappung von Kommunikation und Rechnung möglich ist und wie es in der Implementation realisiert wurde.

4.3 Asynchrones Programmiermodell in PHIST

Um Berechnung und Kommunikation zu überlappen, bietet PHIST ein Tasking-Modell, welches mittels C-Präprozessor-Makros implementiert wurde. Dieses ermöglicht es, Teile des Programms einem Task zuzuweisen und diesen mit einem anderen Task zu überlappen. Dazu wird die Berechnung eines Tasks gestartet und die Kommunikation im Hintergrund ausgeführt. Dabei wird MPI mit OpenMP zusammen verwendet. Schauen wir uns ein Beispiel an:

```
1 PHIST_TASK_BEGIN(Task1)
    α = x^T y
3 PHIST_TASK_END_NOWAIT()
  PHIST_TASK_BEGIN(Task2)
5   β = v^T w
  PHIST_TASK_END_NOWAIT()
7 PHIST_TASK_WAIT(Task1)
  PHIST_TASK_WAIT(Task2)
9   γ = α + β
```

In diesem Beispiel können beide Skalarprodukte unabhängig voneinander berechnet werden. Kommunikation entsteht am Ende der lokalen Berechnungen durch ein MPI_Allreduce. Die Idee hinter den Tasks ist, dass mit einem PHIST_TASK_BEGIN() Teile des Algorithmus in einem Block zusammengefasst werden und diese Blöcke letztlich überlappend berechnet werden können. In dem obigen Beispiel wird ein Skalarprodukt gestartet und mit PHIST_TASK_END_NOWAIT() angewiesen, nicht auf das blockierende MPI_Allreduce zu warten, bevor der nächste Task gestartet wird. Dies bedeutet, dass während das erste Skalarprodukt noch rechnet oder gerade mit der Kommunikation begonnen hat, dass

zweite Skalarprodukt gestartet wird. Da in diesem Beispiel beide Skalarprodukte addiert werden sollen, muss auf beide Ergebnisse mit einem PHIST_TASK_WAIT() gewartet werden, um zu vermeiden, dass gegebenenfalls ein Ergebnis noch nicht verfügbar ist.
Schauen wir uns ein weiteres Beispiel an:

```
1 PHIST_TASK_BEGIN(Task1)
  α = x^T y
3 PHIST_TASK_END_NOWAIT()
  PHIST_TASK_WAIT_STEP(Task1)
5
  y = x + y
7
  PHIST_TASK_WAIT(Task1)
```

In diesem Beispiel soll zuerst ein Skalarprodukt und im Anschluss eine Vektoraddition ausgeführt werden. Das Problem, welches in diesem Beispiel entsteht, ist, dass der Vektor y für die Berechnung des Skalarproduktes benötigt, danach aber überschrieben wird. Hier kann es durch die Überlappung zum Abbruch des Programms oder zu Rechenfehlern kommen, da man nicht garantieren kann, dass das lokale Skalarprodukt fertig gerechnet wurde, bevor die Vektoraddition beginnt. Dazu wird mit dem Makro PHIST_TASK_WAIT_STEP() das Programm angehalten, bis die Rechnung fertig ist. Die Kommunikation kann gleichzeitig mit der Vektoraddition ablaufen.

4.4 Überlappung von Kommunikation und Rechnung im MINRES

Im Folgenden werden einzelne Teile der in Kapitel 2.2 vorgestellten PHIST-Variante des MINRES-Verfahrens getestet, um mögliche Überlappungen zu sehen. Zu diesem Zweck wurde die Zeit gemessen, die der Algorithmus für den Lanczos-Teil sowie für den QR-Teil benötigt. Diese Teile wurden mit SpinSZ[L]-Matrizen getestet, welche in Kapitel 5 vorgestellt werden. Da die QR-Zerlegung insgesamt keine Kommunikation und nur wenige lokale Rechenoperationen pro Iteration benötigt, ist diese für mögliche Überlappungen uninteressant. Dies kann man in den untenstehenden Grafiken sehr gut erkennen. In den Plots wird die durchschnittliche Zeit, welche die Berechnungen der einzelnen Teile benötigen, gegen die Anzahl der MPI-Prozesse geplottet.

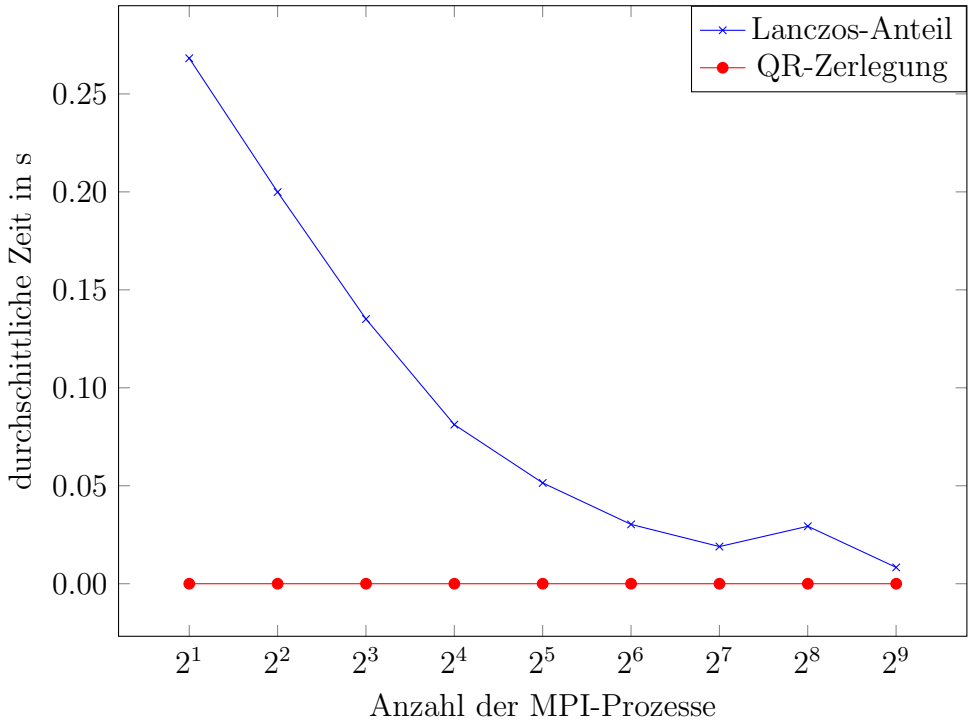

Abbildung 5: Messungen des Lanczos-Teil und der QR-Zerlegung mit einer SpinSZ26 Matrix

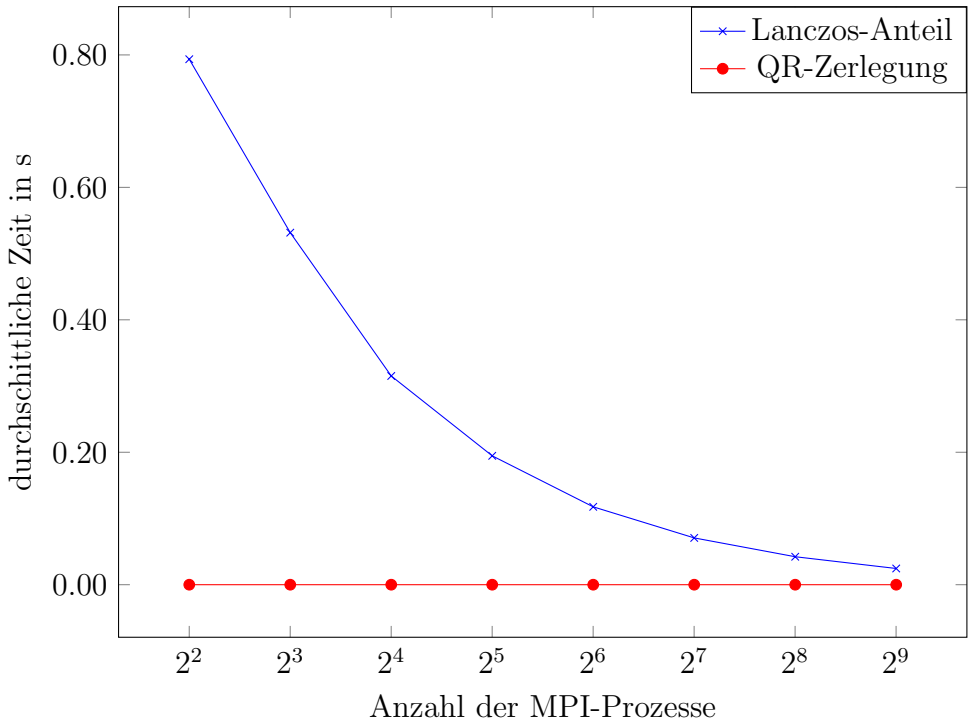

Abbildung 6: Messungen des Lanczos-Teil und der QR-Zerlegung mit einer SpinSZ28 Matrix, welche nur auf mindestens zwei Knoten läuft aufgrund des verfügbaren Speichers

Im Folgenden schauen wir uns den Lanczos-Teil des MINRES-Verfahren etwas genauer an:

Algorithm 3 Lanczos

1: **for** $i = 1 \dots maxit$ **do**
2: $v_i \leftarrow \frac{1}{\beta_i} v_i$
3: $v_{i+1} \leftarrow (\mathbb{1} - qq^T)(A - \sigma\mathbb{1})v_i$
4: $\alpha_i \leftarrow v_i^T v_{i+1}$
5: $v_{i+1} \leftarrow v_{i+1} - \alpha_i v_i - \beta_i v_{i-1}$
6: $\beta_{i+1} \leftarrow ||v_{i+1}||_2$
7: **end**

Die Normalisierung in Zeile 2 kommt ohne Kommunikation aus. Jeder Prozess skaliert dabei im Funktionsaufruf SUBR(mvec_vscale) lokal seine Vektoreinträge. Die PHIST-Funktion welche die Korrekturgleichung des Jacobi-Davidson-Operators auswertet, benötigt Kommunikation insofern, als dass der Operator ein Matrix-Vektor-Produkt sowie ein Skalarprodukt und anschließend eine Vektoraddition mit Skalierung ausführt. Die Skalarprodukte in Zeile 4 und 6 benötigen Kommunikation, wie wir in Kapitel 3.1 gesehen haben. Die Vektoraddition in Zeile 5 benötigt keine Kommunikation. Um eine Überlappung der Kommunikation einfacher zu implementieren, wird die Korrekturgleichung aus Zeile 2 nicht mehr mit dem Jacobi-Davidson-Operator berechnet, sondern einzeln mit den entsprechenden Funktionsaufrufen direkt im Algorithmus. Die Variante, welche nicht mehr die Funktion aufruft die den Jacobi-Davidson-Operator ausführt sieht folgendermaßen aus:

Algorithm 4 PHIST Variante des MINRES

1: $v_1 \leftarrow (A - \sigma\mathbb{1})v_0$
2: $M \leftarrow q^T v_1$
3: **for** $i = 1 \dots maxit$ **do**
4: $v_i \leftarrow \frac{1}{\beta_i} v_i$
5: $v_{i+1} \leftarrow \frac{1}{\beta_i}(v_{i+1} - qM)$
6: $\alpha_i \leftarrow v_i^T v_{i+1}$
7: $v_{i+1} \leftarrow v_{i+1} - \beta_i v_{i-1}$
8: $v_{i+1} \leftarrow v_{i+1} - \alpha_i v_i$
9: $\beta_{i+1} \leftarrow \|v_i\|_2$
10: $v_{i+1} \leftarrow (A - \sigma\mathbb{1})v_i$
11: $M \leftarrow q^T v_{i+1}$
12: # wende Givens Rotationen an
13: **end for**

Schauen wir uns genauer den Aufbau von Algorithmus 4 an. Bevor die Hauptschleife des MINRES startet, wird im ersten Schritt ein Teil der Korrekturgleichung mit dem Startvektor berechnet und mit dieser Berechnung im nächsten Schritt das erste Multivektor-Produkt mit den ersten Projektionsvektoren berechnet. In Zeile 4 wird nach dem Start der Hauptschleife ersteinmal der Basisvektor normalisiert. Anschließend wird im nächsten Schritt der Rest der Korrekturgleichung berechnet. Da der Vektor v_i im ersten Teil der Korrekturgleichung noch nicht normalisiert war, wird das in diesem Schritt nachgeholt. Im Detail heißt das, dass wir in Zeile 1,2,4 und 5 nichts anderes machen, als die Korrekturgleichung zu berechnen, welche in Algorithmus 3 in Zeile 3 als ein Aufruf übernommen wurde. In Zeile 7 und 8 wird der neue Vektor gegenüber den alten orthonormalisiert. Da die Vektoren für die Givens-Rotationen nicht benötigt werden, wird im Anschluss für die nächste Iteration der erste Teil der Korrekturgleichung, sowie das erste Skalarprodukt mit den Projektionsvektoren berechnet. Diese Funktionen wurden so angeordnet, dass sie im asynchronen Falle bestmöglich überlappen ohne den Algorithmus zu verändern. Der folgende Algorithmus zeigt, an welcher Stelle die Tasks angeordnet sind und an welcher Stelle auf die Tasks gewartet wird.

Algorithm 5 asynchrone Variante des MINRES mit Tasks

1: $v_1 \leftarrow (A - \sigma \mathbb{1})v_0$
2: $M \leftarrow q^T v_1$
3: **for** $i = 1 \dots maxit$ **do**
4: $v_i \leftarrow \frac{1}{\beta_i} v_i$
5: wenn $i > 1$ warte auf Task 4
6: $v_{i+1} \leftarrow \frac{1}{\beta_i}(v_{i+1} - VM)$
7: Starte Task 1: $\alpha_i \leftarrow v_i^T v_{i+1}$
8: warte auf lokale Rechnungen von Task 1 und starte Kommunikation
9: $v_{i+1} \leftarrow v_{i+1} - \beta_i v_{i-1}$
10: warte auf Ergebnis von Task 1
11: $v_{i+1} \leftarrow v_{i+1} - \alpha_i v_i$
12: $v_{i-1} \leftarrow v_i$
13: $v_i \leftarrow v_{i+1}$
14: Starte Task 2: beginne Kommunikation des Matrix Vektor Produkts
15: Starte Task 3: $\beta_{i+1} \leftarrow \|v_i\|_2$
16: warte auf lokale Rechnungen von Task 3 und starte Kommunikation
17: warte auf Task 2
18: $v_{i+1} \leftarrow (A - \sigma \mathbb{1})v_i$
19: starte Task 4: $M \leftarrow q^T v_{i+1}$
20: warte auf Task 3
21: # Wende Givens Rotationen an
22: **end for**

Der erste Task wird in Zeile 7 gestartet und enthält das Skalarprodukt α_i. Das Programm wartet hier solange, bis die lokalen Berechnungen abgeschlossen sind, da im nächsten Schritt der Vektor v_i überschrieben wird und es sonst zu Fehlern kommen kann. Wenn der Task seine lokalen Rechnungen beendet hat startet er die Kommunikation und beginnt mit dem ersten Teil der Orthonormalisierung des neuen Vektors gegenüber dem alten, da die Ergebnisse aus dem Task hier noch nicht benötigt werden. Nach diesem Schritt wird auf den Task gewartet und danach der zweite Teil der Orthonormalisierung berechnet. Anschließend startet der zweite Task. Dieser enthält die Kommunikation des Matrix-Vektor-Produkts aus Zeile 19. Im nächsten Schritt wird Task 3 gestartet, welcher die Norm des Vektors v_i berechnet. Hier wird ebenfalls auf die lokalen Ergebnisse gewartet, bevor das Programm weiterläuft. Bevor im nächsten Schritt die Matrix-Vektor-Multiplikation ausgerechnet wird, wird auf das Ende der Kommunikation von Task 2 gewartet. Anschließend wird in Task 4 das Skalarprodukt berechnet. Da diese Ergebnisse

nicht mehr für die Givens Rotationen verwendet werden, wird auf Task 4 erst wieder zu Beginn der nächsten Iteration gewartet. Letztlich muss noch auf die Norm von Task 3 gewartet werden, da diese für die folgenden Givens Rotationen benötigt werden. Das folgende Diagramm veranschaulicht die Nebenläufigkeit von Algorithmus 5.

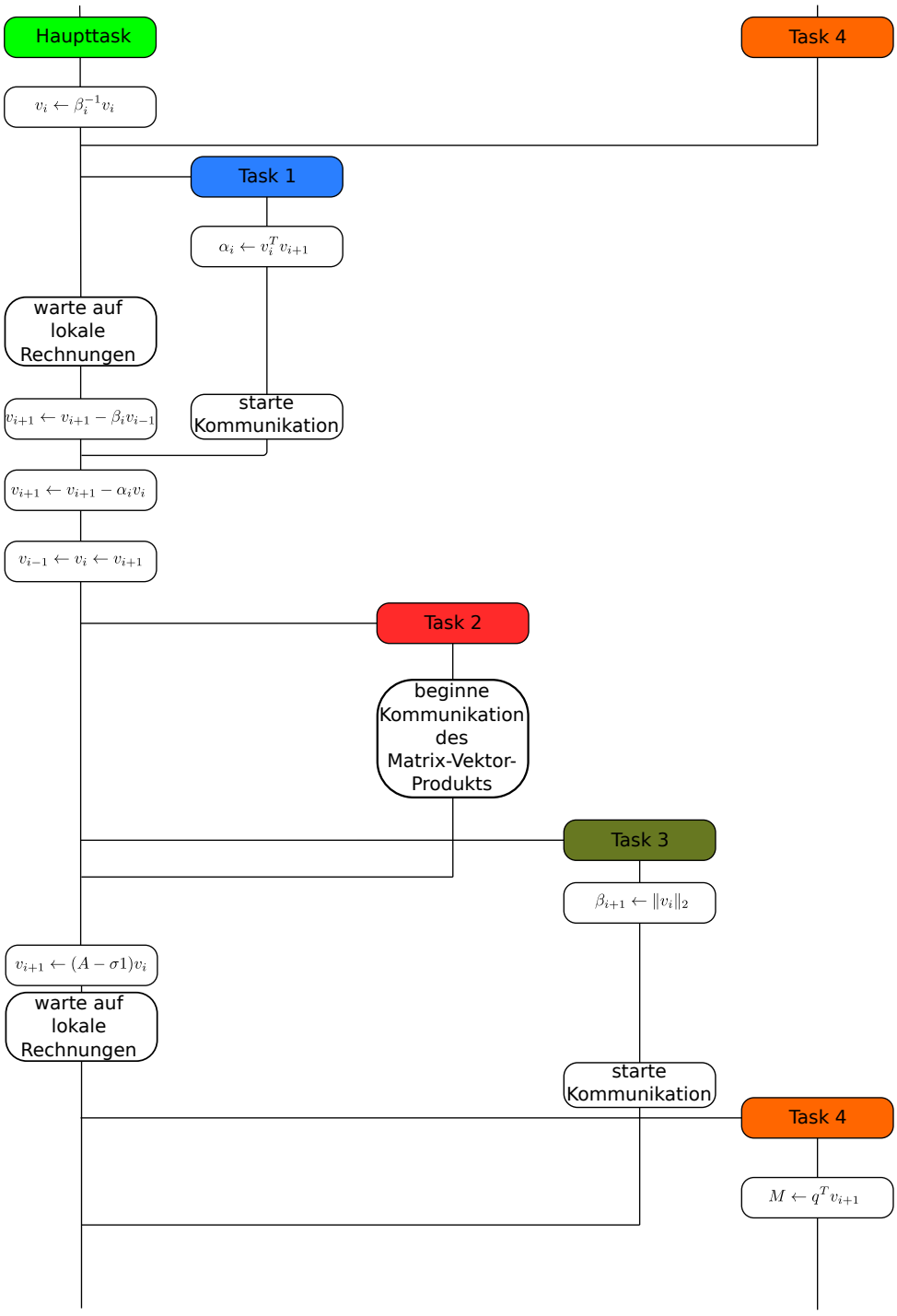

Abbildung 7: Darstellung der Nebenläufigkeit des asynchronen MINRES-Verfahrens als Diagramm

Im nächsten Kapitel werden die Messungen des asynchronen MINRES ausgewertet und einzelne Funktionen genauer betrachtet.

5 Auswertung

Die folgenden Tests wurden auf dem Emmy-Cluster des Rechenzentrums Erlangen[4] gemacht. Dieser besteht aus 560 Rechenknoten mit jeweils 2 Xeon 2660v2 „Ivy Bridge“ CPU's. Diese sind mit jeweils 2.2 Ghz getaktet und besitzen 10 Kerne sowie 26 MB gemeinsamen Cache pro CPU. Jeder Knoten verfügt über 64 GB RAM. Bei den Tests wird auf jeder CPU ein MPI Prozess ausgeführt.
Zuerst betrachten wir die Messungen des MINRES einmal mit und einmal ohne Tasks, bevor wir uns Messungen der einzelnen Funktionen im Algorithmus anschauen. Die Messungen wurden mit symmetrischen SpinSZ[26]- und SpinSZ[28]-Matrizen durchgeführt, wobei eine SpinSZ[L]-Matrix $\binom{L}{\frac{L}{2}}$ Zeilen und Spalten hat. Diese dünn-besetzten Matrizen stammen aus einer Anwendung aus der Quantenphysik und dienen der Modellierung von Interaktionen von Elektronen-Spins.

5.1 Vergleich des MINRES-Algorithmus mit und ohne Tasks

In den folgenden Abbildungen sehen wir die Laufzeit des gesamten MINRES-Algorithmus mit und ohne Tasks auf 1-256 Knoten. Dabei lässt sich beobachten, dass der Durchlauf mit Tasks langsamer ist als ohne Tasks. Insbesondere sieht man, dass der Abstand auf mehreren Knoten größer wird. Idealerweise sollte der asynchrone MINRES schneller sein. Im weiteren Verlauf des Kapitels analysieren wir die einzelnen Funktionen welche in einem Task sind, um zu zeigen, aus welchem Grund der asynchrone MINRES langsamer ist.

[4] https://www.rrze.fau.de/dienste/arbeiten-rechnen/hpc/systeme/emmy-cluster.shtml

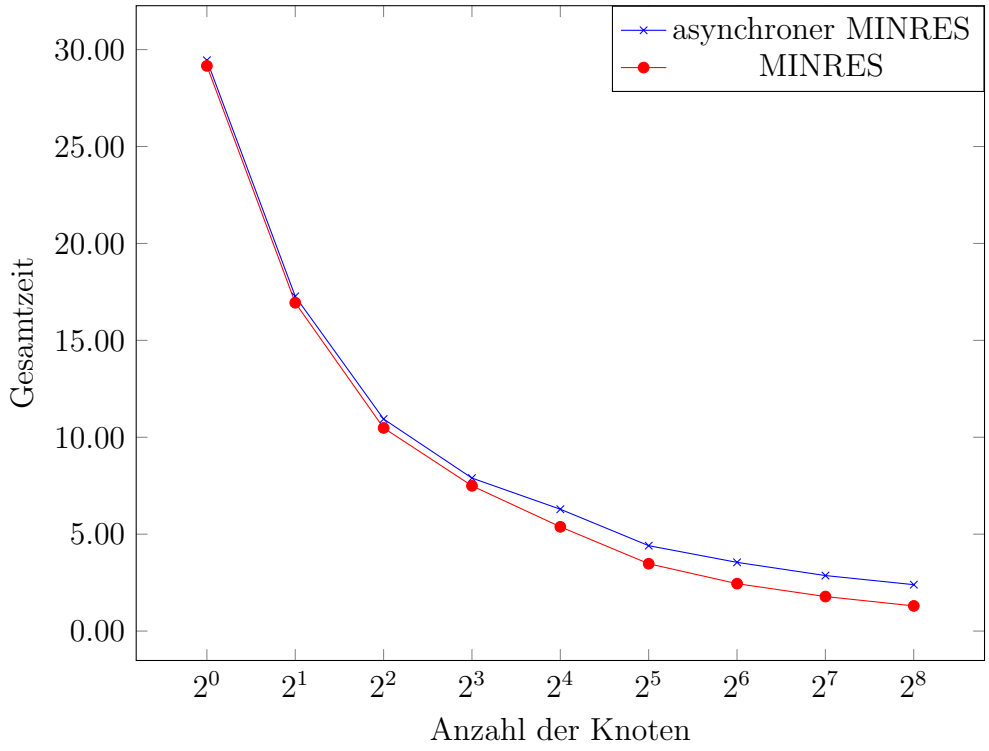

Abbildung 8: Vergleich des MINRES-Verfahrens mit und ohne Task gemessen mit einer SpinSZ26-Matrix

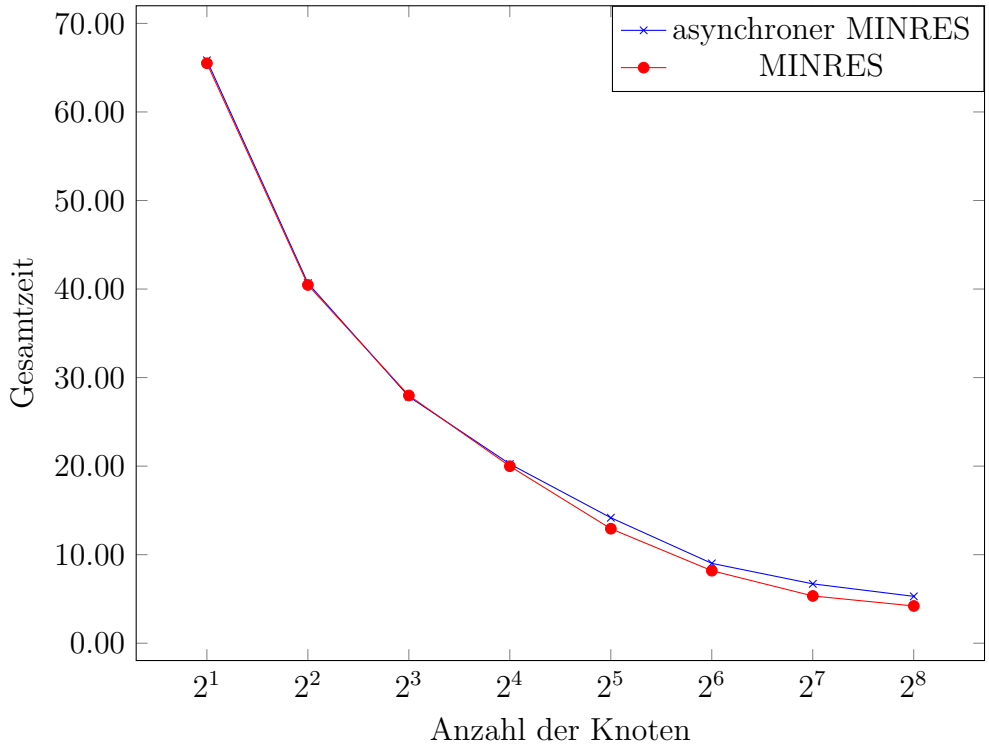

Abbildung 9: Vergleich des MINRES-Verfahrens mit und ohne Task, gemessen mit einer SpinSZ28-Matrix

5.2 Auswertung der Funktionen

Nachfolgend werden Messungen der relevanten Funktionen im MINRES gezeigt. Dabei wurden Messungen mit den Matrizen SpinSZ[26] und SpinSZ[28] durchgeführt. Da die Ergebnisse gleichwertig sind, werden hier nur die Messungen der größeren SpinSZ[28]-Matrix gezeigt.

5.2.1 phist_DsparseMat_times_mvec_vadd_mvec

Zuerst schauen wir uns das Ergebnis der Funktion phist_DsparseMat_times_mvec_vadd_mvec an. Diese führt die Operation

$$y = \alpha(A - \sigma\mathbb{1})x + \beta y \tag{18}$$

mit $A \in \mathbb{R}^{n\times n}$ dünnbesetzt und $x, y \in \mathbb{R}^{n\times m}, m \lll n$, aus.

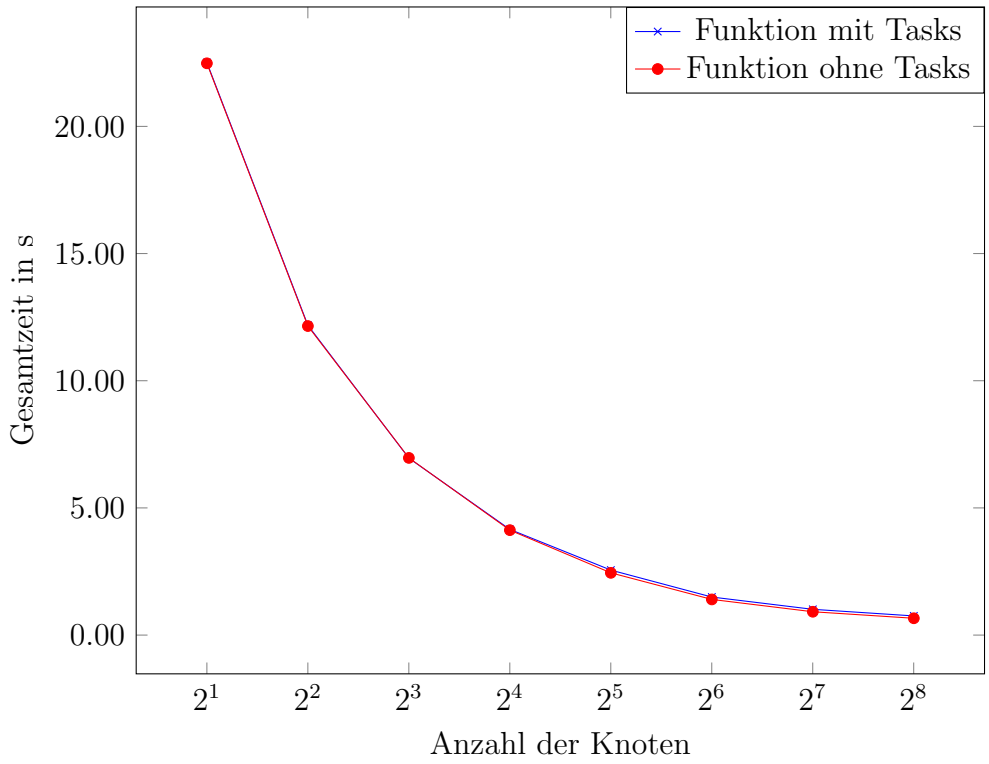

Abbildung 10: Vergleich phist_DsparseMat_times_mvec_vadd_mvec mit und ohne Tasks, gemessen mit einer SpinSZ28-Matrix

Im Plot wird die Gesamtzeit der Rechnungen der Funktion gegen die Anzahl der Knoten, auf denen gerechnet wurde, geplottet. Wie man im Plot sehen kann, beeinflusst der asynchrone Modus die Funktion nicht. Der Grund dafür ist, dass sich der Funktionsaufruf selbst nicht in einem Task befindet, da die Kommunikation der Funktion in einem eigenen Schritt ausgeführt wird (siehe nächstes Kapitel) und die Funktion somit nur lokale

Rechenoperationen ausführt. Damit hätte es keinen Nutzen die Funktion in einen Task zu stecken.

5.2.2 phist_DsparseMat_times_mvec_communicate

Die Funktion phist_DsparseMat_times_mvec_communicate wird vor dem Aufruf der eigentlichen Matrix-Vektor-Multiplikation $Ax = y$ benötigt. Sie ist für den Austausch der von den anderen Prozessen benötigten Elemente des Vektors x verantwortlich. Somit ist es möglich, dass Matrix-Vektor-Produkt lokal durchzuführen. Wie wir im Plot sehen können, steigt die Berechnungszeit der Funktion ohne Tasks bei zwei und vier Knoten an, bevor sie fällt. Der Anstieg bei wenigen Knoten lässt sich dadurch erklären, dass der Kommunikationsaufwand pro Prozess zunächst steigt. Bei vielen Knoten muss jeder Prozess wieder weniger Daten verschicken und empfangen.

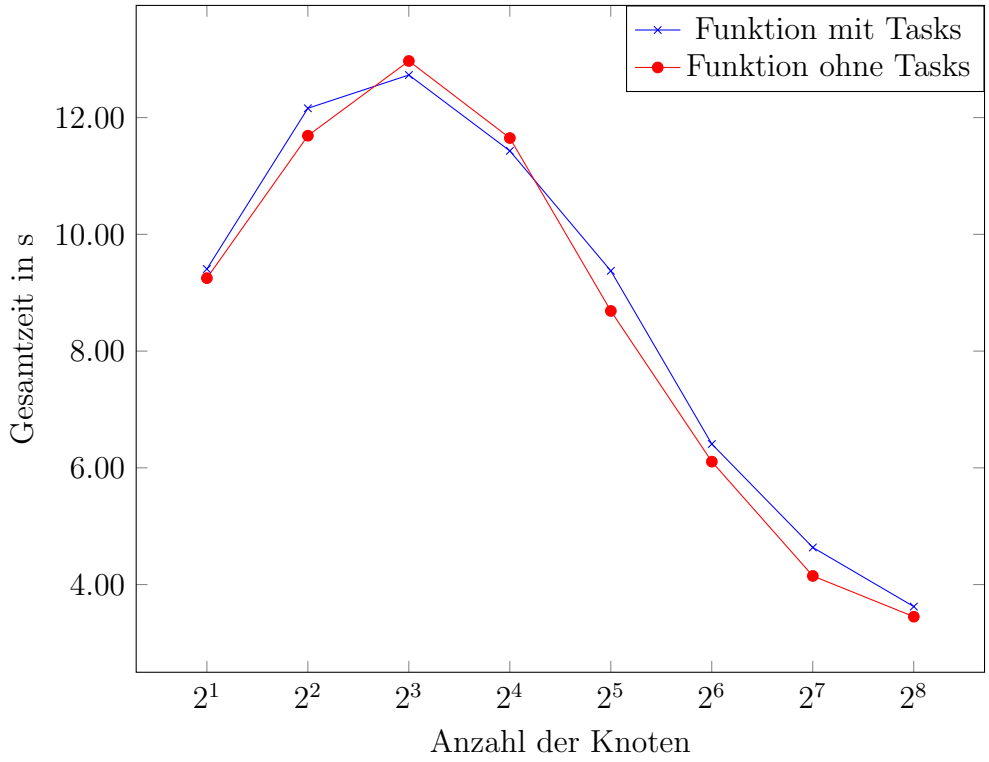

Abbildung 11: Vergleich phist_DsparseMat_times_mvec_communicate mit und ohne Tasks, gemessen mit einer SpinSZ28-Matrix

5.2.3 phist_DmvecT_times_mvec

Die Funktion phist_DmvecT_times_mvec führt ein dichtes Matrix-Matrix-Produkt mit einer MPI_Allreduction aus. Im folgenden Plot ist die Funktion einmal mit und einmal ohne Tasks dargestellt. Was sofort auffällt, ist der große Zeitunterschied bei einem Knoten. Da bei einem Knoten mit zwei Prozessen Kommunikation nur im Knoten aber nicht über

das Netzwerk auftritt, sollte die Berechnungszeit nur unwesentlich abweichen. Die Vermutung ist hier nun, dass es daran liegt wie OpenMP in dieser Funktion genutzt wird. Zum Vergleich dazu wurde ein Testprogramm geschrieben, welches simuliert, wie OpenMP in dieser Funktion aufgerufen wird. Der Code zu diesem Programm ist im Anhang zu finden. Dazu startet das Programm in jeder Iterationen einen neuen Thread. Thread Nummer 1 ruft nun OpenMP auf und lässt parallel eine Schleife abarbeiten. In der nächsten Iteration wird nun Thread 2 gestartet welcher OpenMP aufruft und wieder parallel eine Schleife abarbeitet. Diese Aufrufe von OpenMP von verschiedene Threads führen vermutlich zu einem internen Konflikt bei der Threadverwaltung der OpenMP-Implementierung. Im untenstehenden Balkendiagramm kann man erkennen, dass das Testprogramm mit Tasks um einen Faktor 1.5 langsamer ist als ohne, was auf diesen Konflikt zurückzuführen ist. Somit ist hier ein Problem zwischen OpenMP und dem Tasking zu erkennen, welches von den Entwicklern des ESSEX-Projektes genauer analysiert und behoben werden sollte.

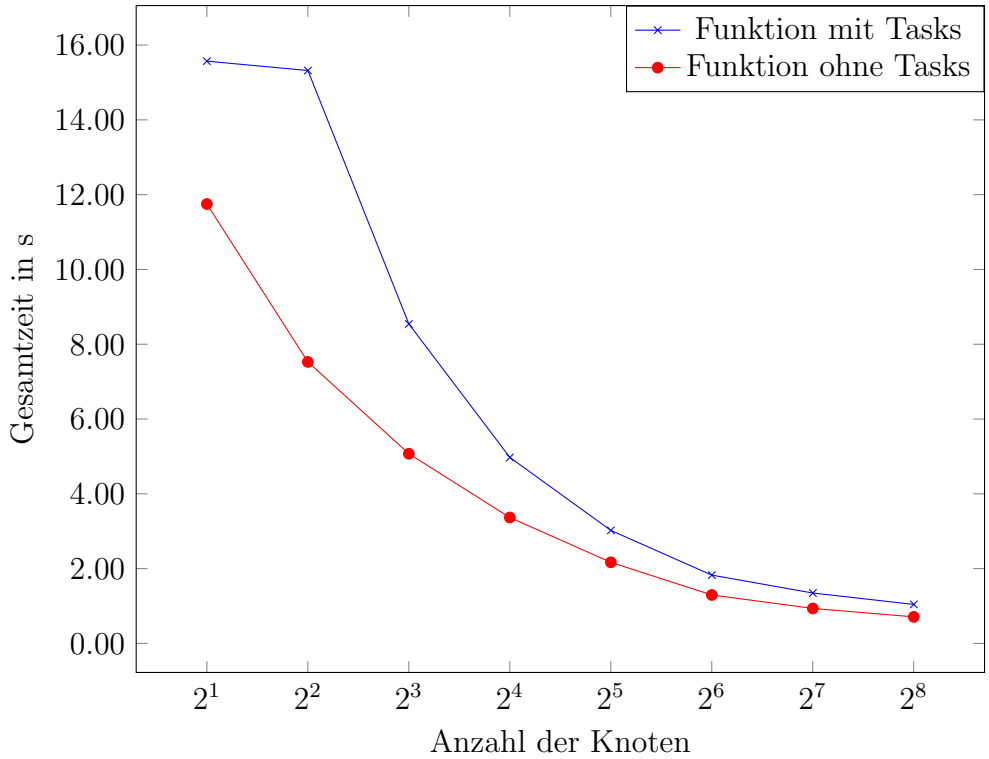

Abbildung 12: Vergleich phist_DmvecT_times_mvec mit und ohne Tasks, gemessen mit einer SpinSZ28-Matrix

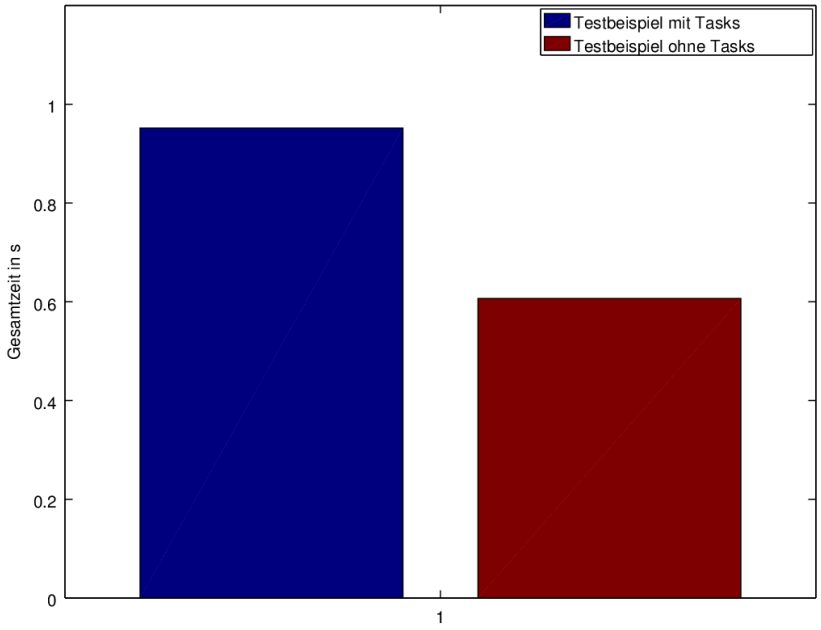

Abbildung 13: Vergleich des Testprogramms auf einem Prozess mit und ohne Tasks

5.2.4 phist_Dmvec_dot_mvec

Bei der Funktion phist_Dmvec_dot_mvec wurden die selben Beobachtungen wie bei der Funktion phist_DmvecT_times_mvec gemacht. Diese Funktion berechnet ein Skalarprodukt

$$m = x^T y \tag{19}$$

mit $x, y \in \mathbb{R}^n$ und $m \in \mathbb{R}$. Hier lässt sich ebenfalls beobachten, dass der Algorithmus mit Tasks länger benötigt als ohne Tasks. Zusätzlich kann man sehen, dass die Operation bei vier und acht Knoten länger benötigt als bei zwei Knoten. Schauen wir uns in der folgenden Tabelle die minimale Zeit an, die ein Prozess zum Ausführen der Funktion benötigt hat, können wir erkennen, dass sich bei der Verdoppelung der Knotenanzahl die Berechnungszeit halbiert, was dem Idealfall entspricht.

Knoten	**Minimale Zeit eines Aufrufs der Funktion**
2	$7.391 \cdot 10^{-3}$
4	$3.600 \cdot 10^{-3}$
8	$1.885 \cdot 10^{-3}$
16	$9.871 \cdot 10^{-4}$
32	$4.840 \cdot 10^{-4}$
64	$2.041 \cdot 10^{-4}$

Der Verlauf der Kurve ohne Tasks ist auf eine schlechte Lastverteilung während der Kommunikation der Funktion phist_DsparseMat_times_mvec_communicate zurückführen. Dies kommt daher, dass einige Prozesse mit mehr Prozessen während der Kommunikation des Matrix-Vektor-Produktes kommunizieren müssen als andere Prozesse. Der Unterschied mit Tasking ist vermutlich wieder auf die OpenMP-Problematik zurückzuführen.

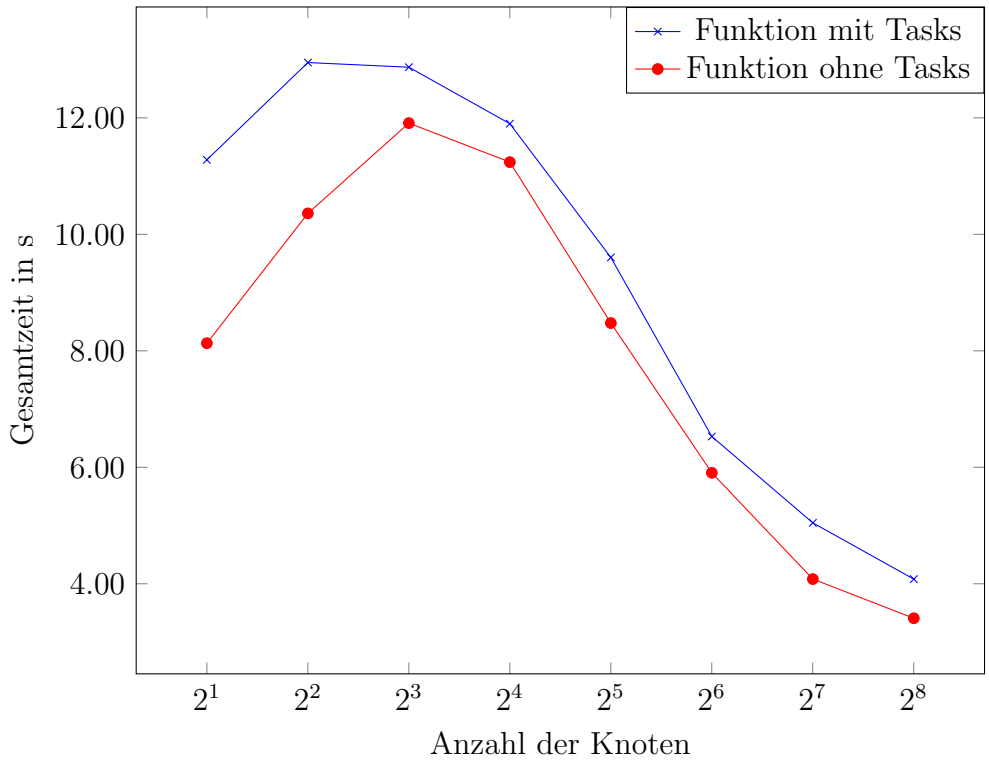

Abbildung 14: Vergleich phist_Dmvec_dot_mvec mit und ohne Tasks, gemessen mit einer SpinSZ28-Matrix

Da der Kommunikationsaufwand der SpinSZ-Matrizen besonders hoch im Vergleich zu Matrizen aus anderen Anwendungsgebieten ist, würde somit auch die Idealzeit der Funktion phist_Dmvec_dot_mvec nicht ausreichen, um die Kommunikation der Funktion phist_DsparseMat_times_mvec_communicate auszufüllen.

6 Fazit

Im Laufe dieser Arbeit wurde eine asynchrone Variante des MINRES-Algorithmus implementiert, welcher in der Zukunft als innerer iterativer Löser des Jacobi-Davidson-Algorithmus in PHIST dienen soll. Wie wir gesehen haben, lässt das Lanczos-Verfahren Spielraum für mögliche Überlappungen von Kommunikation und Rechnung. Daraufhin wurden die entsprechenden Funktion mittels des bereitgestellten Taskings so angeordnet, dass sie bestmöglich überlappen mit dem Ziel den Algorithmus zu beschleunigen.
In den darauffolgenden Test, welche mit symmetrisch indefiniten SpinSZ[L]-Matrizen durchgeführt wurden, wurden die Ergebnisse des MINRES-Verfahrens mit und ohne Tasking gegenübergestellt. Dabei ließ sich beobachten, dass das Tasking den Algorithmus langsamer werden lässt. Um den Grund zu ermitteln, warum der Algorithmus nun langsamer ist, wurden die einzelnen Funktionen, welche während des Lanczos-Prozess aufgerufen werden, genauer untersucht. Diese Tests wurden ebenfalls mit SpinSZ[L]-Matrizen durchgeführt. Diese Tests haben gezeigt, dass hier noch Optimierungsbedarf innerhalb der Funktionsaufrufe besteht. Bei der Funktion phist_DsparseMat_times_mvec_communicate sah man eine schlechte Lastverteilung bedingt dadurch, dass einige Prozesse mehr kommunizieren müssen als andere, was daran liegt wie die Matrix und die Vektoren auf den Prozessen verteilt sind. Dieses Problem beeinflusste auch die Funktion phist_Dmvec_dot_mvec, welche direkt nach einem phist_DsparseMat_times_mvec_communicate aufgerufen wird. Weiterhin entsteht hier das Problem, dass die lokalen Rechnungen nicht ausreichen, um die Kommunikation der Funktion phist_DsparseMat_times_mvec_communicate auszufüllen. In der Funktion phist_DmvecT_times_mvec kam es zu Problemen mit den Aufrufen von OpenMP von verschiedenen Threads nacheinander. Hier muss für die Zukunft geprüft werden, inwiefern dieses Problem besser gelöst werden kann.

Anhang

A Testprogramm zur Simulation von OpenMP Aufrufen

```
1  #include <omp.h>

3  void *thread_fun(void *x_)
   {
5    double x = *(double*)x_;
     #pragma omp parallel for reduction(+:x)
7    for(int i = 0; i < 1000; i++)
     {
9      x += i;
     }
11   *(double*)x_ = x;

13   // do nothing
     return NULL;
15 }

17 int main()
   {
19   double wtime = omp_get_wtime();

21   double x = 0;
     for(int i = 0; i < 1600; i++)
23   {
       #ifndef WITH_THREADS
25     thread_fun(&x);
       #else
27     pthread_t thread;

29     if( pthread_create(&thread, NULL, thread_fun, &x) )
       {
31       printf("Error creating thread\n");
         return 1;
33     }

35     if( pthread_join(thread, NULL) )
       {
37       printf("Error joining threads\n");
         return 1;
39     }
       #endif
41   }
```

```
43
      wtime = omp_get_wtime() - wtime;
45    printf("Needed %e seconds\n",wtime);
      printf("Calculated %e\n", x);
47
      return 0;
49  }
```

B ESSEX-Projekt

Das für diese Arbeit verwendete Softwarepaket PHIST ist auf https://bitbucket.org/essex/phist als Open-Source-Software zu finden.
Der Code, welcher für diese Arbeit geschrieben und getestet wurde ist auf https://bitbucket.org/essex/phist/commits/tag/patrick_ba zu finden.

Erklärung

Hiermit versichere ich, dass ich meine Abschlussarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: .....

(Unterschrift)