Simulation eines Vielkörpersystems auf einem verteilten Rechner

Masterarbeit

Adrian Pegler

14. Juli 2018

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe Scientific Computing

Betreut durch: Prof. Dr. Steffen Börm
Dipl.-Inf. Sven Christophersen

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 14. Juli 2018

Zusammenfassung

Um gravitationelle Wechselwirkungen in unserer Galaxie berechnen zu können, müssen Gleichungen gelöst werden, die großen, vollbesetzten Matrizen entsprechen. Eine Möglichkeit derartige Matrizen anzunähern sind $\mathcal{H}^2$-Matrizen . Als Approximationsmethode kann dazu Interpolation genutzt werden. Durch diese Methodik kann die Berechnung der Gravitationskräfte von n Himmelskörpern und k Interpolationspunkten in $\mathcal{O}(kn)$ Zeiteinheiten gelöst werden. Die Approximation durch Interpolation konvergiert für steigendes k exponentiell gegen die Lösung. Da in der Regel $k \ll n$ gilt, ist dieser Ansatz wesentlich effizienter als das vollbesetzte Problem zu lösen.

Für die Dimension einer Galaxie wäre aber auch dieser Ansatz nicht ausreichend. Daher wird ferner eine Möglichkeit vorgestellt die Berechnung auf viele Rechner zu verteilen, die durch ein Netzwerk verbunden sind. Dabei beschränken wir uns zunächst auf das Message-Passing-Modell und stellen eine Implementierung unter Verwendung der Bibliothek MPI vor.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Vorbemerkungen

Ich verwende in diesem Dokument den nach Herrn Prof. Dr. H. Laue benannten *Lauehaken*: Sei $n \in \mathbb{N}$, dann bezeichnet $\underline{n}$ die Menge $\{1, \ldots, n\}$. In Anlehnung an die Schreibweise $\mathbb{N}_0$ für die natürlichen Zahlen $\mathbb{N} \cup \{0\}$ bezeichnet $\underline{n}_0$ die Menge $\{0, \ldots, n\}$.

1.1. Motivation

Angeblich soll Sir Isaac Newton durch das Fallen eines Apfels die grundlegende Idee gehabt haben, dass die Mechanik des Himmels und der Erde doch die selben sein könnten [Stukeley 1936]. Jeder Schüler hat von dieser Geschichte gehört - und ob sie nun wahr ist oder nicht, sein Gesetz der nicht-relativistischen Gravitation ist bis heute eine wichtige Formel in der Physik.

Seien für zwei Körper die Massen m_1, m_2 und Positionen x_1, x_2 gegeben. Dann wirkt durch die Masse m_2 eine Kraft f auf die Masse m_1, die sich durch

$$f = \gamma m_1 m_2 \frac{x_2 - x_1}{\|x_2 - x_1\|_2^3} \tag{1.1}$$

ergibt [Newton 1833]. Dabei ist $\gamma \approx 6{,}67408 * 10^{-11} m^3 kg^{-1} s^{-2}$ die Gravitationskonstante [Mohr u. a. 2016] und $\|\cdot\|_2 : \mathbb{R}^d \to \mathbb{R}^d$ die euklidische Norm $z \mapsto \sqrt{\sum_{i \in \underline{d}} |z|^2}$. Im weiteren werden die Position eines Körpers und der Körper assoziiert.

Diese Formel für zwei Körper auszurechnen stellt zunächst kein Problem dar. Von größerem Interesse für die Wissenschaft sind aber Mehrkörperprobleme, also die gravitationellen Wechselwirkungen zwischen einer Menge von Körpern. Seien im folgenden also $n \in \mathbb{N}$ und Menge Ω von n Körpern gegeben. Für $x \neq y \in \Omega$ und zugehörigen Massen m_x und m_y wirkt mit Gleichung 1.1 durch den Körper y eine Kraft

$$f_{xy} = \gamma m_x m_y \frac{y - x}{\|y - x\|_2^3} \tag{1.2}$$

auf den Körper x. Will man nun die kumulative Kraft, die durch alle Körper aus Ω auf

den Körper x wirken, berechnen muss folgende Gleichung gelöst werden:

$$f_x = \sum_{\substack{y \in \Omega \\ y \neq x}} \gamma m_x m_y \frac{y - x}{\|y - x\|_2^3} \tag{1.3}$$

[Börm 2016]. Für die Simulation aller auftretenden Gravitationskräfte, müssen also für alle n Körper jeweils $(n-1)$ Therme berechnet werden. Das Problem hat also eine Komplexität von $\mathcal{O}(n^2)$.

Problematisch wird dies, wenn man sich beispielsweise die Dimensionen unseres Sonnensystems vergegenwärtigt. Laut NASA[1] besteht die Milchstraße aus etwa 100 Milliarden (also 10^{11}) Sonnen. Um sämtliche gravitationelle Wechselwirkungen in unserer Galaxie zu berechnen, müssten also ungefähr $(10^{11})^2 = 10^{22}$ Terme gelöst werden. Geht man davon aus, dass ein aktueller Prozessor nicht mehr als eine Milliarde Terme pro Sekunde ausrechnen kann, so benötigt er etwa 10^{13} Sekunden, also mehr als 300.000 Jahre für einen einzigen Simulationsschritt. Ein Problem dieser Größenordnung könnte also nicht in annehmbarer Zeit gelöst werden.[Börm 2016]

In dieser Arbeit wird ein zweigleisiger Ansatz am vorliegenden Beispiel vorgestellt, mit dessen Hilfe Probleme dieser Größen- und Komplexitätsklasse in den Griff zu bekommen sind.

1.2. Ziele

Der erste Teil des Ansatzes ist eine Reduktion der Komplexitätsklasse des Gravitationsproblems. Gleichung 1.2 definiert eine vollbesetzte Matrix. Es gibt Möglichkeiten solche Matrizen speicherplatzsparend durch Matrizen mit deutlich niedrigerem Rang anzunähern. Eine dieser Möglichkeiten ist die Darstellung als hierarchische Matrix. Diese Technik soll genutzt werden, um das vorliegende Problem in linearer Komplexität bewältigen zu können.

Dieser Ansatz alleine wird aber nicht ausreichen um das vorliegende Problem ausreichend schnell zu lösen. Wir werden daher zusätzlich nach einer Möglichkeit suchen, den Algorithmus parallel von vielen Rechnern gleichzeitig lösen zu lassen. Wir werden dazu einen Algorithmus vorstellen, der dies bewerkstelligt, und zeigen, dass dieser die Arbeit optimal auf die beteiligten Prozesse aufteilen kann.

[1] World Book at NASA [2007]: https://web.archive.org/web/20090412172631/http://mynasa.nasa.gov/-worldbook/galaxy_worldbook.html
Archivierte url: http://mynasa.nasa.gov/worldbook/galaxy_worldbook.html vom 12. April 2009. Abgerufen am 1. Mai 2018.

1.3. Aufbau

Zunächst werden die Grundlagen der Arbeit vorgestellt. Den Anfang bilden die hierarchischen Matrizen. Zunächst werden allgemeine Möglichkeiten vorgestellt, eine Matrix als hierarchische Matrix darzustellen. Im Anschluss werden einige Möglichkeiten der Optimierung dieser Darstellung vorgestellt, die uns schließlich zu den sogenannten $\mathcal{H}^2$-Matrizen führt, welche wir für die Darstellung des Gravitationsproblems nutzen wollen.

In Abschnitt 2.2 wird eine Zusammenfassung der Entwicklung der Computer hin zu Parallelität gegeben. Außerdem werden kurz Herausforderungen im Zusammenhang mit parallel arbeitenden Algorithmen erläutert. Im Anschluss wird mit dem Message-Passing-Modell ein Modell vorgestellt, mit dem parallele Algorithmen strukturiert und viele der Herausforderungen umgangen werden können.

Abschnitt 2.3 widmet sich einer Bibliothek, die das Message-Passing-Modell umsetzt: das Messege-Passing-Interface. Diese Bibliothek werden wir später nutzen, um unseren Algorithmus zu parallelisieren.

In Kapitel 3 wird zunächst die Ausgangssituation für unseren Ansatz vorgestellt, also ein nicht-optimierter, nicht-parallel arbeitender Algorithmus zur Berechnung der gravitationellen Wechselwirkungen. Dann werden wir in Abschnitt 3.2 eine Variante vorstellen, die die Struktur der $\mathcal{H}^2$-Matrizen nutzt, um die Berechnungen deutlich effektiver durchzuführen. Schließlich wird in Abschnitt 3.3 dieser Algorithmus noch um Möglichkeiten der Parallelisierung erweitert.

Der erarbeitete Algorithmus muss natürlich noch evalutiert werden. In Kapitel 4 werden wir zunächst eine theoretische Abschätzung der Komplexitätsklasse des parallel arbeitenden Algorithmus vornehmen. Im Anschluss werden die theoretischen Überlegungen durch Laufzeitmessung überprüft.

Den Abschluss bilden Fazit und Ausblick.

ist das wichtig oder kann das Weg?: Hier wird zusammengefasst, was erreicht wurde und aufgeführt, in welchen Richtungen noch weitere Arbeit möglich und notwendig ist.

Kapitel 2

Grundlagen

2.1. Hierarchische Matrizen

Erinnern wir uns an Gleichung 1.2 aus der Einleitung, in welcher die gravitationelle Wechselwirkung zwischen zwei Körpern x und y beschrieben wurde. Die Struktur dieser Formel erinnert stark an Matrizen. Ein Ansatz, um große vollbesetzte Matrizen speicherplatzsparend aufzustellen und Berechnungen zeiteffektiv durchzuführen, besteht darin, sie als hierarchische Matrizen darzustellen. Im Folgenden sollen daher die Grundlagen für allgemeine $\mathcal{H}$- und $\mathcal{H}^2$-Matrizen eingeführt werden. Für ausführlichere Arbeiten zu dem Thema der hierarchischen Matrizen sei auf die Arbeiten von Grasedyck [2001] und Hackbusch [1999] verwiesen.

Quellenangaben einfügen: vgl. Nicht lokale Operationen

Erarbeitet wurde das Konzept der hierarchischen Matrizen unter anderem zur effektiven Lösung Fredholmscher Integralgleichungen:

$$u(x) = \int_\Omega u(x)\, g(x,y)\, \forall x \in \Omega.$$

Dabei ist $\Omega \subseteq \mathbb{R}^d$ und $g\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ eine sogenannte *Kernfunktion*. Diskretisiert man diese Gleichung mit dem Galerkinverfahren unter Verwendung einer Finite-Elemente-Basis $(\phi_i)_{i\in I}$ erhält man die zugehörige Matrix

$$G_{ij} = \int_\Omega \int_\Omega \phi_i(x)\, g(x,y)\, \phi_j(y)\ dy\, dx, \quad \forall i,j \in I.$$

Um die Integration über $\Omega \times \Omega$ aufspalten zu können, ist es notwendig, dass die Variablen in der Kernfunktion getrennt werden können. Die Kernfunktion g muss also in der Form

$$g(x,y) = \sum_{\nu \in \underline{k}} a_\nu(x)\, b_\nu(y)$$

mit $k \in \mathbb{N}$, $a_\nu, b_\nu\colon \mathbb{R}^d \to \mathbb{R}$ vorliegen. Eine Kernfunktion dieser Form heißt *entartete Kernfunktion* von Rang k. Allgemein nennen wir eine Funktion g eine Kernfunktion einer Matrix G, falls die Funktion g die Matrix G erzeugt, also gerade $G_{\mu\nu} = g\mu, \nu$ gilt. Nicht jede Kernfunktion ist entartet, es lassen sich aber oft entartete Kernfunktionen konstruieren, die die Funktion approximieren. Möglichkeiten sind beispielsweise Approximationen durch Taylor-Entwicklung oder Interpolation. [Grasedyck 2001]

Seien X, Y Mengen und $G \in \mathbb{R}^{X \times Y}$ eine Matrix mit Kernfunkion $g\colon X \times Y \to \mathbb{R}$. Ist g entartet, so gilt also für $x \in X$ und $y \in Y$ gerade $G_{xy} = g(x,y) = \sum_{\nu \in \underline{k}_1} a_\nu(x)\, b_\nu(y)$. Die Summe in dieser Gleichung können wir auch durch Multiplikation zweier Matrizen $A \in \mathbb{R}^{X \times k}$ und $B^T \in \mathbb{R}^{Y \times k}$ mit $A_{x\nu} = a_\nu(x)$, $B_{y\nu} = b_\nu(y)$ ausdrücken. Analog dazu, die Kernfunktion als entartete Kernfunktion darzustellen, ist also die Darstellung der Matrix G durch Matrizen A und B durch $G = AB^T$.

Approximation

Im Folgenden wird eine beispielhafte Approximation einer nicht-entarteten Kernfunktion durch Interpolation vorgestellt, um so eine entartete Kernfunktion zu erhalten. Der Aufbau folgt weitestgehend dem von Börm [2018]. Für eine Approximation einer nicht-entarten Kernfunktion durch Taylor-Entwicklung sei wiederum auf Grasedyck [2001] verwiesen.

Sei $g\colon \Omega \times \Omega \to \mathbb{R}$ eine nicht-entartete Kernfunktion. Für eine Interpolationsordnung k und ein Teilgebiet $\sigma \subset \Omega$ wählen wir Interpolationspunkte $\xi_{\sigma,1}, \ldots, \xi_{\sigma,k}$. Nun suchen wir eine Approximation von $g|_{\Omega \times \sigma}$ der Form

$$g_\sigma(x,y) = \sum_{\nu \in \underline{k}_1} g(x, \xi_{\sigma,\nu})\, \mathcal{L}_{\sigma,\nu}(y), \tag{2.1}$$

in der x und y getrennt sind. Somit hätten wir bereits eine entartete Kernfunktion. In eben dieser Form werden Lagrange-Interpolaten in Stützstellen $\xi_{\sigma,\nu}$ mit Lagrange-Basisfunktionen $\mathcal{L}_{\sigma,\nu}$ dargestellt.

Lagrange-Interpolation im mehrdimensionalen Raum

Bevor wir die Funktion g wie in Gleichung 2.1 approximieren können, müssen wir die Lagrange-Basisfunktionen auf mehrdimensionalen Gebieten konstruieren und einige Eigenschaften sicherstellen. Zunächst definieren wir dazu die Lagrange-Basispolynome auf dem Referenzintervall $[-1,1]$. Für $k \in \mathbb{N}$, $\nu \in \underline{k}_0$ und paarweise verschiedene Stützstellen $\xi_0, \ldots, \xi_k \in [-1,1]$ ist durch

$$\mathcal{L}_\nu(x) := \prod_{\substack{\mu \in \underline{k}_0 \\ \mu \neq \nu}} \frac{x - \xi_\mu}{\xi_\nu - \xi_\mu} \qquad \text{für alle } x \in [-1,1]$$

das ν-te Lagrange-Polynom gegeben. Eine Eigenschaft der Lagrange-Polynome, die wir später noch benötigen, ist:

$$\mathcal{L}_\nu(\xi_\mu) = \delta_{\nu\mu} = \begin{cases} 1 & \text{falls } \nu = \mu, \\ 0 & \text{sonst.} \end{cases} \tag{2.2}$$

Wir können durch

$$\mathfrak{I}\colon C[-1,1] \to \Pi_k, \qquad f \mapsto \sum_{\nu \in \underline{k}_0} f(\xi_\nu)\, \mathcal{L}_\nu$$

einen Lagrange-Interpolationsoperator definieren, der auf $[-1,1]$ stetige Funktionen auf Polynome aus Π_k abbildet, also auf Polynome von höchstens Grad k.

Um diesen Operator auf Funktionen auf beliebigen Intervallen $[a,b]$ mit $a < b$ zu übertragen, definieren wir eine Transformation

$$\Phi_{[a,b]}\colon [-1,1] \to [a,b] \qquad x \mapsto \frac{b+a}{2} + \frac{b-a}{2}x,$$

die für alle $x \in [-1,1]$ die Eigenschaften

$$\Phi_{[a,b]}(-1) = a, \quad \Phi_{[a,b]}(1) = b, \quad \Phi'_{[a,b]}(x) = \frac{b-a}{2} > 0$$

hat, also affin und bijektiv ist. Für $f \in C[a,b]$ gilt also $\tilde{f} := f \circ \Phi_{[a,b]} \in C[-1,1]$, sodass wir den auf $[-1,1]$ definierten Interpolationsoperator $\mathfrak{I}$ auf $\tilde{f}$ anwenden können. Wir können also einen Interpolationsoperator auf dem Intervall $[a,b]$ durch

$$\mathfrak{I}_{[a,b]}\colon C[a,b] \to \Pi_k, \qquad f \mapsto \mathfrak{I}[f \circ \Phi_{[a,b]}] \circ \Phi^{-1}_{[a,b]}$$

definieren. Durch Einsetzen der Definition von $\mathfrak{I}$, erhalten wir so die Darstellung:

$$\mathfrak{I}_{[a,b]}[f] = \sum_{\nu \in \underline{k}_0} f \circ \Phi_{[a,b]}(\xi_\nu)\, \mathcal{L}_\nu \circ \Phi^{-1}_{[a,b]} = \sum_{\nu \in \underline{k}_0} f\left(\Phi_{[a,b]}(\xi_\nu)\right) \mathcal{L}_\nu \circ \Phi^{-1}_{[a,b]}.$$

Zur Vereinfachung definieren wir für $\nu \in \underline{k}_0$ und $x \in [a,b]$

$$\xi_{[a,b],\nu} := \Phi_{[a,b]}(\xi_\nu) = \frac{b+a}{2} + \frac{b-a}{2}\xi_\nu$$
$$\mathcal{L}_{[a,b],\nu} := \prod_{\substack{\mu \in \underline{k}_0 \\ \mu \neq \nu}} \frac{x - \xi_{[a,b],\mu}}{\xi_{[a,b],\nu} - \xi_{[a,b],\mu}}.$$

Da mit Gleichung 2.2 folgende Gleichung für alle $\nu, \mu \in \underline{k}_0$ gilt

$$\begin{aligned}\mathcal{L}_\nu \circ \Phi^{-1}_{[a,b]}\left(\xi_{[a,b],\mu}\right) &= \mathcal{L}_\nu \circ \Phi^{-1}_{[a,b]}\left(\Phi_{[a,b]}\left(\xi_{[a,b],\mu}\right)\right) \\ &= \mathcal{L}_\nu(\xi_\mu) = \delta_{\nu\mu} \\ &= \mathcal{L}_{[a,b],\nu}\left(\xi_{[a,b],\mu}\right),\end{aligned}$$

stimmen $\mathcal{L}_\nu \circ \Phi^{-1}_{[a,b]}$ und $\mathcal{L}_{[a,b],\nu}$ in $k+1$ Punkten überein. Da beides Polynome in Π_k sind,

müssen sie identisch sein und wir erhalten die vereinfachte Darstellung

$$\mathfrak{I}_{[a,b]}[f] = \sum_{\nu \in \underline{k}_0} f\left(\xi_{[a,b],\nu}\right) \mathcal{L}_{[a,b],\nu}, \qquad \text{für alle } f \in C[a,b].$$

Nun müssen wir für unser Problem noch Interpolationsoperatoren auf Gebieten im mehrdimensionalen Raum definieren. Der Einfachheit halber, und weil der später vorgestellte Algorithmus ohnehin mit solchen sogenannten *bounding boxes* arbeitet, beschränken wir uns auf achsenparallele Quader $Q = [a_1, b_1] \times \ldots \times [a_d, b_d]$. Sei also eine zu interpolierende Funktion $f \in C(Q)$ gegeben. Für $\iota \in \underline{d}$ und $x \in Q$ gewinnen wir aus f eine Funktion

$$f_{x,\iota} \colon [a_\iota, b\iota] \to \mathbb{R}, \qquad y \mapsto f(x_1, \ldots, x_{\iota-1}, y, x_{\iota+1}, \ldots, x_d),$$

bei der alle Parameter außer dem ι-ten fest sind. Auf diese Funktion auf dem Intervall $[a_\iota, b_\iota]$ können wir den Interpolationsoperator $\mathfrak{I}_{[a_\iota,b_\iota]}$ anwenden und erhalten so einen Operator

$$\mathfrak{I}_{Q,\iota} \colon C(Q) \to C(Q)$$

für den für $f \in C(Q)$ und $x \in Q$

$$\mathfrak{I}_{Q,\iota}[f](x) = \sum_{\nu \in \underline{k}_0} f\left(x_1, \ldots, x_{\iota-1}, \xi_{[a_\iota,b_\iota]}, x_{\iota+1}, \ldots, x_d\right) \mathcal{L}_{[a_\iota,b_\iota],\nu}(x_\iota)$$

gilt. Somit bildet $\mathfrak{I}_{Q,\iota}$ eine stetige Funktion auf eine neue stetige Funktion ab, die sich in der ι-ten Komponente wie ein Polynom verhält. Durch Hintereinanderausführung der Operatoren für jede Dimension erhalten wir ein Polynom in jeder Komponente und mit

$$\mathfrak{I}_Q := \mathfrak{I}_{Q,1} \circ \ldots \circ \mathfrak{I}_{Q,d}$$

einen Interpolationsoperator, der für alle $f \in C(Q)$ und $x \in Q$ die Gleichung

$$\mathfrak{I}_Q[f](x) = \sum_{\nu_1 \in \underline{k}_0} \ldots \sum_{\nu_d \in \underline{k}_0} f\left(\xi_{[a_1,b_1],\nu_1}, \ldots, \xi_{[a_d,b_d],\nu_d}\right) \mathcal{L}_{[a_1,b_1],\nu_1}(x_1) \ldots \mathcal{L}_{[a_d,b_d],\nu_d}(x_d) \tag{2.3}$$

erfüllt. Mit den Definitionen

$$\xi_{Q,\nu} = \left(\xi_{[a_1,b_1],\nu_1}, \ldots, \xi_{[a_d,b_d],\nu_d}\right),$$
$$\mathcal{L}_{Q,\nu} = \mathcal{L}_{[a_1,b_1],\nu_1}(x_1) \ldots \mathcal{L}_{[a_d,b_d],\nu_d}(x_d), \text{ für alle } \nu \in M := \underline{k}_0^d,\ x \in Q$$

erhalten wir die gewohnte Darstellung

$$\mathfrak{I}_Q[f] = \sum_{\nu \in M} f\left(\xi_{Q,\nu}\right) \mathcal{L}_{Q,\nu}, \text{ für alle } f \in C(Q).$$

Mit diesem Interpolationsoperator können wir nun die entartete Approximation der Kernfunktion g aus Gleichung 2.1 konstruieren. Dazu identifizieren wir $\sigma \in T_\Omega$ mit einem überdeckenden Quader Q_σ.

Zulässigkeit

Wir haben also über Interpolation eine entartete Kernfunktion g_σ konstruiert. Für $x \in \Omega$ und $y \in \sigma$ approximiert die Funktion g_σ die Funktion g allerdings nur dann gut, wenn x und σ hinreichend weit voneinander entfernt sind (vgl. Grasedyck [2001]; Börm [2018]). Daher wollen wir nun eine Bedingung einführen, die eine Aussage darüber trifft, ob das Gebiet σ für ein Gebiet τ zulässig, also für alle $x \in \tau$ hinreichend weit entfernt ist. Das Gebiet τ wird auch *Target-* oder *Zielgebiet*, das Gebiet σ auch *Source-* oder *Quellgebiet* genannt.

Definition 2.1. (Zulässigkeitsbedingung)
Wir nennen eine Abbildung, die jedem Paar $(\tau, \sigma) \in \mathcal{P}(\Omega) \times \mathcal{P}(\Omega)$ entweder *"zulässig"* oder *"unzulässig"* zuordnet, *Zulässigkeitsbedingung*.

Für eine konkrete Zulässigkeitsbedingung benötigen wir noch ein paar Definitionen.
Seien $\sigma, \tau \subseteq \Omega$. Dann bezeichnet

$$diam(\sigma) := max\{\|x - y\| \mid x, y \in \sigma\}$$

den Durchmesser eines Teilgebiets und

$$dist(\sigma, \tau) := min\{\|x - y\| \mid x \in \sigma, y \in \tau\}$$

den Abstand der beiden Teilgebiete zueinander.

Damit lässt sich die von Grasedyck [2001] aufgeführte η-Zulässigkeit wie folgt definieren:

Definition 2.2. (η-Zulässigkeit)
Sei $\eta \in \mathbb{R}^+$. Dann heißt die Abbildung $\mathcal{Z}_\eta \colon \mathcal{P}(\Omega) \times \mathcal{P}(\Omega) \to \{zulässig, unzulässig\}$ mit

$$\mathcal{Z}_\eta(\tau, \sigma) = \begin{cases} zulässig & \text{falls } \{diam(\tau), diam(\sigma)\} \leqslant 2\eta dist(\tau, \sigma) \\ unzulässig & \text{sonst} \end{cases}$$

η-Zulässigkeitsbedingung. Paare (τ, σ), für die $\mathcal{Z}(\tau, \sigma) = zulässig$ gilt, werden *η-zulässig* genannt.

Für kleine η zeigen Hackbusch und Börm [2002] die exponentielle Konvergenz der durch Interpolation approximierten Funktion gegen die Kernfunktion und Grasedyck [2001] die exponentielle Konvergenz bei Approximation durch Taylor-Entwicklung.

Börm [2018] weist nach, das für asymptotisch glatte Funktionen die Approximation für jedes $\eta > 0$ exponentiell konvergiert, wenn die obige Zulässigkeitsbedingung erfüllt ist. Eine unendlich oft differenzierbare Kernfunktion g heißt asymptotisch glatt, wenn für alle $\alpha, \beta \in \mathbb{N}_0^d$, $x, y \in \mathbb{R}^d$, $x \neq y$, Konstanten $C \colon \mathbb{N}^2 \to \mathbb{R}_{>0}$ existieren, sodass die partiellen Ableitungen von g wie folgt abgeschätzt werden können:

$$|\partial_x^\alpha \partial_y^\beta g(x, y)| \leqslant C(|\alpha|, |\beta|) \|x - y\|^{-|\alpha| - |\beta|} |g(x, y)|.$$

(vgl. Hackbusch und Börm [2002])

Tschebyscheff-Interpolationspunkte

Die Wahl der Zulässigkeitsbedingung ist nicht der einzige Einflussfaktor auf die Genauigkeit der Approximation. Auch die Wahl der Interpolationspunkte trägt dazu bei. Als optimale Interpolationspunkte haben sich die Tschebyscheff-Interpolationspunkte erwiesen (vgl. Börm [2018]). Daher führen wir diese im Folgenden kurz ein.

Definition 2.3. (Tschebyscheff-Polynome)
Durch

$$T_n = \begin{cases} 1 & \text{falls } n = 0 \\ x & \text{falls } n = 1 \\ 2xT_{n-1}(x) - T_{n-2}(x) & \text{sonst,} \end{cases} \qquad \text{für alle } n \in \mathbb{N}_0,\ x \in \mathbb{R}$$

wird eine Familie $(T_n)_{n\in\mathbb{N}_0}$ von Polynomen definiert. Für $n \in \mathbb{N}_0$ bezeichnen wir T_n als n-tes Tschebyscheff-Polynom. [Börm 2018]

Mit Hilfe des Additionstheorems $\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$ und den Eigenschaften des Cosinus lässt sich leicht die folgende Aussage zeigen.

Lemma und Definition 2.4. Für $n \in \mathbb{N}_0$ und $x \in [-1,1]$ gilt:

$$T_n(x) = \cos(n \arccos(x)).$$

Daraus folgen insbesondere die n reellen einfachen Nullstellen

$$T_n(\xi_\nu) = 0, \qquad \xi_\nu := \cos\left(\pi \frac{2\nu+1}{2n} -\right) \qquad \text{für alle } \nu \in \underline{n}.$$

Diese Nullstellen nennen wir *Tschebyscheff-Interpolationspunkte*. [Börm 2018]

Clusterung

Im Abschnitt Approximation wurde gezeigt, wie die Kernfunktion auf Teilgebieten approximiert werden kann, sofern die Paare von Gebieten zulässig sind. Im Abschnitt Zulässigkeit wurde eine Bedingung aufgestellt, wann Paare von Gebieten zulässig sind und wann nicht. In diesem Kapitel soll eine Struktur vorgestellt werden, mit der solche Paare von Teilgebieten strukturiert werden können. Dazu werden zunächst einige grundlegende Definitionen benötigt.

Definition 2.5. (Baum)
Ein Paar $T = (V, E)$ mit $E \subset V \times V$ nennen wir einen Baum mit Knoten V und Kanten E, wenn die folgenden Bedingungen erfüllt sind:

1. Es gibt genau ein Element $root(T) \in V$, so dass für alle $v \in V$ gilt $(v, root(T)) \notin E$. Dieses Element heißt *Wurzel* des Baumes T.

2. Zu jedem Knoten $v \in V \backslash \{root\,(T)\}$ gibt es $n \in \mathbb{N}$ und einen Weg $(v_i)_{i=0}^{n}$ der Länge n von der Wurzel $root\,(t) = v_0$ zu dem Knoten $v = v_n$ mit $(v_{i-1}, v_i) \in E$ für $i \in \underline{n}$.

3. Es gibt keine Zyklen.

Es gelten folgende Notationen:

1. Die Länge des längsten Weges in T heißt die *Tiefe* des Baumes und wird mit $depth\,(T)$ bezeichnet.

2. Mit "$q \in T$" ist stets "$q \in V$" gemeint.

3. $sons\,(q) := sons_T\,(q) := \{v \in V \mid (q, v) \in E\}$ ist die Menge der *Söhne* eines Knotens $q \in T$.

4. $sons^*\,(q) := \begin{cases} q \cup \bigcup_{\tilde{q} \in sons(q)} sons^*\,(\tilde{q}) & \text{falls } sons(t) \neq \varnothing, \\ q & \text{sonst,} \end{cases}$ ist die Menge aller Nachfahren eines Knotens $q \in T$.

5. $T^{(i)}$ bezeichnet die i-te Ebene des Baumes. Dabei gilt:
   (a) $T^{(0)} := \{root\,(T)\}$ und
   (b) $T^{(i)} := \{\tilde{v} \in V \mid \exists v \in T^{(i-1)} \colon \tilde{v} \in sons\,(v)\}$ für $i \in \underline{depth\,(T)}$.

6. Der Begriff "oberhalb" bedeutet "näher zur Wurzel", der Begriff "unterhalb" respektive "weiter oder gleich weit entfernt von der Wurzel".

7. $\mathcal{L}\,(T) := \{q \in T \mid \forall v \in T \colon (q, v) \notin E\}$ bezeichnet die Menge aller Blätter des Baumes T.

8. $\mathcal{L}\,(T, i) := \mathcal{L}\,(T) \cap T^{(i)}$ ist die Menge der Blätter auf der i-ten Ebene des Baumes.

9. $\mathcal{L}\,(T, \leqslant s) := \mathcal{L}\,(T) \cap \bigcup_{i \in \underline{s}_0} T^{(i)}$ sind alle Blätter der Ebenen $0, \ldots, s \leqslant depth\,(T)$.

[Grasedyck 2001]

Mit Hilfe einer Baumstruktur wollen wir nun eine Struktur definieren, mit der im Allgemeinen eine beliebige Menge M hierarchisch partitioniert werden kann. Diese Struktur wollen wir dann für unsere Menge Ω nutzen.

Definition 2.6. (Clusterbaum)
Wir nennen einen Baum $T = (V, E)$ mit $V \subseteq \mathcal{P}(M) \backslash \{\varnothing\}$ einen *Clusterbaum* einer Menge M, falls die nachfolgenden Bedingungen erfüllt sind:

1. $root\,(T) = M$

2. $\forall t \in T : t = \bigcup_{s \in sons(t)} s$.

3. $\forall t \in T\ \forall s_0, s_1 \in sons\,(t) : s_0 \cap s1 = \varnothing$

Für einen Clusterbaum verwenden wir in der Regel die Notation T_M um auf die verwendete Indexmenge hinzuweisen, und bezeichnen die Knoten $t \in T_M$ als *Cluster*. [Börm 2018]

Grasedyck [2001] nennt Clusterbäume auch hierarchische Partitionsbäume oder kurz $\mathcal{H}$-Bäume der Menge M. Der Name leitet sich direkt von den Eigenschaften des Clusterbaumes her. Es gilt nämlich:

1. Die Menge $P_M^{(i)} := T_M^{(i)} \cup \mathcal{L}(T, \leqslant i)$ für bildet für alle $i \in \underline{depth(T_M)}_0$ eine disjunkte Partition der Menge M.
2. Die Partitionen sind über die Ebenen des Baumes hierarchisch angeordnet. Damit ist gemeint, dass für alle $i \in \underline{depth(T_M)}$ die Partition $P^{(i)}$ mehr Teilmengen beinnhaltet als die Partition $P^{(i-1)}$.

Mit Hilfe eines Clusterbaumes können Partitionen einer Menge also hierarchisch strukturiert werden. Meist werden sie genutzt, um eine Indexmenge oder die geometrische Struktur eines Problems zu partitionieren. Letztere Variante wird auch *geometrischer Clusterbaum* genannt.

Ein weiterer praktischer Aspekt von Clusterbäumen ist ihre gute rekursive Konstruierbarkeit. Beginnend mit der vollständigen Menge M wird ein Cluster jeweils in Sohncluster unterteilt, bis entweder in den Blättern jeweils nur noch ein Element vorhanden ist oder eine anders geartete Abbruchbedingung erfüllt ist. Für praktische Anwendungen ist es in der Regel nicht effizient, bis zu einelementigen Mengen zu teilen. Üblich sind Abbruchbedingungen, die eine gewisse Mächtigkeit für Blattmengen vorgeben. Wird diese unterschritten, wird nicht weiter geteilt.

Das Teilen der Cluster kann ebenfalls unterschiedlich motiviert sein. Naheliegend sind kardinalitätsgesteuerte Zerlegungen, um ein optimales Loadbalancing zu erhalten, sowie die Zerlegung anhand der zugrundeliegenden geometrischen Struktur. Auch kombinierte Zerlegungen sind denkbar.

Definition 2.7. (Blockbaum, (streng) zulässiger Blockbaum)
Seien Menge X, Y und jeweils zugehörige Clusterbäume T_X und T_Y gegeben.

Einen Clusterbaum $T_{X\times Y}$ der Menge $X \times Y$ mit folgenden Eigenschaften:

1. Für alle Knoten $b \in T_{X\times Y}$ existieren Knoten $t \in T_X$ und $s \in T_Y$, sodass $b = t \times s$
2. Falls $b = t \times s \in T_{X\times Y}$ kein Blatt ist, gilt

$$sons(b) = \begin{cases} \{t \times s' \mid s' \in sons(s)\} & \text{falls } sons(t) = \varnothing,\ sons(s) \neq \varnothing \\ \{t' \times s \mid t' \in sons(t)\} & \text{falls } sons(t) \neq \varnothing,\ sons(s) = \varnothing \\ \{t' \times s' \mid t' \in sons(t),\ s' \in sons(s)\} & \text{ansonsten} \end{cases}$$

nennen wir *Blockbaum* und seine Knoten *Blöcke*. Die Komponenten $t \in T_X$ und $s \in T_Y$ werden auch *Zeilen-* und *Spaltencluster* genannt.

Wir nennen einen Block $b = t \times s$ *zulässig* bezüglich einer Zulässigkeitsbedingung $\mathcal{Z}$, wenn $\mathcal{Z}(t,s) = \textit{zulässig}$ erfüllt ist.

Wir nennen einen Blockbaum zulässig, wenn für alle Blattblöcke $b = t \times s \in \mathcal{L}(T_{X\times Y})$ gilt:

$$\mathcal{Z}(t,s) = \textit{zulässig} \text{ oder } \textit{sons}(t) = \varnothing \text{ oder } \textit{sons}(s) = \varnothing.$$

Wir nennen ihn *streng zulässig*, wenn gilt:

$$\mathcal{Z}(t,s) = \textit{zulässig} \text{ oder } \textit{sons}(t) = \varnothing = \textit{sons}(s),$$

Mit

$$\mathcal{L}^+(T_{X\times Y}) = \{b = t \times s \in \mathcal{L}(T_{X\times Y}) \mid \mathcal{Z}(t,s) = \textit{zulässig}\}$$
$$\text{und } \mathcal{L}^-(T_{X\times Y}) = \{b = t \times s \in \mathcal{L}(T_{X\times Y}) \mid \mathcal{Z}(t,s) = \textit{unzulässig}\}$$

bezeichnen wir die Mengen der zulässigen beziehungsweise unzulässigen Blätter eines Blockbaumes. $\mathcal{L}^+(T_{X\times Y})$ wird auch *Fernfeld* und $\mathcal{L}^-(T_{X\times Y})$ *Nahfeld* genannt. Die Namen leiten sich von der η-Zulässigkeitsbedingung her, die auf der relativen Nähe der Cluster zueinander beruht.

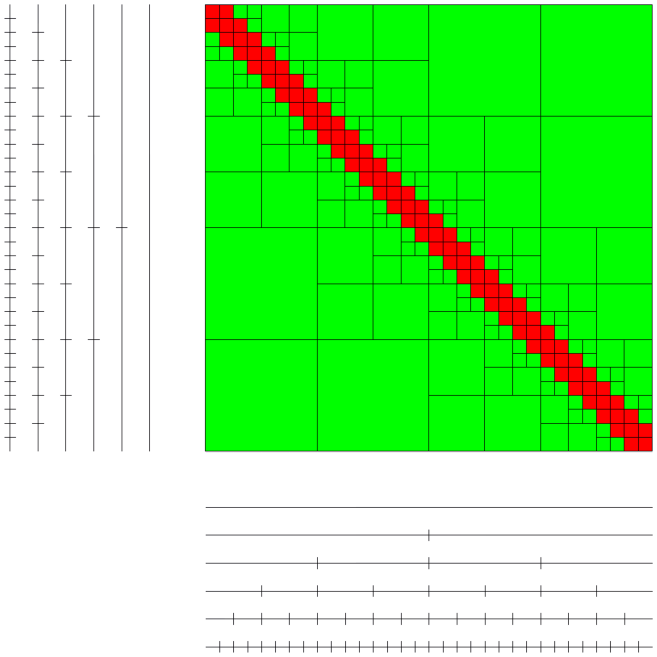


Abbildung 2.1. Schematische Darstellung eines Blockbaumes. In grün sind zulässige, in rot unzulässige Blöcke gekennzeichnet. (Quelle: **h2-slides**)

Mit Hilfe von Blockbäumen können wir nun also Paare von Teilgebieten hierarchisch strukturieren und auf Zulässigkeit prüfen. In Abbildung 2.1 ist ein Blockbaum, beziehungsweise die Blätter eines Blockbaumes, dargestellt. Zulässige Blattblöcke sind grün, unzulässige rot gekennzeichnet. Zur linken den Blockbaumes ist der Clusterbaum mit den Targetclustern, unten der Clusterbaum mit den Sourceclustern schematisch abgebildet.

2.1.1. $\mathcal{H}$-Matrizen

Definition 2.8. (**R**k-Matrix)
Seien $m, n, k \in \mathbb{N}$ sowie $M \in \mathbb{R}^{m \times n}$ gegeben. Wir nennen M eine ***R**k-Matrix*, wenn Matrizen $A \in \mathbb{R}^{m \times k}$ und $B \in \mathbb{R}^{n \times k}$ existieren, sodass

$$M = AB^T$$

gilt. Die Menge aller solcher **R**k-Matrizen bezeichnen wir mit

$$\mathcal{R}(m, n, k) = \{M \in \mathbb{R}^{m \times n} \mid M = AB^T, A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{n \times k}\}.$$

[Börm 2018]

Das praktische an **R**k-Matrizen ist, dass sie sich durch $k(m + n)$ Koeffizienten darstellen lassen. Ist $k \ll m, n$ ist dies wesentlich effizienter als eine volle Darstellung mit $m \cdot n$ Koeffizienten. [Börm 2018]

Definition 2.9. (Hierarchische Matrix oder $\mathcal{H}$-Matrix)
Seien X, Y zwei Mengen, T ein zugehöriger zulässiger Blockbaum mit Zulässigkeitsbedingung $\mathcal{Z}$ und $k : \mathcal{L}(T) \to \mathbb{N}_0$ die Rangverteilung der Blätter.

Eine Matrix $M \in \mathbb{R}^{X \times Y}$ heißt *hierarchische Matrix* oder kurz *$\mathcal{H}$-Matrix* bezüglich T, $\mathcal{Z}$ und k, falls für jeden Blattblock $b \in \mathcal{L}(T)$ der korrespondierende Matrixblock $M_b = \left(M_{ij}\right)_{(i,j) \in b}$ eine **R**k-Matrix ist.

Eine Zahl $k_0 \in \mathbb{N}$ mit $k(b) \leqslant k_0$ für alle $b \in \mathcal{L}^+(T_{X \times Y})$ heißt lokaler Rang der hierarchischen Matrix.

[Grasedyck 2001; Börm 2018]

Für die Darstellung in Computern sind $\mathcal{H}$-Matrizen von besonderem Interesse, da sie sich durch die Zerlegung in **R**k-Matrizen besonders kompakt speichern lassen. Für eine $\mathcal{H}$-Matrix M mit lokalem Rang k existieren nach Definition Familien von Matrizen $(A_b)_{b \in \mathcal{L}^+(T_{X \times Y})}$ und $(B_b)_{b \in \mathcal{L}^+(T_{X \times Y})}$, mit

$$A_b \in \mathbb{R}^{t \times k}, B_b \in \mathbb{R}^{s \times k}, M|_b = A_b B_b^T, \text{ für alle } b = t \times s \in \mathcal{L}^+(T_{X \times Y}).$$

So verbleiben nur noch für unzulässige Blöcke $b \in \mathcal{L}^-(T_{X \times Y})$ vollbesetzte Matrizen $N_b = M|_b$, sogenannte *Nahfeldmatrizen*. Das Tripel $\left((A_b)_{b \in \mathcal{L}^+(T_{X \times Y})}, (B_b)_{b \in \mathcal{L}^+(T_{X \times Y})}, (N_b)_{b \in \mathcal{L}^-(T_{X \times Y})}\right)$ bezeichnen wir als $\mathcal{H}$- Matrix-Darstellung der Matrix M. [Börm 2018]

In Abbildung 2.2 ist eine $\mathcal{H}$-Matrix schematisch dargestellt. Für zulässige Blöcke sind die Matrizen A_b und B_b in blau und die Nahfeldmatrizen N_b in rot gekennzeichnet. Durch die

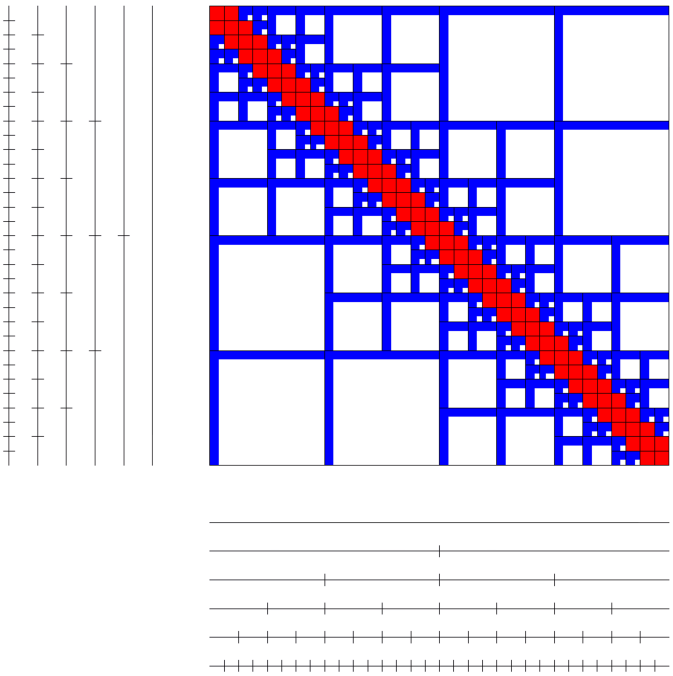


Abbildung 2.2. Schematische Darstellung einer $\mathcal{H}$-Matrix. In blau sind die Matrizen A_b und B_b, in rot die Matrizen N_b gekennzeichnet. (Quelle: Börm u. a. [2014])

Dicke der Matrizen A_b und B_b ist der lokale Rang k dargestellt. die Speicherplatzersparnis ist dadurch gut zu erkennen.

Gerade auch im Zusammenhang mit dieser Arbeit ist aber von noch größerem Interesse, dass auch Operationen wie Matrix-Vektor-Multiplikationen auf **R**k-Matrizen und damit auch auf $\mathcal{H}$-Matrizen wesentlich effektiver implementiert werden können. Grasedyck [2001] zeigt, dass die Matrix-Vektor-Multiplikation nur $k\,(n+m)$ statt $n \cdot m$ Operationen benötigt.

2.1.2. Optimierung

Im Folgenden wollen wir noch einige Überlegungen vorstellen, durch die $\mathcal{H}$-Matrizen noch speicherplatzsparender und noch effizienter in der Berechnung werden.

Zeilen- und Spaltenmatrizen

Bisher wurde eine hierarchische Matrix konstruiert, indem die Kernfunktion g auf zulässigen Blöcken $b = \tau \times \sigma \in \mathcal{L}^+(T_{X\times Y})$ durch die auf σ interpolierte Funktion

$$g_\sigma(x,y) = \sum_{\nu\in\underline{k}_1} g(x,\xi_{\sigma,\nu})\,\mathcal{L}_{\sigma,\nu}(y)$$

approximiert wurde. Somit erhalten wir durch die punktweise Definition

$$(A_b)_{x\nu} := g(x,\xi_{\sigma,\nu}) \text{ und } (B_b)_{y\nu} := \mathcal{L}_{\sigma,\nu}(y) \text{ für alle } x\in\tau \text{ und } y\in\sigma$$

Matrizen $A_b \in \mathbb{R}^{m\times k}$ und $B_b \in \mathbb{R}^{n\times k}$. Damit können wir die Matrix G auf Blöcken $b \in \mathcal{L}^+(T_{X\times Y})$ in der Form

$$(G|_b)_{xy} = g(x,y) \approx g_\sigma(x,y) = \left(A_b B_b^T\right)_{xy},$$

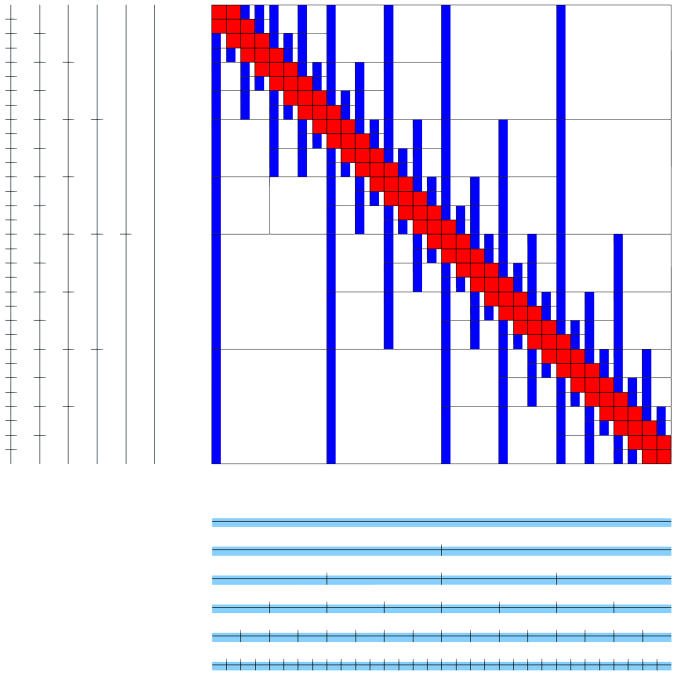


Abbildung 2.3. Darstellung der Ersetzung der Matrizen B_b durch Spaltenmatrizen W_σ. Letztere sind in hellblau auf dem Clusterbaum eingezeichnet. Die verbleibenden dunkelblauen Matrizen sind die Matrizen A_b. (Quelle: Börm u. a. [2014])

also durch **R**k-Matrizen darstellen und erhalten so eine Darstellung als hierarchische Matrix. [Börm 2018]

Ein Blick auf die Definition der Matrix B_b verrät, dass diese nicht von τ sondern ausschließlich von σ abhängt. Wir können sie also für konstantes σ für alle Blöcke $\tilde{b} = \tilde{\tau} \times \sigma \in T_{X \times Y}$ durch eine Matrix

$$(W_\sigma)_{y\nu} := \mathcal{L}_{\sigma,\nu}(y)$$

ersetzen. Anstatt also für jeden Block b wieder eine eigene Matrix aufstellen zu müssen, können sich Blöcke mit gleichem Quellgebiet σ die Matrizen teilen. Insbesondere muss diese Matrix W_σ auch nur einmal berechnet werden, womit ein erheblicher Teil des Rechnungsaufwands bei der Konstruktion von $\mathcal{H}$-Matrizen eingespart werden kann. Zudem können wiederum Operationen wie die Matrix-Vektor-Multiplikation effektiver durchgeführt werden, da auch hier die Matrix W_σ ausgeklammert werden kann, und so nur einmal in die Berechnung einfließt. [Börm 2018]

Definition 2.10. (Zeilen-/Spaltenmatrizen)
Wir nennen Matrizen $W_\sigma \in \mathbb{R}^{\sigma \times k}$ wie oben *Spaltenmatrizen* einer hierarchischen Matrix G. Bei Interpolation in der ersten Komponente nennen wir sie entsprechend *Zeilenmatrizen*.

Dargestellt ist diese Optimierung in Abbildung 2.3. In hellblau wurden die Spaltenmatrizen W_σ auf dem Clusterbaum eingezeichnet. Die verbleibenden dunkelblauen Matrizen sind die nicht optimierten Matrizen A_b. Ein Vergleich der Häufigkeit der Matrizen A_b und W_σ verdeutlicht die zuvor beschriebene Ersparnis von Redundanz.

Transfermatrizen und Clusterbasis

Von Hackbusch und Börm [2002] wird auf folgende Eigenschaft der Interpolationsoperatoren hingewiesen:

Sei $\mathcal{Q}_k := \{\prod_{i \in \underline{i_0}} p_i \mid p_i \in \Pi_k,\ i_0 \in \mathbb{N}\}$ der durch Tensorprodukte von Polynomen von höchstens Grad k aufgespannte Polynomraum. Da alle Cluster durch Polynome der selben Ordnung interpoliert wurden, zeigt Gleichung 2.3, dass $\mathfrak{I}_Q$ gerade eine Projektion auf diesen Raum $\mathcal{Q}_k$ ist.

Wegen dieser Projektionseigenschaft des Interpolationsoperators folg für $\sigma \in T_X \backslash \mathcal{L}(T_X)$, Söhne $\tilde{\sigma} \in sons(\sigma)$ und für alle $\nu \in \underline{k}$:

$$\mathcal{L}_{\sigma,\nu} = \mathfrak{I}_{\tilde{\sigma}}[\mathcal{L}_{\sigma,\nu}] = \sum_{\tilde{\nu} \in \underline{k}} \mathcal{L}_{\sigma,\nu}(\xi_{\tilde{\sigma},\tilde{\nu}})\, \mathcal{L}_{\tilde{\sigma},\tilde{\nu}}.$$

Anstatt also auf jeder Ebene des Clusterbaumes eine vollständige Matrix $W_\sigma \in \mathbb{R}^{\sigma \times k}$ aufzustellen, genügt es für jeden Sohncluster $\tilde{\sigma} \in sons(\sigma)$ Matrizen $E_{\tilde{\sigma}} \in \mathbb{R}^{k \times k}$ mit $(E_{\tilde{\sigma}})_{\tilde{\nu},\nu} := \mathcal{L}_{\sigma,\nu}(\xi_{\tilde{\sigma},\tilde{\nu}})$ zu speichern. Dies führt zu folgender Definition:

Definition 2.11. (Transfermatrix, Clusterbasis)
Sei T_X ein Clusterbaum mit einer Familie von Spalten- (bzw. Zeilen-)matrizen $W =$

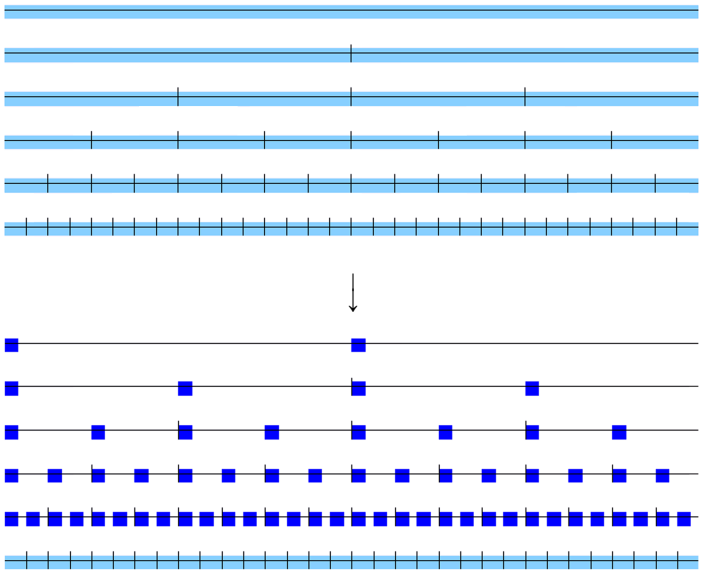


Abbildung 2.4. Ersetzung der Matrizen W_σ durch Transfermatrizen $E_{\tilde{\sigma}}$ auf Nicht-Blattclustern. (Quelle: Börm u. a. [2014])

$(W_\sigma)_{\sigma \in T_X}$. Existiert eine Familie $E = (E_\sigma)_{\sigma \in T_X}$ von $(k \times k)$-Matrizen mit der Eigenschaft

$$V_\sigma|_{\tilde{\sigma} \times k} = V_{\tilde{\sigma}} E_{\tilde{\sigma}} \qquad \text{für alle } \sigma \in T_X \backslash \mathcal{L}(T_X) \text{ und } \tilde{\sigma} \in sons(\sigma),$$

so nennen wir V eine *(geschachtelte) Clusterbasis* (von Rang k) für den Clusterbaum T_X. Die Matrizen der Familie E heißen die korrespondierenden *Transfermatrizen*. [Börm 2018]

Mit der *geschachtelten Darstellung* $\left((W_\sigma)_{\sigma in \mathcal{L}(T_X)}, (E_\sigma)_{\sigma \in T_X \backslash \mathcal{L}(T_X)}\right)$ der Clusterbasen haben wir nun also eine noch kompaktere Darstellung der Matrizen B_b. Dargestellt ist dies in Abbildung 2.4.

Interpolation in beiden Komponenten

Auch in diesem Abschnitt folge ich dem Aufbau von Börm [2018].

Wir haben bereits einige Eigenschaften der Matrizen B_b beziehungsweise W_σ identifiziert, die sich zur Effektivitätssteigerung nutzen lassen. Leider bleiben die Speicherplatzineffizienz und der höhere Rechenaufwand für die Matrizen A_b bestehen, die die positiven Eigenschaften der Matrizen W_σ nicht teilen. Allerdings haben wir die Matrizen W_σ einfach punktweise aus den Lagrange-Polynomen bei der Interpolation definiert. Es liegt also nahe eine Interpolation nicht nur in einer, sondern in beiden Variablen durchzuführen. Wir wählen also Interpolationspunkte $\xi_{\tau,1}, \ldots, \xi_{\tau,k}$ und erhalten wiederum mit den zugehörigen

Lagrange-Polynomen $\mathcal{L}_{\tau,1}, \ldots, \mathcal{L}_{\tau,k}$:

$$g(x,y) \approx \tilde{g}(x,y) = \sum_{\mu \in \underline{k}} \sum_{\nu \in \underline{k}} \mathcal{L}_{\tau,\mu}(x)\, g\left(\xi_{\tau,\mu}, \xi_{\sigma,\nu}\right) \mathcal{L}_{\sigma,\nu}(y).$$

Mit

$$(S_b)_{\mu\nu} := g\left(\xi_{\tau,\mu}, \xi_{\sigma,\nu}\right), \ \mu, \nu \in \underline{k}$$

erhalten wir eine Familie von Koeffizientenmatrizen $(S_b)_{b \in \mathcal{L}^+(T_{X\times Y})} \in \mathbb{R}^{k\times k}$, die weder von x, noch von y anhängt, sondern lediglich von den Interpolationspunkten. Außerdem haben wir durch die neuerliche Interpolation die Abhängigkeit von x wiederum auf Lagrange-Polynome beschränkt und erhalten so Matrizen $V_\tau \in \mathbb{R}^{\tau\times k}$ durch die Festlegung

$$(V_\tau)_{x\mu} := \mathcal{L}_{\tau,\mu}(x).$$

Insgesamt erhalten wir also für alle $b = \tau \times \sigma \in \mathcal{L}^+(T_{X\times Y})$

$$G|_b \approx \tilde{G}\big|_b = V_\tau S_b W_\sigma^T.$$

Die Matrizen V_τ haben dieselben Eigenschaften wie die Matrizen W_σ. Beide lassen sich also nicht nur effizient speichern, sondern auch effizient konstruieren. Außerdem brauchen sie auf Grund der Entkoppelung von τ und σ jeweils nur einmal aufgestellt zu werden. Da die Matrizen S_b, die die Kopplung zwischen den Clustern τ und σ beschreiben, wiederum verhältnismäßig klein sind, lassen sich diese wiederum auch effektiv verarbeiten.

2.1.3. $\mathcal{H}^2$-Matrizen

Die Überlegungen der letzten Abschnitte fließen nun in der Definition der $\mathcal{H}^2$-Matrizen zusammen.

Definition 2.12. ($\mathcal{H}^2$-Matrizen)
Sei $T_{X\times Y}$ ein streng zulässiger Blockbaum, $k \in \mathbb{N}_0$ und seien V und W Clusterbasen des Ranges k für die Clusterbäume T_X und T_Y. Eine Matrix $M \in \mathbb{R}^{X\times Y}$ heißt $\mathcal{H}^2$-*Matrix* mit Zeilenbasis V und Spaltenbasis W, falls für alle zulässigen Blattblöcke $b = \tau \times \sigma \in \mathcal{L}^+(T_{X\times Y})$ eine Matrix $S_b \in R^{k\times k}$ existiert, sodass

$$M|_b = V_\tau S_b W_\sigma^T$$

erfüllt ist. Die Familie $(S_b)_{b\in\mathcal{L}^+(T_{X\times Y})}$ bezeichnen wir als Familie der *Kopplungsmatrizen*.

Für unzulässig Blattblöcke $b = \tau \times \sigma \in \mathcal{L}^-(T_{X\times Y})$ verbleiben, wie auch bei den $\mathcal{H}$-Matrizen , die vollbesetzten Nahfeldmatrizen N_b.

Mit geschachtelten Darstellungen

$$\left((V_\tau)_{\tau in \mathcal{L}(T_X)}, (E_\tau)_{\tau\in T_X\setminus\mathcal{L}(T_X)}\right) \text{ und } \left((W_\sigma)_{\sigma in \mathcal{L}(T_Y)}, (F_\sigma)_{\sigma\in T_Y\setminus\mathcal{L}(T_Y)}\right)$$

der Clusterbasen V und W bezeichnen wir das Tupel

$$\left((S_b)_{b\in\mathcal{L}^+(T_{X\times Y})}, (N_b)_{b\in\mathcal{L}^-(T_{X\times Y})}, (V_\tau)_{\tau in\mathcal{L}(T_X)}, (E_\tau)_{\tau\in T_X\setminus\mathcal{L}(T_X)}, (W_\sigma)_{\sigma in\mathcal{L}(T_Y)}, (F_\sigma)_{\sigma\in T_Y\setminus\mathcal{L}(T_Y)}\right)$$

als $\mathcal{H}^2$-Matrix-Darstellung der Matrix M.

In Abbildung 2.5 ist all dies zusammen dargestellt. Die Familien $(V_\tau)_{\tau in\mathcal{L}(T_X)}$ und $(W_\sigma)_{\sigma in\mathcal{L}(T_Y)}$ links und unten sind hellblau gefärbt. Die Familien der Transfermatrizen $(E_\tau)_{\tau\in T_X\setminus\mathcal{L}(T_X)}$ und $(F_\sigma)_{\sigma\in T_Y\setminus\mathcal{L}(T_Y)}$ sind in dunkelblau abgebildet. Die Kopplungsmatrizen $(S_b)_{b\in\mathcal{L}^+(T_{X\times Y})}$ sind in pink und die Nahfeldmatrizen $(N_b)_{b\in\mathcal{L}^-(T_{X\times Y})}$ unverändert in rot dargestellt. Die Ersparnis an Speicherplatz gegenüber der vollbesetzten Matrix und selbst gegenüber der Darstellung als $\mathcal{H}$-Matrix ist augenscheinlich.

Von Börm [2007] wird eine vollständige Charakterisierung der $\mathcal{H}^2$-Matrizen vorgenommen. Dadurch wird ein für uns entscheidender Unterschied zwischen $\mathcal{H}$-Matrizen und $\mathcal{H}^2$-Matrizen deutlich: Bei $\mathcal{H}$-Matrizen müssen die zu den zulässigen Blöcken $b = \tau \times \sigma \in$

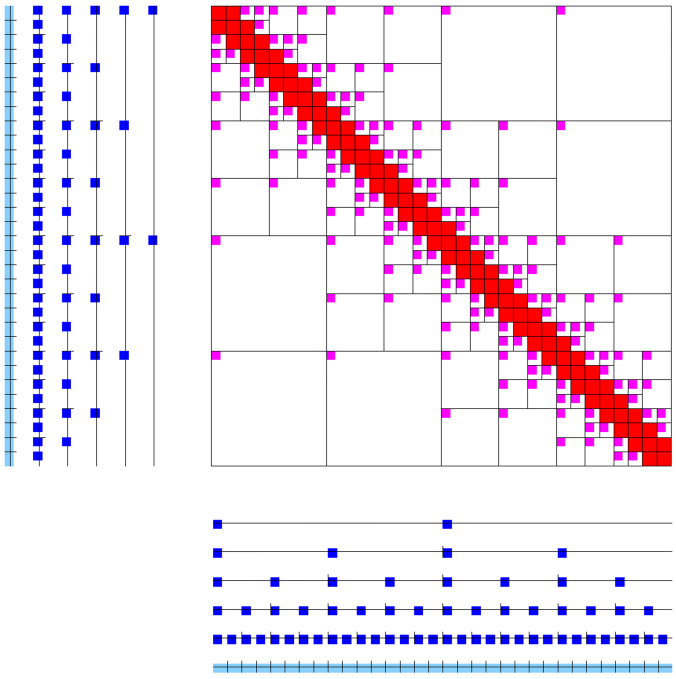


Abbildung 2.5. Darstellung einer $\mathcal{H}^2$-Matrix: In blau die geschachtelten Clusterbasen, in pink die Kopplungsmatrizen, in rot die verbleibenden Nahfeldmatrizen. (Quelle: Börm u. a. [2014])

$\mathcal{L}^+(T_{X\times Y})$ gehörige Matrizen $M|_{\tau\times\sigma}$ niedrigen Rang besitzen. Bei $\mathcal{H}^2$-Matrizen hingegen müssen die deutlich größeren Matrizen $M|_{\tau\times R_\tau}$ und $M|_{C_\sigma\times\sigma}$ für alle Cluster $\tau\in T_X$ und $\sigma\in T_Y$ niedrigen Rang besitzen. Dabei sind für alle $\tau\in T_X$ und $\sigma\in T_Y$ definiert:

$$R_\tau := \bigcup_{r\in row*(\tau)} r, \qquad C_\sigma := \bigcup_{c\in col(\sigma)} c,$$
$$row*(\tau) := \{s\in T_Y \mid \exists\tilde{\tau}\in T_X\colon\ \tilde{\tau}\times s\in\mathcal{L}^+(T_{X\times Y})\wedge\tau\in sons^*(\tilde{\tau})\},$$
$$col*(\sigma) := \{t\in T_X \mid \exists\tilde{\sigma}\in T_Y\colon\ t\times\tilde{\sigma}\in\mathcal{L}^+(T_{X\times Y})\wedge\sigma\in sons^*(\tilde{\sigma})\}.$$

Dieser Unterschied ist in Abbildung 2.6 bildlich dargestellt:

Mit den $\mathcal{H}^2$-Matrizen haben wir also eine sehr effiziente Teilmenge der hierarchischen Matrizen gefunden, die sich noch kompakter speichern lassen und mit denen Berechnungen noch effizienter durchgeführt werden können.

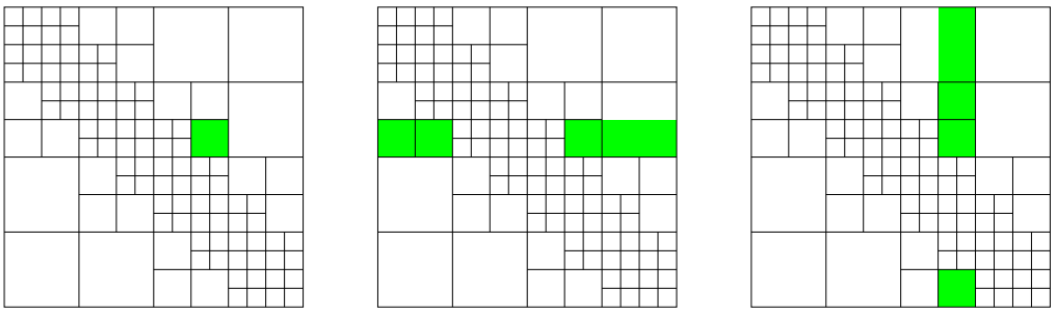


Abbildung 2.6. Vergleich zwischen $\mathcal{H}$- und $\mathcal{H}^2$-Matrix: Bei ersterer müssen nur einzelne Teilmatrizen niedrigen Rang besitzen, bei letzterer ganze Zeilen- und Spaltenblöcke. (Quelle: Börm [2018])

2.2. Paralleles Rechnen

Bereits in Abschnitt 1.1 wurde gezeigt, dass das Problem der gravitationellen Wechselwirkungen in unserer Galaxie mit einem aktuellen Computer nicht ohne weiteres lösbar ist. Konnten sich früher Programmierer darauf verlassen, dass in einigen Jahren eine deutlich schnellere Prozessorgeneration erscheinen würde, so dass Probleme ohne weitere Änderungen schneller gelöst werden könnten, ist dies heute leider nicht mehr der Fall.

Grund hierfür ist die Physik. Dazu ein kleines Rechenexperiment: Nehmen wir einen Prozessorchip von etwa einem Zentimeter Durchmesser an und vergegenwärtigen wir uns die Lichtgeschwindigkeit $c \approx 3 \cdot 10^{10}$ [BIPM 1975]. Licht könnte diesen Chip also etwa 30 Milliarden mal pro Sekunde durchqueren, was 30 Gigahertz entspräche. Bedenkt man, dass man in Prozessoren mit Elektrizität arbeitet und somit beispielsweise mit kapizitären Effekten und proportional zur Taktfrequenz steigender Verlustleistung zu kämpfen hat, sind die 3 Gigahertz aktueller Mittelklasseprozessoren durchaus nicht zu verachten.

Zwar wäre eine Möglichkeit die Chips weiter zu verkleinern, jedoch machen dem Fortschritt hier quantenmechanische Phänomene aktuell noch einen Strich durch die Rechnung. Ein gängiger und gangbarer Lösungsweg besteht in der Parallelisierung von Soft- und Hardware. Durch das gleichmäßige verteilen eines Algorithmus auf mehrere Prozessoren kann die Gesamtlaufzeit entsprechend verringert werden. Unter optimalen Bedingungen könnte theoretisch eine Verdopplung der Prozessoranzahl eine Halbierung der Laufzeit bewirken. Heute haben daher die meisten Prozessoren nicht mehr nur einen Rechenkern, sondern mehrere und für die Berechnung großer mathematischer Probleme arbeiten oft viele Rechner zusammen an der Lösung.[Börm 2017]

Ein weiterer Grund für parallele, beziehungsweise verteilte Rechnersysteme ist die Diskrepanz zwischen vorhandenem und benötigtem Speicher. Beispielsweise hat der diesbezüglich größte Computer im Rechenzentrum der Christian-Albrecht-Universität Kiel einen Hauptspeicher von 768 GB. Ein aktuelles, speicherintensives Problem ist ein engmaschiges Klimamodell der Erde. Bei einer Dichte von einer Masche alle 1,56 km in Äquatornähe wird der Gesamtspeicherbedarf auf etwa 24000 GB geschätzt [Wehner u. a. 2008]. Ein Problem dieser Größe passt nicht mehr in den Hauptspeicher eines Computers. Hinzu kommt die Skalierbarkeit solcher Probleme: Angenommen ein Hauptspeicher von 24000 GB stünde auf einem Computer zur Verfügung. Warum bei einer Maschendichte von 1,56 km stehen bleiben und nicht noch feiner auflösen um noch genauere Berechnungen durchzuführen? Warum bei der Berechnung der gravitationellen Wechselwirkungen bei den Sonnen unserer Galaxie stehen bleiben und nicht die Planeten, Monde oder noch kleinere Himmelkörper mit einbeziehen? Diese Probleme, und damit auch der Speicherbedarf, lassen sich also nahezu beliebig vergrößern.

Im folgenden Kapitel wird kurz skizziert, wie parallel arbeitende Hardware gestalten sein kann und welche Vor- und Nachteile und welche Herausforderungen damit einhergehen.

Tabelle 2.1. Flynnsche Klassifikation mit Beispielen.

	eine Instruktion (single instruction)	mehrere Instruktionen (multiple instruction)
ein Datensatz (single data)	SISD: konventioneller einkerniger Prozessor	MISD: Spezialrechner
mehrere Datensätze (multiple data)	SIMD: Vektorrechner	MIMD: Multicore; Clustersystem

2.2.1. Parallele Hardware

Prozessor

Flynn [1972] klassifiziert 4 unterschiedliche Architekturen von Rechnern. Dabei unterteilt er nach der Möglichkeit des Prozessors zu einem Zeitpunkt einzelne oder mehrere Instruktionen beziehungsweise Datensätze zu verarbeiten (vgl. Tabelle 2.1).

Vor einigen Jahrzehnten waren die meisten Prozessoren in PCs noch SISD-Architekturen. Sie hatten einen Prozessor mit einem einzelnen Kern; konnten also zu jedem Zeitpunkt nur eine Instruktion auf einem Datensatz ausführen. Heute sind Multicore-Prozessoren üblich. Auf einem Prozessorchip finden sich hierbei mehrere Rechenkerne. Die einzelnen Kerne sind unabhängig von einander; sie verwalten eigenen Register und führen eigene Instruktionen aus. Daher zählen Multicore-Prozessoren zu den MIMD-Architekturen. Zu diesen Architekturen gehören ebenfalls die später erläuterten Clustersysteme. [Flynn 1972; Barbic 2006]

MIMD-Architecturen sind auf Grund ihrer autonom arbeitenden und programmierbaren Prozessoren sehr flexibel und können viele Probleme effizient lösen. Die Autonomie der Prozessoren stellt die Programmierung aber auch vor Herausforderungen. Durch die asynchrone Ausführung der Programme wird das Verhalten des gesamten Systems schwer vorhersagbar.[Börm 2017]

MISD-Architecturen, also solche in denen parallel auf einem Datum mehrere Instruktionen ausgeführt werden können sind nur in Spezialrechnern zu finden und für die Lösung der meisten Probleme nicht geeignet.

SIMD-Architekturen hingegen, in denen auf mehreren Daten parallel die gleiche Instruktion ausgeführt werden kann, sind durchaus gängig. Hintergrund sind Algorithmen, die auf ganzen Reihen von Daten immer wieder die selben Operation ausführen. Während beim Ansatz der MIMD-Archituren die Operationen dynamisch auf die einzelnen Recheneinheiten verteilt werden, ist der Ansatz von SIMD-Architekturen, beziehungsweise von Vektorrechnern, die Anzahl der Rechenwerke in den Recheneinheiten zu erhöhen und eine Operation für mehrere Daten gleichzeitig auszuführen. Zum Beispiel werden für die Berechnung der Gravitationskräfte für viele Körper immer wieder die selben Operationen verarbeitet. Anstatt diese also für jedes einzelne Paar von Körpern einzeln durchzuführen könnten sie für mehrere Paare von Körpern gleichzeitig durchgeführt werden.

Der Name Vektorrechner leitet sich von der Anlehnung an die kartesische Geometrie her, in der ein Vektor ein Tupel von Zahlen ist. Entsprechend werden Daten des selben Typs ebenfalls als Vektoren bezeichnet. Um solche Vektoren handhaben zu können, verfügen Vektorrechner über spezielle Vektorregister, die mehrere Daten gleichzeitig fassen können. Für jedes einzelne Datum innerhalb dieser Register ist dann jeweils ein eigenes Rechenwerk im Kern zuständig.

Viele aktuelle Multicore-Prozessoren verfügen zusätzlich über einige Möglichkeiten der Vektorisierung. Es existieren aber auch auf diese Art der Parallelisierung spezialisierte Prozessoren.

verifizieren

Prozessoren laden in der Regel nicht nur ein einzelnes Datum, sondern gleich eine sogenannte *cache line* aus dem Hauptspeicher. Daher ist es sinnvoll, um SIMD-Parallelisierung optimal ausnutzen zu können, die Daten im Hauptspeicher anhand dieser cache lines anzuordnen. Dazu müssen in der Regel explizite Methoden zum Anfordern von *aligned memory* verwendet werden. Außerdem müssen die parallel zu verarbeitenden Daten im Hauptspeicher direkt hintereinander und nicht durch andere Daten unterbrochen auftauchen. Nehmen wir an, wir wollten die Daten der Körper, die für die Berechnung der Gravitationskräfte benötigt werden, in einer Struktur zusammenfassen. Wollen wir Daten zu vielen dieser Körper speichern so wäre es in Bezug auf Vektorrechner nicht sinnvoll ein Array dieser Strukturen anzulegen (engl.: *array of structs*). Dabei würden die im Sinne der Vektorisierung zusammengehörigen Daten durch andere getrennt. Um eine Vektorisierung zu ermöglichen ist es daher sinnvoll die Struktur so zu definieren, dass sie Array der benötigten Daten beinhaltet (engl.: *struct of arrays*). So kann sichergestellt werden, dass diese Daten im Hauptspeicher hintereinander stehen.

Referenz und vielleicht eine erklärende Grafik

Speicherverwaltung

genauer ausarbeiten

In Abschnitt 2.2 wurde bereits kurz erwähnt, dass bei Multicore-Prozessoren jeder Kern seine eigenen Register hat. Genau genommen verfügen die Kerne meist auch über privaten L1-, seltener über L2- oder L3-Cache. Diese sind meist *shared* (dt.: gemeinsam genutzt). Shared Memory wird von allen Prozessoren gemeinsam verwaltet.

Grundsätzlich können die heute üblichen MIMD Architekturen weiter anhand ihrer Speicherverwaltung unterteilt werden. Beim Shared Memory Modell wird ein globaler Speicher gemeinsam genutzt. Sie teilen sich also einen gemeinsamen Adressraum. Dies trifft in der Regel auf den Hauptspeicher, aber eventuell auch, wie bereits erwähnt, auf den Cache zu. Anders ist dies beim Distributed Memory Modell. Hier besitzt jeder Prozessor seinen eigenen Speicher. Shared Memory Systeme sind unter anderem die heute üblichen Multicore-Prozessoren, aber auch Computer mit mehreren Prozessoren, wie sie häufig in Servern und Großrechnern vorkommen. Distributed Memory Systeme sind in der Regel vernetzte Rechnersysteme. Allerdings wird auch auf Shared-Memory-Systemen durch moderne Betriebssysteme ein Distributed Memory Modell simuliert. Durch die

Virtualisierung des Hauptspeichers etwa wird jedes Programm behandelt, als liefe es alleine. Dabei wird jedem Programm ein privater Bereich des Hauptspeichers zugewiesen. Das Programm nutzt diesen Bereich, als sei es der gesamte Hauptspeicher. [Körbler 2007]

Der Vorteil von gemeinsam genutztem Speicher ist, vorausgesetzt das Programm wurde entsprechend entworfen, dass die Ergebnisse eines Kerns den anderen automatisch auch zur Verfügung stehen. Arbeiten also mehrere Recheneinheiten gemeinsam an einem Problem, müssen die Ergebnisse einer Einheit den andern also gegebenenfalls nicht explizit mitgeteilt werden. Dies vermeidet Overhead durch das Austauschen und kopieren von Daten. Jedoch ist es bei privatem Cache möglich, diesen in größerer Nähe zum zugehörigen Kern auf dem Chip zu platzieren. Dies führt zu besseren Zugriffszeiten, als gänzlich gemeinsam genutzter Cache und wird daher auch oft in verwendet.

Ein Nachteil von Shared Memory Systemen ist, dass der Schaltungsaufwand für den Zugriff auf den gemeinsam genutzten Speicher steigt mit wachsender Anzahl Recheneinheiten rapide an [Börm 2017]. Außerdem können Race Conditions auftreten, die durch den gemeinsam genutzten Speicher verursacht werden.

Beim schreiben von parallelen Programmen ist also besondere Vorsicht geboten. Es muss klar sein welche Daten sich wo befinden, und ob sie gemeinsam genutzt werden, oder nicht. Dies ist gegebenenfalls eine noch größere Herausforderung, wenn der Hauptspeicher innerhalb eines parallelen Programms teilweise gemeinsam genutzt wird und teilweise verteilt ist.

Netzwerk

Wie bereits erwähnt gibt es Probleme, die für einen einzelnen Computer zu viel Hauptspeicher benötigen oder zu viel Rechenaufwand bedeuten. Daher ist es naheliegend Computer zu Netzwerken, sogenannten *Clustersystemen*, zusammenzuschließen. Hierbei können die vernetzten Computer, die als *Knoten* des Clustersystems bezeichnet werden, über die Netzwerkverbindung kommunizieren. Clustersysteme gehören zur Klasse der MIMD-Architekturen und stellen meist eine Mischform aus Distributed- und Shared-Memory-Modell dar: Jeder Knoten hat seinen eignen Hauptspeicher, der nicht mit den anderen Knoten geteilt wird, die einzelnen Knoten sind in der Regel aber Shared-Memory-Systeme. Meist Verfügen die einzelnen Knoten über einen oder mehrere Multicore-Prozessor(en). Referenz!

2.2.2. Parallele Software

Im vorherigen Kapitel wurde die Entwicklung von einzelnen Computern mit einer Recheneinheit hin zu Clustersystemen, deren Knoten mit mehreren Recheneinheiten bestückt sind, skizzert.

Solange es sich um einen Computer mit Multicore-Prozessor handelt, ist bis dato keine Änderung an der Software notwendig um auf diesem Multicore-Prozessor lauffähig zu sein. Auf einem Computer läuft in der Regel nicht nur ein einzelnes Programm, sondern

zumindest noch ein Betriebssystem und meist auch noch einige weitere Programme. Der Prozessor kann nun lediglich mehrere Programme echt parallel verarbeiten.

Allerdings profitiert ein Programm bis dato auch wenig von parallel arbeitender Technologie. Es wird nun lediglich gegebenenfalls seltener von anderen Programmen unterbrochen. Soll sich das Potential der Parallelität für ein Programm voll entfalten, so muss das Programm selbst parallel arbeiten und so mehrere Kern beziehungsweise mehrere Knoten eines Clusters nutzen können. Dazu sind natürlich auch entsprechende Entwicklungen im Bereich der Software notwendig. Zum einen muss es überhaupt möglich sein mehrere Instanzen eines Prozesses zu starten und diese zusammenarbeiten zu lassen. Dies impliziert aber wiederum die Schaffung von Möglichkeiten, die oben erwähnten Race Conditions zu vermeiden. Schließlich müssen für Programme, die auf Clustersystemen arbeiten, Möglichkeiten geschaffen werden, zu kommunizieren. Durch diese Kommunikation wäre es dann auch nicht mehr notwendig alle Daten lokal abrufbar zu haben. Statt dessen wäre es Möglich die Datenmenge auf die Knoten zu verteilen und bei Bedarf zwischen Knoten auszutauschen. Erst dadurch sind Probleme wie die Berechnung der Gravitationskräfte oder engmaschige Klimamodelle zu bewältigen.

All diese Konzepte vorzustellen wäre im Rahmen dieser Arbeit nicht zielführend. Im folgenden Kapitel wird daher nur das für diese Arbeit relevante Konzept vorgestellt.

Das Message-Passing-Modell

Für unser Beispielproblem der Berechnung der Gravitationskräfte ist ein Programm vonnöten, das auf einem Clustersystem läuft und dessen Speicher verteilt verwaltet wird. Daher wird im folgenden Kapitel ein Parallel-Programming-Modell vorgestellt, das die Schwierigkeiten von Shared Memory Systemen umgeht und ein Konzept der Speicherverwaltung und Kommunikation auf verteilten Rechnersystemen liefert. Dieses Konzept ist das sogenannte *Messege-Passing-Modell*. Dieses Modell geht von einer Menge von *autonomen* Prozessen aus, die

1. eindeutig benannt sind,
2. jeweils über *privaten Speicher* verfügen.

Dies spezifiziert nicht, um welche Art Rechner(system) es sich handelt. Bei gemeinsam genutztem Hauptspeicher wird davon ausgegagen, dass jeder Prozess einen eigenen privaten Bereich zugewiesen bekommt.

Um nun Daten von einem Prozess P_i an einen Prozess P_j zu kommunizieren, muss P_i explizit im Programmlauf seine Daten in Form einer Nachricht (engl.: message) senden, und respektive P_j diese Nachricht und damit die Daten in Empfang nehmen.

Ein Nachteil des Message-Passing-Modells besteht in der Grundannahme von nicht gemeinsam genutztem Speicher. Jeder Prozess hat seine eigenen Kopien der beötigten Daten und Ergebnisse müssen explizit durch Nachrichten mitgeteilt und in die Speicherbereiche der anderen Prozesse kopiert werden. Dies macht das Programm aber auch weniger anfällig für Race-Conditions. Weitere Vorteile des Message-Passing-Modells sind:

Portabilität Message-Passing ist spätestens seit MPI (vgl.: Abschnitt 2.3) auf den meisten parallelen Plattformen einheitlich implementiert.

Universalität Das Messege-Passing-Modell stellt nur minimale Anforderungen an die zugrundeliegende Hardware. Es funktioniert einheitlich für vernetzte Systeme mit verteiltem Speicher ebenso, wie für Shared-Memory-Systeme oder Kombinationen aus diesen.

Einfachheit Das Modell unterstützt explizite Kontrolle über Referenzen zu Speicherzellen und erleichtert so auch Debugging.

Diese Vorteile machen das Message-Passing-Modell zu einem der Standard-Modelle im High-Performance-Computing.[IBM 2017; Foster 1995; van Engelen 2017]

2.3. MPI

MPI (*Message-Passing Interface*) ist eine Spezifikation für den Nachrichtenaustausch eines verteilten Systems. MPI richtet sich hauptsächlich nach dem *Message-Passing-Modell* (siehe Abschnitt 2.2.2). Erweitert wird das "klassische" Message-Passing-Modell unter anderem durch kollektive Kommunikationsmöglichkeiten, Remote-Speicherzugriff und parallele I/O-Operationen. Als Interface bietet MPI selbst keine Implementierung des Standards, sondern beschreibt Methoden und ihre Semantik.

Das Ziel von MPI ist es, einen Standard für Programme zu liefern, die sich des Message-Passing-Modells bedienen, und somit zu Effizienz, Portabilität und Flexibilität beizutragen. [MPI-Forum 2015]

2.3.1. Geschichte

Bereits vor 1992 gab es Bibliotheken für paralleles Rechnen. Jedoch gab es keinen einheitlichen Standard und die meisten Bibliotheken waren systemspezifisch, sodass das Portieren von Programmen auf ein anderes System zumindest eine aufwendige Aufgabe war. Auch die Ansätze der Bibliotheken unterschieden sich teilweise stark. Ein weit verbreiteter Ansatz war allerdings das Message-Passing-Model, bei dem die verteilten Prozessteile Nachrichten austauschen, um eine gemeinsame Aufgabe zu erfüllen.[Kendall 2017]

Am 29. und 30. April 1992 begann mit dem *Workshop on Standards for Message-Passing in a Distributed Memory Environment* am *Center for Research on Parallel Computing* in Williamsburg (Virginia) ein Prozess zur Standardisierung des Message-Passing-Ansatzes [Walker 1992]. Hier wurden die essentiellen Bestandteile eines standardisierten Message-Passing-Interfaces diskutiert. An diesem Prozess waren rund 60 Personen von 40 verschiedenen Organisationen beteiligt, darunter die bedeutendsten Anbieter von Parallelrechnern sowie Forscher aus Universitäten, staatlichen Laboren und der Industrie. Die Ergebnisse wurden zunächst in einem vorläufigen Entwurf im November 1992 und schließlich in revidierter Fassung in einem Proposal, bekannt als MPI-1, veröffentlicht. [Dongarra u. a. 1993]

Eine Hauptabsicht von MPI-1 war es, erst einmal den "Ball in's Rollen zu bringen" und eine Diskussion anzuregen. Daher beschäftigte es sich noch hauptsächlich mit Point-to-Point-Kommunikation. Zu diesem Zeitpunkt war MPI weder thread safe, noch bot es Methoden zur kollektiven Kommunikation. Aktuell liegt MPI in der Version 3.1 vor und bietet neben der Point-to-Point-Kommunikation auch kollektive Routinen, nicht-blockierende Methoden, automatische Puffer-Verwaltung und vieles mehr. [MPI-Forum 2015]

Umsetzung in C und Fortran
später auch bindings für c++

2.3.2. Point-to-Point-Kommunikation

Die Point-to-Point-Kommunikation ist das durch das Message-Passing-Modell beschriebene Herzstück von MPI. Die Standardmethoden in C-Syntax sind:

Listing 2.1. Die Syntax der standard Sende- und Empfangsoperationen

```
MPI_Send(
  void* data,
  int count,
  MPI_Datatype datatype,
  int destination,
  int tag,
  MPI_Comm communicator)

MPI_Recv(
  void* data,
  int count,
  MPI_Datatype datatype,
  int source,
  int tag,
  MPI_Comm communicator,
  MPI_Status* status)
```

Jede Nachricht besteht aus einem Daten-Teil sowie einem *Umschlag* (engl.: envelope). Der Daten-Teil beinhaltet die Daten, die vom Sende- in den Empfangspuffer kopiert werden sollen (`void* data`), deren Anzahl (`int count`) sowie den Datentyp der zu sendenden Elemente (`MPI_Datatype datatype`). [MPI-Forum 2015]

Der Umschlag besteht aus:

Sender — Dieser bei der Point-to-Point-Kommunikation automatisch hinzugefügt, muss aber bei der kollektiven Kommunikation explizit angegeben werden (vgl.: Abschnitt 2.3.3).

Empfänger — Jeder Prozess im Kommunikator hat eine eindeutige Nummer. Über diese Nummer (`int destination`) wird der Empfänger festgelegt.

Kennzeichnung — Eine Kennzeichnung (engl.: tag) kann zur Differenzierung der Nachrichten eingesetzt werden (`int tag`).

Kommunikator — Der `MPI_Comm communicator` spezifiziert eine Menge von p Prozessen, die sich diesen Kommunikator teilen. Die *Prozessgruppe* ist geordnet und die Prozesse sind durch ihren Rang `destination` $\in \underline{p-1}_0$ innerhalb der Gruppe spezifiziert.

Tabelle 2.2. MPIs Point-to-Point-Kommunikationsvarianten (Quelle: Akinci [2012])

Kommunikationsmodus	blockierende Methode	nicht-blockierende Methode
Standard	MPI_Send	MPI_Isend
Synchro	MPI_Ssend	MPI_Issend
Ready	MPI_Rsend	MPI_Irsend
Buffered	MPI_Bsend	MPI_Ibsend
	MPI_Recv	MPI_Irecv

Diese Informationen ermöglichen es, Nachrichten voneinander unterscheiden und selektiv empfangen zu können. Der zugehörige Empfängeraufruf muss genau zum Sendeaufruf passen. [MPI-Forum 2015]

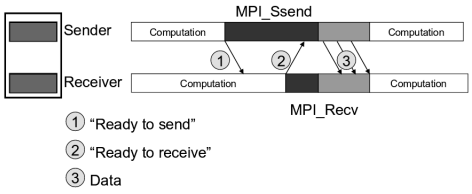
`MPI_Status* status` dient dem empfangenden Prozess dazu eventuelle Fehler zu erhalten. Dies ist besonders dann wichtig, wenn der Sender oder die Kennzeichnung, zum Beispiel auf Grund der Nutzung von Wildcards, nicht bekannt ist. Der Datentyp `MPI_Status` enthält dazu die Member `MPI_SOURCE`, `MPI_TAG` und `MPI_ERROR`. [MPI-Forum 2015]

Bei den in Listing 2.1 dargestellten Standard-Point-to-Point-Methoden handelt es sich um blockierende Operationen. Das heißt, dass das Programm aus `MPI_Send` erst zurückkehrt, wenn sicher ist, dass die gesendeten Daten wieder verändert werden dürfen. Entweder, weil sie in einen temporären Systempuffer, oder weil sie bereits in den Empfängerspeicher übertragen wurden. Dies kann bei manchen Programmen zu viel Wartezeit führen, andere benötigen eventuell mehr Sicherheit im Ablauf. Daher bietet MPI Variationen dieser Standard-Sendeoperation an. Diese sind in Tabelle 2.2 aufgelistet und werden im Folgenden kurz erläutert.

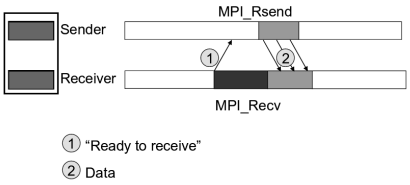
Beim synchronen Senden wird, wie in Abbildung 2.7.(a) dargestellt, zunächst vom sendenden Prozess eine “Ready to send”-Mitteilung verschickt. Auf diese Antwortet der empfangende Prozess mit einer “Ready to receive”-Mitteilung. Im Anschluss findet das Senden und Empfangen der Daten statt. Durch dieses Vorgehen müssen die Prozesse aber gegebenenfalls auf einander warten. Die Wartezeit ist in der Abbildung dunkel eingefärbt. Dafür wird aber nicht nur sichergestellt, dass die gesendeten Daten verändert werden dürfen, sondern auch, dass der Empfänger-Prozess zumindest damit begonnen hat die Daten auch zu empfangen. [Akinci 2012; MPI-Forum 2015]

Das Ready-Send sieht wie eine einfachere synchrone Sende-Variante aus (vgl. Abbildung 2.7.(b)), ist aber restriktiver. Die Methoden `MPI_Rsend` beziehungsweise `MPI_Irsend` erwarten, dass die “Ready to receive”-Mitteilung bereits geschickt wurde, wenn sie aufgerufen werden. Nur wenn diese Mitteilung bereits eingetroffen ist, werden die Daten auch gesendet, ansonsten wird ein Fehler gemeldet. Diese Methode soll den System-Overhead durch Sendeoperationen und Synchronisation minimieren. Es wird jedoch dazu geraten diese Methode nur zu verwenden, wenn die zeitliche Abfolge garantiert ist. [Akinci 2012; MPI-Forum 2015]

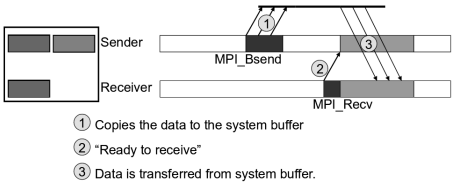
Beim Buffered-Send werden die Daten in einem gesonderten Puffer zwischengespei-



(a) Ablauf der synchronen Sendeoperation.



(b) Ablauf der Ready-Send-Operation.



(c) Ablauf der gepufferten Sendeoperation.

Abbildung 2.7. Quelle: Akinci [2012]

chert. Dies führt durch das zusätzliche Kopieren gegebenenfalls zu weiterem Overhead, dafür kann der Prozess, wie in Abbildung 2.7.(c) zu erkennen, im Anschluss an den Kopiervorgang seine Arbeit fortsetzen und auch den Sendepuffer bereits verändern. Die eigentliche Sendeoperation wird dann zu einem späteren Zeitpunkt nach der "Ready to receive"-Mitteilung des Empfängers durchgeführt. Der Puffer zum Zwischenspeichern muss allerdings auch durch den Nutzer zur Verfügung gestellt werden und wird nicht automatisch durch das System verwaltet. [Akinci 2012; MPI-Forum 2015]

Bei allen blockierenden Methoden können Deadlocks auftreten. Diese können beispielsweise entstehen, wenn zwei Prozesse Daten austauschen wollen, jedoch beide mit einer Sendeoperation beginnen, die nicht zurückkehrt, bevor nicht auch das Empfangen der Daten begonnen wurde. Beide Prozesse werden niemals beginnen Daten zu empfangen und werden somit auch nie aus dem Senden zurückkehren. Abhilfe können die nicht-blockierenden Methoden liefern.

Die nicht-blockierenden Methoden bestehen grundsätzlich aus zwei Aufrufen. Einer Initialisierung des Sendens/Empfangens, ohne dass auf dessen Ausführung gewartet wird, und einer Abschlussmethode, wie `MPI_Wait`, `MPI_Probe` oder `MPI_Test`. Dies ermöglicht es dem sendenden und dem empfangenden Prozess weitere Arbeiten auszuführen, um so möglichst Wartezeiten zu vermeiden beziehungsweise produktiv zu nutzen. Allerdings sollten weder der Sende- noch der Empfangspuffer zwischen Initialisierung und Abschluss verändert werden. [Akinci 2012; MPI-Forum 2015]

2.3.3. Kollektive Kommunikation

Zusätzlich zur klassischen Point-to-Point-Kommuniklation bietet MPI Möglichkeiten für kollektive Kommunikation. Ein Unterschied zur direkten Kommunikation zwischen zwei Knoten ist, dass hier keine getrennten Send- und Receive-Operationen durchgeführt werden. Bei der kollektiven Kommunikation rufen alle Prozesse eines Kommunikators dieselbe Methode auf. Daher sind auch die Parameter `destination` und `tag` nicht mehr notwendig. Wie auch bei der Point-to-Point-Kommunikation gibt es bei diesen Methoden jeweils eine blockierende und eine nicht-blockierende Variante.

Die einfachste kollektive Methode ist `MPI_Barrier`. Diese dient nicht dem Austausch von Daten, sondern ausschließlich der Synchronisation der Prozesse eines Kommunikators. Jeder Prozess, der diese Methode aufruft, wartet, bis jeder Prozess des Kommunikators diese Methode ebenfalls aufgerufen hat. Eine Deadlockgefahr besteht, falls nicht alle Prozesse des Kommunikators die Methode aufrufen. [MPI-Forum 2015]

In Abbildung 2.11 sind einige kollektive Methoden und ihre Funktionsweise veranschaulicht und werden im Folgenden kurz erläutert.

Der Broadcast dient dazu, Daten von einem Prozess an alle anderen zu verteilen. Die Syntax ist Listing 2.2 zu finden.

Die meisten Parameter sind bereits aus dem vorherigen Abschnitt bekannt. Neu ist der Parameter **`int`** `root`. Dieser bezeichnet den Rang des Prozesses im Kommunikator, der die Daten an die anderen verteilen möchte. Er legt also letztlich fest welcher Prozess sendet und welcher empfängt. Auch wenn es auf den ersten Blick so aussehen mag, werden die Daten nicht ausschließlich vom Sender nacheinander an alle Empfänger gesendet, was

Listing (2.2) Die Syntax von `MPI_Bcast`

```
MPI_Bcast(
  void* data,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm communicator)
```

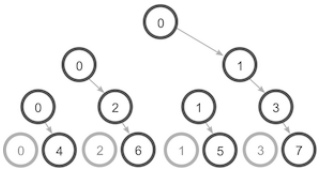


Abbildung 2.9. Mögliche Verteilung der Daten durch `MPI_Bcast`. (Quelle: Kendall [2017])

Listing (2.3) Die Syntax von `MPI_Alltoallv`

```
int MPI_Alltoallv(
  const void* sendbuffer,
  const int sendcounts[],
  const int senddispls[],
  MPI_Datatype sendtype,
  void* recvbuffer,
  const int recvcounts[],
  const int recvdispls[],
  MPI_Datatype recvtype,
  MPI_Comm comm)
```

linearen Aufwand bedeuten würde, sondern baumartig weitergegeben. In Abbildung 2.9 wird ein möglicher Ablauf dargestellt. Durch dieses Vorgehen ist logarithmische Laufzeit erreichbar. [Kendall 2017; MPI-Forum 2015]

`MPI_Scatter` dient ebenfalls dem Verteilen von Daten von einem Prozess auf die anderen, jedoch wird hier ein Array auf alle Prozesse eines Kommunikators verteilt. Dies kann zum Beispiel genutzt werden, um Testdaten in einem Root-Prozess einzulesen und dann an die anderen Prozesse zu verteilen. Umgekehrt zieht `MPI_Gather` die Daten aller Prozesse eines Kommunikators auf einem Prozess zusammen, beispielsweise um die Ergebnisse einer verteilten Berechnung auf einem Prozess zu aggregieren und auszugeben. Beim darunter abgebildeten `MPI_Allgather` wird das Ergebnis nicht nur auf einem, sondern auf allen Prozessen gesammelt. [Kendall 2017; MPI-Forum 2015]

Ähnlich zu den beiden Gather-Methoden sind die Methoden `MPI_Reduce` und `MPI_Allreduce` . Auch diese sammeln Daten von allen Prozessen auf einen, respektive auf alle Prozesse, zusammen. Jedoch werden hierbei die Daten nicht einfach in einem Array gebündelt, sondern über den Parameter `MPI_Op op` eine Operation mitgegeben, die auf die gesammelten Daten angewandt wird. So können aus den gesendet Daten direkt das Maximum, Minimum, die Summe und vieles mehr bestimmt werden. [Kendall 2017; MPI-Forum 2015]

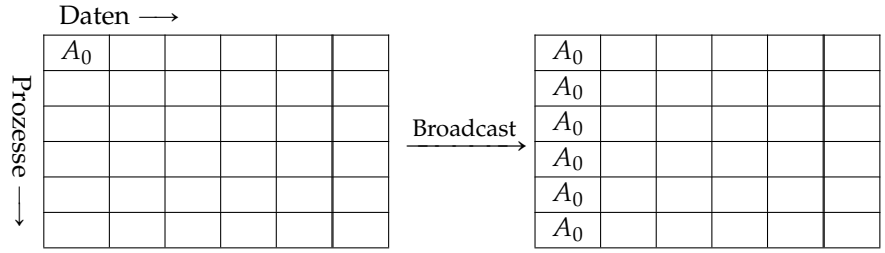
Zuletzt ist `MPI_Alltoall` in Abbildung 2.11 dargestellt. Hierbei führt quasi jeder Prozess ein Scatter durch. Man könnte es auch als *Transponieren* der "Prozess-Daten-Matrix" bezeichnen. [MPI-Forum 2015]

Von jeder dieser kollektiven Methoden gibt es wie bereits erwähnt ebenfalls eine nichtblockierende Variante, die, der Namenskonvention folgend, durch ein eingeschobenes `I` gekennzeichnet ist.

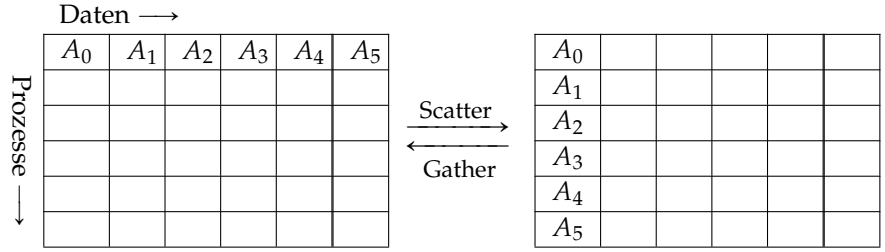
Außerdem gibt es für die Daten austauschenden Methoden eine *vektorisierte* Variante. Alle zuvor erläuterten Methoden erwarten eine feste Anzahl an zu kommunizierenden Elementen pro Prozess. Es kann aber vorkommen, dass zwischen unterschiedlichen Prozessen unterschiedliche Anzahlen von Elementen ausgetauscht werden müssen. Diese vektorisierten Varianten, zu erkennen an einem hinter dem Namen angefügten `v`, erwarten

ein Array von Anzahlen. Als Beispiel ist die Syntax der vektorisierten Alltoall-Methode in Listing 2.3 aufgeführt.

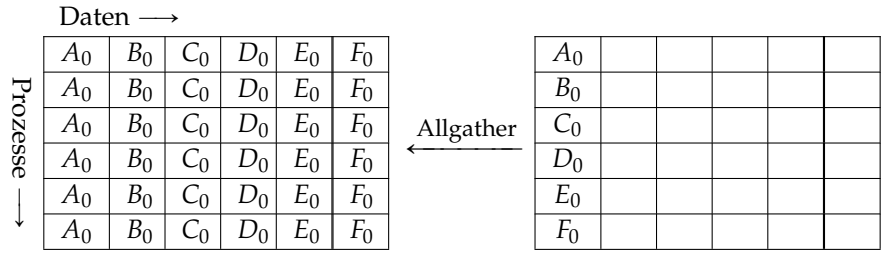
Für $i \in \underline{p-1}_0$ legt der i-te Eintrag der Arrays `sendcounts` und `recvcounts` fest, wie viele Elemente an den i-ten Prozess gesendet, respektive vom i-ten Prozess empfangen werden. Neu sind außerdem die **`int`**-Arrays `senddispls` und `recvdispls`. Der i-te Eintrag legt hier fest, ab welchem Index die Daten im `sendbuffer` für den i-ten Prozess bestimmt sind, beziehungsweise ab welchem Index im `recvbuffer` die Daten des i-ten Prozesses zu empfangen sind. [MPI-Forum 2015]



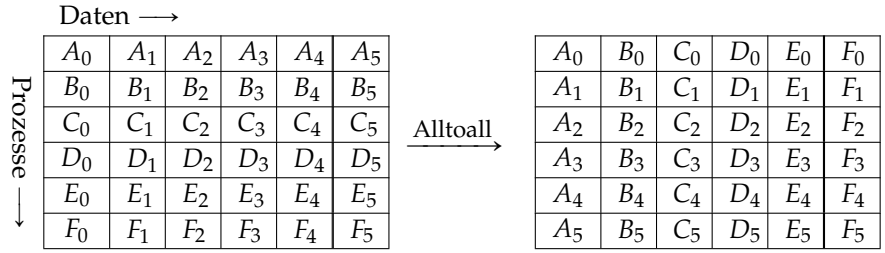
(a) Beim `MPI_Bcast` wird ein Datensatz von einem Root-Prozess an alle anderen Prozesse gesendet und von diesen empfangen.



(b) `MPI_Scatter` verteilt ein Array von Daten auf die Prozesse. `MPI_Gather` vereinigt Daten aller Prozesse in einem Array auf einem Root-Prozess.



(c) `MPI_Allgather` vereint ebenfalls Daten aller Prozesse, allerdings erhält jeder Prozess eine Kopie des Arrays.



(d) `MPI_Alltoall` ist ein simultanes MPI_Scatter aller Prozesse. Betrachtet man die Prozesse und Daten als Matrix, wird eine Matrixtranformation durchgeführt.

Abbildung 2.11. Möglichkeiten der kollektiven Kommunikation in MPI. (Quelle: Schwarz [2011])

Kapitel 3

Hauptteil

3.1. Ausgangssituation

Zur Erinnerung: Wir wollen für $n \in \mathbb{N}_{\geqslant 2}$ die wechselseitigen Gravitationskräfte von n Himmelskörpern berechnen. Da die größten davon die Sonnen sind und es davon bereits mehr als genug in unserer Galaxie gibt, beschränken wir uns zunächst auf diese. Die Menge unserer n Sonnen ist im Folgenden mit der Menge Ω dargestellt. Für jede Sonne $x \in \Omega$ muss die Gleichung

$$f_x = \sum_{\substack{y \in \Omega \\ y \neq x}} \gamma m_x m_y \frac{y - x}{\|y - x\|_2^3} \tag{3.1}$$

gelöst werden. Zunächst möchte ich die Grundstruktur des Programms vorstellen. Das Programm ist in C geschrieben und wurde in der nicht-optimierten Version von Sven Christophersen zur Verfügung gestellt.

Wir benötigen als ersten Schritt eine Datenstruktur, die unsere Sonnen darstellt. Diese ist in Listing 3.1 aufgeführt. In Vorbereitung einer eventuell anschließenden Vektorisierung (vgl.) wurde diese Struktur als struct of arrays (vgl. Abschnitt 2.2.1) angelegt.

Ref auf Ausblick

Die Komponenten der Struktur sind:

1. das Array `int *id`, das einer einfachen Identifizierung der Sonnen dient,
2. die Arrays `double *x, *y, *z`, in denen die Position der Sonnen gespeichert werden,
3. die Arrays `double *Fx, *Fy, *Fz`, in denen die Kraftvektoren, die auf die einzelnen Sonne wirken, gespeichert werden,
4. die Arrays `double *vx, *vy, *vz`, in denen der Geschwindigkeitsvektor jeder Sonne gespeichert wird,
5. das Array `double *m`, in dem die Massen der Sonnen gespeichert werden, und zuletzt
6. die Werte `int n` und `double th`, die die Anzahl an Sonnen und die Länge eines Zeitschritts in Sekunden enthalten.

Die Körper entsprechen keinen reellen Himmelkörpern, zu denen uns keine Daten vorliegen, sondern werden zufällig generiert. Die Sonnen werden in einem Gebiet von ± 1

Listing (3.1) Die Struktur `bodies` dient der Speicherung aller mit den einzelnen Sonnen zusammenhängenden Daten.

```
/* Struct that holds coordinates x, masses m and forces F for all
    particles. */
struct _bodies {
  int    *id; // id to identify each single body
  double *x;  // x-components of the masses.
  double *y;  // y-components of the masses.
  double *z;  // z-components of the masses.
  double *Fx; // x-components of the forces.
  double *Fy; // y-components of the forces.
  double *Fz; // z-components of the forces.
  double *vx; // x-components of the velocities.
  double *vy; // y-components of the velocities.
  double *vz; // z-components of the velocities.
  double *m;  // The masses.
  int     n;  // Number of masses.
  double th;  // length of one timestep.
};
typedef struct _bodies bodies;
```

Lichtjahr in jeder Koordinatenrichtung um einen angenommenen Ursprung mit Massen zwischen einer und 15 Sonnenmassen verteilt. echte Werte

Um nun alle gravitationellen Wechselwirkungen zu berechnen wird die Methode `compute_forces_bodies(...)` (vgl. Listing 3.2) ausgeführt. Gut zu erkennen ist die matrixähnliche Struktur: Die beiden **`for`**-Schleifen entsprechen dem zeilen- und spaltenweisen Durchgehen der zugehörigen $\Omega \times \Omega$-Matrix. Explizit aufgestellt wird diese Matrix allerdings nicht, da dies in Bezug auf den Speicherplatz äußerst ineffizient wäre. Statt dessen wird direkt, wie in Gleichung 3.1 angegeben, die kumulative Kraftwirkung für jede Sonne berechnet und in der `bodies`-Struktur gespeichert. Ein Teil der Kernfunktion ist in den Aufruf von `calc_potential(...)` in Zeile 19 ausgegliedert. Diese Methode berechnet das rein auf der Entfernung der beiden Sonnen beruhende Gravitationspotential $P(x,y)$, sodass die Formel aus Gleichung 3.1 wie folgt dargestellt werden kann:

$$f_x = \sum_{\substack{y \in \Omega \\ y \neq x}} \gamma m_x m_y P(x,y), \quad P(x,y) := \frac{y-x}{\|y-x\|_2^3}.$$

Wie bereits in der Einleitung beschrieben ist das Problem mit diesem Algorithmus, dass er in Bezug auf die Laufzeit quadratisch skaliert. Läuft dieser Algorithmus mit $n = 2^{13} \approx 8.000$ Sonnen noch in $1,097$ Sekunden durch, so braucht er bereits für $n = 2^{15} \approx 32.000$

Sonnen 17,531 Sekunden und für $n = 2^{16} \approx 64.000$ Sonnen bereits über 70 Sekunden.[1] Und diese Größenordnung ist weit entfernt von dem reellen Problem, die Gravitationskräfte in unserer Galaxie mit etwa 100.000.000.000 Sonnen zu berechnen.

Listing (3.2) Diese Methode berechnet die auftretenden Gravitationskräfte.

```
1  /* Computation of the resulting forces for every particle.*/
2  void compute_forces_bodies(bodies *b) {
3    int n = b->n;
4    int i, j;
5    double F[3],P[3];
6    double *x, *y, *z, *m;
7    x = b->x;
8    y = b->y;
9    z = b->z;
10   m = b->m;
11   for (i = 0; i < n; ++i) {
12     F[0] = 0.0;
13     F[1] = 0.0;
14     F[2] = 0.0;
15     for (j = 0; j < n; ++j) {
16       if (i == j) {
17    continue;
18       }
19       calc_potential(x[i], y[i], z[i], x[j], y[j], z[j], P);
20       F[0] += m[j] * P[0];
21       F[1] += m[j] * P[1];
22       F[2] += m[j] * P[2];
23     }
24     b->Fx[i] += GAMMA * m[i] * F[0];
25     b->Fy[i] += GAMMA * m[i] * F[1];
26     b->Fz[i] += GAMMA * m[i] * F[2];
27   }
28 }
```

[1] Ausgeführt auf einem Computer mit 16 Gb Arbeitsspeicher und einem Intel Core i7-4710HQ.

3.2. $\mathcal{H}^2$-Matrixstruktur und Approximation

Nun wollen wir die in Abschnitt 2.1 vorgestellte Struktur der hierarchischen Matrizen nutzen, um den Algorithmus zu beschleunigen. Wir wollen also die Kernfunktion

$$f : \Omega \times \Omega \to \mathbb{R}^3,\ (x,y) \mapsto \gamma m_x m_y \frac{y-x}{\|y-x\|_2^3}$$

durch Lagrange-Interpolation der Ordnung k in jeder Koordinatenrichtung auf Clustern τ und σ des zulässigen Blocks $b = \tau \times \sigma \in \mathcal{L}^+(T_{\Omega\times\Omega})$ approximieren. Dazu konstruieren wir Tschebyscheff-Interpolationspunkte $\xi_{\mathfrak{g},\mathfrak{i}}$, $\mathfrak{g} \in \{\tau,\sigma\}, \mathfrak{i} \in \underline{k}^3 =: M \subset \mathbb{N}^3$ und erhalten so:

$$\begin{aligned} f(x,y) \approx \tilde{f}(x,y) &= \sum_{\nu\in M}\sum_{\mu\in M} \mathcal{L}_{\tau,\nu}(x)\gamma m_x m_y \frac{\xi_{\sigma,\mu}-\xi_{\tau,\nu}}{\|\xi_{\sigma,\mu}-\xi_{\tau,\nu}\|_2^3}\mathcal{L}_{\sigma,\mu}(y) \\ &= \gamma \sum_{\nu\in M} \mathcal{L}_{\tau,\nu}(x) m_x \sum_{\mu\in M} \frac{\xi_{\sigma,\mu}-\xi_{\tau,\nu}}{\|\xi_{\sigma,\mu}-\xi_{\tau,\nu}\|_2^3}\mathcal{L}_{\sigma,\mu}(y) m_y \\ &= \gamma \sum_{\nu\in M} \mathcal{L}_{\tau,\nu}(x) m_x \sum_{\mu\in M} P(\xi_{\tau,\nu},\xi_{\sigma,\mu})\mathcal{L}_{\sigma,\mu}(y) m_y. \end{aligned}$$

Anders ausgedrückt wollen wir die Matrix $(F)_{x,y} := f(x,y)$ durch eine $\mathcal{H}^2$-Matrix $\tilde{F}$ approximieren. Für einen streng zulässigen Blockbaum $T_{\Omega\times\Omega}$ und jeden zulässigen Block $b = \tau \times \sigma \in \mathcal{L}^+(T_{\Omega\times\Omega})$ konstruieren wir die nötigen Matrizen mit $\mu,\nu \in M$ wie folgt:

$$\begin{aligned} E_{\tilde{\tau}} &:= (E_{\tilde{\tau}})_{\mu\nu} := \mathcal{L}_{\tau,\nu}(\xi_{\tilde{\tau},\mu}) && \text{für alle } \tilde{\tau} \in sons(\tau) \text{ falls } \tau \notin \mathcal{L}(T_\Omega), \\ V_\tau &:= \begin{cases} (V_\tau)_{x\nu} := \mathcal{L}_{\tau,\nu}(x) & \text{für alle } x \in \tau \text{ falls } \tau \in \mathcal{L}(T_\Omega), \\ \sum_{\tilde{\tau}\in sons(\tau)} E_{\tilde{\tau}} V_{\tilde{\tau}} & \text{falls } \tau \notin \mathcal{L}(T_\Omega), \end{cases} \\ S_b &:= (S_b)_{\mu\nu} := P(\xi_{\tau,\nu},\xi_{\sigma,\mu}), \end{aligned} \tag{3.2}$$

$$\begin{aligned} E_{\tilde{\sigma}} &:= (E_{\tilde{\sigma}})_{\mu\nu} := \mathcal{L}_{\sigma,\nu}(\xi_{\tilde{\sigma},\mu}) && \text{für alle } \tilde{\sigma} \in sons(\sigma) \text{ falls } \tau \notin \mathcal{L}(T_\Omega), \\ W_\sigma &:= \begin{cases} (W_\sigma)_{y\nu} := \mathcal{L}_{\sigma,\nu}(y) & \text{für alle } y \in \sigma \text{ falls } \sigma \in \mathcal{L}(T_\Omega), \\ \sum_{\tilde{\sigma}\in sons(\sigma)} E_{\tilde{\sigma}} W_{\tilde{\sigma}} & \text{falls } \sigma \notin \mathcal{L}(T_\Omega). \end{cases} \end{aligned} \tag{3.3}$$

Für die unzulässigen Blöcke $b = \tau \times \sigma \in \mathcal{L}^-(T_{\Omega\times\Omega})$ müssen wir weiterhin vollbesetzte Matrizen

$$N_b := (N_b)_{xy} := \gamma m_x m_y P(x,y)$$

auswerten.

Für die Anschauung mag es helfen, sich die Faktoren $\mathcal{L}_{\tau,\nu} m_x =: h_x$ beziehungsweise $\mathcal{L}_{\sigma,\mu} m_y =: h_y$ als Anteil der Masse der Sonne x beziehungsweise y an *Ersatzmassen* $m_{\xi_{\mathfrak{g},\mathfrak{i}}}$ vorzustellen, wobei $\mathfrak{g} \in \{\tau,\sigma\}$ und $\mathfrak{i} \in M$ gilt. Durch die Interpolation werden also für weit genug entfernte Gebiete statt der echten Sonnen Ersatzkörper betrachtet, die sich an der Position der Interpolationspunkte befinden und eben diese Ersatzmassen haben.

Listing (3.3) Die Struktur `Cluster` vereint die Definition eines Clusters mit der eines Clusterbaumes.

```
struct _Cluster{
  bodies *bodies;             //related bodies
  int id;                     //id of the Cluster
  int start;                  //start index of the according subarray
  int n;                      //number of bodies in this Cluster
  double a[3];                //"upper left" corner of the bounding box
  double b[3];                //"lower right" corner of the bounding box
  double center[3];           //center of the bounding box
  double diam;                //diameter of the bounding box
  double *m;                  //substitution masses
  double *F;                  //substitution forces
  double *xs;                 //coordinates of the locations of submasses
  int num_sons;               //number of son clusters
  struct _Cluster *son[2];    //pointers to the son
};
typedef struct _Cluster Cluster;
```

3.2.1. Clusterung

Für die oben beschriebene Approximation müssen wir einen Clusterbaum und einen streng zulässigen Blockbaum konstruieren. Dazu benötigen wir zunächst eine Datenstruktur für Clusterbäume. Diese sind in Listing 3.3 zu finden.

Die Datenstruktur

Die Datenstruktur speichert die für die Cluster notwendigen Daten sowie jeweils Zeiger auf die Sohncluster `Cluster *son[2]`. Jedes `Cluster` verweist auf die zum Baum gehörenden `bodies *bodies`, und nutzt einen Startindex **`int`** `start` sowie die Anzahl **`int`** `n`, um auf die zum Cluster gehörigen Sonnen zu verweisen und zuzugreifen. Auch denkbar wäre es, nicht getrennt das `bodies`-Array und den Startindex zu speichern, sondern direkt auf das erste Element des zum Cluster gehörenden Teilarrays zu zeigen. Für jedes Cluster ein eigenes Array anzulegen würde hingegen zu Redundanz und somit unnötigem Speicherbedarf führen, da jede Ebene des Clusterbaumes eine Partition von Ω erzeugt. Die hier vorgestellte Variante erscheint mir am vielseitigsten, ohne nennenswert unnötig Speicher zu belegen.

Außerdem enthält das Cluster die die Elemente des Clusters umschließende bounding box (vgl. Seite 8). Diese wird durch zwei Punkte in der Gestalt von **`double`** `a[3]` und **`double`** `b[3]` aufgespannt. Da das Zentrum mehrfach bei der Konstruktion und Teilung auftaucht ist dieses zur Laufzeitoptimierung in **`double`** `center[3]` abgespeichert. Ebenso ist der Durchmesser der bounding box in **`double`** `diam` hinterlegt, um ihn nur einmal bei der

Konstruktion und nicht bei jeder Abfrage erneut berechnen zu müssen.

Die drei Arrays **double** *m, **double** *F und **double** *xs repräsentieren die in den Clustern genutzten Ersatzkörper. In *m sind die n Massen, in *F Ersatzkräfte und in *xs die Positionen der Ersatzkörper, beziehungsweise Interpolationspunkte $\xi_{\mathfrak{g},\mathfrak{i}}$, gespeichert. Für jeden Interpolationspunkt muss eine Ersatzmasse gespeichert werden können, sodass *m k^3 Einträge hat. Da *F ein Array von dreidimensionalen Vektoren ist, enthält es $3k^3$ Einträge. Zwar sind *xs ebenfalls dreidimensionale Punkte, jedoch bilden diese ein regelmäßiges Raster. Indem wir die Koordinatenprojektionen der Punkte speichern, reichen $3k$ Einträge. Die Interpolationspunkte können dann durch $\{\mathtt{xs}_0,\ldots,\mathtt{xs}_{k-1}\} \times \{\mathtt{xs}_0,\ldots,\mathtt{xs}_{k-1}\} \times \{\mathtt{xs}_0,\ldots,\mathtt{xs}_{k-1}\}$ rekonstruiert werden. Die Einträge sind jeweils nach Koordinatenrichtung zusammengefasst, also erst alle x-Koordinaten, dann die y-Koordinaten usw.. Dies bildet wieder eine "struct of arrays"-Struktur, um eine einfache Vektorisierung zu ermöglichen.

Die Membervariable **int** num_sons dient in diesem Fall der einfacheren Abfrage, ob es sich bei dem Cluster um ein Blatt oder einen inneren Knoten handelt. In allgemeineren Kontexten ist es durchaus möglich, dass Cluster unterschiedlich viele Sohncluster haben. Beispielweise hat der zugehörige Blockbaum $T_{\Omega\times\Omega}$ vier Sohnblöcke pro Nicht-Blattblock.

Die Konstruktion

Nun, da wir eine Datenstruktur für Clusterbäume definiert haben, müssen wir diesen noch erzeugen. Sei dazu eine bodies Struktur mit zufällig generierten Sonnen gegeben.

Nun beginnen wir, indem wir ein Wurzelcluster *root* mit $\mathtt{id}_{root} = 0$ konstruieren. Per Definition enthält es alle in bodies gegebenen Sonnen, womit die Member *bodies, start und n bereits festgelegt sind. Als nächstes erzeugen wir für die gegebenen Sonnen eine bounding box mit minimalen Ausmaßen. Dazu fassen wir für jede Koordinatenrichtung die kleinsten (beziehungsweise größten) Positionswerte der bodies in a (beziehungsweise b) zusammen. Alternativ könnte man, wegen der zufälligen Konstruktion der Sonnen, auch dessen Grenzen als bounding box wählen. Zu dieser bounding box werden nun die Werte center[3] und diam berechnet. Die Member **double** *m und **double** *F bleiben mit 0 inizialisiert. Die Interpolationspunkte xs werden allerdings ebenfalls jetzt berechnet. Dazu wurden beim Start des Programmes bereits Tschebyscheff-Interpolationspunkte auf dem Intervall $[-1,1]$ berechnet. Diese werden jetzt für jede Koordinatenrichtung ι auf das entsprechende Intervall $[a_\iota, b_\iota]$ des Quaders der bounding box transformiert.

Um nun mit dieser Wurzel einen Clusterbaum zu konstruieren, unterteilen wir ein Cluster jeweils in Sohncluster bis eine Abbruchbedingung erfüllt ist. Die wichtigsten Auszüge aus dem Code zur rekursiven Konstruktion des Clusterbaumes sind in Listing 3.4, Listing 3.5 und Listing 3.6 aufgeführt.

Da wir mit zufällig generierten und recht großen Datenmengen arbeiten, ist es nicht notwendig, die Unterteilung eines Clusters in Sohncluster kardinalitätsgesteuert vorzunehmen, um ein gutes load balancing zu erhalten. Dies könnte bei reellen Daten möglicherweise notwendig werden, wenn es in der betrachteten Menge Ω große Bereiche mit sehr wenigen Körpern gibt. Der hier vorgestellte Algorithmus unterteilt die Cluster auf jeder Stufe des

Listing (3.4) Dies sind die Kernmethoden der Konstruktion des Clusterbaumes. Die Teilung der bounding box wird hier vorgenommen sowie die Sortierung der Elemente und Konstruktion der Sohncluster gestartet.

```
1  void _setup(Cluster *c, int depth){
2  [...]
3    if(depth < MAX_DEPTH){      //split as long as condition is not met
4      _setup_nonLeafCluster(c, depth);
5    }
6  [...]
7  }
8  void _setup_nonLeafCluster(Cluster *c, int depth){
9    int dir = _largest_direction(c);
10   int border = _sortIndices(c, dir);   //number of elements in the first
         son cluster
11   double b1[3],a2[3];         //new bounding box points
12   for(int d = 0; d < 3; d++){
13     if(d == dir){
14       b1[d] = c->center[d];
15       a2[d] = c->center[d];
16     } else {
17       b1[d] = c->b[d];
18       a2[d] = c->a[d];
19     }
20   }
21   c->num_sons = 2;
22   c->son[0] = _new_bound_Cluster(c->start, border, c->bodies, c->a, b1);
23   _setup(c->son[0], depth+1);
24   c->son[1] = _new_bound_Cluster(c->start + border, c->n - border, c->
         bodies, a2, c->b);
25   _setup(c->son[1], depth+1);
26 }
```

Baumes rein geometrisch durch eine Mittelebene. Dazu wird zunächst die Richtung der größten Ausdehnung der bounding box bestimmt und dann in dieser Koordinatenrichtung mittig unterteilt. Während dieser Unterteilung werden die zunächst ungeordneten `bodies` gemäß ihrer Zugehörigkeit zu einem der beiden Sohncluster sukzessive sortiert.

Die Methode `_sortIndices` beruht, über die gesamte Konstruktion des Clusterbaumes betrachtet, auf der Grundidee von Quicksort. Für die größte Richtung $[a_\iota, b_\iota]$ des Quaders, der die bounding box darstellt, wird als Quasi-Pivotelement jeweils die Mitte $c_\iota = \frac{a_\iota + b_\iota}{2}$ dieser Richtung gesetzt und die Sonnen entsprechend ihrer Position in die linke oder rechte Hälfte einsortiert. Anders als bei Quicksort wird auf Grund der gewählten Abbruchbedin-

Listing (3.5) Hier wird der Wert MAX_DEPTH für die Abbruchbedingung berechnet.

```
1  void init(int argc, String* argv){
2  [...]
3  switch(argc){
4    case 3:
5      pot = atoi(argv[2]);
6    case 2:
7      INTERPOLATION_POINTS = atoi(argv[1]);
8      [...]
9  }
10   NUM_SUB_MASSES = INTERPOLATION_POINTS * INTERPOLATION_POINTS *
         INTERPOLATION_POINTS;
11 [...]
12   int leaf_pot = INTERPOLATION_POINTS - 1 ? ceil(log2(2 * NUM_SUB_MASSES))
          : 4;
13   MAX_DEPTH = pot - leaf_pot;
14 [...]
15 }
```

gung die Liste nicht vollständig sortiert. Für den Algorithmus ist die Zugehörigkeit der Sonnen zu bestimmten Clustern die entscheidende Information.

Nach Definition 2.6 wurden zwei Beispiele für Abbruchbedingungen erwähnt. Wegen der zufälligen Generierung lässt sich die ungefähre Anzahl Elemente in den Blattclustern anhand der Tiefe des Baumes schätzen. Indem wir als Abbruchbedingung eine maximale Tiefe für den Clusterbaum festlegen, stellen wir gleichzeitig sicher, dass der Clusterbaum ein vollständiger Binärbaum wird. Der Wert MAX_DEPTH für die Abbruchbedingung wird gleich zu Beginn des Programmlaufs auf Basis der Anzahl an Sonnen ($|\Omega| = 2^{\mathtt{pot}}$) und der Interpolationsordnung k berechnet. Dabei gilt:

$$\mathtt{MAX\_DEPTH} := \mathtt{pot} - \mathfrak{l}, \qquad \mathfrak{l} := \begin{cases} \lceil \log_2(2k^3) \rceil & \text{falls } k > 1 \\ 4 & \text{falls } k = 1 \end{cases} \tag{3.4}$$

Bei einer Baumtiefe von pot wären die Blattcluster im Schnitt gerade einelementig. Indem die Abbruchbedingung die Tiefe des Clusterbaumes auf $\mathtt{pot} - \mathfrak{l}$ begrenzt, enthalten die Blattknoten im Schnitt gerade $2^{\mathfrak{l}}$ Elemente. Durch diese Baumtiefenregulierung wird für $k > 1$ die nächsthöhere Zweierpotenz zur doppelten Anzahl an Interpolationspunkten als durchschnittliche Anzahl Sonnen pro Blattcluster festgelegt. Für die Interpolationsordnung $k = 1$ wäre dies eine zu feine Zerlegung. Daher wird hier $\mathfrak{l}$ auf den Wert 4 und damit die durchschnittliche Blattgröße auf $2^4 = 8$ gesetzt. Indem die Blattgröße im Schnitt auf ungefähr das Doppelte der Anzahl an Interpolationspunkten festgelegt wird, soll eine gute Balance zwischen Genauigkeit und Laufzeit erreicht werden.

Listing (3.6) Diese Methode nimmt die Sortierung der `bodies` anhand einer vorgegebenen Koordinatenrichtung vor.

```
1  int _sortIndices(Cluster *c, int dir){
2    if(0 == c->n){
3      return 0;
4    }
5    int j, front, back;
6    double *assoc_coord;
7
8    //get associated coordinates of bodies
9    assoc_coord = _getAssocCoord(c->bodies, dir);
10
11   //sort the bodies according to clusters center
12   front = 0;
13   back  = c->n;
14   back -= 1;
15   do {
16     j = c->start + front;
17
18     if(assoc_coord[j] < c->center[dir]){
19       front++;
20     } else {
21       if(assoc_coord[c->start + back] <= c->center[dir]){
22    swap_bodies(c->bodies, c->start + front, c->start + back);
23       }
24       back--;
25     }
26   } while(front <= back);
27
28   return front; //front resembles the number of elements in the left half
29 }
```

Für die Blattcluster ist keine eigene setup-Methode notwendig, da der Konstruktor `_new_bound_Cluster(...)` zunächst alle Werte für ein Blattcluster setzt. Die Methode `_setup(...)` entscheidet also anhand der Stufe ob ein Cluster ein Blattcluster bleibt, oder durch Unterteilung zu einem Nicht-Blattcluster wird.

3.2.2. Vorwärtstransformation

Nun können wir uns an die Berechnung der Gravitation machen. Um eine Matrix-Vektor-Multiplikation mit der Matrix $\tilde{F}$ auf zulässigen Blöcken auszuwerten, muss diese Operation

Listing (3.7) Diese Methode dient dem rekursiven Aufruf der Vorwärtstransformation und delegiert die Konstruktion der Ersatzmassen. Dabei wird zwischen Blatt-und Nicht-Blattclustern unterschieden.

```
1 void forward(Cluster *c){
2   if(c->num_sons){
3       _forward_nonLeaf(c);
4   } else {
5       _forward_leaf(c);
6   }
7 }
```

als erstes auf den Matrizen W_σ ausgewertet werden. Die Auswertung der Matrizen W_σ nennt man auch *Vorwärtstransformation*.

Da wir die Gravitationskräfte für alle Sonnen in Ω berechnen wollen, ist es sinnvoll die Vorwärtstransformation einfach für alle $\sigma \in T_\Omega$ durchzuführen. Dies geschieht im vorliegenden Programm durch einen Aufruf von `forward(Cluster *c)` mit $root(T_\Omega)$. Die Methode `forward(...)` unterscheidet zwischen Blatt- und Nicht-Blattclustern, und ruft entsprechend die Methode `_forward_Leaf(Cluster *c)` oder `_forward_nonLeaf(Cluster *c)` auf. Die Methode `_forward_Leaf(...)` entspricht der Konstruktion und Auswertung der Matrizen W_σ anhand der Sonnen $x \in \sigma$, die Methode `_forward_nonLeaf(Cluster *c)` der anhand der Sohncluster. Für letztere müssen daher die Ergebnisse der Söhne bereits vorliegen. Daher ruft diese als ersten Schritt `forward(cs)` für alle Sohncluster `cs` auf. Auf diese Weise durchlaufen die Methoden gemeinsam betrachtet den Clusterbaum rekursiv. Der Code der drei Methoden ist in Listing 3.7, Listing 3.8 und Listing 3.9 zu finden.

Durch die Methode `_forward_Leaf(Cluster *c)` werden, wie auf Seite 40 beschrieben, für alle Interpolationspunkte $\xi_{c\mu} \in \bigtimes_{i=0}^{2} \{\mathtt{xs}_{i\cdot k}, \ldots, \mathtt{xs}_{(i+1)\cdot k-1}\}, \mu \in M$ durch

$$\mathtt{m}_{\xi_{c\mu}} = \mathtt{m}_\mu := \sum_{y \in \mathtt{c}} \mathcal{L}_{\mathtt{c},\mu}(y) m_y$$

die Ersatzmassen in der Clusterbasis berechnet und gespeichert. Die Methode `_forward_nonLeaf(Cluster *c)` berechnet ebenfalls Ersatzmassen, jedoch nicht aus allen Sonnen des Clusters, sondern aus den Interpolationspunkten und Ersatzmassen der Sohncluster (vgl. Abschnitt 2.1.2).

Stellt man sich den Baum von der Wurzel aus nach unten hängend vor, werden also die Massen der Sonnen in den Baum hochgezogen. Je höher die Stufe des Baumes, um so gröber ist die Approximation.

Listing (3.8) Diese Methode arbeitet auf den Transfermatrizen und ruft daher zuert rekursiv die Vorwärtstransformation für die Sohncluster auf.

```
1  void _forward_nonLeaf(Cluster *c){
2    int i, j, k, w, wj, son, ison, json, kson, wson, wsonj;
3    double sum, l1, l2, l3;
4
5    forward(c->son[0]);   //recursiv call for first son
6    forward(c->son[1]);   //recursiv call for second son
7
8    //for all submasses
9    for(i = 0; i < INTERPOLATION_POINTS; i++){
10     for(j = 0; j < INTERPOLATION_POINTS; j++){
11       wj = i * INTERPOLATION_POINTS + j;
12       for(k = 0; k < INTERPOLATION_POINTS; k++){
13     w = wj * INTERPOLATION_POINTS + k;
14     sum = 0.0;
15
16     //for all sons
17     for(son = 0; son < 2; son++){
18       //for all sons submasses
19       for(ison = 0; ison < INTERPOLATION_POINTS; ison++){
20         l1 = lagrange(c, i, 0, c->son[son]->xs[ison]);
21         for(json = 0; json < INTERPOLATION_POINTS; json++){
22           wsonj = ison * INTERPOLATION_POINTS + json;
23           l2 = l1 * lagrange(c, j, 1, c->son[son]->xs[INTERPOLATION_POINTS
                 + json]);
24           for(kson = 0; kson < INTERPOLATION_POINTS; kson++){
25        wson = wsonj * INTERPOLATION_POINTS + kson;
26        l3 = l2 * lagrange(c, k, 2, c->son[son]->xs[2 * INTERPOLATION_POINTS
               + kson]);
27
28        //calculate the substitution mass
29        sum += c->son[son]->m[wson] * l3;
30           }
31         }
32       }
33     }
34     c->m[w] = sum;
35        }
36      }
37    }
38 }
```

Listing (3.9) Diese Methode erstellt die Clusterbasis des Clusterbaumes.

```
1  void _forward_leaf(Cluster *c){
2    if(0 == c->n){ return; }
3    int i, j, k, l, m, w, wj;
4    double x, y, z, *mass, l1, l2, l3;
5    bodies *b = c->bodies;
6    mass = b->m;
7
8    for(i = 0; i < NUM_SUB_MASSES; i++){
9      c->m[i] = 0.0;
10   }
11
12   //for all linked actual masses
13   for(l = 0; l < c->n; l++){
14     m = c->start + l;
15     x = b->x[m];
16     y = b->y[m];
17     z = b->z[m];
18
19     //for all interpolation points:
20     for(i = 0; i < INTERPOLATION_POINTS; i++){
21       l1 = lagrange(c, i, 0, x);
22       for(j = 0; j < INTERPOLATION_POINTS; j++){
23    wj = i * INTERPOLATION_POINTS + j;
24    l2 = l1 * lagrange(c, j, 1, y);
25    for(k = 0; k < INTERPOLATION_POINTS; k++){
26      //actual interpolation point:
27      w = wj * INTERPOLATION_POINTS + k;
28      //actual lagrange weight:
29      l3 = l2 * lagrange(c, k, 2, z);
30
31      //calculate the substitution mass
32      c->m[w] += mass[m] * l3;
33    }
34       }
35     }
36   }
37 }
```

3.2.3. Auswertung der Kopplungsmatrizen

Der nächste Schritt in der Auswertung ist die Multiplikation mit den Kopplungsmatrizen S_b. Da diese Matrizen, im Gegensatz zu den Clusterbasen V_τ und W_σ, vom Block und damit von beiden und nicht nur von einem Cluster abhängen, müssen wir nun den Blockbaum $T_{\Omega\times\Omega}$ durchlaufen. Gleichzeitig können wir in diesem Schritt die Nahfeldmatrizen N_b, die ebenfalls von den Blöcken abhängen, auswerten.

Den Blockbaum haben wir nicht explizit aufgestellt. Das ist auch nicht notwendig, da wir ihn implizit konstruieren können und somit keinen Speicherplatz benötigen. Wir durchlaufen unseren impliziten Blockbaum, indem wir, beginnend mit dem Wurzelblock $b_{root} = root(T_\Omega) \times root(T_\Omega)$, für jeden Block $b = \tau \times \sigma$ prüfen, ob

1. $\mathcal{Z}_\eta(\tau, \sigma) = \textit{zulässig}$ erfüllt ist. Ist dies der Fall, haben wir, unabhängig davon, ob $\tau \in \mathcal{L}(T_\Omega)$ oder $\sigma \in \mathcal{L}(T_\Omega)$ gilt, einen Blattblock erreicht und können S_b auswerten.
2. Ist $\mathcal{Z}_\eta(\tau, \sigma) = \textit{unzulässig}$, prüfen wir, ob
   (a) $\tau, \sigma \in \mathcal{L}(T_\Omega)$. Falls τ und σ Blattcluster sind, haben wir ebenfalls einen Blattblock erreicht, können aber nicht approximieren und müssen daher N_b auswerten.
   (b) Sind τ und σ aber keine Blattcluster, so können wir durch folgende Zuweisung rekursiv fortfahren:

Für $\tilde{\tau}_0 \neq \tilde{\tau}_1 \in sons(\tau)$ und $\tilde{\sigma}_0 \neq \tilde{\sigma}_1 \in sons(\sigma)$ definieren wir Sohnblöcke

$$\tilde{b}_0 := \tilde{\tau}_0 \times \tilde{\sigma}_0 \qquad \tilde{b}_1 := \tilde{\tau}_0 \times \tilde{\sigma}_1$$
$$\tilde{b}_2 := \tilde{\tau}_1 \times \tilde{\sigma}_0 \qquad \tilde{b}_3 := \tilde{\tau}_1 \times \tilde{\sigma}_1.$$

Durch dieses Vorgehen konstruieren wir implizit einen streng zulässigen Blockbaum. Zulässige Blöcke werden automatisch Blattblöcke, da die Rekursion dort nicht fortgesetzt wird. Außerdem werden die Sohnblöcke immer aus genau einer Stufe des Clusterbaumes gebildet, bei welchem wir sichergestellt haben, dass er als vollständiger Binärbaum konstruiert wird. Daher ist auch sichergestellt, dass unzulässige Blattblöcke nur aus Blättern des Clusterbaumes bestehen können.

Für jeden Blattblock $b = \tau \times \sigma$ wird also

1. für alle $x \in \tau$ und $y \in \sigma$ die Funktion f ausgewertet und direkt in den `bodies` gespeichert, falls b unzulässig ist, oder
2. für alle Interpolationspunkte $\xi_{\tau,\nu}$, $\xi_{\tau,\nu}$, mit $\nu, \mu \in M$ der Ausdruck $P(\xi_{\tau,\nu}, \xi_{\tau,\nu})$ ausgewertet und mit den in der Vorwärtstransformation berechneten Ersatzmassen $\mathtt{m}_\mu$ multipliziert, falls b zulässig ist.

Anschaulich werden in diesem Schritt in den zulässigen Blöcken aus den Ersatzmassen und Interpolationspunkten Quasi-Ersatzkräfte[2] berechnet.

[2] An dieser Stelle wird nur die Masse des Source-Clusters σ verrechnet. Daher handelt es sich im physikalischen Sinne noch nicht um eine Kraft.

Listing (3.10) Durch diese rekursive Struktur wird ein impliziter Blockbaum durchlaufen, um die Matrizen S_b bzw. N_b auszuwerten.

```
1  void _eval(Cluster *ct, Cluster *cs){
2    if(admissable(ct, cs)){
3        _eval_CC(ct, cs);
4    } else {
5      if (ct->num_sons && cs->num_sons){ //both clusters have sons left:
6        _eval(ct->son[0], cs->son[0]);
7        _eval(ct->son[0], cs->son[1]);
8        _eval(ct->son[1], cs->son[0]);
9        _eval(ct->son[1], cs->son[1]);
10     } else {                          //no son clusters left but still
           not admissable:
11       _eval_full(ct, cs);
12     }
13   }
14   // world.rank?:current_level?:printf("\nfirst admissable block on level
         %d\n", first_admissable);
15   // world.rank?:current_level?:printf("split depth: %d\n", SPLIT_DEPTH);
16 }
```

3.2.4. Rückwärtstransformation

Der letzte Schritt der Berechnung ist die Auswertung der Matrizen V_τ. Diesen Schritt nennt man auch *Rückwärtstranformation*.

Auch dieser Schritt hängt wieder ausschließlich von den Clustern ab. Daher kann die Rückwärtstranformation wieder unabhängig von den Blöcken für den gesamten Clusterbaum T_Ω ausgeführt werden. Wir rufen dazu wieder mit $root(T_\Omega)$ die Methode `backward( Cluster *c)` auf. Wie bereits bei der Vorwärtstransformation unterscheidet diese Methode, ob es sich bei dem aktuellen Cluster um ein Blatt- oder ein Nicht-Blattcluster handelt und delegiert die Arbeit entsprechend an die Methoden `_backward_Leaf(Cluster *c)` und `_backward_nonLeaf(Cluster *c)`. Anders als bei der Vorwärtstransformation müssen bei der Rückwärtstransformation die Ergebnisse des Vaterclusters für die Söhne vorliegen. Daher gibt die Methode `_backward_nonLeaf(Cluster *c)` die approximierten Quasi-Kräfte, gewichtet über die Lagrange-Polynome, an die Sohncluster weiter. Für alle Söhne $\tilde{c} \in sons(c)$ und alle $\mu \in M$ wird

$$\mathsf{F}_{c,\tilde{c},\mu} = \sum_{\nu \in M} \mathcal{L}_{c,\nu}(\xi_{\tilde{c},\mu} \mathsf{F}_{c,\nu}$$

zu den dort eventuell bereits aus der Auswertung der Kopplungsmatrizen vorhandenen Quasi-Ersatzkräfte addiert.

Schließlich werden in der Methode `_backward_Leaf(Cluser *c)` die Quasi-Ersatzkräfte

$\mathsf{F}_{\mathsf{c},\nu}$, $\nu \in M$ über die Lagrange-Polynome gewichtet auf alle Sonnen $x \in \mathsf{c}$ verteilt und mit deren Masse m_x multipliziert. Da auch hier bereits aus der Nahfeldauswertung Kräfte vorhanden sind, müssen diese approximierten Kräfte[3] zu den vorhandenen addiert werden.

Da diese Methoden analog zu denjenigen der Vorwärtstransformation funktionieren sind diese hier nicht aufgeführt.

[3] Durch die Multiplikation mit m_x werden die Quasi-Kräfte im physikalischen Sinne zu Kräften vervollständigt.

3.3. Parallelisierung

Im vorherigen Kapitel haben wir die Berechnung der Gravitationskräfte durch Interpolation approximiert. Dadurch konnten wir die Technik der $\mathcal{H}^2$-Matrizen nutzen, um den Rechenaufwand des Algorithmus zu optimieren. Um die absolute Laufzeit aber noch weiter zu reduzieren, sind wir an einer parallel arbeitenden Variante dieses Algorithmus interessiert. In diesem Kapitel wird ein Ansatz dazu vorgestellt.

Ziel ist es, zunächst einen einfachen Algorithmus zu entwerfen. Daher beschränken wir uns auf Parallelisierung nach dem Message-Passing-Modell unter Verwendung von MPI. Wir können also $p \in \mathbb{N}$ Prozesse starten, von denen jeder eine eindeutige $id \in \underline{p-1}_0$ und seinen privaten Speicherbereich besitzt. Insbesondere ist in diesem Modell Shared-Memory ausgeschlossen. Außerdem können die Prozesse miteinander kommunizieren, um Daten auszutauschen. Die Menge der Prozesse wird im Folgenden mit $\mathfrak{P}$ bezeichnet.

3.3.1. Arbeitsverteilung

Damit wir von parallel arbeitenden Prozessen profitieren können, muss die Arbeit möglichst gleichmäßig auf diese Prozesse verteilt werden. Wir folgen, in modifizierter Variante, dem von Börm und Bendoraityte [2008] vorgestellten Cluster-zentrierten Ansatz. Dieser basiert auf dem Grundgedanken, dass für Vorwärts- und Rückwärtstransformation ausschließlich Daten zwischen Vater- und Sohnclustern ausgetauscht werden müssen. Um möglichst viel Kommunikation zu sparen, ist es daher besonders effektiv, wenn möglichst viele Söhne durch denselben Prozess verarbeitet werden wie der Vater. Besonders einfach wird das Verteilen der Cluster und das Loadbalancing, wenn wir p als Zweierpotenz $p = 2^q$ wählen. Da dann die Anzahl von Clustern in $T_\Omega^{(q)}$ gerade p entspricht, können wir diese Ebene, zuzüglich der Sohncluster, optimal auf die Prozesse verteilen. Wir gehen im Folgenden immer von einer so gewählten Anzahl Prozesse aus.

Um diesen Ansatz auf den Algorithmus zu übertragen, wird in der in Listing 3.5 aufgeführten `init`-Methode die globale Variable `SPLIT_DEPTH` $= log_2(p) = q$ gesetzt. Außerdem bekommt die Datenstruktur `Cluster` einen weiteren Member: **`int`** `active`. In diesem wird die *id* des für dieses Cluster zuständigen Prozesses gespeichert.

Zudem gibt es aber noch Cluster auf den Ebenen $T_\Omega^{<q} := T_\Omega^{(0)}, \ldots, T_\Omega^{(q-1)}$. Um ein Cluster $C \in T_\Omega^{<q}$ zu klassifizieren nutzt jeder Prozess $P \in \mathfrak{P}$ mit id_P zwei Konstanten. Gilt für alle Nachfahren $\overline{C} \in sons^*(C)$: $\overline{C}$.`active` $\neq id_P$, so wird der Member C.`active` auf die **`int`**-Konstante `inactive` gesetzt. Gibt es aber Nachfahren $\overline{C} \in sons^*(C)$ mit $\overline{C}$.`active` $= id_P$, so wird der Member C.`active` stattdessen auf die **`int`**-Konstante `semi_active` gesetzt. Inaktive Cluster werden während der Vorwärts- und Rückwärtstransformation nicht weiter beachtet. Semi-aktive Cluster berechnen zwar Ersatzmassen, aber nur aus den Ersatzmassen des (semi-)aktiven und nicht des inaktiven Sohnclusters.

Um diese Einteilung vorzunehmen, wird vorrangig der Code der Methode `_setup(...)` (vgl. Listing 3.4) angepasst. Zudem bekommt der Konstruktor `_new_bound_Cluster` einen

Listing (3.11) Für die Verteilung der Cluster auf die Prozesse angepasste _setup-Methode.

```
1  void _setup(Cluster *c, int depth){
2    if(depth == SPLIT_DEPTH){
3      c->active = ++split_count;
4    }
5
6    if(depth < MAX_DEPTH){
7      _setup_nonLeafCluster(c, depth);
8    }
9
10   if(c->active == semi_active){
11     if(c->son[0]->active == inactive && c->son[1]->active == inactive){
12       c->active = inactive;
13     } else{
14       if(c->son[0]->active != world.rank
15    && c->son[0]->active != semi_active
16    && c->son[1]->active != world.rank
17    && c->son[1]->active != semi_active) {
18       c->active = inactive;
19       }
20     }
21   }
22 }
```

neuen Parameter, um den Member `active` aller Cluster zu initialisieren. Für die Konstruktion der Wurzel ist dieser auf `semi_active` gesetzt. Danach wird durch die Methode `_setup_nonLeafCluster(...)` immer der Wert des Vaters an die Söhne weitergegeben. Der angepasste Code der `_setup`-Methode ist in Listing 3.11 aufgeführt.

Zunächst wird überprüft, ob die `SPLIT_DEPTH` erreicht wurde. Ist dies der Fall, wird der Member `active` auf die *id* des zuständigen Prozesses gesetzt. Dies geschieht einfach durch Abzählen. Als nächstes folgt der bereits aus Listing 3.4 bekannte Aufruf, der das Teilen in Sohncluster, das Sortieren der `bodies` und die Rekursion beinhaltet. Der letzte Teil wird beim Abbau der Rekursion durchgeführt. Hier wird für eben die Cluster in $T_\Omega^{(<q)}$ überprüft, ob diese auf `semi-active` bleiben, oder, falls kein Nachfahre aktiv ist, der Member `active` auf `inactive` gesetzt wird.

Die Idee der semi-aktiven Cluster beruht darauf, die Gestalt der Spalten- bzw. Zeilenmatrizen V_τ und W_σ auszunutzen. Diese werden für Nicht-Blattcluster durch die Transfermatrizen aus den Söhnen konstruiert (vgl. Abschnitt 3.2). Während der Vorwärtstransformation werden für ein solches Cluster $C_{semi} \in T_\Omega^{<q}$ die Ersatzmassen nur aus (semi-)aktiven Söhnen errechnet. Da diese Cluster für mehrere Prozesse als semi-aktiv gekennzeichnet sind, wer-

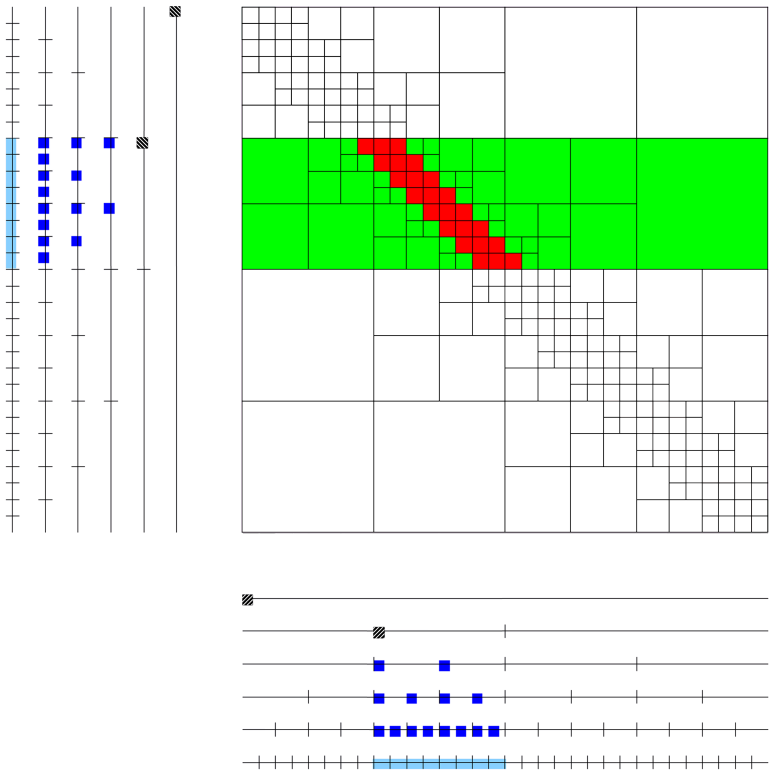


Abbildung 3.12. Für einen Prozess P mit $id_P = 1$ und die Anzahl Prozesse $p = 4$ ist hier die Zuständigkeit des Prozesses P für Blöcke eines Blockbaumes farbig dargestellt. (Quelle: Börm u. a. [2014])

den global betrachtet alle Söhne in der Vorwärtstransformation beachtet. Ist eines dieser Cluster Bestandteil eines zulässigen Blockes $b_0 = C_{semi} \times C$ oder $b_1 = C \times C_{semi}$, so können die Prozesse ihre Berechnungen untereinander austauschen. Somit werden die Definitionen der Matrizen V_τ und W_σ aus Gleichung 3.2 beziehungsweise Gleichung 3.3 lediglich auf mehrere Prozesse verteilt und die Summation bei Bedarf aus den Teilergebnissen gebildet.

Unter der Voraussetzung, dass alle notwendigen Informationen für jeden Prozess vorhanden sind, ist die einzige Anpassung der Auswertung der Kopplungsmatrizen, dass sich die Auswertung auf (semi-)aktive Targetcluster beschränkt. Auch die Rückwärtstransformation braucht sich lediglich auf (semi-)aktive Cluster beschränken.

Letztlich verteilt sich die Arbeit durch diesen Ansatz sehr natürlich auf die Prozesse. In Abbildung 3.12 ist dies veranschaulicht. Hier ist dies für einen Prozess P mit $id_P = 1$

und $p = 4$ dargestellt, welche Zuständigkeiten sich für die Blöcke des Blockbaumes aus der Aufteilung der Targetcluster ergeben. In den Clusterbäumen links und unten sind in hellblau die Clusterbasen, in dunkelblau die Transfermatrizen und in schwarz schraffiert die Transfermatrizen der semi-aktiven Cluster dargestellt.

3.3.2. Datenverteilung

Eine weitere grundlegende Frage ist, wo welche Daten vorhanden sein sollten. Ein möglicher Ansatz wäre, dass jeder Prozess eine Kopie aller Daten hat. Doch obwohl das Gravitationsproblem kein all zu speicherhungriges ist, würde der Hauptspeicher bei 32 Prozessen auf einem Knoten[4] sehr schnell knapp werden. Außerdem sei an dieser Stelle an die weitere Skalierbarkeit des Problems erinnert (vgl. Abschnitt 2.2). Daher müssen die Daten über die Prozesse verteilt werden.

Da wir im vorigen Kapitel bereits beschrieben haben, wie die Arbeit effektiv verteilt werden kann, ist es nur naheliegend die Daten auf die gleiche Weise zu verteilen. Jeder Prozess soll also genau die Teile der `bodies`-Struktur beinhalten, die zu seinen aktiven Clustern gehören. Dies entspricht den hellblau gekennzeichneten Teilen des Clusterbaumes links in Abbildung 3.12.

Weder bei reellen Daten noch bei unseren zufällig generierten Testdaten können wir davon ausgehen, dass diese nach Clustern sortiert vorliegen. Da ferner bei reellen Daten

Listing (3.12) Ausschnitt aus der parallelen Konstruktion des Clusterbaumes.

```
1  Cluster *constructClusterTree(bodies *b){
2    [...]
3    ////////////////////////////////////////////////////
4    // Sort the bodies; first locally, then globally.
5    preSort(root, 0);
6    new_bs = new_bodies(new_n);
7    alltoall_bodies(my_bs, send_count, send_displ,
8          new_bs, recv_count, recv_displ);
9    del_bodies(my_bs);
10   my_bs = new_bs;
11   //reset roots bodies*
12   root->bodies = my_bs;
13   root->n      = my_bs->n;
14
15   //clean up
16   [...]
17 }
```

[4] Dies entspricht den Spezifikationen der meisten Knoten des RZ-Clusters der Uni Kiel.

Listing (3.13) Diese Methode sortiert die lokalen `bodies` nach Prozesszugehörigkeit.

```
1  void preSort(Cluster *c, int depth){
2    if(depth == SPLIT_DEPTH){
3      c->active = ++split_count;
4
5      //get count and start index of data to send to process #split_count
6      send_count[split_count] = c->n;
7      send_displ[split_count] = split_count == 0 ? 0 :
8                 send_displ[split_count - 1] + send_count[split_count - 1];
9
10     //if the current knot is the active one: gather counts and displs from
           the other knots:
11     MPI_Gather(&send_count[split_count], 1, MPI_INT, recv_count, 1,
           MPI_INT, split_count, MPI_COMM_WORLD);
12     if(world.rank == c->active){
13       new_n = 0;
14       for(int i = 0; i < world.size; i++){
15    recv_displ[i] = new_n;
16    new_n        += recv_count[i];
17       }
18     }
19   }
20   [...}] // Sortierung und rekursiver Aufruf
21 }
```

davon auszugehen ist, dass diese aus Gründen des Speicherbedarfs auch verteilt eingelesen werden müssen, generiert in unserem Programm jeder Prozesse selbst zufällige Testdaten. Es wäre möglich die Daten so verteilt zu belassen, wie sie generiert beziehungsweise eingelesen wurden. Jedoch wäre dann während der Vorwärts- und Rückwärtstransformation eine große Menge an Kommunikation notwendig um diese entsprechend der Arbeitsverteilung durchzuführen. Daher ist es sinnvoll diese Daten einmal nach Prozesszugehörigkeit auszutauschen. Dies bedeutet zwar einen erheblichen Kommunikationsaufwand, dafür kann aber im Anschluss die Vorwärts- und Rückwärtstransformation komplett lokal und ohne weitere Kommunikation durchgeführt werden. Außerdem ist nicht davon auszugehen, dass viele Sonnen ihre bounding box während weniger Simulationsschritte verlassen. Daher kann dieser Kommunikationsaufwand als einmalig gewertet werden.

Diese Kommunikation wird während der Konstruktion des Clusterbaumes vorgenommen. Der zugehörige Quellcode ist in Listing 3.12 und Listing 3.13 aufgeführt.

Die Methode `preSort(...)` sortiert die `bodies`, indem sie den Clusterbaum bis zur Ebene `SPLIT_DEPTH` konstruiert. In den Arrays `send_count[]` und `send_displ[]` merkt sich jeder

Prozess, wie viele Elemente seiner `bodies` und ab welcher Position er an welchen anderen Prozess zu senden hat. Respektive werden in den Arrays `recv_count[]` und `recv_displ[]` die Anzahlen und Pufferpositionen für die zu empfangenden Daten gespeichert. Diese werden über die Methode `MPI_Gather(...)` (vgl. Abbildung 2.11) zwischen den Prozessen ausgetauscht. So bekommt jeder Prozess von den anderen mitgeteilt, wie viele Elemente er von ihnen gesendet bekommen wird. Die Gesamtgröße der künftigen `bodies`-Memberarrays wird in `new_n` gespeichert.

Der Datenaustausch findet über die Methode `MPI_Alltoallv(..)` (vgl. Abbildung 2.11 und Listing 2.3) statt. Diese führt mit Hilfe der zuvor konstruierten `count`- und `displ`-Arrays gerade eine Transformation der "Prozess-Daten-Matrix" aus, wodurch jeder Prozess jedem anderen die zugehörigen Daten schickt und respektive von diesem erhält. Danach werden noch einige Variablen aktualisiert.

Schließlich wird der Clusterbaum wie gehabt vollständig bis zur Ebene `MAX_DEPTH` konstruiert. Hierfür sind zwar die ausgetauschten Daten noch kaum notwendig, da die Konstruktion des Clusterbaumes durch mittige Teilung der Nicht-Blattcluster durchgeführt wird und zu diesem Zeitpunkt lediglich die Interpolationspunkte erstellt werden. Jedoch werden so gleich die korrekten Startindizes und Anzahlen in den Clustern gespeichert.

In Abschnitt 3.2.1 wurde der Programmparameter `pot` vorgestellt, der die Anzahl an Testsonnen bestimmt. Bei der parallelen Variante generiert jeder Prozess gerade $2^{\texttt{pot}}$ Sonnen. Die globale Menge Ω hat also $|\Omega| = 2^q 2^{\texttt{pot}} = 2^{q+\texttt{pot}}$ Elemente. Daher wird die maximale Baumtiefe $\texttt{MAX\_DEPTH} = q + \texttt{pot} - \ell$ gesetzt. Dadurch wird die durchschnittliche Blattgröße von 2^ℓ Elementen beibehalten. Im Folgenden wird die Zahl $n := |\Omega| = 2^{q+\texttt{pot}}$ für alle `bodies` auf allen Prozessen und die Zahl $m := 2^{\texttt{pot}}$ für die durchschnittliche Anzahl `bodies` pro Prozess.

3.3.3. Kommunikation

Als Resultat der Aufteilung der Daten müssen wir uns nun Gedanken machen, welche Daten für die Auswertungen der Matrizen zusätzlich zu den lokal vorhandenen gebraucht und damit kommuniziert werden müssen.

In Abbildung 3.15 sind diese Zusammenhänge in Bezug auf einen Prozess $P_1 \in \mathfrak{P}$ visualisiert. Die dunkel schraffierten Blöcke sind diejenigen, die Daten von P_1 benötigen: Entweder Berechnungen aus der Vorwärtstransformation für zulässige oder die Daten von Sonnen für unzulässige Blöcke. Der Prozess P_1 muss die Daten aus den zugehörigen Clustern also an die anderen Prozesse übermitteln. Umgekehrt benötigt P_1 die Daten zu den hell schraffierten Blöcken für seine Auswertung der Kopplungsmatrizen und Rückwärtstransformation. Lediglich die nicht schraffierten Blöcke benötigen keine Kommunikation, da alle Daten lokal verfügbar sind. Der Datenaustausch kann wie zuvor wieder durch eine Transformation der "Prozess-Daten-Matrix" und damit über `MPI_Alltoallv(...)` bewerkstelligt werden.

Diesmal liegen die Daten allerdings nicht sequentiell hintereinander, sondern sind über den Clusterbaum und die `bodies`- Arrays verteilt. Hinzu kommt, dass für zulässige und

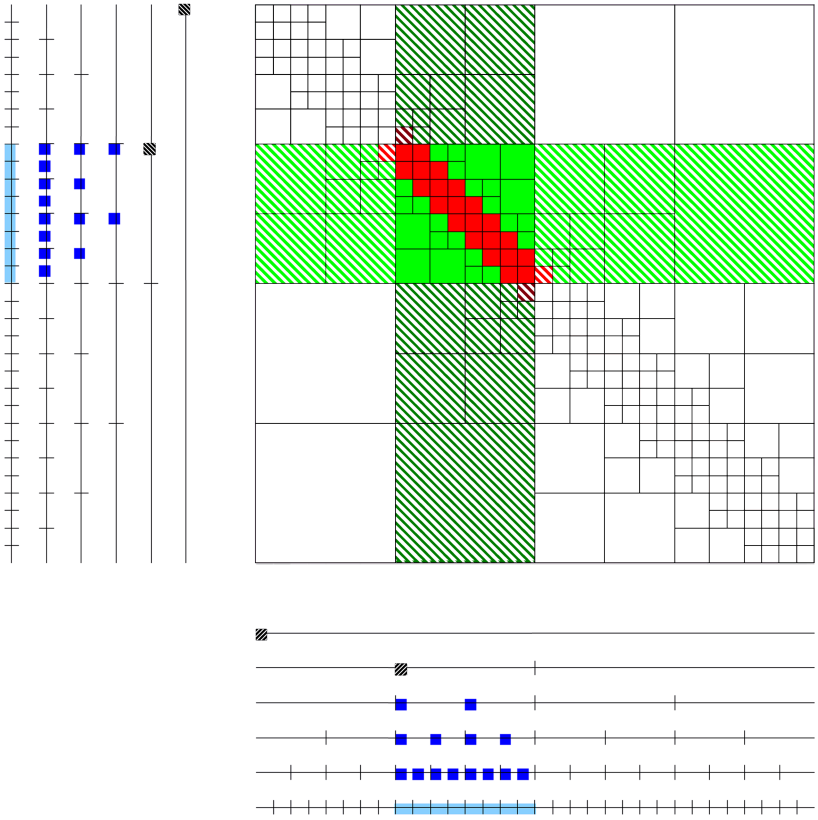


Abbildung 3.15. Hier sind einige Implikationen der Datenverteilung visualisiert. Die schraffierten Blöcke benötigen Datenaustausch mit Prozess P_1 (Quelle: Börm u. a. [2014])

unzulässige Blöcke unterschiedliche Daten ausgetauscht werden müssen. Für zulässige Blöcke brauchen lediglich die errechneten Ersatzmassen ausgetauscht werden, da die Interpolationspunkte bereits von jedem Prozess berechnet wurden. Dies ist also nur ein **`double`**-Array mit verhältnismäßig wenigen Daten. Für unzulässige Blöcke hingegen müssen alle zugehörigen Daten aus den Memberarrays `x`, `y`, `z` und `m` sowie die Anzahl der zum Cluster gehörenden Sonnen übertragen werden.

Diesmal müssen also für die Kommunikation manuell Sende- und Empfangspuffer angelegt werden. Der zugehörige Code ist in der Datei `eval.c` zu finden und wird im Folgenden erläutert.

Die Methode **`void`** `_prep_comm(Cluster *ct, Cluster *cs)` durchläuft ebenso wie die

in Listing 3.10 dargestellte Methode `_eval(...)` den impliziten Blockbaum und sucht nach zulässigen und unzulässigen Blöcken mit (semi-)aktivem Targetcluster. Pointer zu diesen Source- und Targetclustern werden in Arrays von Vektoren[5] gespeichert.[6] Dabei wird darauf geachtet, dass die Daten aus (semi-)aktiven Clustern, für die mehrere Prozesse zuständig sind, für alle diese Prozesse vermerkt werden.

Im Anschluss nutzt die Methode **`void`** `_prep_buffers()` diese gesammelten Daten, um die Sende- und Empfangspuffer zu allozieren und mit den zu sendenden Daten zu füllen. Sind diese Puffer bereit, kann die Kommunikation durchgeführt werden. Dazu wird die Methode **`void`** `_communicate()` aufgerufen. Dies führt mit den entsprechenden Puffern und den gesammelten Anzahl- und Positionsdaten insgesamt sechs Aufrufe von `MPI_Alltoallv(...)` durch: Eine für die Ersatzmassen der Cluster, und fünf für die benötigten `bodies`-Daten, da die einzelnen Koordinatenrichtungen, die Anzahlen pro Cluster und die Massen in einzelnen Arrays vorliegen.

Nachdem die Kommunikation vollendet wurde, müssen die Daten noch an die richtigen Stellen vermittelt werden. Die Methode **`void`** `_finalize_comm()` kopiert die empfangenen Ersatzmassen in die Cluster, in denen diese benötigt werden. Dabei werden die Ersatzmassen von Clustern, die sich aus den Ersatzmassen mehrerer Prozesse zusammensetzen, summiert (vgl. Gleichung 3.2 bzw. Gleichung 3.3). Die empfangenen `bodies`-Daten werden nicht kopiert, um Speicherplatz zu sparen. Die zugehörigen Cluster müssen aber angepasst werden. Der Startindex und die Anzahl wird entsprechend der Empfangspuffer gesetzt. Später können so die Daten direkt aus diesen Puffern abgerufen werden. Dies wäre für die Cluster zwar auch möglich, da hier die Ersatzmassen aber bereits alloziert wurden, belegen wir durch das Kopieren keinen zusätzlichen Speicher. Außerdem hat dies den Vorteil, dass wir so später nicht zu unterscheiden brauchen, wo welche Ersatzmassen zu finden sind.

Nun kann die Auswertung der Kopplungsmatrizen wie gehabt durch Aufrufen der Methode `_eval(...)` durchgeführt werden. Die einzigen Anpassungen sind das Auslassen von Blöcken mit inaktivem Targetcluster und die Unterscheidung, ob die `bodies`-Daten in der lokalen `bodies` Struct oder in den Empfangspuffern zu finden sind.

Abschließend werden die Daten in inaktiven Clustern durch die Methode **`void`** `_clear_inactive_clusters()` wieder auf 0 gesetzt, da diese beim nächsten Simulationsschritt nicht überschrieben werden würden. (Semi-)aktive Cluster werden während der Vorwärtstransformation neu berechnet und Cluster, für die genau ein anderer Prozess zuständig ist, werden nach der Kommunikation mit den neuen Daten überschrieben. Diese brauchen also nicht eigens geleert werden, um sicherzustellen, dass keine alten Daten verbleiben und zu fehlerhaften Berechnungen führen.

Sbschließender Satz?!

[5] Die Implementierung der vector-Struktur sowie zugehörige Methoden wurden von https://gist.github.com/-EmilHernvall/953968 übernommen und leicht modifiziert und erweitert.

[6] Mit einem Vektor ist in diesem Fall eine Struktur gemeint, die ein Array mit flexibler Länge darstellt.

Kapitel 4

Evaluierung

Börm und Bendoraityte [2008] haben gezeigt, dass deren paralleler Ansatz sehr nah an die optimale parallele Effizienz von $\mathcal{O}(\frac{nk}{p})$ herankommt, falls n deutlich größer als p ist. Dabei gilt: $n := |\Omega|$ ist die Anzahl Sonnen, $p := |\mathfrak{P}|$ ist die Anzahl Prozesse und $k := k_0^3$ ist die Anzahl Interpolationspunkte, mit dem Grad der eindimensionalen Lagrange-Polynomen k_0. Außerdem sei $m := \frac{n}{p}$ die durchschnittliche Anzahl Elemente pro Prozess.

Im folgenden gilt es diese Argumentation auf den vorliegenden Algorithmus zu übertragen und durch Daten aus praktischen Laufzeitmessungen zu unterstützen. Da unser Algorithmus mit impliziten Blockbäumen und $\mathcal{H}^2$-Matrizen arbeitet, wird die Konstruktion nicht weiter behandelt. Statt dessen konzentrieren wir uns auf eine Abschätzung für die Laufzeit.

4.1. Theoretische Abschätzung

Zunächst treffen wir auch für diesen Abschnitt eine Annahme, die bereits bei der Vorstellung des Algorithmus zielführend war:

Annahme 1.
Es existiert $q \in \mathbb{N}$ sodass für $p := |\mathfrak{P}|$ gilt:

$$p = 2^q$$

Zusätzlich zu der sichergestellten vollständigen Binärbaumstruktur benötigt der Clusterbaum für einige Abschätzungen weitere Eigenschaften:

Annahme 2.
Es existiert eine Konstante C_{st}, sodass für alle Teilbäume T_{sub} mit $\tau := root(T_{sub}) \in T_\Omega^{(q)}$ gilt

$$|T_{sub}|^1 \leqslant C_{st}\frac{n}{kp} \text{ und}$$
$$|\tau| \leqslant C_{st}\frac{n}{p}$$

Diese Annahme wurde durch die Wahl der Abbruchbedingung bei der Konstruktion des Clusterbaumes sichergestellt. Es gilt $|T_{sub}| = \frac{1}{2}\frac{n}{kp}$ (vgl. Gleichung 3.4), da jeder Prozess

[1] $|T_{sub}|$ bezeichnet die Anzahl Knoten im Baum

gerade einen Teilbaum konstruiert, wie er zuvor vom gesamten nicht-parallelisierten Algorithmus vorgenommen wurde (vgl. letzter Absatz in Abschnitt 3.3.2). Zwar lässt sich die Anzahl Elemente der Cluster $\tau \in T_\Omega^{(q)}$ nicht exakt angeben, da die Unterteilung in Sohncluster nicht nach Kardinalität vorgenommen wird, jedoch beträgt diese im Schnitt gerade $m = \frac{n}{p}$. Die Konstante C_{st} kann also als ≈ 1 angenommen werden.

Die Methode `_setup(Cluster *c, int depth)`, die in unserem Algorithmus die Verteilung der Cluster auf Prozesse vornimmt, gewährleistet folgende Eigenschaften:

Bemerkung 4.1. (Zuständigkeiten)
Für den verteilten Clusterbaum T_Ω gelten folgende Eigenschaften:

$$\text{Für } \tau \in T_\Omega^{(\geqslant q)} \text{ existiert genau ein } P \in \mathfrak{P} \text{ mit } \tau.\texttt{activ} = id_P. \tag{4.1}$$

Jeder Prozess $P \in \mathfrak{P}$ berechnet auf jeder Ebene $T_\Omega^{(q')}$, $q' \in \underline{q-1}_0$ genau eine Transfermatrix. (4.2)

Aus Gleichung 4.2 folgt direkt mit $q = \log_2 p$ (vgl. Abschnitt 3.3.1):

$$\text{Die Anzahl Transfermatrizen } E_\tau,\ \tau \in T_\Omega^{(\leqslant q)} \text{ pro Prozess beträgt } q. \tag{4.2'}$$

Da wir einen parallel arbeitenden Algorithmus haben ist es für die Abschätzung des Rechenaufwandes nicht ausreichend die Anzahl an Operationen zu zählen. Da die Prozesse kommunizieren müssen ist regelmäßig eine Synchronisation der Prozesse notwendig. So wird es vorkommen, dass ein Prozess P_i schneller seine Berechnungen durchführt als ein Prozess P_j und dann auf diesen warten muss bevor die Kommunikation stattfinden kann.

Daher verwenden wir einen ähnlichen Ansatz, wie er auch beim BSP-Modell [Valiant 1990] verwendet wurde: Die gesamte Berechnung wird in eine Sequenz von $s \in \mathbb{N}_0$ *Superschritten* eingeteilt, die jeweils unabhängig von den anderen Prozessen von einem Prozess durchgeführt werden können. Der i-te Superschritt startet simultan, sobald alle Prozesse den $(i-1)$-ten Superschritt abgeschlossen haben, $i \in \underline{s}$. Als Zeiteinheit verwenden wir die abstrahierte Einheit *Zyklus*. Ein Zyklus sei dabei lang genug um eine arithmetische Operation, einen Speicherzugriff oder eine Sende- oder Empfangsoperation für einen **`double`**-Wert durchzuführen.

Lemma 4.2. (Vorwärtstransformation)
Die parallele Vorwärtstransformation benötigt $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen.

Beweis: Die gesamte Vorwärtstransformation kann in einem Superschritt abgehandelt werden, da keinerlei Kommunikation notwendig ist.

Zunächst werden alle aktiven Cluster ausgewertet. Seien also $P \in \mathfrak{P}$ und T_Ω^P der aktive Teilbaum dieses Prozesses. Für ein Blattcluster $\sigma \in \mathcal{L}(T_\Omega^P)$ müssen für alle k Ersatzmassen $|\sigma|$ Berechnungen durchgeführt werden. Dann gilt mit Annahme 2, dass insgesamt

$$\sum_{\sigma \in \mathcal{L}(T_\Omega^P)} k|\sigma| = k \sum_{\sigma \in \mathcal{L}(T_\Omega^P)} |\sigma| \approx km$$

$$\leqslant C_{st}\frac{nk}{p}$$

Operationen ausgeführt werden müssen.

Für Nicht-Blattcluster $\sigma \in T_\Omega^P \backslash mathcal L(T_\Omega^P)$ sind für die k Ersatzmassen und beide Sohncluster wiederum k Auswertungen notwendig. Damit folgt, dass weitere

$$\sum_{\sigma\in T_\Omega^P\backslash\mathcal{L}(T_\Omega^P)}\ \sum_{\tilde\sigma\in sons(\sigma)} 2k^2 \leqslant \sum_{\tilde\sigma\in T_\Omega^P} 2k^2$$
$$= 2k^2|T_\Omega^P| \leqslant 2k^2 C_{st}\frac{n}{kp} = 2C_{st}\frac{nk}{p}$$

Operationen ausgeführt werden müssen.

Für die Ebenen $T_\Omega^{(<q)}$ werden nun jeweils die k Ersatzmassen aus den k Ersatzmassen eines Sohnclusters berechnet. Da es gerade $q = log_2(p)$ Ebenen gibt führt dies insgesamt zu obiger Abschätzung. □

Lemma 4.3. (Rückwärtstransformation)
Die parallele Rückwärtstransformation benötigt $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen.

Beweis: Der Beweis läuft analog zu dem von 4.2, die Berechnung verläuft lediglich von der Wurzel zu den Blättern statt wie bei der Vorwärtstransformation umgekehrt. □

Lemma 4.4. (Kopplungsmatrizen und Nahfeld) Die Auswertung der Kopplungs- und Nahfeldmatrizen benötigt $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen.

Beweis: Von Börm und Bendoraityte [2008] wurde bewiesen, dass der Speicherbedarf für Nah- und Fernfeldmatrizen $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ beträgt. Da der Algorithmus für jedes Element der Matrizen S_b und N_b nicht mehr als zwei Auswertungen vornimmt, liegt auch die benötigte Anzahl Zyklen in $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$. Auch in diesem Schritt ist keine Kommunikation notwendig, sodass auch diese Auswertung in einem Superschritt stattfinden kann. □

Bleibt noch die Kommunikation. Diese gliedert sich in zwei Superschritte: Das Vorbereiten mit dem anschließenden Senden der Daten und das Empfangen der Daten.

Lemma 4.5. (Kommunikation) Die Kommunikation benötigt $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen.

Beweis: Da die Vorbereitung der Kommunikation nach der selben Struktur wie die Auswertung der Kopplungs- und Nahfeldmatrizen arbeitet, fallen dafür ebenfalls $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen an. Da ebenso viele Elemente gesendet und empfangen werden müssen liegt der Gesamtaufwand für die Kommunikation in eben dieser Komplexitätsklasse. □

Für jeden Superschritt wurde also gezeigt, dass dieser $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen benötigt. Damit folgt für den gesamten Algorithmus:

Theorem 4.6. *Die Berechnung des Algorithmus benötigt $\mathcal{O}(\frac{nk}{p} + k^2 log_2 p)$ Zyklen.*

Börm und Bendoraityte [2008] merken noch an, dass für genügend große Probleme, also $n \geqslant k^2 log_2 p$, die optimale Komplexitätsordnung von $\mathcal{O}(\frac{nk}{p})$ erreicht wird.

4.2. Laufzeitmessung

Die theoretischen Argumente des letzten Abschnitts wollen wir nun durch die Messung von Laufzeitdaten des Algorithmus untermauern.

Um diese Daten zu sammeln, wurde das vorliegende auf dem NEC HPC-Linux-Cluster der CAU Kiel ausgeführt. Jeder Knoten dieses Clusters ist mit 192 GB Arbeitsspeicher und zwei Intel Xeon Gold 6130 bestückt, die einen Kerntakt von 2,1 GHz aufweisen. Verbunden sind die Knoten über EDR infiniband. Als Compiler wurden der Intel-C-Compiler 17.0.4 sowie der Intel-MPI-Compiler 17.0.4 verwendet.

Es wurden unterschiedliche Testreihen durchgeführt, bei denen entweder die Anzahl Prozesse oder die Anzahl Sonnen variiert wurde, um beide Ansätze unabhängig voneinander testen zu können. Alle Tests wurden mit Lagrange-Polynomen von Grad 3 in jeder Richtung approximiert, was zu $3^3 = 27$ Interpolationspunkten führt. Außerdem wurde jeder Testlauf 10 Mal wiederholt und immer vollständige Knoten im Clustersystem angefordert, um stabile Daten zu ermitteln.

Approximation

In den ersten beiden Testreihen wurde die Anzahl Sonnen pro Prozess bei konstanter Anzahl Prozesse variiert. Die Messergebnisse sind in Tabelle 4.1 sowie in Abbildung 4.1 aufgeführt.

Abbildung beziehungsweise Tabelle **(a)** bezieht sich jeweils auf die Testreihe mit einem Prozess, **(b)** auf die Testreihe mit 32 Prozessen. Trotz einiger Schwankungen, deren Herkunft nicht festgestellt werden konnte, ist der lineare Zuwachs gut zu erkennen. Bei der eingezeichneten Geraden handelt es sich um eine von gnuplot berechnete Regressionsgrade. Der Testlauf mit 32 Prozessen zeigt eine etwas stärkere Steigung, die vermutlich auf den erhöhten Kommunikationsaufwand zurückzuführen ist. Dennoch ist zu erkennen, dass die Approximation durch Interpolation und die Nutzung der $\mathcal{H}^2$-Struktur den theoretischen Berechnungen auch in reellen Anwendungen gerecht wird und die Komplexität von $\mathcal{O}(n^2)$ auf $\mathcal{O}(kn)$ senken kann. Da k wesentlich kleiner als n ist, ist damit viel gewonnen.

Um diese $\mathcal{H}^2$-Struktur nutzen zu können hatten wir die eigentliche Kernfunktion durch Interpolation approximiert. Zwar haben wir über eine Zulässigkeitsbedingung sichergestellt, dass die Ergebnisse "vernünftig" sind und Beweise angeführt, nach denen diese Approximation miot steigender Ordnung exponentiell gegen die eigentliche Kernfunktion konvergiert, aber wie ungenau wird die Berechnung durch die Approximation? Dazu wurden im kleineren Maßstab[2] einige Testläufe durchgeführt, die ebenfalls sehr stabile Resultate gezeigt haben. Die Ergebnisse sind in Tabelle 4.2 zu finden.

[2] Ausgeführt auf einem Computer mit 16 Gb Arbeitsspeicher und einem Intel Core i7-4710HQ.

Tabelle 4.1. In dieser Tabelle sind die Laufzeitmessungen der ersten beiden Testläufe aufgeführt.

#Elemente	Laufzeit [s]
2^{10}	0,008324
2^{11}	0,02569
2^{12}	0,04406
2^{13}	0,1348
2^{14}	0,503
2^{15}	0,6866
2^{16}	1,905
2^{17}	5,271
2^{18}	6,454
2^{19}	17,52
2^{20}	46,63
2^{21}	55,57
2^{22}	154,5
2^{23}	413,1
2^{24}	476
2^{25}	1270

(a) Dieser Testlauf wurde mit $p = 1$ durchgeführt und entspricht damit der nichtparallelen Variante.

#Elemente	Laufzeit [s]
2^{10}	0,02034
2^{11}	0,04135
2^{12}	0,1327
2^{13}	0,1744
2^{14}	0,4502
2^{15}	1,87
2^{16}	2,235
2^{17}	6,181
2^{18}	15,54
2^{19}	18,61
2^{20}	49,45
2^{21}	132,111
2^{22}	190,8

(b) Dieser Testlauf wurde mit $p = 32$ durchgeführt, genau ein Knoten des Rechenclusters auszulasten.

Wie an den Daten zu erkennen ist, nimmt der Fehler bei jeder Erhöhung der Interpolationsordnung k_0 um etwa Faktor 8 ab. Unsere Approximation konvergiert also wie vorgesehen exponentiell. Natürlich dauert die Berechnung für mehr Interpolationspunkte länger. Zu erkennen ist außerdem, dass bei diesem Testlauf bei $k_0 = 7$ eine Art Plateau erreicht wurde. Weitere Erhöhung hat keine Signifikante Änderung der Fehlerwerte mehr ergeben. Dies scheint mit den Ebenen des Clusterbaumes zusammenzuhängen. Wird die Interpolationsordnung gegenüber der Anzahl Sonnen groß genug, werden durch den vorliegenden Algorithmus zu wenig Ebenen im Clusterbaum erzeugt, sodass entsprechend wenig zulässige Blöcke gefunden werden können. Dies erklärt sowohl den plötzlichen Anstieg an Genauigkeit und die anschließende Stagnation.

Tabelle 4.2. Approximationsfehler und Laufzeit in Abhängigkeit zur Interpolationsordnung.

k	1	8	27	64	125	216	343
Fehler	$5,12e^{-2}$	$8,58e^{-3}$	$8,11e^{-4}$	$9,38e^{-5}$	$9,67e^{-6}$	$1,21e^{-6}$	$7,4e^{-15}$
Laufzeit [s]	0,0486	0,0967	0,259	0,454	0,864	1,09	1,3

Parallelität

Bereits durch die höhere Steigung der Graden in Abbildung 4.1.**(b)** gegenüber der in **(a)** lässt sich erahnen, dass durch die Kommunikation bei der Parallelisierung ein gewisser Overhead verursacht wird. Um dies genauer zu untersuchen wurde die Anzahl Sonnen pro Prozess[3] auf $m = 2^{20}$ fixiert und die Anzahl Prozesse variiert. Die Messdaten sind in Tabelle 4.3 und Abbildung 4.2 zu finden.

Die Daten unterliegen leider recht großen Schwankungen. Woher diese Schwankungen stammen muss an anderer Stelle genauer untersucht werden, da es den Umfang und die Zielsetzung dieser Arbeit übersteigt. Allerdings waren diese Ergebnisse in sich sehr stabil. Auch mehrfache Wiederholungen der einzelnen Testläufe haben immer bis auf wenige Zehntelsekunden die selbe Zeit benötigt. Trotzdem scheint mir, dass die errechnete Regressionsgrade die allgemeine Tendenz der Laufzeit einigermaßen akkurat wiedergibt.

Ein gewisser Anstieg der Laufzeit ist zu erwarten, da der Kommunikationsaufwand steigt. Insgesamt zeigt sich aber die fast optimale Ausnutzung der Parallelität.

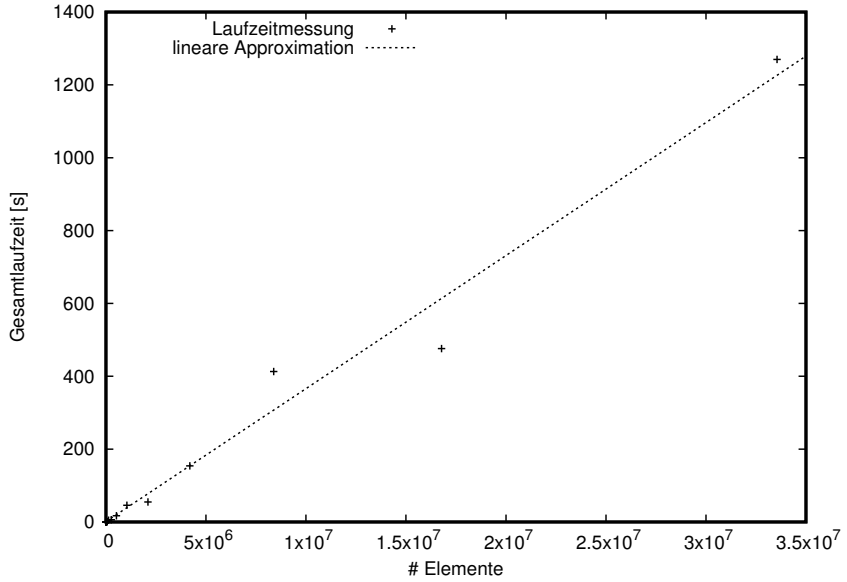
Ein interessanter Effekt kann an den mit Fußnoten *a* und *b* gekennzeichneten Testläufen beobachtet werden. Während bei *a* alle Prozesse auf einem Knoten gearbeitet haben, wurden diese für den Testlauf *b* auf zwei Knoten verteilt. Eine an sich naheliegende Vermutung wäre, dass die Testparameter für *a* bessere Laufzeiten ergeben müssten, da die Kommunikation auf einem Rechner und ohne Netzwerkbeteiligung stattfinden kann. In der Realität hat sich aber gezeigt, dass es gerade umgekehrt ist. Vermutlich behindern sich die vielen Speicherzugriffe bei dem Testlauf auf einem Knoten so stark, dass es effektiver ist einen Teil der Kommunikation über das Netzwerk zu führen. Auch dieser Effekt müsste aber an anderer Stelle genauer untersucht werden.

Tabelle 4.3. Tabelle der gemessenen Laufzeitdaten in Abhängigkeit zur Anzahl Prozesse.

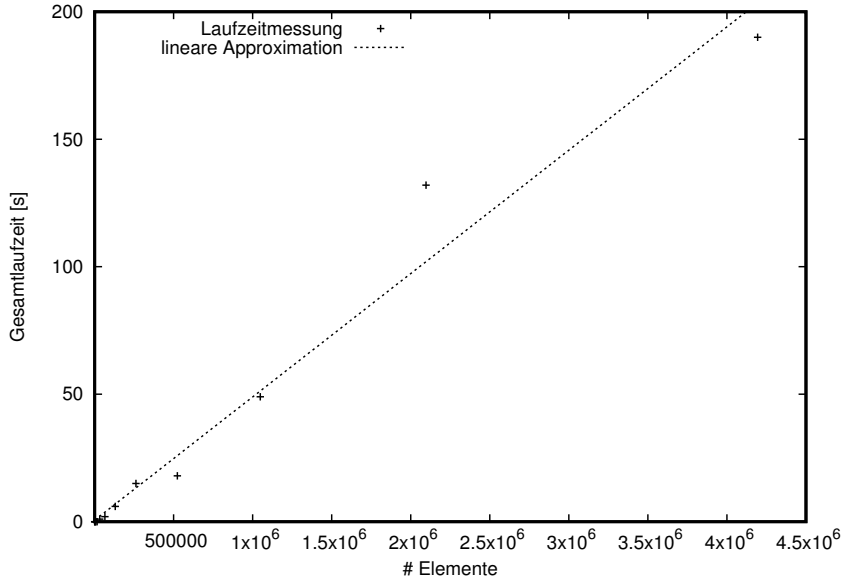
#Prozesse	Laufzeit [s]
1	46,54
2	27,73
4	36,94
8	51,94
16	31,29
32[a]	49,42
32[b]	40,82
64	64,96
128	42,27

[a] Die 32 Prozesse befanden sich auf einem Knoten des Clustersystems.
[b] Die 32 Prozesse verteilten sich zu je 16 auf zwei Knoten des Clustersystems.

[3] Mit einer Verdoppelung der Prozessanzahl geht also auch eine Verdopplung der Anzahl Sonnen einher.



(a) Dieser Testlauf wurde mit $p = 1$ durchgeführt und entspricht damit der nicht-parallelen Variante.



(b) Dieser Testlauf wurde mit $p = 32$ durchgeführt, genau ein Knoten des Rechenclusters auszulasten.

Abbildung 4.1. In dieser Abbildung ist der Zusammenhang der Laufzeit mit der Anzahl an Elementen pro Prozess dargestellt.

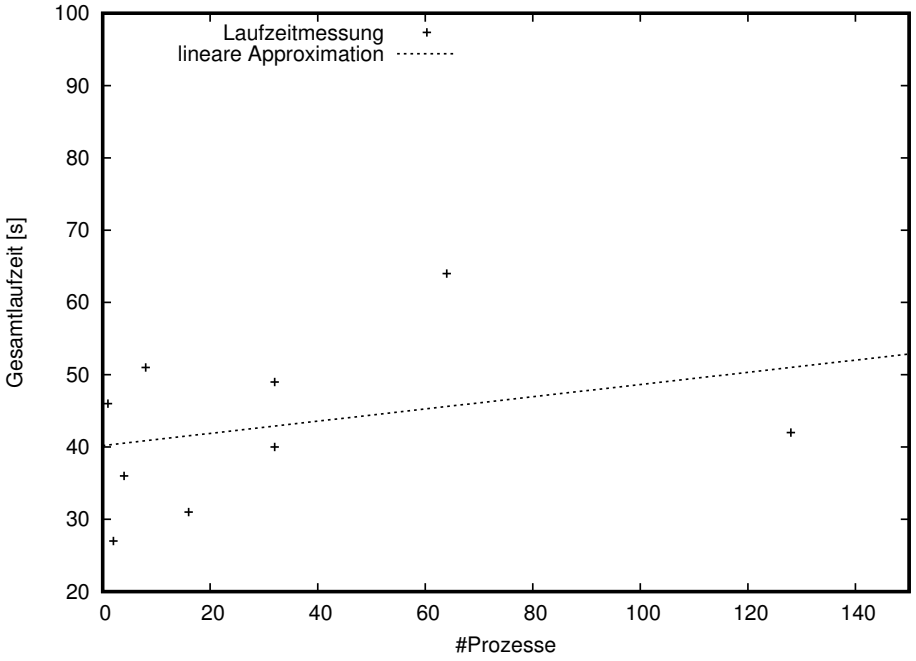


Abbildung 4.2. Dieses Diagramm zeigt den Anstieg der Laufzeit in Abhängigkeit zur Anzahl Prozesse.

Speicherbedarf der Kommunikation

Bei all den positiven Eigenschaften sei auch auf einen Nachteil des vorgestellten Algorithmus hingewiesen: Die verwendete Kommunikation benötigt viel unnötigen Speicher. Mein Hauptaugenmerk lag darauf, die Kommunikation so einfach wie möglich zu gestalten, um die Laufzeit möglichst gering zu halten. Da die `MPI_Alltoallv(..)`-Methode exakt das benötigte Kommunikationsmuster liefert und davon auszugehen ist, dass bei der Implementierung dieser Methode mehr Zeit und Know-How eingeflossen ist, als ich im Rahmen dieser Arbeit hätte investieren können, habe ich mich dazu entschieden diese Methode zu nutzen. Die dafür notwendigen Puffer waren recht schnell konstruiert und wurden von mir zunächst nicht weiter beachtet.

Es hat sich aber gezeigt, dass bei 32 Prozessen auf einem Knoten mit 2^{22} Sonnen pro Prozess bereits 182 GB Arbeitsspeicher benötigt werden. Das sind $5,68$ GB pro Prozess. Vergleicht man dies mit dem Testlauf mit nur einem Prozess, der lediglich 52.37 MB Hauptspeicher belegt, ist schnell einzusehen, dass die Kommunikation mit derart riesigem Pufferbedarf nicht tragbar ist.

Ganz ohne Puffer wird die Kommunikation nie auskommen. Es wäre aber möglich die Kommunikation durch nicht-blockierende Methoden derart zu verteilen, dass keinerlei Sendepuffer und lediglich Empfangspuffer für nicht-zulässige Blöcke benötigt würden. Dieser Ansatz ähnelt stärker dem von Börm und Bendoraityte [2008] vorgestellten Algorithmus, jedoch wäre weiterhin keine Kommunikation für Vorwärts- und Rückwärtstransformation notwendig. Eine genaue Ausarbeitung eines solchen Ansatzes muss leider aus Zeit- und Umfangsgründen an anderer Stelle fortgesetzt werden.

Kapitel 5

Fazit und Ausblick

5.1. Fazit

Wir haben gesehen, dass sich die Berechnung der Gravitationskräfte gut durch Interpolation approximieren und die zu berechnende vollbesetzte Matrix damit auch gut durch hierarchische Matrizen darstellen lässt. Dadurch kann dieses Problem, das an sich quadratische Komplexität besitzt, mit linearem Zeitaufwand gelöst werden.

Außerdem wurde gezeigt, dass sich der entstandene Algorithmus gut auf viele Rechner verteilen lässt. Hat hat der hier vorgestellte Ansatz einen zu hohen Speicherbedarf um Praxistauglich zu sein. Dennoch konnte gezeigt werden, dass durch die Parallelisierung des Algorithmus nahezu optimale Komplexität erreicht werden kann. Kann der Kommunikationsaufwand weiter reduziert werden, so besteht Grund zur Annahme, dass schwache Skalierbarkeit erreicht werden. Das heißt, dass der Algorithmus bei Verdopplung der Anzahl an Sonnen bei gleichzeitiger Verdopplung der Anzahl Prozesse konstante Laufzeit aufweist.

5.2. Ausblick

Trotzdem bleiben noch viele Optimierungsmöglichkeiten.

In dieser Arbeit haben wir einen Algorithmus erarbeitet, der sich auf das Message-Passing-Modell beschränkt. Das hat Vorteile, aber auch Nachteile. Beispielsweise wird durch das Message-Passing-Modell der Nutzen von Shared-Memory ignoriert. Das erhöht den Speicherbedarf, da jeder Prozess Kopien von Daten von anderen Prozessen benötigt. Das erhöht aber auch die Laufzeit, da auch Prozesse, die auf einem gemeinsamen Knoten arbeiten Daten explizit austauschen und kopieren müssen. Ein kombinierter Ansatz, der auf den einzelnen Knoten die Arbeit, aber nicht die Daten aufteilt und Messege-Passing zur Kommunikation zwischen Knoten, könnte die Laufzeit weiter senken.

Außerdem wurde bei dem Algorithmus darauf geachtet, dass die Datenstrukturen eine Vektorisierung möglich machen. Viele der Berechnungen in diesem Algorithmus eignen sich dafür als Vektoren verarbeitet zu werden. Dies würde zumindest diese Teile der Berechnung nochmals um einen Faktor beschleunigen.

Eine weitere Option, die in Zukunft zu prüfen wäre, ist, die Technologie von Grafikkarten für dieses Problem zu nutzen. Ob dies möglich und effektiv wäre muss aber an anderer

Stelle geklärt werden.

Es bleibt zu hoffen, dass sich viele Probleme in Zukunft mit einem derartigen Ansatz, wie er in dieser Arbeit vorgestellt wurde, effizient lösen lassen.

Literaturverzeichnis

[Akinci 2012] Ö. Akinci. MPI - Point to Point Communication. Juni 2012. URL: http://training.uhem.itu.edu.tr/docs/14haziranmpi/01.1_14Haz2012_MPI_P2P_III_Block-NonBlock-Comm_e2.pdf. (Siehe Seiten 30, 31)

[Barbic 2006] J. Barbic. Multi-core architectures. Mai 2006. URL: https://www.cs.cmu.edu/~fp/courses/15213-s06/lectures/27-multicore.pdf. (Siehe Seite 23)

[BIPM 1975] BIPM. Resolution 2 of the 15th CGPM. 1975. URL: https://www.bipm.org/en/CGPM/db/15/2/. (Siehe Seite 22)

[Börm 2007] S. Börm. Data-sparse approximation of non-local operators by $\mathcal{H}^2$-matrices. *Linear Algebra and its Applications* 422 (2007), Seiten 380–403. (Siehe Seite 20)

[Börm 2016] S. Börm. Wissenschaftliches Rechnen. Feb. 2016. URL: https://www.math.uni-kiel.de/scicom/de/lehre/wissenschaftliches-rechnen. (Siehe Seite 2)

[Börm 2017] S. Börm. Hochleistungsrechnen. Jan. 2017. URL: https://www.math.uni-kiel.de/scicom/de/lehre/hpc. (Siehe Seiten 22, 23 und 25)

[Börm 2018] S. Börm. Numerik nicht-lokaler Operatoren. Feb. 2018. URL: https://www.math.uni-kiel.de/scicom/de/lehre/nichtlokal. (Siehe Seiten 6, 9, 10, 12, 14, 17, 18 und 21)

[Börm und Bendoraityte 2008] S. Börm und J. Bendoraityte. Distributed $\mathcal{H}^2$-matrices for non-local operators. *Comput Visual Sci* 11 (2008), Seiten 237–249. (Siehe Seiten 52, 61, 63, 64 und 69)

[Börm u. a. 2014] S. Börm, S. Christophersen, J.Gördes, K.Reimer und D.Boysen. $\mathcal{H}^2$-Matrices. 2014. URL: https://www.math.uni-kiel.de/scicom/de/vortraege/dateien/2014/141205-h2matrices. (Siehe Seiten 15, 16, 18, 20, 54 und 58)

[Dongarra u. a. 1993] J. J. Dongarra, R. Hempel, A. J. Hey und D. W. Walker. A proposal for a user-level, message passing interface in a distributed memory environment. Technischer Bericht. Oak Ridge National Lab., TN (United States), 1993. (Siehe Seite 28)

[Flynn 1972] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers* 100.9 (1972), Seiten 948–960. (Siehe Seite 23)

[Foster 1995] I. Foster. 1.3 A Parallel Programming Model. 1995. URL: http://www.mcs.anl.gov/~itf/dbpp/text/node9.html. (Siehe Seite 27)

[Grasedyck 2001] L. Grasedyck. Theorie und Anwendungen hierarchischer Matrizen. Dissertation. Christian-Albrechts Universität Kiel, 2001. (Siehe Seiten 5, 6, 9, 11, 12, 14, 15)

[Hackbusch 1999] W. Hackbusch. A Sparse Matrix Arithmetic Based on $\mathcal{H}$-Matrices. Part I: Introduction to $\mathcal{H}$-Matrices. *Computing* 62.2 (1999), Seiten 89–108. (Siehe Seite 5)

[Hackbusch und Börm 2002] W. Hackbusch und S. Börm. $\mathcal{H}^2$-matrix approximation of integral operators by interpolation. *Applied Numerical Mathematics* 43 (2002), Seiten 129–143. (Siehe Seiten 9 und 17)

[IBM 2017] IBM. The message passing model. 2017. URL: https://www.ibm.com/support/knowledgecenter/en/SSF4ZA_9.1.3/pmpi_guide/mesage_passing_model.html. (Siehe Seite 27)

[Kendall 2017] W. Kendall. A Comprehensive MPI Tutorial Resource. 2017. URL: http://mpitutorial.com/. (Siehe Seiten 28, 32, 33)

[Körbler 2007] S. Körbler. Parallel Computing-Systemarchitekturen und Methoden der Programmierung (2007). (Siehe Seite 25)

[Mohr u. a. 2016] P. J. Mohr, D. B. Newell und B. N. Taylor. CODATA recommended values of the fundamental physical constants: 2014. *Journal of Physical and Chemical Reference Data* 45.4 (2016), Seite 043102. (Siehe Seite 1)

[MPI-Forum 2015] MPI-Forum. MPI: A Message-Passing Interface Standard Version 3.1. Technischer Bericht. Message Passing Interface Forum, Juni 2015. (Siehe Seiten 28–34)

[Newton 1833] I. Newton. Philosophiae naturalis principia mathematica. Band 1. G. Brookman, 1833. (Siehe Seite 1)

[Schwarz 2011] S. Schwarz. Intro to MPI. Feb. 2011. URL: http://www.dartmouth.edu/~rc/classes/intro_mpi/print_pages.shtml. (Siehe Seite 35)

[Stukeley 1936] W. Stukeley. Memoirs of Sir Isaac Newton's Life. Taylor und Francis, 1936. (Siehe Seite 1)

[Valiant 1990] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM* 33.8 (1990), Seiten 103–111. (Siehe Seite 62)

[Van Engelen 2017] R. van Engelen. Parallel Programming Models. 2017. URL: https://www.cs.fsu.edu/~engelen/courses/HPC/Models.pdf. (Siehe Seite 27)

[Walker 1992] D. W. Walker. Standards for message-passing in a distributed memory environment. Technischer Bericht. Oak Ridge National Lab., TN (United States), 1992. (Siehe Seite 28)

[Wehner u. a. 2008] M. Wehner, L. Oliker und J. Shalf. Towards ultra-high resolution models of climate and weather. *The International Journal of High Performance Computing Applications* 22.2 (2008), Seiten 149–165. (Siehe Seite 22)