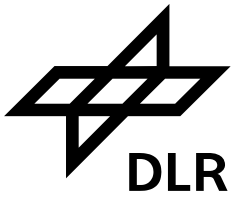


Entwicklung und Aufbau eines verteilten Systems zur Reduktion der Rechenzeit für Gradienten basierte Ersatzmodell Algorithmen

BACHELORARBEIT

für die Prüfung zum
Bachelor of Engineering

im Studiengang Informatik - Informationstechnik
an der Dualen Hochschule Baden-Württemberg Mannheim

von
Fabian Küppers

Abgabedatum: 19. September 2016

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	4182884, TINF13ITIN
Ausbildungsfirma	Deutsches Zentrum für Luft- und Raumfahrt e.V.
Betreuer der Ausbildungsfirma	Dr.-Ing Dipl.-Math. Christian Voß
Gutachter der Dualen Hochschule	Prof. Dr. Rainer Colgen

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich
die vorliegende Arbeit selbstständig und nur unter
Verwendung der angegebenen Quellen und
Hilfsmittel angefertigt habe.

Köln, der 6. September 2016

Kurzfassung

Zur Entwicklung neuer Konzepte für Turbomaschinen sind moderne Simulations- und Optimierungsverfahren, wie sie im Institut für Antriebstechnik des DLR verwendet werden, unerlässlich. Innerhalb dieser Optimierungsprozesse werden verschiedene so genannte Ersatzmodelle verwendet, die im Rahmen dieses Prozesses kontinuierlich verbessert bzw. trainiert werden. Für das Training solcher Ersatzmodelle sind allerdings diverse Matrizenoperationen notwendig, die je nach Problemgröße äußerst aufwendig zu berechnen sind. Werden für dieses Training zusätzlich Gradienteninformationen aus vorangegangen Rechnungen mit einbezogen, vergrößern sich die zugrunde liegenden Matrizen noch einmal erheblich, sodass die Rechenzeit des gesamten Optimierungsprozesses darunter leidet.

Es soll im Rahmen dieser Arbeit daher ermöglicht werden, einerseits diese Matrizenoperationen in geeignetem Maße auf einen leistungsstarken Rechner auszulagern, der mit einer GPU zu Rechenzwecken ausgestattet ist, und andererseits parallel ausführbare Teile des zugrunde liegenden Quellcodes des Ersatzmodelltrainings zu identifiziert und auf mehrere Recheneinheiten zu verteilen.
Die Auslagerung bzw. Verteilung von Rechenoperationen konnte durch Erweiterung und Optimierung von bestehender Software erreicht werden, indem wichtige Teile des Ersatzmodelltrainings mithilfe eines Clients als Schnittstelle an ein geeignetes Netzwerk mit beliebig vielen Servern angebunden wurde. Je nach Problemstellung sind so Verbindungen in einer 1:1 Beziehung zu nur einem Server und 1:n zu vielen Servern möglich. Die Ausführungsdauer eines Ersatzmodelltrainings und somit die des gesamten Optimierungsprozesses konnte so durch eine bessere Auslastung von bereits zur Verfügung stehenden Ressourcen deutlich gesenkt werden.

Abstract

The design of new turbomachinery needs modern simulation and optimization methods which are developed by the Institute of Propulsion Technology at the German Aerospace Center. Various so-called surrogate models are trained and continuously used for the selection of promising configurations within these optimization processes. However, various matrix operations are necessary for the exercise of such a training and the numerical effort increases dramatically with the problem size. If additional gradient information from preceded calculations is used for training, the size of the underlying matrix is increased further which results in a large impact to the whole optimization process.

Therefore it should be possible to outsource these matrix operations on the one hand in a proper way to a powerful server which is equipped with a GPU for computational purposes. On the other hand parallel executable parts of the source code of the surrogate model training should be identified and distributed to many processing units.
This was achieved by expanding and optimizing important parts of the source code using a client as an interface to a suitable network with many servers. Connections in a 1:1 relation to a single server and 1:n relation to many servers are enabled, depending on the current problem. The execution time of a surrogate model training and thus the execution time of the whole optimization process is significantly reduced. This is achieved by a better utilization of available resources.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

DLR	Deutsches Zentrum für Luft- und Raumfahrt e.V.
AT	Institut für Antriebstechnik
AT-FUV	Institut für Antriebstechnik - Abteilung Fan und Verdichter
CPU	Central Processing Unit
GPU	Graphics Processing Unit
UML	Unified Modeling Language
RPC	Remote Procedure Call
RMI	Remote Method Invocation
SIMD	Single Instruction, Multiple Data - Konzept zur Prozessorbeschleunigung
SSE	Streaming SIMD Extensions - Umsetzung des SIMD-Konzepts
Flop/s	Floating Point Operations per Second - Gleitkommaoperationen pro Sekunde
XML	Extensible Markup Language

1 Einleitung

Das Deutsche Zentrum für Luft- und Raumfahrt e.V. (DLR) ist das zentrale Forschungsinstitut der Bundesrepublik Deutschland und forscht im Auftrag der Bundesregierung im Bereich Luft- und Raumfahrt, Energie, Verkehr und Sicherheit. Es ist in insgesamt 33 verschiedene Institute mit verschiedenen Themengebieten an 16 verschiedenen Standorten unterteilt. Diese Arbeit wurde im Institut für Antriebstechnik (AT) in der Abteilung Fan und Verdichter (AT-FUV) am Standort Köln geschrieben. Im Folgenden wird eine kurze Übersicht über die Arbeitsweise und Strategien zur Entwicklung neuer Antriebstechniken und den damit einhergehenden Problemstellungen gegeben. Ferner wird das Thema und die Frage- bzw. Problemstellung dieser Arbeit sowie die aktuelle technische Ausgangslage erläutert und das weitere Vorgehen beschrieben.

1.1 Das Institut für Antriebstechnik - Abteilung Fan und Verdichter

Das Institut für Antriebstechnik (AT) forscht an Turbomaschinen in Form von Flugzeugtriebwerken bzw. Gasturbinen. Dabei werden verschiedene Aspekte wie z.B. Effizienz und Umwelteinflüsse untersucht. Das Institut unterteilt sich in verschiedene Abteilungen, wobei in jeder ein anderes Maschinenteil einer Turbomaschine untersucht wird. In der Abteilung Fan und Verdichter (AT-FUV) werden verschiedene Aspekte bzgl. der in Turbomaschinen eingesetzten Verdichter, die die Luft des so genannten Hauptstroms für die Verbrennung in der Brennkammer verdichten, und dem davor gelagerten Fan, der die Umgebungsluft ansaugt und beschleunigt, untersucht und optimiert. Der typische Aufbau eines Flugzeugantriebs ist zur Veranschaulichung in Abbildung 1 dargestellt. Werden Änderungen an einer Triebwerkskomponente vorgenommen, so müssen diese vor Serienreife zuerst untersucht und getestet werden. Da allerdings aus Zeit- und Kostengründen nicht jede kleinste Änderung praktisch getestet werden kann, wird hierzu moderne und größtenteils im DLR intern entwickelte Software zur Modellierung und Simulation verwendet. Erst nach erfolgreichem Durchlauf dieses Prozesses können Änderungen bzw. neu ausgelegte Maschinenteile in praktischen Messreihen

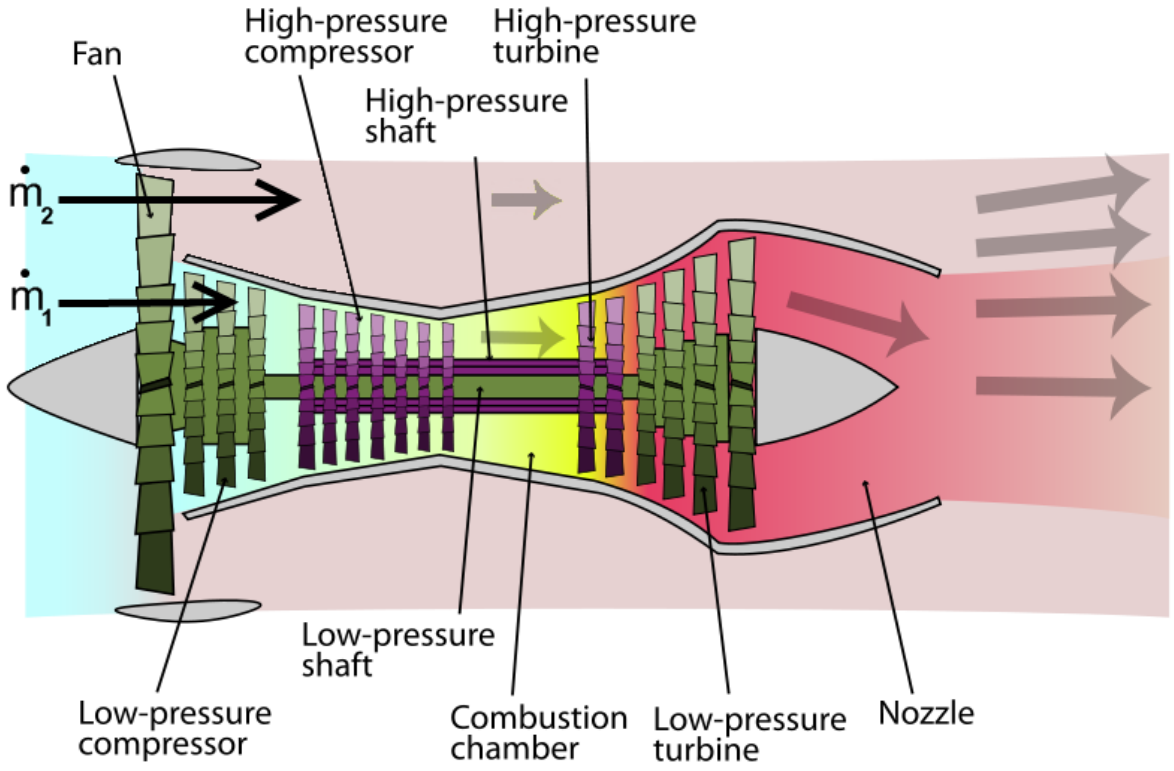


Abbildung 1: Typischer Aufbau eines Flugzeugtriebwerks [1]

getestet werden. Um noch bessere Resultate in kürzerer Zeit erzielen zu können, wird in einem ständigen Prozess Simulationssoftware weiterentwickelt und verbessert. Vor allem die Beschleunigung und Optimierung bestehenden Codes im Rahmen dieses Prozesses ist Kern der nachfolgend beschriebenen Problemstellung und letztendlich dieser Arbeit.

1.2 Problemstellung und Motivation

Die Abteilung AT-FUV verwendet zur aerodynamischen Optimierung von Turbomaschinen das Programm AutoOpti. Dieses wiederum besteht aus einer rechenintensiven Prozesskette, bestehend u.a. aus Geometrieerzeuger, Vernetzer, Strömungslöser, Strukturlöser und der Nachbereitung der Daten in einem Postprocessing. Damit können zu einem gegebenen Parametersatz (verschiedene physikalische Eigenschaften, Geometrien, etc.) Zielfunktionen bestimmt und ausgewertet werden. Diese Zielfunktionen wiederum repräsentieren weitere physikalische Eigenschaften, die nach Möglichkeit optimiert werden sollen (Wirkungsgrad, Lärm, etc.). Aus diesen Informationen werden mathematische Ersatzmodelle gebildet, anhand derer weitere Berechnungen durchgeführt werden können. Im Laufe des Optimierungsprozesses werden diese Ersatzmodelle durch neue Informationen immer weiter verbessert bzw. trainiert. Die Anzahl der Eingabeparameter und damit auch der Suchraum sind dabei in der Regel sehr groß. Aeromechanische Bewertungen mittels CFD-Verfahren (computational fluid dynamics) können daher meist nur mit einem enormen numerischem Aufwand durchgeführt werden und sogar auf einem Rechencluster mit zahlreichen Prozessoren viele Wochen benötigen.
Der Einsatz von adjungierten Lösern bietet dazu eine deutlich schnellere Alternative. Dieser bietet die Möglichkeit, dass neben den eigentlichen Zielfunktionalen auch deren Gradient mit relativ niedrigem numerischem Mehraufwand ermittelt werden kann (für weitergehende Informationen hierzu vgl. [2]. Dort wird das Konzept der adjungierten Strömungslöser näher erklärt). Diese zusätzlichen Informationen können dem Ersatzmodell-Training zur Verfügung gestellt werden, wodurch dieses präziser arbeiten kann aber auch in den Berechnungen deutlich aufwendiger wird - dies betrifft vor allem die in dem Ersatzmodell-Training eingesetzten und notwendigen Kriging-Modelle, welche im Institut entwickelt worden sind (Kriging: geostatisches Verfahren zur Interpolation von Zwischenwerten).
Grund hierfür sind die innerhalb dieser Modelle verwendeten und notwendigen Kovarianzmatrizen. Im Zuge der Berechnung der Kriging-Modelle müssen diese Matrizen nach dem Cholesky-Verfahren zerlegt und teilweise auch invertiert werden. Neuere Verfahren, die parallel zu dieser Arbeit entwickelt wurden, ersetzen die Notwendigkeit einer Matrixinvertierung durch eine so genannte Rückwärtsdifferenzierung des Cholesky-Verfahrens.
Die Größe dieser Matrizen K ergibt sich bei einem herkömmlichen Kriging-Modell (auch Ordinary Kriging) ohne die Verwendung von Gradienten aus der Anzahl der bisher erzeugten

Ersatzmodell-Trainingsdaten N:

$$K \in \mathbb{R}^{N^2} \tag{1.1}$$

Werden allerdings Gradienten beim Ersatzmodell-Training für so genannte Gradient-Enhanced-Kriging-Ersatzmodelle mit berücksichtigt, so vergrößert sich die Matrixgröße enorm. Werden Optimierungen mit n Parametern und für alle bisher erzeugten Trainingsdaten alle partiellen Ableitungen berücksichtigt, ergibt sich eine Kovarianzmatrix der folgenden Größe:

$$K \in \mathbb{R}^{N^2*(n+1)^2} \tag{1.2}$$

Bei großen Optimierungsproblemen und zahlreichen Trainingsdaten ergibt sich daraus eine enorme Matrixgröße (Größen $\sim 20.000^2$ mit 8 Byte Gleitkommawerten können durchaus vorkommen).
Neben den Gradient-Enhanced-Kriging-Ersatzmodellen gibt es noch Multifidelity-Kriging-Ersatzmodelle. Deren Training basiert auf kleineren Matrizen als das des Gradient-Enhanced-Krigings. Allerdings werden für die Berechnungen erheblich mehr Variablen verwendet. Dies hat zahlreiche kleinere Operationen zur Folge, die für jede Variable durchgeführt werden.
Die für das Training notwendigen Berechnungen werden somit bei beiden Ausprägungen des Ersatzmodells äußerst aufwendig und können den gesamten Optimierungsprozess ausbremsen. Es wäre zukünftig aber wünschenswert, dass auch größere Problemstellungen (große Matrizen bzw. viele Variablen) in annehmbarer Zeit berechnet werden können. Die Herangehensweise bzw. Strategie zur Reduktion der Rechenzeit eines Trainings muss allerdings auf das jeweilige Kriging-Verfahren abgestimmt werden, da bei diesen verschiedene Aspekte im Vordergrund stehen.

1.3 Fragestellung, Anforderungen und Ziele

Das oben beschriebene Kriging-Verfahren ist Teil des Optimierungsprozesses und kann je nach aktueller Problemstellung unterschiedliche Ersatzmodelle verwenden. Allerdings ist dieses Verfahren bei großen Matrizen sehr rechen- und zeitintensiv. Daher gilt es zu untersuchen, ob und in welchem Umfang sich eine mögliche Auslagerung der Rechnungen auf einen oder mehrere entfernt liegende Prozessoren auf die Ausführungszeit des Kriging auswirkt.

Da aber auch ein Training, welches kleinere Matrizen aber dafür viele Variablen verwendet, sehr aufwendig zu berechnen ist, soll untersucht werden, ob einige Berechnungen parallel und in einem nächsten Schritt in einem Netzwerk verteilt durchgeführt werden können.
Darüber hinaus ist es notwendig, dass solch ein System stabil arbeitet und auf eventuelle Ausfälle von Netzwerkteilnehmern reagieren kann. Dies muss bei der Implementierung berücksichtigt werden. Es soll auf bestehenden Verfahren und Programmen aufgebaut und der Frage nachgegangen werden, in welchem Umfang sich diese weiter optimieren und besser beschleunigen lassen.

1.4 Ausgangssituation

Die Abteilung Fan und Verdichter im Institut für Antriebstechnik benutzt für rechenintensive Operationen ein externes Rechencluster mit zahlreichen Knoten. Während die Verbindungsgeschwindigkeiten der Knoten untereinander äußerst hoch ist (Anbindung über InfiniBand), können für den Kommunikationsweg zwischen Cluster und Abteilungsnetzwerk vergleichsweise nur geringe Netzwerkgeschwindigkeiten erreicht werden. Neben dem Rechencluster wird aber auch lokal derzeit aufgerüstet: in [3] wird der Einsatz einer leistungsstarken GPU zu Rechenzwecken beschrieben und in der Abteilung AT-FUV derzeit in Betracht gezogen. Die Eingliederung solch einer GPU im Zusammenspiel mit dem Rechencluster soll daher ermöglicht werden und ein Teil dieser Arbeit sein.
Die vorliegende Arbeit baut auf dem Quellcode des im DLR intern entwickelten Krigings auf (vgl. [4]) und versucht dieses durch geeignete Maßnahmen weiter zu optimieren. Der Code ist in C++ geschrieben, um einerseits systemnah performant arbeiten und andererseits die Vorteile objektorientierter Programmierung genießen zu können. Es existieren neben dem eigentlichen Quellcode bereits mehrere Klassen zur Repräsentation von Matrizen, die von einer Superklasse `Matrix` erben und sämtliche Funktionen zu Matrizenoperationen bereitstellen (vgl. [3] S.9 f. bzw. [4] S.22 ff.):

- `OpenMPMatrix`: Implementierung der Matrizenfunktionen unter Verwendung der Parallelisierungsbibliothek „OpenMP“
- `CudaMatrix`: diese Klasse bindet die Bibliothek „CUDA“ von Nvidia® ein, um für Rechenoperationen - falls vorhanden - die GPU verwenden zu können

- `MKLMatrix`: analog zu der Klasse `CudaMatrix` werden auch hier externe Funktionen eingebunden, nämlich die Math Kernel Library von Intel®. Diese stellt hoch optimierte Methoden für diverse mathematische Probleme bereit, welche auf Intel® CPUs zugeschnitten sind und eine bestmögliche Auslastung erzielen

Darüber hinaus existiert eine weitere abstrakte Klasse `SaveableOnServer`, welche noch über der Superklasse `Matrix` angesiedelt ist und die Methoden für die Verwendung von Objekten in einem Netzwerk als Interface bereitstellen soll. Die Methoden dieses Interfaces müssen jedoch noch weiter ausgebaut und getestet werden.
Die Beziehungen der einzelnen Klassen untereinander sind in Abbildung 2 als UML-Diagramm dargestellt.
Für nachrichtenbasierte Kommunikation mittels der Messaging-Middleware „ZeroMQ“ (weitere Ausführungen dazu vgl. Abschnitt 3.2.1) gibt es in der Abteilung Fan und Verdichter bereits erste Erfahrungswerte. Zur Verwendung dieser Middleware in Kommunikationssystemen hat sich dabei konzeptionell folgendes Klassenmodell durchgesetzt:

- `ZMQConnection`: abstrahiert die Methodenaufrufe von ZeroMQ und stellt eigene Funktionen für eine Client/Server-Verbindung bereit, um Nachrichten zu versenden/empfangen
- `ZMQServer`: übernimmt die Rolle des Servers und ist für Datenhaltung und -verwaltung zuständig
- `ZMQClient`: übernimmt die Rolle des Clients; auch hier steht die Datenverwaltung im Fokus
- `MatrixServerFunctions`: serverseitige Implementierung von Berechnungsaufgaben und ggf. Anbindung an weitere Software
- `MatrixClientFunctions`: Verwaltung der Verteilungslogik von Rechenaufgaben an angeschlossene Server mit anschließender Zusammenführung der Ergebnisse

Diese Kommunikationsstruktur sowie die Implementierungen der Matrizenklassen sollen nun im Zuge dieser Arbeit um die benötigten Funktionalitäten erweitert und deutlich fehlertoleranter gestaltet werden.

[1] in Anlehnung an: [3], S. 9 f. bzw. [4], S. 22 ff.

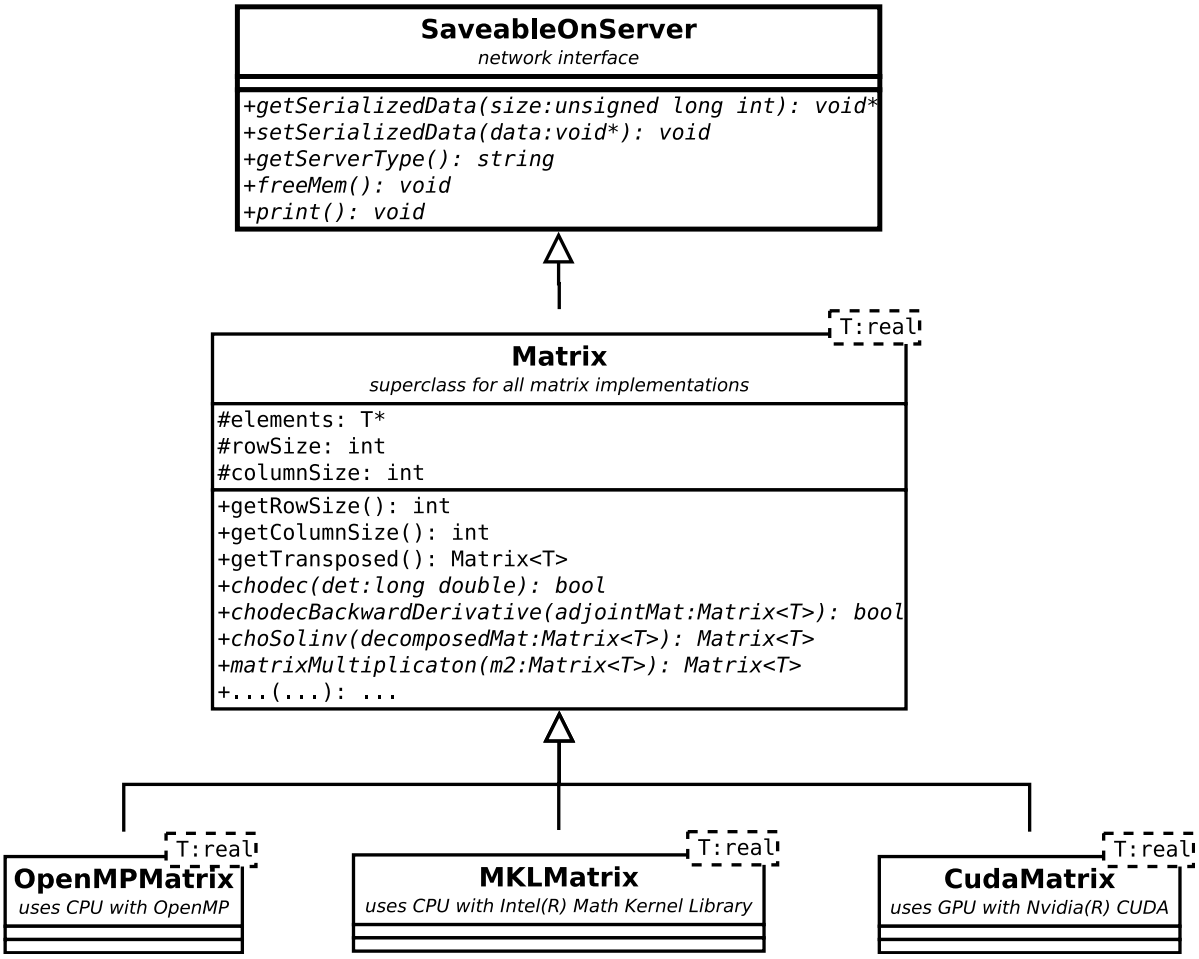


Abbildung 2: UML-Diagramm der virtuellen Matrixklasse mit drei Implementierungen (OpenMPMatrix, CudaMatrix, MKLMatrix) sowie der darüber gelagerten virtuellen Netzwerkklasse als Netzwerkinterface[1]

1.5 Aufbau der Arbeit

In Kapitel 2 wird näher auf die für diese Arbeit relevanten mathematischen Grundlagen des Kriging-Verfahrens eingegangen. Anschließend beschreibt Kapitel 3 die Grundlagen von verteilten Programmen und bezieht diese auf die aktuelle Problemstellung. Es werden dabei verschiedene Netzwerkaspekte und -topologien auf Vor- und Nachteile untersucht und geeignete Verteilungen von Rechenoperationen erarbeitet.
Das darauf folgende Kapitel 4 beschreibt die Implementierung und Lösung der Aufgabenstellung anhand der zuvor beschriebenen Grundlagen, während sich Kapitel 5 hauptsächlich der Überprüfung der erzielten Ergebnisse durch Geschwindigkeitsvergleiche (vorher/nachher) widmet. Schlussendlich werden in Kapitel 6 die im Rahmen dieser Arbeit erzielten Ergebnisse zusammengefasst und hinsichtlich der in Abschnitt 1.3 vorliegenden Fragestellung bewertet.

2 Mathematischer Hintergrund

Im Folgenden wird auf den mathematischen Hintergrund des Kriging-Verfahrens näher eingegangen. Es werden allerdings nur die für diese Arbeit relevanten Zusammenhänge beschrieben. Für detaillierte Informationen zu dem gesamten Verfahren vgl. [4]. Die nachfolgend beschriebenen mathematischen Zusammenhänge sowie die Beschreibung des Kriging-Verfahrens sind dort von S. 14., S. 27 und S. 40 ff. entnommen worden.

Der Optimierungsprozess besteht aus vielen verschiedenen Teilen, die auch für das Kriging relevant sind. Alle Eigenschaften und Rahmenparameter eines Verdichters werden in dem Prozess über freie Variablen gesteuert, die bestimmte Maschineneigenschaften wie z.B. Position und Winkel von Verdichterschaufeln repräsentieren. In diesem Prozess wird versucht, diese freien Variablen zu einem bestimmten Ziel hin zu verändern und zu optimieren. Dazu existieren die Zielfunktionen, die ihrerseits wieder für verschiedene Eigenschaften stehen (bspw. Lärmbelastung, Wirkungsgrad, etc.) und optimiert werden sollen. Ein Member besteht aus einem Satz an freien Variablen und dazugehörigen Zielfunktionen.
Mit den bereits bestehenden Membern kann nun ein Ersatzmodell trainiert werden, anhand dessen anschließend Vorhersagen zur Erzeugung neuer Member getroffen werden können. Im Vergleich zur Vorhersage neuer Member anhand des Ersatzmodells ist dessen Training mit dem Kriging-Verfahren sehr rechenintensiv und bei ausreichend großer Problemstellung damit auch sehr langsam.
Sämtliche rechenintensive Operationen des Ersatzmodelltrainings werden mit symmetrischen und positiv definiten Kovarianzmatrizen R durchgeführt, für die also gilt:

$$\vec{v} * R * \vec{v} > 0 \; \forall \; \vec{v} \neq 0 \tag{2.1}$$

$$R = R^T \tag{2.2}$$

wobei alle Eigenwerte der Matrix > 0 sind. Die Größe dieser Kovarianzmatrizen skaliert mit der Anzahl an bisher erzeugten Membern, sodass der Aufwand für ein Training im Laufe der Prozesskette weiter steigt.

Die Funktionen des Krigings arbeiten mit so genannten Hyperparametern, deren Anzahl in etwa mit der Anzahl an freien Variablen skaliert (kann bei verschiedenen Kriging-Verfahren abweichen). Diese sind zu Beginn des Krigings unbekannt und werden meist zufällig initialisiert. Im Laufe des Trainings werden diese Hyperparameter immer weiter verbessert, um so optimale Schätzwerte für das Ersatzmodell zu finden. Dies wird durch die Maximum Likelihood Methode erreicht (nähere Erläuterung vgl. [4], S.40 f.). In mehreren Iteration wird dabei stets folgender Prozess durchlaufen:

1. Mithilfe einer Wahrscheinlichkeits- oder auch Dichtefunktion wird aus der Kovarianzmatrix der Likelihood N (nachfolgend auch Dichte) bestimmt. Im Rahmen dieses Prozesses wird eine Cholesky-Zerlegung durchgeführt. Diese Operation ist vor allem bei großen Matrizen sehr rechenintensiv.
2. Anschließend wird für alle Hyperparameter θ die partiellen Ableitungen der Dichte nach dem Schema $\frac{\partial N}{\partial \theta_i}$ bestimmt. Dies kann für jeden Hyperparameter unabhängig berechnet werden. Dazu muss zuerst die Inverse der Cholesky zerlegten Matrix einmal bestimmt werden. Ein neueres Verfahren ersetzt die Notwendigkeit einer inversen Matrix durch die einer adjungierten Matrix der Cholesky zerlegten Kovarianzmatrix, mit der dann die partiellen Ableitungen bestimmt werden. Die Berechnung der adjungierten Matrix erfolgt anhand einer Rückwärtsdifferenzierung des Cholesky-Verfahrens, in welchem die ursprüngliche Cholesky-Zerlegung unter Zuhilfenahme von Gradienteninformationen rückwärts durchlaufen wird. Die theoretische Grundlage hierzu wird in [5] bzw.[6] beschrieben. Auch diese Operation muss in diesem Schritt nur ein einziges mal ausgeführt werden. Da bei der Bestimmung aller partiellen Ableitungen noch weitere Matrizenoperationen notwendig sind, die dann allerdings für jeden Hyperparameter durchgeführt werden müssen, ist dieser Schritt vor allem bei Problemstellungen mit zahlreichen Hyperparametern rechenlastig.
3. Im letzten Schritt einer Iteration wird nun mithilfe der Gradienteninformationen der Hyperparameter versucht, den berechneten Likelihood mittels diverser Verfahren zu minimieren. Dies soll an dieser Stelle jedoch nicht weiter betrachtet werden.

Für die Zerlegung einer Matrix R nach dem Cholesky-Verfahren muss diese symmetrisch und positiv definit sein. Dies ist bei der Kovarianzmatrix stets der Fall. Die zerlegte Matrix L ist eine untere Dreiecksmatrix (alle Werte oberhalb der Hauptdiagonalen sind 0) und der

Berechnung liegen folgende Gleichungen zugrunde:

$$L_{ij} = \begin{cases} 0 & j > i \\ \sqrt{R_{jj} - \sum_{k=0}^{j-1} L_{jk}^2} & j = i \\ \frac{1}{L_{jj}} * (R_{ij} - \sum_{k=0}^{j-1} L_{jk} L_{ik}) & j < i \end{cases} \tag{2.3}$$

(vgl. [7], S.100)

Das Produkt der zerlegten Matrix L mit deren Transponierten ergibt wiederum die Ursprungsmatrix R:

$$R = L * L^T \tag{2.4}$$

Die Komplexität dieser Operation beträgt bei n Zeilen bzw. Spalten etwa $\frac{1}{6}n^3$. Anhand der Gleichung 2.3 lässt sich erkennen, dass jeder Berechnungsschritt auf vorherigen aufbaut, sodass diese Operation nur sequentiell erfolgen kann. Dies gilt auch für die Berechnung der adjungierten Matrix durch Rückwärtsdifferenzierung der Cholesky-Zerlegung.
Die zerlegte Matrix kann nun verwendet werden, um durch Vorwärts- und Rückwärtseinsetzen lineare Gleichungssysteme der Form $R * \vec{x} = \vec{b}$ zu lösen. Wird nun der Parameter $\vec{x}$ durch R^{-1} und $\vec{b}$ durch die Einheitsmatrix E ersetzt, so lässt sich mit der Zerlegung die Inverse R^{-1} der Matrix R anhand eines Hilfsvektors $\vec{y}$ berechnen:

$$L * \vec{y} = \vec{b} \text{ (Vorwärtseinsetzen)} \tag{2.5}$$

$$L^T * \vec{x} = \vec{y} \text{ (Rückwärtseinsetzen)} \tag{2.6}$$

Die Invertierung der zerlegten Matrix ist im Gegensatz zur Cholesky-Zerlegung und deren Rückwärtsdifferenzierung für jede Spalte einzeln durchführbar und somit parallelisierbar, sie benötigt allerdings etwa n^3 Rechenoperationen, Die Bestimmung einer adjungierten Matrix durch Rückwärtsdifferenzierung benötigt lediglich nur etwa $\frac{1}{3}n^3$ Rechenoperationen (vgl. [6]). Es wurde daher grundsätzlich entschieden, auf eine Matrixinvertierung trotz Parallelisierbarkeit vollständig zu verzichten, sie wird nachfolgend im Rahmen dieser Arbeit nicht weiter

berücksichtigt. Nichtsdestotrotz ist auch eine Rückwärtsdifferenzierung noch immer sehr rechenlastig.

3 Grundlagen und Möglichkeiten zur Prozessverteilung

Es gibt zahlreiche Möglichkeiten, um ein Problem wie z.B. eine Rechenoperation auf mehreren Prozessoren bzw. Rechnern zu bearbeiten und damit zu beschleunigen. Es müssen allerdings einige wichtige Aspekte (z.B. Synchronisation) beachtet werden. Jedes Problem benötigt dabei eine exakt auf dieses zugeschnittene Lösung, insbesondere bei der Arbeit mit bereits bestehender Software. Die nachfolgenden Abschnitte geben einen Einblick in die Thematik zur Prozessverteilung einerseits lokal und andererseits in einem Netzwerk. Außerdem wird mit ZeroMQ eine Messaging-Middleware vorgestellt, die für die Kommunikation im Netzwerk verwendet werden soll. Es werden darüber hinaus verschiedene Netzwerktopologien betrachtet und geeignete Verteilungsalgorithmen erarbeitet.

3.1 Lokale Prozessverteilung

Der erste Schritt, um Programme schneller und effizienter arbeiten zu lassen, besteht in der Verteilung einzelner Operationen auf der lokalen Maschine. In der Programmiersprache C/C++ ist dies auf mehrere Arten umsetzbar:

- Posix-Thread (unter Unix-Betriebssystemen), auch Pthread genannt: mithilfe dieser API können einzelne Funktionen dediziert in einem neuen Thread aufgerufen werden. Die Verwaltung, Steuerung und Synchronisation dieser Threads obliegt hierbei dem Programmierer
- Multithreading mit der Bibliothek Boost (unter C++, vgl. [8]): ähnlich zu Pthread können auch hier einzelne Threads manuell gestartet werden. Es gibt hier allerdings mehr Möglichkeiten zur Kontrolle und Steuerung. Zudem ist die Handhabung deutlich einfacher im Vergleich zu Pthreads.

- Parallelisierung einzelner Code-Segmente mit OpenMP: im Gegensatz zu den beiden oben erwähnten Modellen lassen sich mit OpenMP nur einzelne Programmabschnitte innerhalb einer Funktion parallelisieren. Dies erfolgt durch geeignete Präprozessordirektiven (Befehle, die vor der Kompilierung des Codes ausgeführt werden), die vor dem zu parallelisierendem Abschnitt eingefügt werden. Dies ist vor allem für Programmschleifen mit unabhängigen Iterationen interessant. Die eigentliche Parallelisierung und Erzeugung von Threads wird von der API autonom übernommen

Mithilfe oben genannter Konzepte wurde ein erheblicher Teil des Krigings bereits auf mehrere Threads verteilt. Die Grenzen dieser Parallelisierung und der damit erreichbaren möglichen Beschleunigung liegen allerdings in der Rechenkapazität der Prozessoren, der Anzahl an Prozessoren selbst und dem beschränkten parallelisierbaren Anteil eines Programms. Dies wird in dem Gesetz von Amdahl deutlich. Nach Amdahl wird der Geschwindigkeitszuwachs eines Programms bei Verteilung auf mehrere Prozessoren/Threads vor allem durch den enthaltenen sequentiellen Anteil begrenzt. Zudem steigt die benötigte Zeit zur Verwaltung und Synchronisation der einzelnen Prozessoren mit steigender Anzahl. Dies lässt sich anhand folgender Gleichung darstellen (vgl. hierzu [9]):

$$\eta_s = \frac{T}{t_s + t_{\text{sync}} + \frac{t_p}{n_p}} \leq \frac{T}{t_s} \tag{3.1}$$

η_s Speedup

T Gesamtlaufzeit

t_s Laufzeit des sequenziellen Programmabschnitts

t_p Laufzeit des parallelen Programmabschnitts

t_{sync} Zeit zur Synchronisation des Programms

n_p Anzahl der verwendeten Prozessoren

Durch den zusätzlichen Synchroniersierungsaufwand und den sequentiellen Anteil ist der mögliche Speedup nach oben hin begrenzt und nimmt bei übermäßig großer Prozessoranzahl sogar wieder ab:

$$\lim_{(n_p, t_{\text{sync}}) \to \infty} \eta_s = 0 \tag{3.2}$$

Große Rechnungen lassen sich daher nicht mit einer beliebig großen Anzahl an Prozessoren „erschlagen“. Es muss daher genau abgewägt werden, wie sich welcher Prozess verteilen lässt. In der Realität stehen zudem nicht beliebig viele Prozessoren in einer Maschine zur Verfügung. Zahlreiche Threads auf einigen wenigen Prozessoren können aber zu enorm hohen Verwaltungsaufwand führen, was wiederum den Gesamtprozess ausbremst. Der Speedup ist zudem von der Rechenkapazität der Prozessoren bzw. einer Maschine abhängig und beschränkt. Auch wenn diese mehrere Threads und Befehlserweiterungen (bspw. SSE) verwenden, übersteigt irgendwann der lokale Verwaltungsaufwand den erzielten Speedup. Darüber hinaus wird der Nutzen, der beim Einsatz weiterer Prozessoren erreicht werden kann, gegenüber den dafür notwendigen Kosten immer kleiner, das Preis-/Leistungsverhältnis wird demnach immer schlechter.
Daher kann es sinnvoll sein, große Rechnungen nicht ausschließlich lokal, sondern verteilt in einem Netzwerk zu berechnen, insbesondere wenn die Infrastruktur für solch ein Netzwerk bereits vorhanden ist und so eine bessere Auslastung erzielt werden kann. Dies wirkt sich positiv auf das Preis-/Leistungsverhältnis des gesamten Systems aus. Hier gibt es zwar auch Verwaltungsaufwand - ein Netzwerk kann aber bei moderatem Verwaltungsaufwand beliebig erweitert werden, ohne dass ein Prozessor bzw. eine ganze Maschine mit Rechenoperationen überfordert wird. Es ist in so einem System daher eher möglich, ein hohes Maß an Parallelität mit wachsendem Speedup zu erreichen als lokal auf einer Maschine.

3.2 Messaging und Datenverwaltung in verteilten Anwendungen

Grundlage für die Verteilung von Aufgaben in einem System ist die Kommunikation der einzelnen Teilnehmer untereinander. Dazu wird eine geeignete Middleware benötigt, die Methoden dazu bereitstellt. Mit ZeroMQ wird nachfolgend solch eine Middleware vorgestellt. Darüber hinaus gibt es verschiedene Aspekte bzgl. der Datenverwaltung, die innerhalb einer verteilten Anwendung berücksichtigt werden müssen und nachfolgend diskutiert werden.

3.2.1 Distributed Messaging mit ZeroMQ als Middleware

ZeroMQ (auch ∅MQ, 0MQ oder ZMQ) ist eine asynchrone Messaging-Bibliothek, welche für effiziente Nachrichtenübermittlung entwickelt wurde. Es ist mit dieser API als Middleware (Vermittlungsschicht zwischen der eigentlichen Anwendung und der Netzwerkschnittstelle) möglich, nachrichtenbasierte Kommunikation sprachübergreifend zu implementieren. ZeroMQ ist für Problemstellungen entwickelt worden, bei denen es auf eine hoch optimierte und schnelle Datenübertragung ankommt. Für die Übertragung von Daten können verschiedene Protokolle verwendet werden (bspw. TCP). Es werden verschiedene Netzwerkarchitekturen und -topologien unterstützt:

- Request/Reply: Client-Server-Architektur, bei der der Client Anfragen an den Server stellt und dieser darauf antwortet
- Publisher/Subscriber: der Publisher veröffentlicht eine Nachricht bzw. Daten, sodass diese für alle Subscriber zugänglich sind. Die Kommunikation findet allerdings nur unidirektional statt
- Push/Pull (Pipeline): empfange Nachrichten bzw. Daten werden verarbeitet und modifiziert an den nächsten Teilnehmer weitergegeben
- Exclusive Pair: zwei Teilnehmer im selben Netzwerk verbinden sich dauerhaft miteinander (Verbindung besteht nur 1:1)

Je nach Problemstellung kann zwischen diesen Kommunikationsmodellen gewählt werden. Aufgrund des hohen Durchsatzes, der mit ZeroMQ erreicht werden kann und der einfachen Handhabung der Programm-API eignet sich diese Bibliothek hervorragend als Middleware

für ein verteiltes System, welches große Datenmengen in möglichst kurzer Zeit verarbeiten können soll.

3.2.2 Datenserialisierung

Damit eine Messaging Middleware wie ZeroMQ Daten versenden kann, um in einem Netzwerk mit verteilten Algorithmen und Programminstanzen arbeiten zu können, müssen diese Daten für den Transport vorbereitet werden. In diesem Kontext wird von Serialisierung und Deserialisierung gesprochen. Die Serialisierung von Daten und Parametern muss im Zusammenhang mit verteilten Anwendungen in der Regel von einem geeigneten Frameword (z.B. JSON, Protocol Buffers) übernommen werden. Im Rahmen des Krigings werden relativ wenige Objekte verwendet, die versendet werden müssen. Zudem verwalten diese meist linear im Speicher liegende Arrays. Daher wäre der Entwicklungsaufwand bzw. Overhead zur Integration von weiteren Frameword bzw. Interfaces zur Datenserialisierung in den Quellcode des Krigings zu hoch. Stattdessen wurde auf das eigens für diesen Zweck entwickeltes Interface `SaveableOnServer` (vgl. Abschnitt 1.4) zurückgegriffen und weiterentwickelt. Dieses stellt sämtliche virtuelle Funktionen bereit, die für die Datenserialisierung und -deserialisierung benötigt werden.

3.3 Topologie des Systems

Der Abteilung AT-FUV stehen für komplexe Rechenaufgaben sowohl ein externes Rechencluster mit zahlreichen CPUs als auch ein abteilungsinterner Rechner mit einer leistungsstarken GPU zur Verfügung. In der Regel wird die Prozesskette rund um das Programm AutoOpti auf dem externen Rechencluster gestartet. Wie in Abschnitt 1.4 bereits erwähnt wurde, ist die Anbindung des Abteilungsnetzwerks an das Cluster vergleichsweise langsam. Dadurch könnte der Geschwindigkeitsvorteil, der durch die Auslagerung von rechenintensiven Operationen auf die abteilungsinterne GPU erreicht werden kann, zunichte gemacht werden. Abhängig von der gegebenen Problemstellung muss daher vor der Wahl der geeigneten Netzwerktopologie geklärt werden, ob die abteilungsinterne GPU aufgrund ihrer langsamen Netzwerkanbindung überhaupt verwendet werden soll oder ob alle Rechnungen auf dem Cluster selbst auf mehrere CPUs verteilt werden sollen. Es ergeben sich für jeden dieser beiden Anwendungsfälle daher unterschiedliche Topologien:

- ein Client, ein Server (1:1 Beziehung) (vgl. Abbildung 3): da die Verbindung zwischen Client, der auf dem Cluster liegt, und dem Server, der abteilungsintern ausgeführt wird und eine leistungsstarke GPU zur Verfügung hat, als relativ langsam angenommen wird, sollte der Datenverkehr im Netz möglichst niedrig gehalten werden. Dies wird erreicht, indem die Datenpakete nur ein einziges Mal diese Verbindung passieren und lediglich Endergebnisse zurückgesendet werden (Größe eines `double`)
- ein Client, ein Broker/Balancer, n Server (1:1 und 1:n Beziehung) (vgl. Abbildung 4): der obere Fall reduziert zwar die zu übertragende Datenmenge zwischen dem externen Cluster und dem Abteilungsnetzwerk, bezieht aber nur eine einzige abteilungsinterne Recheneinheit in das System ein, obwohl bei geeigneter Problemaufteilung Teile der Rechnungen verteilt auf anderen Einheiten laufen könnten. Es wäre daher eine Aufteilung auf verschiedene Abteilungsrechner möglich. Da das Abteilungsnetzwerk allerdings nicht für große Datenmengen ausgelegt ist, soll dieser Ansatz nicht weiter verfolgt werden.
- ein Client, n Server (1:n Beziehung) (vgl. Abbildung 5): auf dem Cluster selbst ist die Netzwerkgeschwindigkeit der Knoten untereinander sehr hoch und beinahe vernachlässigbar. Daher können auch mehrere Server auf verschiedenen Knoten mit Daten bedient werden, ohne größere Einbußen befürchten zu müssen. So kann bei geeigneter Problemaufteilung ein hoher Grad an Parallelität erreicht werden. Die abteilungsinterne GPU wird in diesem Fall vernachlässigt.

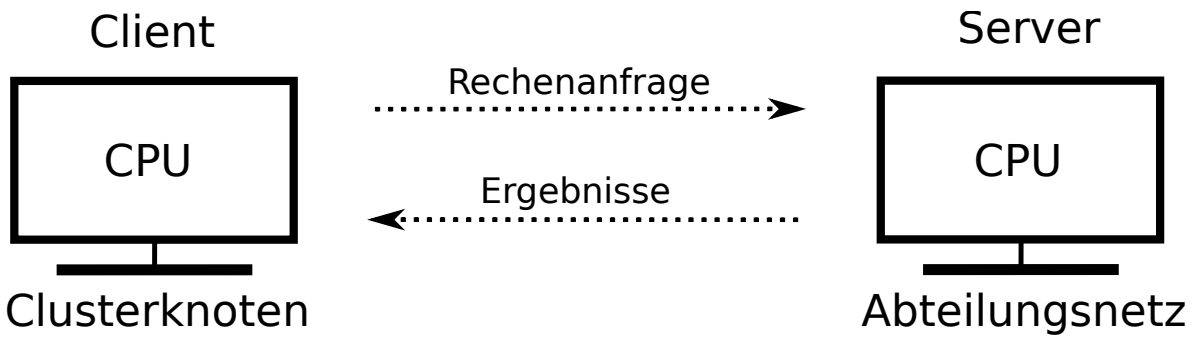


Abbildung 3: Netzwerktopologie mit einfacher 1:1-Beziehung zwischen Client (auf einem Clusterknoten) und Server (auf einem Rechner im Abteilungsnetzwerk) zur Integration einer leistungsstarken GPU

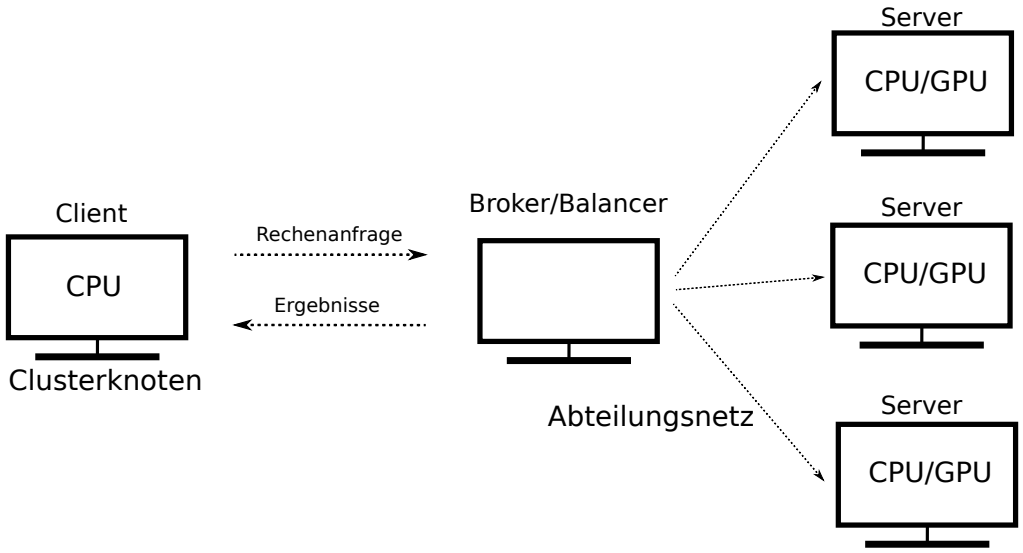


Abbildung 4: Netzwerktopologie mit 1:1-Beziehung zwischen Client (auf einem Clusterknoten) und einem Broker/Balancer im Abteilungsnetz und 1:n-Beziehung zu dahinter liegenden Servern zur Integration von n GPUs

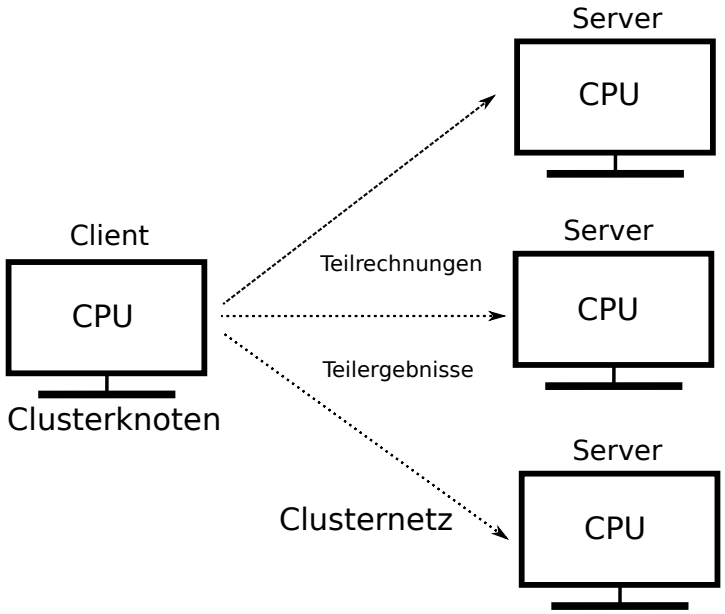


Abbildung 5: Netzwerktopologie mit 1:n-Beziehung zwischen Client und n Servern zur Lastverteilung auf dem Cluster intern

Theoretisch wäre auch eine Cluster-übergreifende 1:n-Beziehung zur Integration weiterer GPUs möglich. Derzeit steht allerdings die dafür notwendige Infrastruktur nicht zur Verfügung. Da die Netzwerkgeschwindigkeit zwischen Abteilungsnetz und Cluster je nach Auslastung zudem Schwankungen unterliegt und der mögliche Nutzen somit nur schwer abzuschätzen ist, wurde diese Variante nicht weiter in Betracht gezogen.

Bei der späteren Implementierung muss zudem erörtert werden, welche Aspekte der Transparenz eines verteilten Systems (vgl. [10], S. 14 ff.) und der Fehlersemantik (vgl. [11]) für welche Topologien im Rahmen des Krigings relevant sind. Da die hier vorgestellten Topologien grundsätzlich einer Client-Server-Architektur entsprechen, bietet sich für die aktuelle Problemstellung die Verwendung des Request/Reply-Patterns von ZeroMQ an (vgl. vorherigen Abschnitt). Neben einer einfachen 1:1 Verbindung kann mit diesem Pattern auch ein 1:n Netzwerkmodell umgesetzt werden.

3.4 Problemauslagerung und -verteilung

Für die Entwicklung des Clients ist es notwendig, Überlegungen zur sinnvollen Verteilung eines Problems anzustellen, um die gesamte Dauer des Programms so gering wie möglich zu halten. Die mathematischen Operationen, die für das Kriging benötigt werden, bauen hauptsächlich auf Matrizenrechnungen auf. Einige dieser Operationen können recht gut auslagert werden. Zusätzlich können auch einzelne Schleifen verteilt berechnet werden, deren Iterationen unabhängig voneinander sind. Diese Operationen werden in diesem Abschnitt nachfolgend näher betrachtet.

Die rechenintensiven Matrizenoperationen finden mit der Kovarianzmatrix **Cov** statt. Abhängig von der Anzahl der bisherigen Trainingsdaten N und der Anzahl an Parametern n hat diese eine Größe von $\mathbb{R}^{N^2}$, bei Gradient-Enhanced-Kriging-Ersatzmodellen sogar von $\mathbb{R}^{N^2*(n+1)^2}$. Zur Durchführung der mathematischen Operationen benötigt jeder Server diese Kovarianzmatrix. Es stellt sich daher die Frage, ob diese nun über das Netzwerk verschickt werden soll oder sich jeder Rechenknoten diese Matrix selbst aufbaut. Letzterer Fall benötigt deutlich weniger Informationen, die über das Netzwerk versendet werden müssen. Allerdings erhöht dies die

Rechendauer jedes einzelnen Netzwerkteilnehmers. Es soll später untersucht werden, welcher dieser beiden Fälle eher geeignet ist.

Auslagerung von sequentiellen Matrizenoperationen

Ein wesentlicher Bestandteil des Krigings ist die Cholesky-Zerlegung und deren Rückwärtsdifferenzierung. Die Berechnung einer Cholesky-Zerlegung ist anhand der Gleichung 2.3 dargestellt. Daran lässt sich allerdings erkennen, dass jeder Rechenschritt auf die Ergebnisse der vorherigen Berechnungen zurückgreift. Gleiches gilt für die Berechnung der adjungierten Matrix, da hier lediglich die Umkehrung des Cholesky-Algorithmus angewendet wird. Daher können beide Operationen ausschließlich sequentiell erfolgen, sodass eine sinnvolle Verteilung auf mehrere Recheneinheiten nicht umgesetzt werden kann. Stattdessen bietet sich für Berechnungen, denen großen Matrizen zugrunde liegen, eher eine 1:1 Verbindung zu einem Server mit einer leistungsstarken GPU an, um diesen Rechenoperationen zu beschleunigen. Vor allem das Training von Gradient-Enhanced-Kriging-Ersatzmodellen (vgl. Abschnitt 1.2) kann davon profitieren, da dort besonders große Matrizen auftreten können.

Verteilung von parallel ausführbaren Programmabschnitten

Eine Möglichkeit zur parallelen Verarbeitung von Rechnungen besteht in der Verteilung unabhängiger Programmabschnitte an mehrere Rechner. Im Laufe des Kriging-Verfahrens wird für jeden Hyperparameter eine partielle Ableitung der Dichtefunktion bestimmt. Die Bildung dieser partiellen Ableitungen erfolgt dabei unabhängig von den übrigen Hyperparametern. Dies ist ein rechenintensiver Schritt, da die Kovarianzmatrix in jeder Iteration einmal neu aufgestellt, Cholesky zerlegt und rückwärts differenziert werden muss. Da die übrigen Matrizenoperationen zur Bestimmung der partiellen Ableitungen für jeden Hyperparameter separat durchgeführt werden können, lässt sich dieser Prozess als 1:n Beziehung an mehrere Server im Netzwerk verteilen. Dies bietet sich vor allem bei Multifidelity-Kriging-Ersatzmodellen (vgl. Abschnitt 1.2) an, da diese besonders viele Hyperparameter verwenden. Die einzelnen Teilergebnisse müssen anschließend wieder an den Client zurückgeschickt und zu einem Gesamtergebnis zusammengefasst werden.

Zusammenfassend können so zwei Anwendungsfälle abhängig vom jeweiligen Ersatzmodell im Kriging betrachtet werden, die sich gut mit unterschiedlichen Topologien berechnen lassen:

- Ersatzmodelle mit großen Matrizen ausgelagert in einer 1:1 Beziehung
- Ersatzmodelle mit einer großen Anzahl an Hyperparametern verteilt in einer 1:n Beziehung

Ein dritter Anwendungsfall (große Matrizen mit einer großen Anzahl an Hyperparametern) könnte in einer Mischform mit Verbindungen der Art 1:1 und 1:n arbeiten. Wie eingangs allerdings schon erwähnt wurde, ist die dafür notwendige Infrastruktur derzeit nicht vorhanden. Da dieser Anwendungsfall darüber hinaus vergleichsweise selten auftritt, soll dieser nachfolgend nicht weiter betrachtet werden.

3.5 Verfahren zur Lastverteilung

Für eine optimale Ausnutzung der zur Verfügung stehenden Ressourcen werden effiziente Algorithmen zur Verteilung der Berechnungen benötigt. Dabei müssen die Datenmenge und die Komplexität der durchzuführenden Berechnungen bekannt sein. Es sollte jedoch auch beachtet werden, dass das Verfahren zur Lastverteilung selbst nicht beliebig kompliziert wird, da sonst die Berechnungszeit einer geeigneten Verteilung den beabsichtigten Nutzen zunichte machen kann.
Nachfolgend werden Algorithmen und Modelle zur Lastverteilung dargestellt und näher erläutert. Diese Verfahren werden unabhängig von der gewählten Netzwerktopologie betrachtet und können sowohl für Topologien (aus Client-Sicht) mit 1:n Beziehungen (direkte Anbindung des Clients an n Server) als auch für Topologien mit 1:1:n Beziehungen (Client kommuniziert mit einem Broker/Balancer, welcher die Lastverteilung übernimmt) angewendet werden. Es werden stets die Knoten im Netzwerk betrachtet, die die Verteilung der Daten und Berechnungen übernehmen. Dies kann entweder der Client selbst in einer 1:n Beziehung sein oder ein zwischengeschalteter Broker/Balancer in einer 1:1:n Beziehung.

3.5.1 Theoretische Betrachtung einer statischen Verteilung

Um eine geeignete Netzwerkauslastung in einem verteilten System mit n Servern zu erreichen, bedarf es vorab einiger theoretischer Betrachtungen. Die theoretische Rechenleistung eines

Prozessors wird üblicherweise vereinfacht in der Einheit „Gleitkommaoperationen pro Sekunde“ (flop/s) angegeben. Die Art der Operationen ist für die nachfolgenden Betrachtungen allerdings unerheblich. So wird auch die Komplexität eines Algorithmus in der Anzahl an benötigten Gleitkommaoperationen in Abhängigkeit der Problemgröße angegeben. Nachfolgende Überlegungen werden für einen beliebigen Server i angestellt, der Teil des Gesamtnetzwerkes ist. Grundsätzlich gilt für die Zeit t_i, die eine bestimmte Operation an einen Server i von der Anfrage bis zur Antwort benötigt:

$$t_i = t_n + t_r \tag{3.3}$$

wobei t_n für die Übertragungszeit der benötigten Daten und t_r für die Dauer der eigentlichen Berechnungen stehen. In der Theorie lässt sich diese Zeit mithilfe folgender zusätzlicher Informationen vorhersagen:

D Gesamtdatenmenge in Byte (sowohl Anfrage als auch Antwort)

C_i theoretische Datenrate zu Server i in $\frac{\text{Byte}}{s}$

$K(f)$ Komplexität der Funktion f mit $f : \mathbb{R}^k \to \mathbb{R}^m$ in flop

R_i theoretische Rechenleistung des Servers i in $\frac{\text{flop}}{s}$

Für die Zeit t_n, die der Transfer der Daten auf den Server i benötigt gilt somit:

$$t_n = \frac{D}{C_i} \tag{3.4}$$

und für die theoretische Rechendauer t_r des Servers i:

$$t_r = \frac{K(f)}{R_i} \tag{3.5}$$

sodass für die Gesamtzeit t_i gilt:

$$t_i = \frac{D}{C_i} + \frac{K(f)}{R_i} \tag{3.6}$$

Sei nun nachfolgend die Funktion f das aktuell zu berechnende Problem. Wird nun ein System mit mehr als einem Server betrachtet, stellt sich die Frage der geeigneten Partitionierung von

f. Bedingung für die Aufteilung einer Rechnung ist, dass diese parallelisiert und unabhängig von Zwischenergebnissen berechnet werden kann. Unter der Annahme, dass die Funktion f parallelisierbar ist und die vollständige Datenbasis D für die Berechnung auf dem Server benötigt werden, kann die theoretisch benötigte Gesamtzeit als Funktion von dem Anteil x_i des Servers i an der Gesamtberechnung wie folgt dargestellt werden:

$$t_i(x_i) = \frac{D}{C_i} + \frac{K(f) * x_i}{R_i} \quad \text{mit } x_i \in \mathbb{R},\ x_i \leq 1 \tag{3.7}$$

wobei

$$\sum_{i=1}^{n} x_i = 1 \tag{3.8}$$

Mithilfe der theoretischen Rechenzeit eines Servers i für einen Teil des Gesamtproblems x_i lässt sich nun die theoretische Dauer des gesamten Problems t_{ges} in Abhängigkeit der einzelnen Anteile $x_1, x_2, \ldots, x_n$ ermitteln:

$$t_{ges}(x_1, x_2, \ldots, x_n) = max(t_i(x_i)) + t_{seq} \quad \text{mit } i = 1 \ldots n \tag{3.9}$$

wobei t_{seq} den sequentiellen und nicht parallelisierbaren Programmteil darstellt. Dieser ist für die Berechnung einer geeigneten Verteilung allerdings nicht relevant. Die Gesamtdauer wird maßgeblich von dem Server mit der höchsten Laufzeit bestimmt, da alle Netzwerkteilnehmer parallel arbeiten. Es wäre daher von Vorteil, dass leistungsstarke Server einen möglichst großen Anteil an der Gesamtrechnung haben im Vergleich zu schwächeren Servern. Eine gute Lastverteilung soll daher bewirken, dass nach Möglichkeit alle zur Verfügung stehenden Netzwerkteilnehmer die gleiche Rechendauer t_0 haben sollen, da hier die größte Performance des Gesamtsystems zu erwarten ist:

$$t_i(x_i) = \frac{D}{C_i} + \frac{K(f) * x_i}{R_i} = t_0 \quad \text{mit } i = 1 \ldots n \tag{3.10}$$

Diese Gleichung kann nun nach dem variablen Anteil x_i mit dem Vorfaktor a_i und dem restlichen konstanten Anteil c_i wie folgt umgestellt werden:

$$a_i * x_i = t_0 - c_i \tag{3.11}$$

mit $a_i = \frac{K(f)}{R_i}$ und $c_i = \frac{D}{C_i}$, sodass sich unter Einbeziehung der Nebenbedingung aus 3.8 folgendes Gleichungssystem der Form $A * \vec{x} = \vec{b}$ ergibt:

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ a_1 & 0 & \dots & 0 \\ 0 & a_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & a_n \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} 1 \\ t_0 - c_1 \\ t_0 - c_2 \\ \vdots \\ t_0 - c_n \end{pmatrix} \tag{3.12}$$

Wie sich hier erkennen lässt, ist dieses Gleichungssystem überbestimmt. Zudem existiert im Zielvektor ein unbekannter, wenn auch konstanter Faktor. Berechnungen dieser Art können daher äußerst komplex und im ungünstigen Falle gar nicht oder nur mit numerischen Verfahren gelöst werden. Die Berechnung einer günstigen Lastverteilung wird so selbst zu einem Optimierungsproblem und kann den eigentlichen Prozess ausbremsen. An dieser Stelle muss zudem die Frage gestellt werden, ob dieser Ansatz überhaupt zielführend ist. Einerseits entsteht durch die Frage nach einer geeigneten Lastverteilung ein neues Optimierungsproblem im Raum $\mathbb{R}^n$, welches sich nicht ohne umfangreiche Berechnungen ermitteln lässt. Andererseits ist auch nicht sichergestellt, dass sowohl die angegebene Netzwerkgeschwindigkeit als auch die theoretische Rechenleistung der Netzwerkteilnehmer bzw. deren Datenübertragungsraten der Realität entsprechen oder konstant sind. Die vorangegangenen Überlegungen zur theoretischen Rechenleistung eines Servers berücksichtigen nämlich nicht wichtige Aspekte wie Hardwareeigenschaften (Anzahl an Prozessoren, Unterstützung von SSE-Beschleunigung, Multithreading, etc.), Auslastung, Art des Prozessors (CPU/GPU), Umsetzung der Berechnungen (Standardprozeduren, optimierte Bibliotheken), etc., welche u.a. auch von der Problemgröße D abhängen können. Eine GPU bspw. hat einen viel höheren Datendurchsatz (flop/s) bei großen Problemstellungen. Es ergibt sich für jede GPU, aber auch CPU, ein charakteristischer Verlauf des Datendurchsatzes über der Datenmenge - unabhängig von dem theoretischen Nennwert des Durchsatzes. Auch ein möglicher Ausfall eines Servers während der Laufzeit wird hier nicht betrachtet.

Abschließend ist es ist daher nur sehr schwer möglich, eine allgemeine Aussage bzw. Gesetzmäßigkeit für die voraussichtliche Rechengeschwindigkeit und damit auch eine Vorhersage für für die Gesamtzeit treffen zu können. Dieser Ansatz kann allerdings zur Entwicklung neuer

Strategien verwendet werden, die hierauf aufbauen und auf die hier beschriebenen Probleme eingehen.

3.5.2 Umsetzung einer statischen Lastverteilung

Mit dem Ansatz der statischen Lastverteilung wird versucht, die in Abschnitt 3.5.1 angestellten Überlegungen (konkret: Gleichungen 3.9 bzw. 3.12) zu vereinfachen und umzusetzen. Die zuvor beschriebenen Probleme konnten mit diesem Ansatz im Kern allerdings nicht gelöst werden.
Der prozentuale Anteil an der Gesamtberechnung, welcher ein an das Netzwerk angeschlossener Server zu leisten hat, wird noch vor der Verteilung der Daten errechnet. Es wird zudem davon ausgegangen, dass der aufrufende Netzwerkknoten (Client) nicht nur Datenpakete verteilt, sondern auch selbst Berechnungen durchführen kann. Zu Beginn der Netzwerkarbitrierung schickt dabei jeder Server seine theoretische Rechenleistung an den Client zurück (in flop/s). Anschließend werden zufällig erzeugte Datenmengen hin und zurück geschickt, um die Geschwindigkeit der Verbindung zu messen. Anhand dieser Daten wird dann das Verhältnis η_i der zu erwartenden Rechenzeiten zwischen dem Server i und dem Client nach folgendem Schema ermittelt:

$$\eta_i = \frac{\frac{K(f)}{R_m}}{\frac{K(f)}{R_i} + \frac{D}{C_i}} \tag{3.13}$$

R_m Rechenleistung des Clients in Gflop/s

R_i Rechenleistung des Servers i in Gflop/s

$K(f)$ Komplexität der Berechnung für Funktion f (Anzahl der benötigten Rechenoperationen in Gflop)

D Datenmenge (Anzahl der zu verschickenden double-Werte)

C_i Netzwerkgeschwindigkeit in double/s

η_i Verhältnis der theoretischen Berechnungszeiten zwischen Server i und Client

Diese Berechnung wird für jeden an den Client angeschlossenen Server i durchgeführt. Aus diesen Verhältnissen lassen sich nun jeweils alle prozentualen Anteile an der einzelnen Server

an der Gesamtrechnung bestimmen. Diese werden wie folgt bestimmt:

$$p_i = \frac{\eta_i}{\sum_j^n \eta_j} * 100 \tag{3.14}$$

n Anzahl der Server

p_i Anteil an der Gesamtberechnung für Server i in Prozent

η Verhältnis der theoretischen Berechnungszeiten zwischen Server und Client

Das Prinzip ist in Abbildung 6 schematisch dargestellt. Der Vorteil dieser Methode zur Berechnung einer geeigneten Lastverteilung ist, dass stärkere Netzwerkteilnehmer bzw. solche mit höherer Verbindungsgeschwindigkeit stärker belastet werden. Darüber hinaus ist es relativ einfach umzusetzen.
Da die eingangs erwähnten Probleme aber auch bei diesem Ansatz bestehen bleiben, müssen nachfolgend andere Konzepte betrachtet werden, die für dynamische Verteilungsprozesse besser geeignet sind.

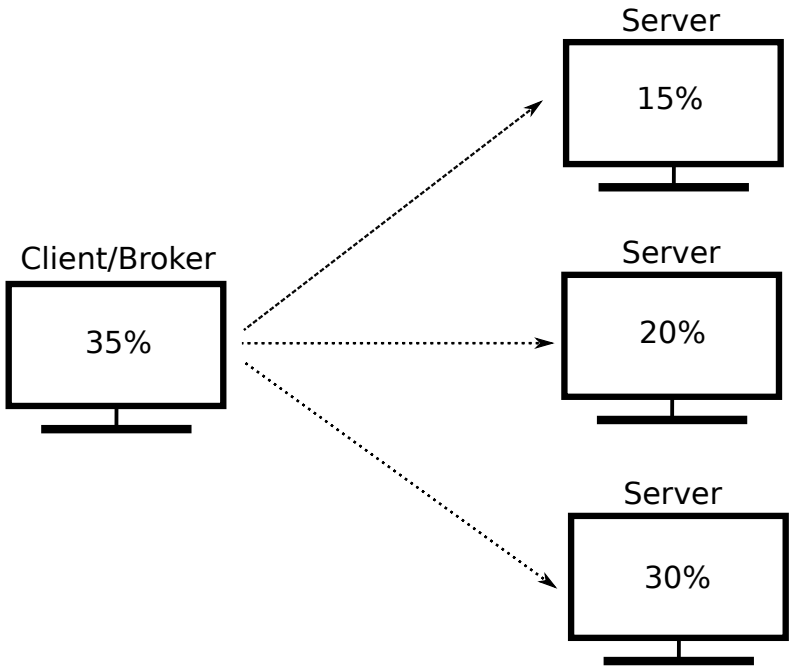


Abbildung 6: Statische Lastverteilung an n Server

3.5.3 Rundlauf-Verfahren/Round Robin

Bei dem Round-Robin-Verfahren wird die durchzuführende Rechenoperation in zahlreiche kleinere Pakete eingeteilt und an mehrere Server verteilt. Jedes Paket besteht aus einer Anfrage, welche Anweisungen zur Berechnung eines Anteils an der gesamten Rechenoperation enthält. Die Paketgröße, hier die jeweilige Größe des zu berechnenden Anteils am Gesamtproblem, wird vorher festgelegt und bleibt konstant. Anschließend erhält jeder Server ein Rechenpaket, welches er zu bearbeiten hat. Sobald die Rückmeldung eines Servers i an den Client kommt, dass dieser die Berechnung abgeschlossen hat, erhält er ein neues Rechenpaket. Bei einem Netzwerkfehler wird das fehlgeschlagene Paket einfach wieder auf den Stack abgelegt, sodass es von einem anderen Server bearbeitet werden kann. Starke Server mit einer guten Netzwerkanbindung tragen somit deutlich mehr zu dem Gesamtergebnis bei als langsame Netzwerkteilnehmer. Dem Client müssen nicht einmal die genauen Rechenleistungen sowie Verbindungsgeschwindigkeiten der einzelnen Server bekannt sein, um eine gute Lastverteilung zu erreichen. Der Grundgedanke dieser Methode wird in Abbildung 7 noch einmal dargestellt. Es muss allerdings eine Paketgröße gefunden werden, welche die geringste Ausführungsdauer für ein gegebenes Problem erreicht. Bei Netzwerken mit unterschiedlich starken Servern wären kleinere Paketgrößen von Vorteil. Bei kleinen Paketgrößen ist zwar mehr Netzwerkkommunikation notwendig, die Server können so aber in Abhängigkeit von ihren Rechenleistungen besser ausgelastet werden. Da im Rahmen dieser Arbeit aber auf einem Cluster mit Knoten gearbeitet wird, die in ihrer Rechenleistung identisch sind, kann hier mit deutlich größeren Paketen gearbeitet werden, um so die Netzwerkkommunikation zu verringern. Bei gleich bleibender Paketgröße benötigt hier jeder Server so die gleiche Zeit, sodass unnötige Wartezeiten, die den Gesamtprozess ausbremsen, so gering wie möglich gehalten werden können.

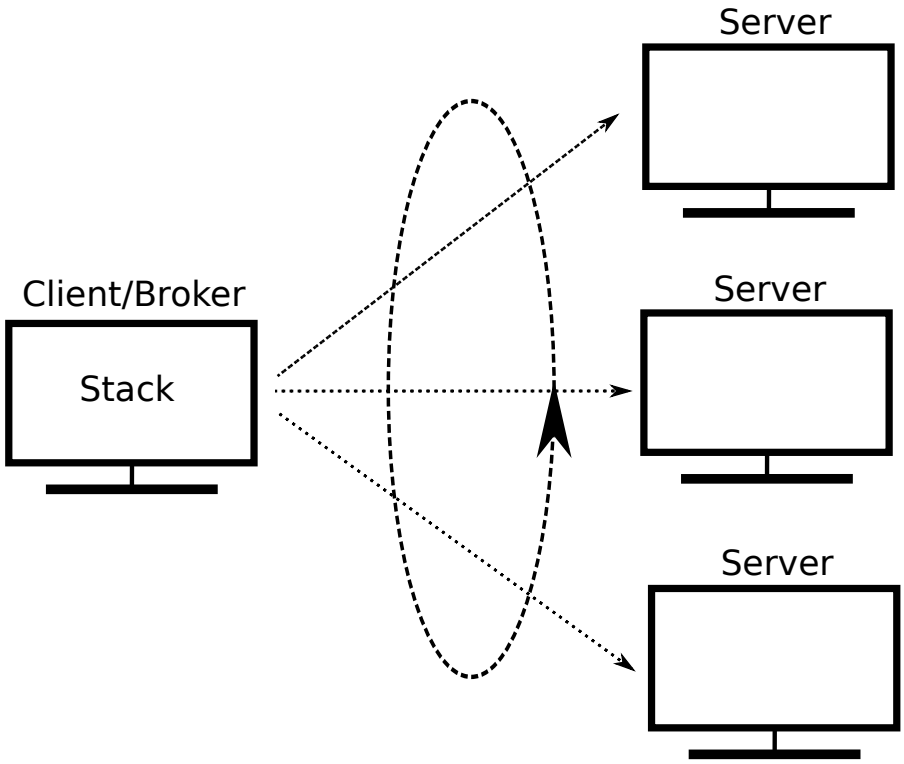


Abbildung 7: Lastverteilung nach einfachem Rundlauf-Verfahren/Round Robin an n Server anhand eines zuvor erzeugten Stacks. Dieser wird periodisch an bereite Server verteilt

4 Umsetzung

Nachdem im vorherigen Kapitel die Grundlagen zum Thema Prozessverteilung erörtert wurden, beschreibt dieses Kapitel die Implementierung dieser in einer für das Kriging-Programm geeigneten Form. Im Zuge dieser Arbeit ist es notwendig gewesen, sich in den Quellcode des in [4] beschriebenen Krigings als Teil der Optimierungsprozesskette einzuarbeiten und darauf aufzubauen. Die Entwicklung der Verteilungsalgorithmen und einzelner Quellcodeoptimierungen erfolgte in iterativ inkrementeller Herangehensweise durch ständige Neudefinition, Implementierung, Überprüfung und Bewertung von kleineren Teilaufgaben über den gesamten Arbeitszeitraum hinweg.

4.1 Struktur der Netzwerkschnittstelle

Zur Kommunikation mit anderen Netzwerkteilnehmern wird die Messaging-Middleware ZeroMQ verwendet. Die Implementierung der Netzwerkschnittstelle folgt dabei dem in Abschnitt 1.4 beschriebenen Klassenaufbau.
Zum Verbindungsaufbau mithilfe der Methoden aus dieser Middleware wurden diese Funktionsaufrufe in einer zwischengelagerten Klasse `ZMQConnection` zusammengefasst und abstrahiert. Dies vereinfacht die spätere Arbeit mit der Schnittstelle ungemein und erlaubt zudem mehrere Verbindungsinstanzen zu n Servern. Da es sich hier um ein klassisches Client-Server-Modell handelt, wurden die einzelnen ZeroMQ-Verbindungen dementsprechend konfiguriert. Auf eine bestimmte Anfrage wird eine Antwort erwartet (Request/Reply). Nachfolgend werden die wichtigsten Methodenaufrufe, die diese Klasse bereitstellt, aufgelistet:

- `send(...)` - sendet ein Datenarray des Typs `void`
- `recvMem(...)` - empfängt gesendete Daten
- `sendOk()` - sendet ein OK-Signal
- `sendNotOk()` - sendet ein Fehlersignal
- `recvOk()` - erwartet ein OK-Signal

Diese Klasse stellt die Grundlage für alle weiteren Klassen dar, die weitergehende Netzwerkfunktionalitäten bereitstellen.

4.1.1 Netzwerkobjekte und Datenrepräsentation

Programmobjekte bzw. bestimmte Instanzen von Klassen, die im Netzwerk verteilt werden, werden nachfolgend als Netzwerkobjekte bezeichnet. Zur Verwaltung dieser Objekte in einem Netzwerk werden zusätzliche Funktionalitäten benötigt - hauptsächlich zur Serialisierung und Deserialisierung für den Transfer. Zu Beginn der Arbeit wurde bewusst auf den Einsatz von weiteren APIs/Middleware verzichtet, da der Implementierungsaufwand im Vergleich zum späteren Nutzen minimal gewesen wäre und nur wenige der API-Funktionen letztendlich benötigt werden.
In [4], S.22 ff. wird eine Möglichkeit beschrieben, mittels Vererbung verschiedene Objekte für Netzwerkkommunikation nutzbar zu machen. Zur Datenrepräsentation wird daher eine neue abstrakte Klasse `SaveableOnServer` den zu verteilenden Objekten als Interface vorgelagert. Diese sorgt dafür, dass folgende Methoden in den Subklassen implementiert werden und bereitstehen:

- `getSerializedData(...)` - serialisiert das Objekt und stellt dieses als Array des Typs `void` bereit
- `setSerializedData(...)` - erwartet serialisierte Daten, deserialisiert diese und setzt die neuen Objekteigenschaften
- `getServerType()` - liefert einen klassenspezifischen Objektbezeichner zurück, um die Art des Objektes bestimmen zu können
- `freeMem()` - löscht das Objekt und gibt dessen Speicher frei
- `print()` - stellt das Objekt auf der Standardausgabe (stdout) dar (sofern darstellbar)

Mithilfe dieser Funktionen lassen sich alle benötigten Netzwerkfunktionalitäten umsetzen. Folgenden Klassen wurde die abstrakte Netzwerk-Superklasse `SaveableOnServer` vorgelagert, um mit diesen in einem Netzwerk verteilt arbeiten zu können (in Abbildung 8 zusätzlich dargestellt):

- `Matrix` - Superklasse für Matrizenobjekte
- `DensityFunction` Superklasse für Dichtefunktionen
- `Point` - repräsentiert die Punkte/Stützstellen des Krigings
- `config` - Konfigurationsobjekt zur Steuerung des Krigings

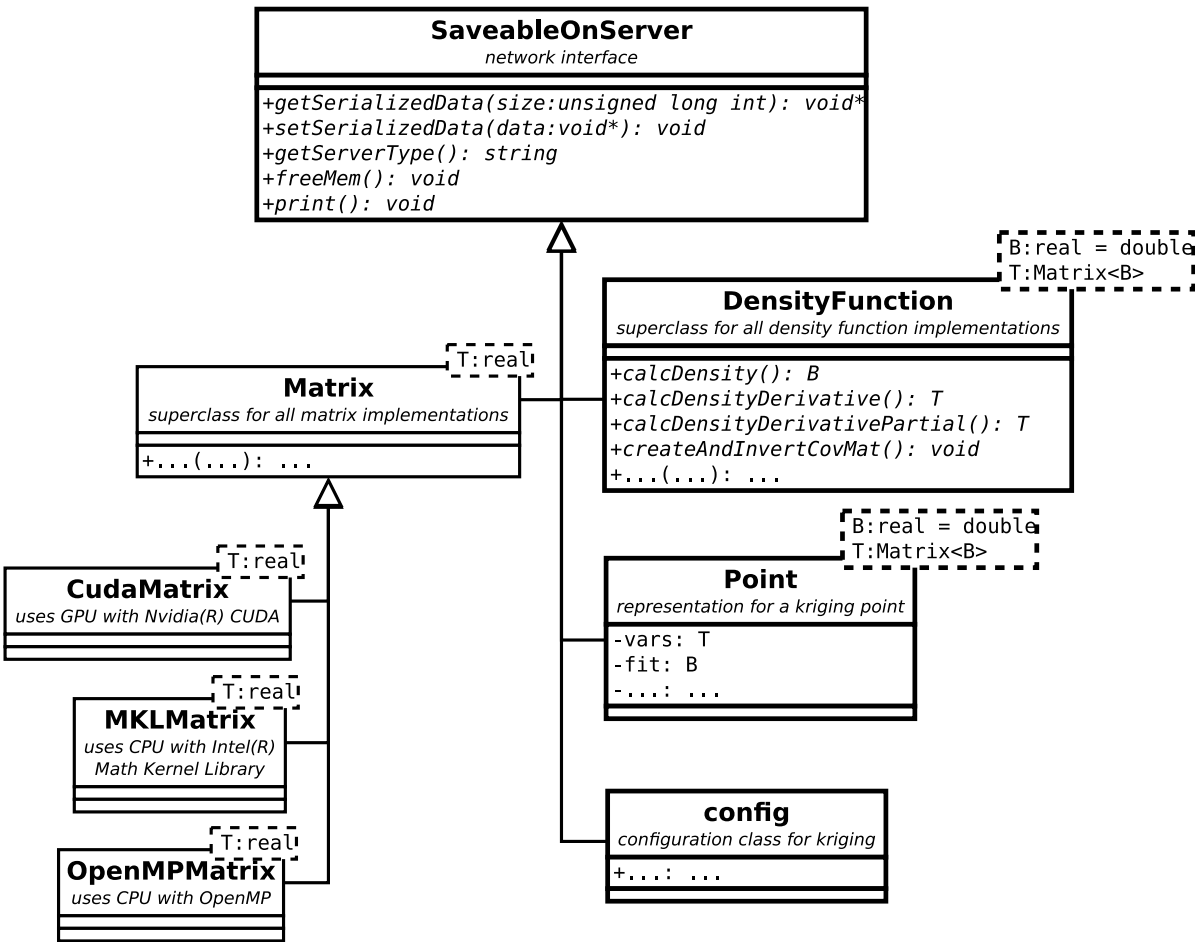


Abbildung 8: UML-Diagramm aller Klassen, die von der abstrakten Netzwerkklasse `SaveableOnServer` erben

Das ursprüngliche Ziel war es, lediglich Matrizenoperationen auf externe Netzwerkteilnehmer auszulagern. Die Geschwindigkeit des internen Netzwerks ist für große Matrizen allerdings zu gering. Die Matrizen können aber anhand weniger Daten mit Methoden einer Dichtefunktion erzeugt werden. Es wurde daher entschieden, statt der Matrizen direkt die Dichtefunktionen auf einen Server auszulagern, um diese auf in einem Netzwerk nutzen zu können. Hierfür werden deutlich weniger Informationen benötigt, um solch eine Dichtefunktion nach Versand auf einem entfernten Server wieder zu deserialisieren und nutzbar zu machen (u.a. die Stützstellen des Krigings in Form der Klasse `Point` und die Konfiguration mit der Klasse `config`). Bei geringer Netzwerkbandbreite beschleunigt dies den Gesamtprozess deutlich.

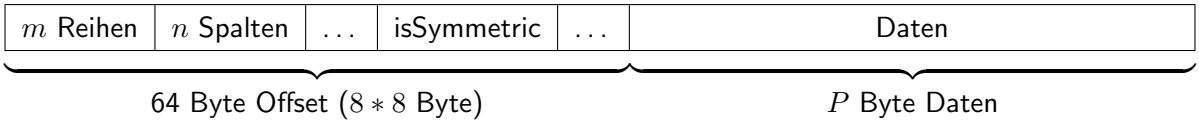
Serialisierung und Deserialisierung

Das Interface in Form der Klasse `SaveableOnServer` stellt die Funktionen zur Datenserialisierung/-deserialisierung bereit. Diese Vorgänge unterscheiden sich allerdings von Netzwerkobjekt zu Netzwerkobjekt, da die Struktur der Daten für jede Klasse anders aussieht.

Matrix:

Die Datenstruktur der Klasse `Matrix` und aller Subklassen erlaubt eine recht einfache Form der Datenserialisierung und -deserialisierung. Sämtliche Informationen werden in der Klassenvariable „elements“ in Form eines Datenarrays gespeichert (vgl. auch [4], S.23 f.). Dieses besteht aus einem acht Werte großen Offset, welches Informationen über Zeilen- und Spaltenanzahl sowie Eigenschaften der Matrix enthält, gefolgt von den eigentlichen Matrixeinträgen. Dieses Array kann einfach der Methode `ZMQConnection.send(...)` übergeben werden (nach vorherigem Cast auf `void*`) und umgekehrt von der Methode `ZMQConnection.recv(...)` nach Cast auf den geforderten Datentyp eingelesen werden. Eine Besonderheit von Kovarianz- bzw. Cholesky zerlegten Matrizen besteht in deren Symmetrie. Bei symmetrischen Matrizen reicht es aus, wenn entweder die obere oder die untere Hälfte der Matrix entlang der Hauptdiagonalen gesendet wird - die andere Hälfte ergibt sich dem entsprechend. Dies wird sich auch in den Methoden zur Datenserialisierung/-deserialisierung zu Nutze gemacht: ist zu Beginn des Datenarrays das Flag zur Kennzeichnung der Symmetrie auf „true“ gesetzt, so wird statt der gesamten Matrix lediglich die Hälfte der Daten über das Netzwerk verschickt. Dies erfordert zwar geringen Aufwand in der Speicherverwaltung durch Umkopieren des Datenarrays, erspart allerdings die Hälfte an über das Netzwerk zu sendenden Daten. Dies bringt noch einmal

einen großen Geschwindigkeitsvorteil. Die gesendeten Daten haben daher folgende Struktur:



Die Gesamtgröße der Daten P (in Byte) berechnet sich wie folgt: bei einer nicht symmetrischen Matrix mit m Zeilen und n Spalten mit:

$$P = 8 * m * n \quad \text{(nicht symmetrisch)} \tag{4.1}$$

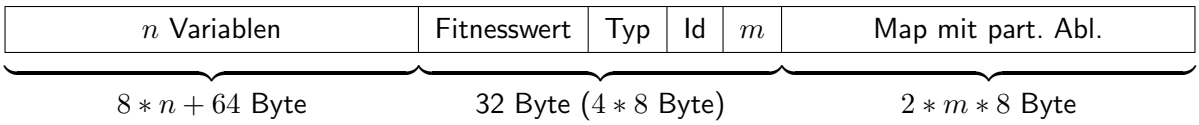
$$P = 4 * n * (n + 1) \quad \text{(symmetrisch)} \tag{4.2}$$

Die Gesamtgröße einer Matrix ergibt sich insgesamt mit $P + 64$ Byte.

Point:
Ein einzelner Punkt enthält folgende Daten, die versendet werden müssen:

- punktspezifische Variablen (wenige Metadaten, die in einem Vektor der Größe n gespeichert sind) - `Matrix` (nicht symmetrisch)
- Fitnesswert (optimierungsspezifische Kenngröße) - double
- Typ (0/1) - integer
- Id - integer
- Map mit partiellen Ableitungen (Größe: m) - unsigned integer zu double zugeordnet

Diese Werte werden in einem Array vom Typ `double` zusammengefasst und versendet. Zuerst wird die Funktion zur Serialisierung/Deserialisierung der punktspezifischen Variablen aufgerufen, da es sich hier um eine Instanz der Superklasse `Matrix` handelt. Die restlichen Daten sind diesen angehangen. Dabei wird auf folgender seriellen Datenstruktur gearbeitet:



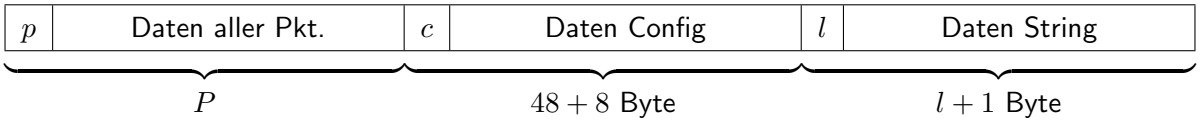
Die Gesamtgröße eines Punktes beträgt damit $8 * (2 * m + n) + 96$ Byte.

Config:
Die Klasse `config` steuert das Verhalten des Krigings. Diese enthält zahlreiche einzelne Variablen, die einzeln verwaltet werden, wobei insgesamt zwölf davon als relevant für die Übertragung über das Netzwerk identifiziert werden konnten. Diese werden in ein einzelnes Datenarray vom Typ `integer` geschrieben, sodass die Gesamtdatenmenge für die Datenserialisierung/-deserialisierung insgesamt nur 48 Bytes ($12 * 4$ Bytes) beträgt.

Density Function:
Die Serialisierung/Deserialisierung einer ganzen Dichtefunktion gestaltet sich etwas schwieriger. Es existierten aber bereits Funktionen, um ein Abbild einer Dichtefunktionen mit einem String im XML-Format zu erzeugen bzw. ein solches einzulesen (`getSerializedString` bzw. `parseSerializedString`). Diese können nun dazu verwendet werden, um eine Instanz einer Dichtefunktion zu versenden und auf einem entfernten Server wieder aufzubauen. Das XML-Format ist nicht sonderlich effizient für den Datenverkehr. Allerdings ist der String, der mithilfe dieser Funktionen verwaltet wird, in der Regel verschwindend (wenige Kilobyte), sodass dies weiteren Entwicklungsaufwand zur Datenserialisierung/-deserialisierung nicht notwendig macht. Es werden generell drei wichtige Teile benötigt:

- Punkte - diese müssen allerdings nur einmal über das Netzwerk geschickt werden, da diese sich im Verlauf des Krigings nicht ändern (Anzahl der Punkte: p)
- Config - zur Steuerung des Krigings (Datenmenge: c - insgesamt 12 Werte)
- serialisierter Datenstring der Größe l

Da der Versand aller Punkte nur ein einziges Mal auf einen Server erfolgen muss, wird dies durch ein spezielles Flag gekennzeichnet. Ist dieses auf den Wert „true“ gesetzt, so werden die Punkte beim nächsten Versand zu dem gleichen Server nicht mehr mitgesendet. Alle Daten werden serialisiert und byteweise in ein einziges Datenarray vom Typ `char` abgespeichert. Es ergibt sich aus allen Teilen daher folgende Datenstruktur (mit m partiellen Ableitungen für n Hyperparameter):



wobei für die Datengröße P aller Punkte p gilt:

$$P = p * (8 * (2 * m + n) + 96) + 8 \text{ Byte}$$

Die Größe D einer zu sendenden Dichtefunktion ergibt sich demnach wie folgt:

$$D = P + l + 65 \text{ Byte} \tag{4.3}$$

wobei $P = 0$ ist, wenn keine Punkte mitgesendet werden. In der Regel ist die Länge l des serialisierten Datenstrings äußerst gering, weshalb die zu sendende Datenmenge D allgemein relativ gering ist.

4.1.2 Client

Der Client wird in dem Kriging zur Verteilung der Rechenaufgaben benötigt. Zur Implementierung wurden zwei Klassen geschrieben, welche unterschiedliche Aufgaben wahrnehmen. Die Klasse `ZMQClient` verwaltet die Verbindungen zu den Servern über Instanzen von `ZMQConnection` und übernimmt die Speicherverwaltung von zu sendenden und empfangenen Objekten. So stellt diese Methoden wie z.B. `mallocVarOnServer(...)` (Objekt an einen Server versenden) oder `getVarFromServer(...)` (Objekt von einem Server anfordern) bereit. Es sind hier Verbindungen zu beliebig vielen Servern möglich. Darauf aufbauend existiert die zweite Klasse `MatrixClientFunctions` (vgl. auch UML-Diagramm in Abbildung 9).
Hiermit lassen sich Matrizenoperationen wie Additionen an alle im Netzwerk verfügbaren Server verteilen. Die Funktion `distributedDensityDerivatePartial(...)` bspw. verteilt die Berechnung der partiellen Ableitungen der Dichtefunktionen für die einzelnen Hyperparameter an mehrere Server. Dazu wird im ersten Schritt die adjungierte Matrix (vgl. Kapitel 2) an alle Server verschickt. Anschließend werden die Berechnungen auf diesen Servern vom Client aus gestartet und die Ergebnisse im Anschluss eingesammelt und zusammengefügt. Damit kann nun weiter gearbeitet werden.
Grundsätzlich stellt der Client keine permanente Verbindung zu Servern auf, sondern belegt diese nur bei Bedarf. Damit dieser weiß, mit welchen Servern er sich verbinden kann, gibt es eine spezielle Konfigurationsdatei, die bei der Initialisierung eingelesen wird. Dort sind neben der eigenen Rechneradresse alle möglichen Netzwerkteilnehmer inklusive deren Ports auf-

geführt. Darüber hinaus kann dort ein Server mit dem Stichwort „extern“ hinter dem Namen als externer Server markiert werden. Dies ist für den weiteren Verlauf des Krigings wichtig, da externe Server anders behandelt werden und die zu sendende Datenmenge möglichst gering gehalten wird.

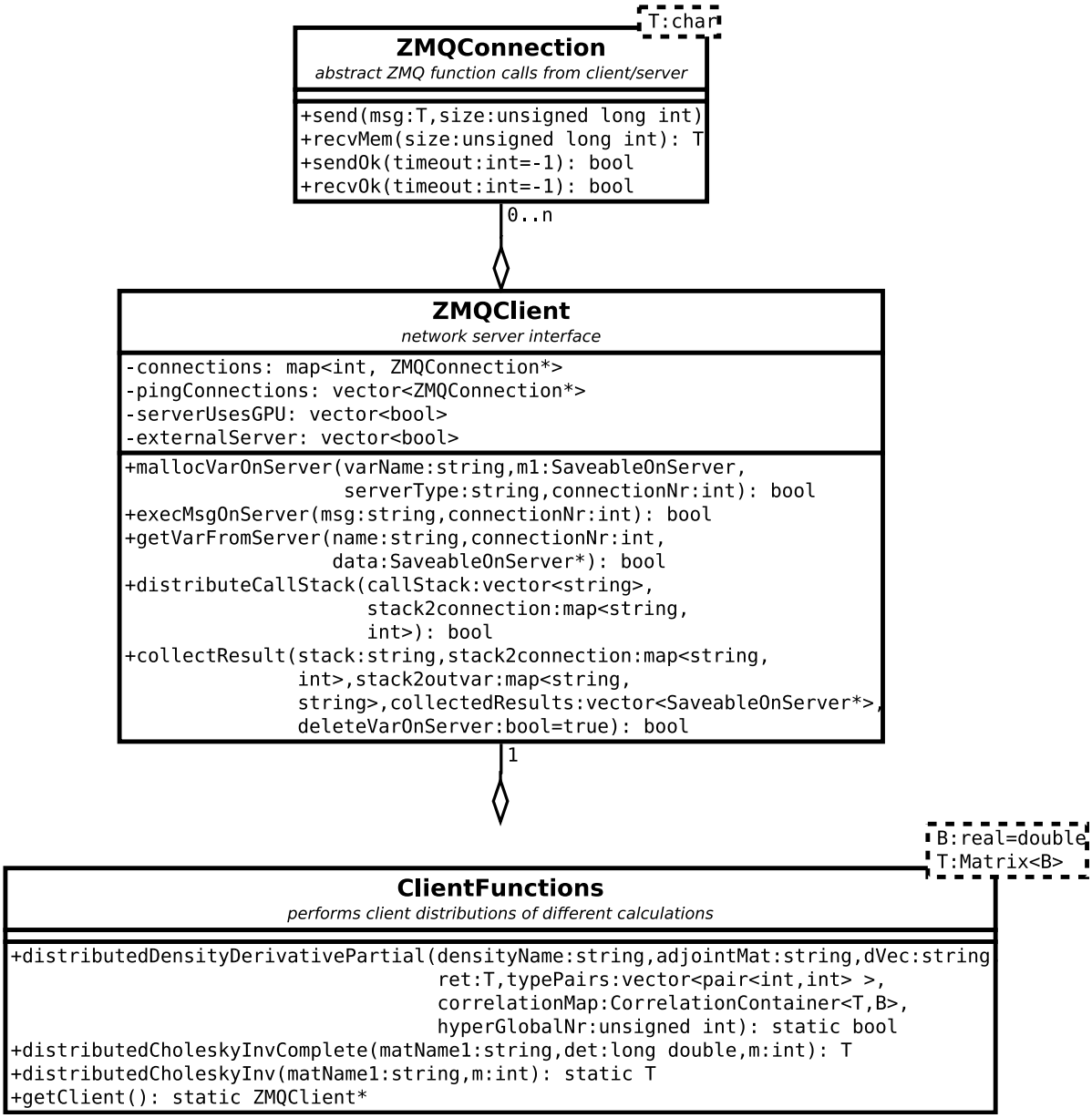


Abbildung 9: UML-Diagramm des Client-Interface. Im Programm selbst wird über die Klasse `ClientFunctions` mittels der Funktion `getClient()` auf den Client zugegriffen

4.1.3 Server

Der Server kann als Pendant zum Client angesehen werden und ist analog dazu aufgebaut. Er gliedert sich wieder in zwei unterschiedliche Klassen. In `ZMQServer` wird sämtliche Kommunikation mit dem Client sowie die Speicherverwaltung der verteilten Objekte abgewickelt. Darüber hinausgehende Anfragen, welche mathematische Operationen an zuvor gesendeten verteilten Objekten beinhalten, werden an eine Instanz der Klasse `MatrixServerFunctions` weitergeleitet (vgl. UML-Diagramm in Abbildung 10). Diese beinhaltet alle benötigten Methoden (partielle Berechnung einer Matrixinversen, Berechnung der Dichten/Gradienten der Dichtefunktionen). So kann eine strikte Trennung von Programmverwaltung und den eigentlichen Rechnungen realisiert werden.
Anders als zu herkömmlichen Servern ist allerdings nur eine Verbindung gleichzeitig zu einem Client erlaubt, da die Daten anhand ihrer Namen verwaltet werden, die der Client vergibt, und ein Server so die Daten nicht einem Client zuordnen kann. Es könnte so zu konkurrenten Zugriffen kommen und die Integrität der Daten gefährden, wenn mehrere Clients zugelassen werden.
Bei Programmstart des Servers kann über einen Parameter gesteuert werden, ob dieser für seine Berechnungen die GPU mittels der Klasse `CudaMatrix` oder die CPU über `MKLMatrix` verwenden soll (ist weder die Bibliothek Nvidia® CUDA noch Intel® MKL vorhanden, so wird die ursprüngliche Klasse `OpenMPMatrix` verwendet). Es ist daher möglich, auf einem Rechner mehrere Server zu starten, die verschiedene Prozessoren verwenden können. Dies kann insbesondere auf dem Abteilungsrechner mit der leistungsstarken GPU genutzt werden, um daneben auch die CPU nutzen zu können und um so eine bessere Auslastung der zur Verfügung stehenden Ressourcen erreichen zu können.

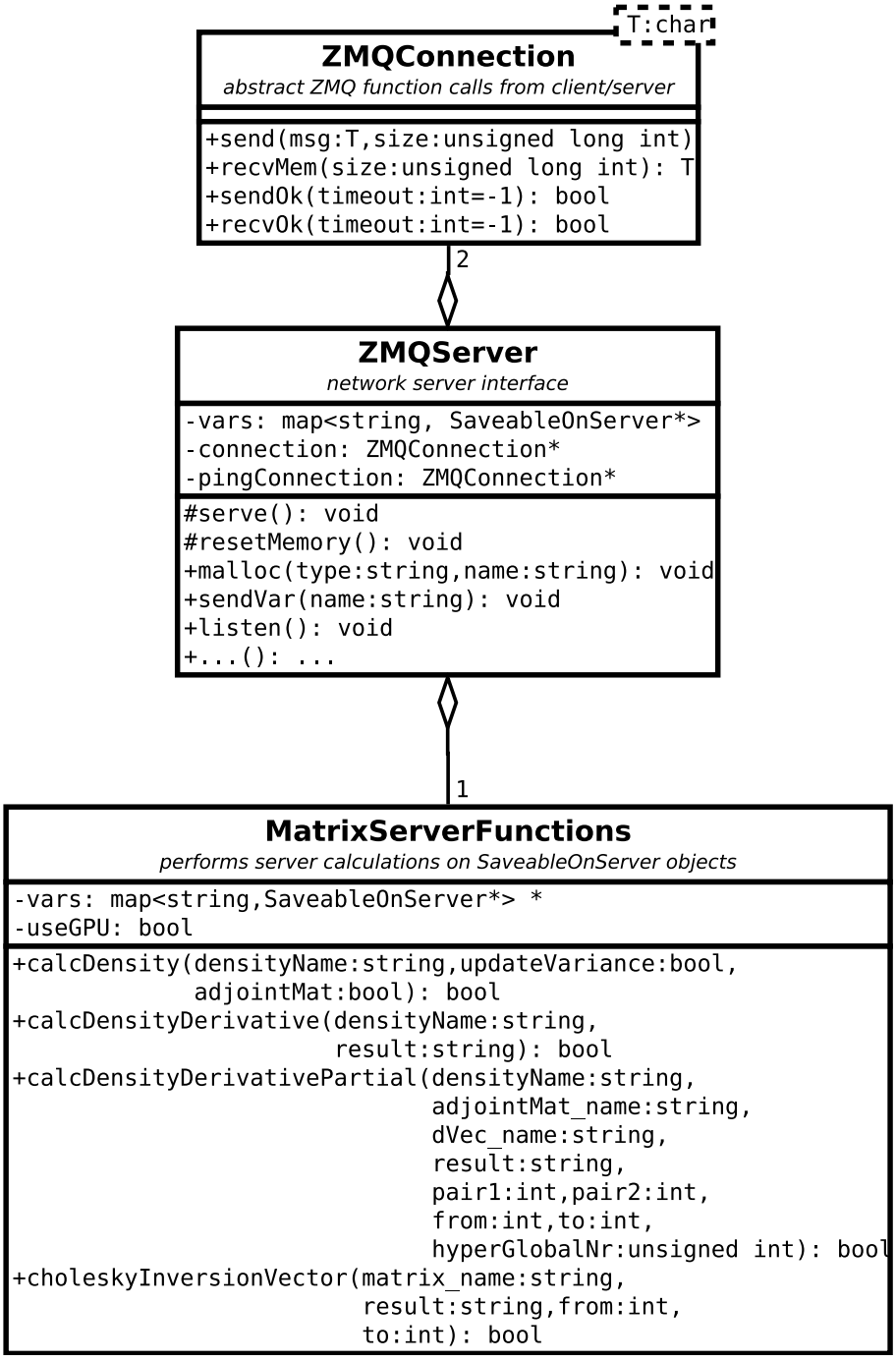


Abbildung 10: UML-Diagramm des Server-Interface. Der Server wird mit der Funktion `listen()` gestartet

4.1.4 Kommunikationsprotokoll

Ähnlich zu anderen textbasierten Protokollen (z.B. STOMP - Simple/Streaming Textoriented Messaging Protocol) wurde auch hier ein unidirektionales Kommunikationsverfahren auf Textbasis implementiert (der Client formuliert die Anfragen und der Server sendet entweder ein OK-/Fehlersignal oder serialisierte Objekte zurück). Dazu schickt der Client je nach Anfrage verschiedene Schlüsselwörter, die dann auf dem Server unterschiedliche Methodenaufrufe auslösen. Weitere Parameter, die zu solch einer Anfrage gehören, werden mit einem Unterstrich „_“ getrennt. So lautet bspw. der Befehl zum Senden eines verteilten Objektes „malloc_[name]“. Der Server reserviert nun in seinem internen Speicher eine Variable mit dem angegebenen Namen und sendet anschließend ein OK-Signal zurück. Anschließend werden die serialisierten Daten des Objektes über den Kommunikationskanal erwartet und nach Abschluss der Übertragung wird erneut ein OK-Signal an den Client zurück geschickt. Alle weiteren Befehle erfolgen analog dazu (ggf. mit mehr Parametern).
Anfragen, die mit dem Schlüsselwort „call“ starten, werden an die Instanz der Klasse `MatrixServerFunctions` weitergeleitet. Diese übernimmt die weitere Interpretation der Anfrage und führt die dort formulierten Rechenoperationen aus.

Ein Beispielaufruf:

$$\underbrace{\text{call\_MatrixServerFunctions}}_{\text{Schlüsselwort}}\_\underbrace{\text{additionRow}}_{\text{math. Operation}}\_\underbrace{\text{inputmatrix}}_{\text{Eingabevariable}}\_\underbrace{\text{outputmatrix}}_{\text{Ergebnis}}\_\underbrace{10}_{\text{von}}\_\underbrace{40}_{\text{bis}}$$

Die auf dem Server erzeugten Objekte sind anschließend über ihren Namen abrufbar. Das Ergebnis des vorherigen Beispielaufrufs würde dann als einfache Methode zur Datenverwaltung so vom Client angefragt werden:

$$\underbrace{\text{sendVar}}_{\text{Schlüsselwort}}\_\underbrace{\text{outputmatrix}}_{\text{Objekt}}$$

4.2 Einbindung in das Kriging

Das Kriging verwendet zur Steuerung des Programms die Klasse `config`. Die Einstellungen werden sowohl über Kommandozeilenparameter als auch über Eingabedateien getätigt. Wird bei Aufruf des Krigings nun der Parameter „- -useLocalNetwork“ mit übergeben, so wird die entsprechende Einstellung in dieser Klasse übernommen und auf „true“ gesetzt. Bei der Berechnung der Dichten (Funktion `calcDensity(...)`) wird anschließend auf diese Einstellung hin überprüft und ggf. auf die Verteilungsfunktionen zur Verwendung des Netzwerks zurückgegriffen. Ansonsten wird die Funktion lokal ausgeführt.
Werden nun große Matrizen verwendet (bspw. bei Gradient-Enhanced-Kriging-Ersatzmodellen), eignet sich für die Berechnung der Dichten die Netzwerkbeziehung 1:1, um die aufwändigen Operationen einer Cholesky-Zerlegung und Bestimmung einer adjungierten Matrix auf einen leistungsstarken Server außerhalb des Clusters auf einen Abteilungsrechner auszulagern. Da hier zudem die Instanz einer Dichtefunktion `DenstiyFunction` versendet werden kann statt der Matrizen und lediglich Endergebnisse wieder zurückgesendet werden müssen, die nur aus einzelnen `double`-Werten bestehen, kann so mit einer erheblich reduzierten Datenmenge gearbeitet werden, was die langsame Netzwerkverbindung zwischen Cluster und Abteilungsnetz ausgleicht.
Um Code-Duplizierungen zu vermeiden, verwendet der Server die gleiche Funktion `calcDensity (...)`, um mit einer entfernt liegenden Dichtefunktion im Netzwerk die Dichten bestimmen zu können. Dazu wird diese Funktion auf dem Server von der Klasse `MatrixServerFunctions` erneut aufgerufen und als lokal ausführbar interpretiert. Dies erleichtert die spätere Pflege des Quellcodes.

Der Algorithmus für die Berechnung der partiellen Ableitungen der Dichtefunktionen für die einzelnen Hyperparameter verläuft nach einem ähnlichem Prinzip. Da hier die partiellen Ableitungen der Hyperparameter unabhängig voneinander bestimmt werden, kann diese Operation in einem Netzwerk mit n Servern auf dem Cluster intern verteilt werden. Dies ist vor allem für Problemstellungen mit vielen Hyperparametern (bspw. bei Multifidelity-Kriging-Ersatzmodellen) interessant. Zur Berechnung der partiellen Ableitungen wird die adjungierte Matrix der Cholesky zerlegten Kovarianzmatrix benötigt. Da die Netzwerkgeschwindigkeit auf dem Cluster aber ausreichend hoch ist, kann die adjungierte Matrix vor Beginn der Berechnungen problemlos an alle Server verteilt werden.

Für die weitere Vorgehensweise wurde der parallelisierbare Teil der ursprünglichen Funktion `calcDensityDerivative(...)` in eine zweite Funktion `calcDensityDerivatePartial(...)` ausgelagert, welche als zusätzlichen Parameter die Anzahl an zu berechnenden Hyperparameter erwartet. Bei rein lokalem Betrieb wird diese Funktion von der ursprünglichen `calcDensityDerivate(...)` mit der vollen Anzahl an Hyperparametern aufgerufen. Stehen dem Programm mehrere Server zur Verfügung, so wird diese Funktion 1:n an alle angeschlossenen Server über das Rundlaufverfahren (auch Round Robin) verteilt. Dieses Vorgehen ist nachfolgend näher beschrieben.

Round Robin zur Verteilung an mehrere Server

Die theoretische Grundlage dieses Verfahrens ist in Abschnitt 3.5.3 bereits näher beschrieben worden. Der Client implementiert dies in der Klasse `ZMQConnection` mithilfe der Funktionen `distributeCallStack(...)` zur Verteilung der aktuell benötigten Operationen und `collectResult(...)` zur Zusammenführung der einzelnen Teilergebnisse. Um nun diesen Prozess in Gang zu setzen, wird das Problem in kleinere Pakete unterteilt, die dem in Abschnitt 4.1.4 beschrieben Protokoll entsprechen und Serveranfragen enthalten. Diese Pakete werden in einem Vektor gespeichert, der als Stack (Last in First out) verwaltet wird. Anschließend wird in der Funktion `distributeCallStack(...)` dieser Stack abgearbeitet. Dazu wird eine mit OpenMP parallelisierte for-Schleife ausgeführt. Die Anzahl der Threads entspricht der Anzahl der internen Server, sodass jeder parallel angesprochen werden kann. In dieser for-Schleife werden wiederum die Serveraufrufe des Stacks abgearbeitet, indem diese an die einzelnen Server verteilt werden. Schlägt eine Anfrage fehl, so wird die betreffende Verbindung vorzeitig geschlossen und der entsprechende Serverbefehl auf den Stack zurück gelegt. Zusätzlich wird mitverfolgt, welche Anfrage an welchen Server heraus gegangen ist und in welcher Reihenfolge die zu erwartenden Ergebnisse von der Funktion `collectResult(...)` zusammengeführt werden sollen.
Damit einzelne Teile einer Operation nun sinnvoll ausgelagert werden können, muss allerdings zuvor eine geeignete Paketgröße (Größe des Anteils am Gesamtproblem) bzw. Anzahl an zu berechnenden Hyperparametern bestimmt werden, die ein einzelner Server zu leisten hat. Da hier auf dem Cluster intern gearbeitet wird, kann für alle angeschlossenen Server die gleiche Rechenkapazität sowie die gleiche Netzwerkgeschwindigkeit angenommen werden. Dies vereinfacht den Verteilungsprozess.

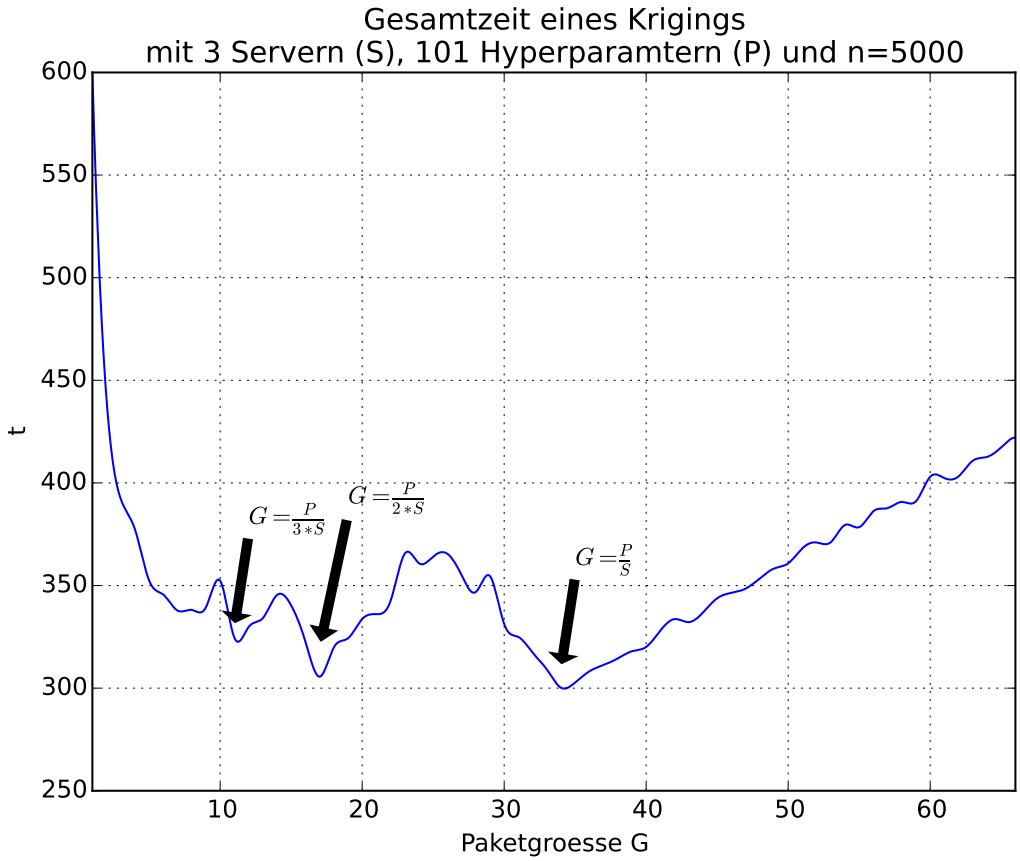


Abbildung 11: Benötigte Zeit für die Verteilung der Berechnung der Gradienten für die Dichtefunktionen in Abhängigkeit von der Paketgröße (101 Hyperparameter, 3 Server)

Die in Abbildung 11 dargestellten Messungen zeigen, dass eine gleiche Lastverteilung auf alle Server mit möglichst wenigen Aufrufen einen entscheidenden Einfluss auf die Gesamtgeschwindigkeit hat. Wird ein einzelner Server zu stark ausgelastet bzw. andere zu wenig, so verlängert sich die benötigte Rechendauer. Daher wird zur Berechnung der Paketgröße derzeit wie folgt vorgegangen:

```
1 int free_vars_amount; // known parameter
2 int server_amount = getClient()->getAllInternalServers().size();
3
4 int package_size = free_vars_amount / server_amount;
5
6 // avoid small leftovers
7 if (free_vars_amount%server_amount != 0)
8   package_size++;
```

Jeder interne Server erhält somit die gleiche Rechenlast und es werden kleine Reste vermieden, die als eigenständige Anfrage neu formuliert werden müssten.

4.3 Verbindungssicherheit und Fehlertoleranz

In der Abteilung AT-FUV finden häufig zahlreiche Optimierungsprozesse parallel statt. Es ist daher wichtig, einen Server vor parallelen Zugriffen zu schützen, um die Konsistenz der Daten zu erhalten. Darüber hinaus muss ein Client wissen, ob ein Server, der in der Client-Konfiguration angegeben wurde, überhaupt verfügbar ist. Diese Problemstellung wurde gelöst, indem der Client neben jeder Hauptverbindung zu einem Server jeweils eine weitere Instanz der Klasse `ZMQConnection` erzeugt, die die Verfügbarkeit der Server im Netzwerks überprüft. Dazu wird über diese zusätzlichen Verbindungen ein periodisches OK-Signal an die Server gesendet. Die Server besitzen wiederum auch jeweils eine zweite Verbindung neben der eigentlichen Hauptverbindung, die auf dieses Signal bzw. auf diesen Ping antwortet. Diese zusätzliche Instanz von `ZMQConnection` kann bei Bedarf auch mehreren Clients antworten, sofern mehrere auf einen Server zugreifen möchten. Ist der Server gerade nicht belegt, so wird auf den Ping ein OK-Signal zurückgesendet. Dies signalisiert dem Client, dass dieser nun die exklusiven Rechte auf dem Server für alle implementierten Operationen hat. Ist der angefragte Server aber gerade belegt, so wird ein Fehlersignal zurückgesendet, sodass der Client diesen für seine Algorithmen nicht weiter berücksichtigt. Auch ein Server, der aktuell offline ist, wird durch diesen periodischen Ping als nicht verfügbar identifiziert. Darüber hinaus ist es möglich, einen möglichen Absturz eines Servers frühzeitig zu erkennen, sodass eine eventuell fehlgeschlagene Anfrage erkannt und ggf. an einen anderen Server weitergeleitet werden kann.

Das Fehlersignal eines Servers kann zudem dazu verwendet werden, den Client über mögliche Schwierigkeiten bei bestimmten Abläufen auf dem Server zu informieren. Bedingung einer Cholesky-Zerlegung bspw. ist es, dass die zu zerlegende Matrix positiv definit ist. Im ursprünglichen Kriging-Code wird dies überprüft und im Fehlerfall eine Exception geworfen, auf die das Programm reagieren kann. Tritt dieser Fall aber auf einem Server ein (bspw. bei einer 1:1 Verbindung, wo die zu zerlegende Matrix erst auf dem Server erzeugt wird), so muss diese Exception an den Client weitergegeben werden, damit dort das Programm entsprechend darauf reagieren kann. Dazu wird die geworfene Exception auf dem Server abgefangen und über die Hauptverbindung mittels eines spezifizierten Fehlersignals an den Client versendet.

4.4 Transparenz und Fehlersemantik

Allgemein ist das Ziel eines verteilten Systems, dass die Geschwindigkeit und Verfügbarkeit des gesamten Systems erhöht wird. Es wird dabei versucht, dass die Anwendungsprogramme so wenig wie möglich von der eigentlichen Arbeit des Systems mitbekommen, dass die meisten Vorgänge innerhalb dieses Systems also transparent sind. Im Zusammenhang von verteilten Systemen gibt es verschiedene Ausprägungen der Transparenz, wovon die wichtigsten hier kurz erläutert und auf ihre Bedeutung bzgl. des hier entwickelten Systems hin untersucht werden (die Definitionen der verschiedenen Transparenzbegriffe sind aus [10], S. 14 ff. entnommen). Zudem wird die hier umgesetzte Fehlersemantik des Systems untersucht, d.h. das Verhalten des Systems bei Auftreten eines Fehlers im Zusammenhang mit der Netzwerkkommunikation (Definitionen nach [11]).

4.4.1 Zugriffstransparenz

Der Zugriff auf Dienste erfolgt stets in der selben Weise, unabhängig davon, ob diese lokal oder auf einem entfernten Rechner zur Verfügung stehen. Dem Netzwerkinterface ist der Ort aller Server bekannt. Es spielt allerdings keine Rolle, ob diese lokal oder in dem Netzwerk ausgeführt werden. Daher ist die Zugriffstransparenz auf allen Ebenen gegeben. Dies ist wichtig, um die Wartbarkeit des Quellcodes zu erhalten.

4.4.2 Ortstransparenz

Der Ort einer Ressource oder eines Dienstes ist der Anwendung nicht bekannt. Je nach Sichtweise ist diese Form der Transparenz gegeben. Das eigentliche Kriging greift auf das Netzwerkinterface zu, welches die Netzwerkverwaltung vollständig abstrahiert. Diesem ist nur die Anzahl der Server bekannt. Daher ist der Ort der Ressourcen bzw. der Server für das Kriging nicht relevant, sondern nur für das Netzwerkinterface.

4.4.3 Skalierungstransparenz

Systeme können flexibel auf Erweiterungen bzw. Änderungen der Hard- und Softwarebasis reagieren. Durch den nebenläufigen Ping, welcher die Verfügbarkeit aller bekannten Server periodisch überprüft, kann die Skalierungstransparenz für dieses System sichergestellt werden. Ausfallende Server können während der Laufzeit entfernt und neu verfügbare Server dem Verteilungsprozess hinzugefügt werden, ohne dass dies Auswirkungen auf das Kriging hat. Dies ist für die Stabilität des Programms enorm wichtig und kann dazu verwendet werden, dynamisch weitere Ressourcen in den Programmablauf mit einzubinden.

4.4.4 Migrationstransparenz

Aus verschiedenen Gründen kann es nötig sein, einen Prozess oder Daten von einem Rechner auf einen anderen zu transferieren. Dies geschieht für die Anwendung verdeckt und unbemerkt. Aktuell ist die Migrationstransparenz für das Kriging nicht gegeben, da die Verwaltung der Verteilung der eigenen Daten nicht von dem Netzwerkinterface, sondern vom Kriging selbst übernommen wird. Das Netzwerkinterface ist nur für die Verwaltung des Netzwerks zuständig. Es muss vor jeder Auslagerung von Rechenoperationen daher überprüft werden, ob auf diesem Server bereits alle notwendigen Daten vorhanden sind.

4.4.5 Fehler- und Ausfalltransparenz

Fehler oder Ausfälle im System sollten für die Anwendung maskiert werden. Im Rahmen des Krigings ist es zudem wichtig, dass Verarbeitungsfehler der Server bzgl. einzelner mathematischer Operationen an den Client weitergegeben werden. Vollständige Ausfalltransparenz ist für

das Kriging in diesem Kontext nicht gegeben, da bspw. der Ausfall eines Servers im Rahmen einer 1:1 Beziehung erkannt werden muss, um wenigstens lokal weiterarbeiten zu können. Ausfälle innerhalb von Verteilungsprozessen an viele Server werden allerdings transparent zu dem Kriging gehalten und vom Netzwerkinterface abgefangen. Erst bei vollständigem Ausfall des Netzwerks wird der Client informiert, sodass dieser geeignet weiterarbeiten kann.

4.4.6 Fehlersemantik

Nach [11] gibt es verschiedene Definitionen zu Fehlersemantik in verteilten Systemen - dem Vorgehen des Programms also im Fehlerfall:

- Maybe: eine Operation wird durchgeführt, ohne dass deren Erfolg bekannt bzw. relevant ist
- At most once: eine Anfrage wird maximal einmal versendet. Anfragen, deren Ausgang nicht sicher ist, werden nicht neu formuliert. Es kann hier vorkommen, dass Anfragen vollständig verloren gehen und ein Server deshalb eventuell gar nicht arbeitet
- At least once: ist der Ausgang einer Anfrage unbekannt, so wird diese solange erneut gesendet, bis diese erfolgreich durchgeführt wurde. Es kann allerdings passieren, dass Operationen mehrfach auf einem Server durchgeführt werden.
- Exactly once: eine Kombination aus „At most once“ und „At least once“ - es wird versucht, dass eine Anfrage genau einmal auf dem Server durchgeführt wird, ohne dabei den Verlust bzw. Mehrfachausführung einer Operation auf einem Server in Kauf zu nehmen.

In dem hier betrachteten Kontext ist es allerdings nur relevant, ob und wie oft ein Ergebnis von einem Server zurück gesendet wird. Eine Ausführung einer Operation auf einem Server, die nichts zurückliefert, hat auf den Zustand des Netzwerks keinerlei Auswirkungen, sodass auch der Zeitpunkt, an dem ein Server ausfallen sollte, irrelevant ist. Ein Server, der kein Ergebnis liefert, wird für nachfolgende Verteilungsoperationen nicht weiter berücksichtigt. Daher werden die nachfolgenden Betrachtungen lediglich für das Ergebnis einer Operation, welches von einem Server zurück gesendet wurde, durchgeführt.
Schlägt eine Anfrage zu einem Server in einer 1:1 Beziehung fehl und liefert kein Ergebnis, so wird diese Operation nicht erneut versendet und stattdessen lokal berechnet. Dies erspart ggf. unnötige Wartezeiten, sollte ein Server vollständig ausgefallen sein. Es wurde so Fehlersemantik

„At most once“ für 1:1 Verbindungen zu einem einzigen Server umgesetzt.

Fällt ein Server in einer 1:n Beziehung an einer beliebigen Stelle der Verarbeitung aus, so wird eine bereits versendete Anfrage an diesen Server nach der Semantik „At most once“ nicht neu übertragen. Eine fehlgeschlagene Anfrage kann aber an einen anderen Server gesendet werden, sodass für die Anfrage selbst die Semantik „At least once“ gilt. Es wird so erreicht, dass eine Anfrage genau einmal, also nach dem Prinzip „Exactly once“ in einer 1:n Beziehung zum Ergebnis beiträgt.

4.5 Qualität der Software durch Testing

Da diese Arbeit eng mit dem bestehendem Kriging-Code verzahnt ist und dieser zusätzlich angepasst werden musste, waren umfangreiche Tests zur Funktion des Kernalgorithmus notwendig. Dazu wurden nach jeder Anpassung des Codes Regressionstests zur Überprüfung durchgeführt, um eventuelle Programmfehler aufzuspüren. Um allerdings Laufzeit- bzw. Systemfehler vermeiden bzw. detektieren zu können - bspw. invalide Speicherzugriffe - ist der Einsatz von weiterer Analysesoftware notwendig, die ein fehlerhaftes Programmverhalten erkennen kann. Insbesondere bei systemnahen Programmiersprachen wie C/C++ kann es schnell zu Fehlern in der Programmierung kommen. Zur Analyse wurden daher folgende Programme verwendet:

- GNU Project Debugger (GDB) - dieses Tool erlaubt Einblicke in den Programmablauf und kann bei Fehlern einen so genannten Stacktrace einsehen (Hierarchie von Funktionsaufrufen) (vgl. [12])
- Valgrind - ähnlich zum GDB ist auch hier die Anzeige eines Stacktrace im Fehlerfall möglich. Dieses Programm benötigt allerdings deutlich länger zur Laufzeitanalyse, erkennt dafür aber auch Speicherbereiche, die im Programmverlauf nicht korrekt beseitigt werden und daher unnötigen Speicherplatz verbrauchen (vgl. [13])

Vollständiges Testen ist in der Regel nicht möglich. Es können prinzipiell auch keine Softwarefehler ausgeschlossen werden. Durch den Einsatz dieser Programme und weiterer geeigneter Softwaretests war es allerdings möglich, die Anzahl potenzieller Fehler deutlich zu senken und mehr Stabilität zu gewährleisten.

5 Auswertung durch Geschwindigkeitsmessungen

In diesem Kapitel werden verschiedene Szenarien betrachtet, wie sie im typischen Optimierungsprozess in der Abteilung AT-FUV vorkommen. Die Ausführungsgeschwindigkeiten einzelner Operationen, wie die Durchführung einer Cholesky-Zerlegung und Berechnung der partiellen Ableitungen, die im Rahmen der Arbeit ausgelagert wurden, werden dabei untersucht und bzgl. verschiedener Aspekte verglichen. Bei den Messungen wurde zwischen den zwei wesentlichen Anwendungsfällen (große Matrizen vs. große Anzahl an Hyperparametern) unterschieden und darüber hinaus die Rechengeschwindigkeit der GPU im Vergleich zu einer CPU verglichen. Die durchgeführten Messungen wurden zuvor spezifiziert wie folgt zusammengetragen:

Nr.	min(n)	max(n)	$n+$[1]	IT[2]	HP[3]	Beschreibung
1	28000	28000	0	10	50	Machbarkeit bei großen Matrizen
2	100	2000	100	10	20	OpenMPMatrix lokal
3	1000	28000	2000	10	20	MKL lokal
4	1000	28000	2000	10	20	CUDA lokal
5	1000	28000	2000	10	20	MKL Clusterknoten
6	500	5000	500	10	500	MKL Clusterknoten
7	100	2000	100	10	20	OpenMPMatrix Cluster $\rightarrow$ lokal
8	1000	28000	2000	10	20	MKL Cluster $\rightarrow$ lokal
9	1000	28000	2000	10	20	CUDA Cluster $\rightarrow$ lokal
10	500	5000	500	10	500	MKL Cluster 1:n

Die Ergebnisse dieser Messungen werden nachfolgend miteinander verglichen und auf ihre Bedeutung hin untersucht. Sowohl auf der CPU als auch auf der GPU wurden - wenn nicht anders angegeben - jeweils spezielle Routinen für lineare Algebra verwendet (Basic Linear Algebra Subprograms - auch BLAS-Routinen), um diese besser vergleichen zu können (Intel®

[1] Schrittweite
[2] Anzahl an Wiederholungen (Iterationen)
[3] Anzahl an Hyperparameter

Math Kernel Library (MKL) auf der CPU, Nvidia® CUDA auf der GPU). Für die lokalen Berechnungen und den leistungsstarken Server, der eine GPU enthält, wurden auf einem Rechner folgende Prozessoren eingesetzt:

- CPU: 2x Intel® Xeon® E5-2650 v3 mit 2,3 GHz (ca. 2x 368 Gflop/s)
- GPU: 1x Nvidia® Quadro K6000 mit 601 MHz (ca. 1,1 Tflop/s bei dieser Taktrate)

In den nachfolgenden Abschnitten wird auch die Ausführungsdauer eines Krigings auf einem Clusterknoten untersucht. Ein Clusterknoten besitzt folgende CPU:

- 2x Intel® Xeon® E5-2695 v2 mit 2.4 GHz (ca. 2x 230 Gflop/s)

5.1 Vergleich zwischen CPU und GPU

Um zu überprüfen, ob sich der Einsatz eines Servers mit einer GPU lohnt, wurden im Vorfeld vergleichbare Messungen lokal auf einem Rechner zuerst auf der CPU und anschließend auf der GPU ausgeführt (Spez. Nr. 3 & 4). Die Abbildungen 12 und 13 zeigen die Zeit und den mit der GPU erreichten Speedup für unterschiedliche Matrixgrößen einerseits für die Gesamtberechnung eines Krigings und andererseits für die dort durchgeführte Berechnung einer Dichtefunktion, die hauptsächlich aus einer Cholesky-Zerlegung besteht. Der hier erreichte Speedup für die Berechnung einer Dichtefunktion liegt je nach Matrixgröße zwischen 1,3 und 1,4, der Gesamtspeedup etwas niedriger zwischen 1,1 und 1,28. Dies entspricht etwa dem Leistungsverhältnis zwischen CPU und GPU.

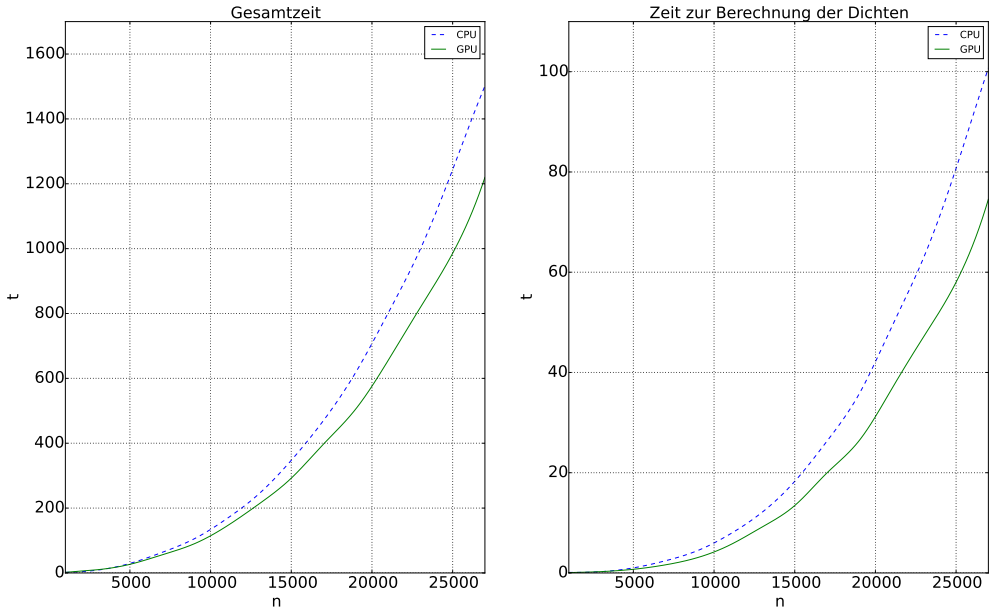


Abbildung 12: Geschwindigkeitsvergleich CPU/GPU über Matrixgröße n nach Spez. 3/4 links: Gesamtzeit; rechts: Zeit zur Berechnung einer Dichtefunktion (hauptsächlich Cholesky-Zerlegung)

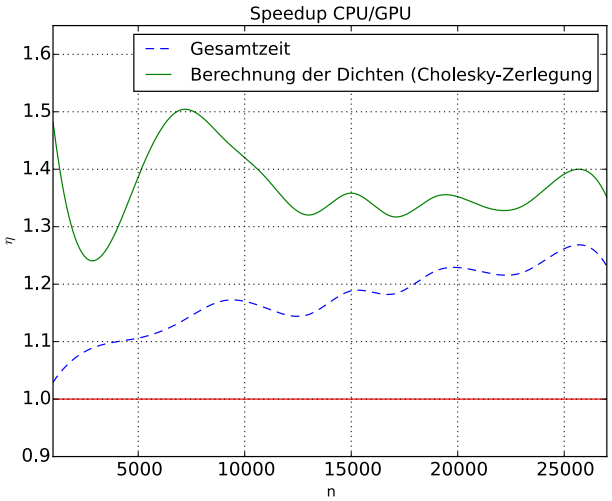


Abbildung 13: Speedup η eines Krigings bzw. einzelner Operationen von CPU/GPU über Matrixgröße n nach Spez. 3/4)

5.2 Messungen für eine 1:1 Verbindungen zwischen einem Clusterknoten und einem Abteilungsrechner

Bei Problemstellungen mit großen Matrizen (bspw. bei Gradient-Enhanced-Kriging-Ersatzmodellen) eignet sich die Verwendung einer 1:1 Verbindung zu einem Rechner mit einer GPU. Nachfolgend wird einerseits der Zeitverlust untersucht, der allein durch die Netzwerkkommunikation zu einem GPU-Server entsteht. Andererseits wird der Speedup ermittelt, der durch so eine Verbindung im Vergleich zur alleinigen Ausführung auf dem Clusterknoten erzielt werden kann.

5.2.1 Zeitverlust durch Netzwerkkommunikation

Ein wichtiger Aspekt, der untersucht wurde, ist der Zeitverlust, der bei der Netzwerkkommunikation auftritt. Dazu wurden zwei identische Testfälle zuerst lokal auf dem Rechner mit der CPU bzw. GPU durchgeführt, der zu Beginn dieses Kapitels beschrieben wurde (Spez. 3 bzw. 4). Anschließend wurde dieser Rechner als Server verwendet und der Testfall auf einem Clusterknoten gestartet, der dann auf diesen Server zugreift (Spez. 8 bzw. 9). So konnte ermittelt werden, wie groß der eigentliche Zeitverlust ist, der für die Benutzung des Netzwerkes bei einer 1:1 Beziehung anfällt. Dieser Zeitverlust ist in Abbildung 14 für die Gesamtzeit eines Krigings bei unterschiedlicher zugrunde liegender Matrizengröße abgebildet. Die hier verwendeten Testfälle sind nach den Spezifikationsnummern 4 und 9 durchgeführt worden. Bei der Verwendung eines Servers in einer 1:1 Beziehung ist die Netzwerkkommunikation deutlich reduziert, indem statt der Matrizen die Instanzen der Dichtefunktionen, mit denen die Berechnungen stattfinden, serialisiert und über das Netzwerk versendet werden. Die Matrizen werden anschließend auf dem Server aufgebaut. Anhand der Abbildung 14 lässt sich daher erkennen, dass vor allem bei Problemen mit großen Matrizen die Netzwerkkommunikation kaum noch ins Gewicht fällt und vernachlässigbar gering wird.

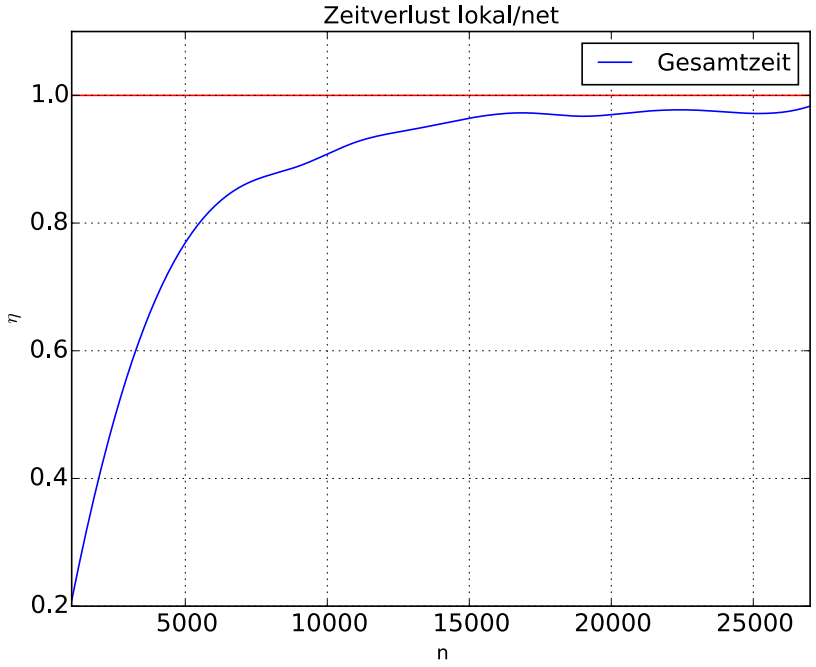


Abbildung 14: Zeitverlust η durch Netzwerkkommunikation in einer 1:1 Client-Server-Beziehung über Matrixgröße n

5.2.2 Vergleich der Ausführungszeiten Netzwerk/GPU vs. lokal/CPU

In der Regel wird eine Optimierung auf dem Cluster durchgeführt. Daher muss sich das neue System, in welchem das Kriging auf einem Clusterknoten ausgeführt wird und einen Rechner im Abteilungsnetzwerk als Server verwendet, an der bisherigen Ausführungszeit eines Krigings auf einem Clusterknoten messen können. Dieser Vergleich ist in Abbildung 15 abgebildet (Spez. Nr. 4 & 5). Es lässt sich erkennen, dass sich bei derzeitiger Infrastruktur vor allem bei großen Problemen der Einsatz einer externen GPU zu Rechenzwecken lohnt und eine Halbierung der Gesamtrechenzeit im Vergleich zum herkömmlichen Kriging erreicht werden kann. Lediglich die Berechnung der partiellen Ableitungen ist geringfügig langsamer im Vergleich zum Clusterknoten. Dies kann allerdings daran liegen, dass die Prozessoren des Clusterknotens diesen Schritt intern besser parallelisieren können. Die Durchführung einer Cholesky-Zerlegung im Rahmen der Berechnung einer Dichtefunktion kann allerdings hier

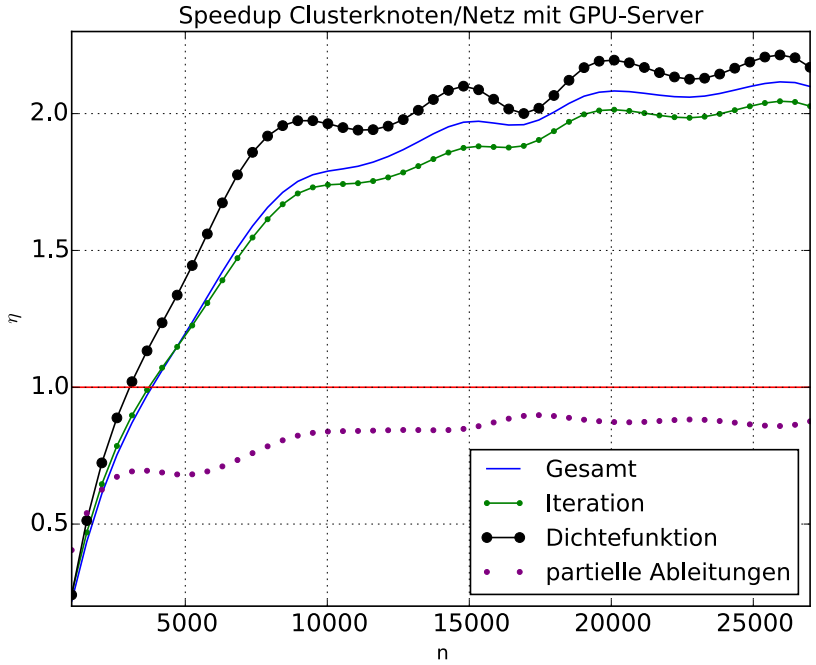


Abbildung 15: Speedup η einer 1:1 Verbindung im Vergleich zur herkömmlichen Ausführung eines Krigings auf einem Clusterknoten über Matrixgröße n nach Spez. 4 und 5

deutlich beschleunigt werden. Daher ist der Einsatz einer 1:1 Verbindung zur Integration eines Servers mit einer GPU bei Problemen mit großen Matrizen gut geeignet.

5.3 Messungen für ein 1:n Netzwerk auf dem Cluster intern

In den vorherigen Abschnitten wurden Probleme mit großen zugrunde liegenden Matrizen bei relativ geringer Anzahl an Hyperparametern untersucht. Dieser Abschnitt befasst sich mit dem Vergleich des umgekehrten Falls: moderate Matrizengrößen bei vielen Hyperparametern (bspw. bei Multifidelity-Kriging-Ersatzmodellen, Spez. 6 & 10). Dazu wurde der zu betrachtende Testfall für 0 (lokale Ausführung) bis 7 Server durchgeführt, wobei jeder Server auf einem anderen Clusterknoten ausgeführt wurde. Die Abbildungen 16, 17 und 18 präsentieren dabei

den erreichten Speedup bei unterschiedlichen Matrizengrößen und unterschiedlicher Anzahl an Servern für die Gesamtzeit, die Berechnung einer Dichtefunktion und die Bestimmung der partiellen Ableitung (jeweils für eine Iteration).

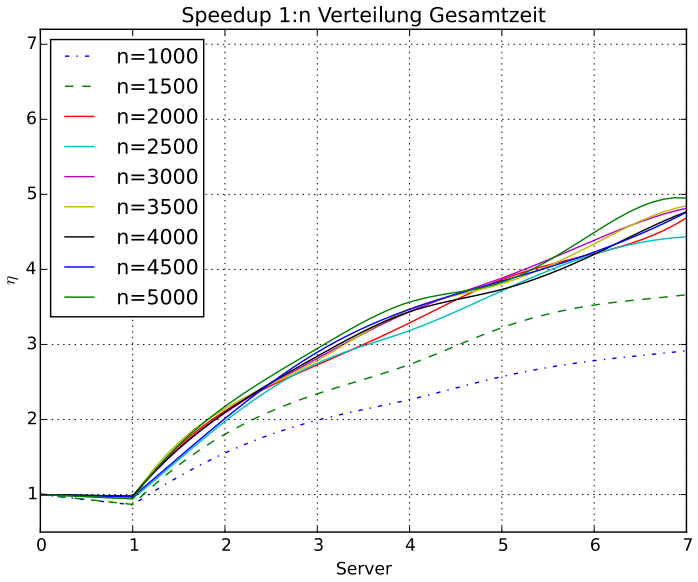


Abbildung 16: Speedup η der Gesamtzeit in einer 1:n Client-Server-Beziehung für verschiedene Matrixgrößen n und Anzahl an Servern nach Spez. 6 und 10

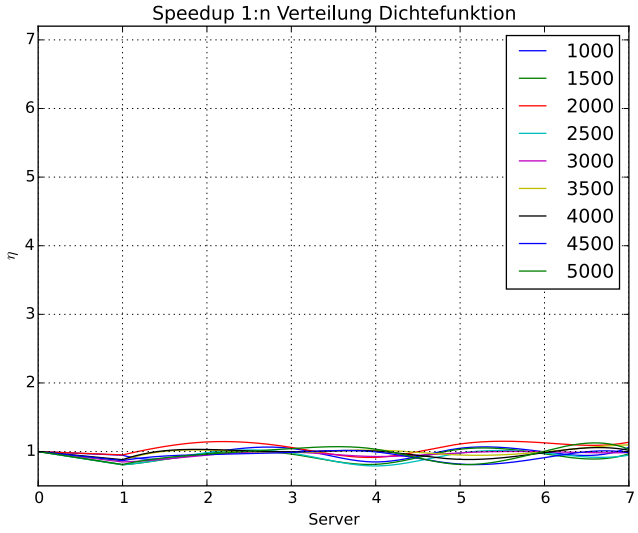


Abbildung 17: Speedup η der Berechnung einer Dichtefunktion in einer 1:n Client-Server-Beziehung für verschiedene Matrixgrößen n und Anzahl an Servern nach Spez. 6 und 10

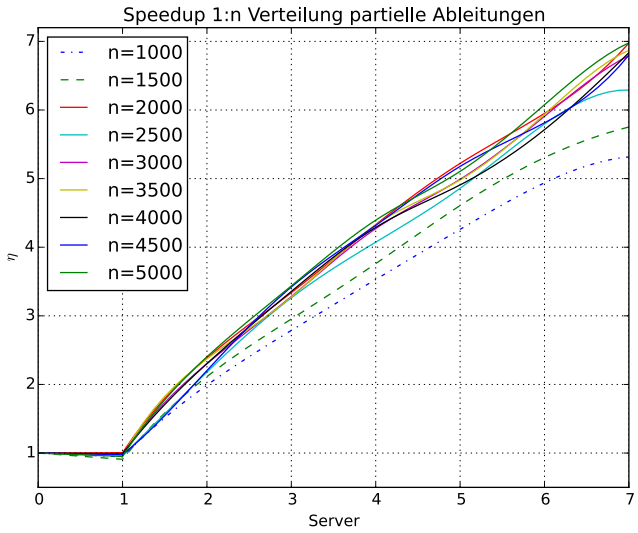


Abbildung 18: Speedup η der Berechnung der Gradienten in einer 1:n Client-Server-Beziehung für verschiedene Matrixgrößen n und Anzahl an Servern nach Spez. 6 und 10

Während bei der Verwendung eines Netzwerks in einer 1:n Beziehung im Vergleich zur Ausführung auf einem einzigen Clusterknoten bei der Berechnung einer Dichtefunktion innerhalb einer Iteration kein Speedup erreicht werden kann (da keine Verteilung von Rechnungen vorgesehen), wirkt sich eine Verteilung bei der Berechnung der partiellen Ableitungen umso mehr auf die Gesamtzeit aus. Abbildung 18 zeigt hier eindrucksvoll, dass der Speedup beinahe linear mit der Anzahl an verwendeten Servern wächst. Dies wirkt sich insgesamt deutlich auf den Speedup der Gesamtzeit (vgl. Abbildung 16) aus. Es lässt sich außerdem erkennen, dass dieser bei ausreichender Anzahl an Hyperparametern beinahe unabhängig von der zugrunde liegenden Matrixgröße ist. Zudem ist der Zeitverlust, der bei der Netzwerkkommunikation auf dem Cluster intern auftritt, vernachlässigbar gering. Dies lässt sich in allen drei Abbildungen im Vergleich zwischen dem Speedup bei lokaler Ausführung (0 Server) und der Beteiligung eines Servers an den Berechnungen (1 Server) erkennen. Die Verwendung vieler Server in einer 1:n Beziehung ist daher bestens für Probleme mit einer großen Anzahl an Hyperparametern geeignet.

6 Ergebnis, Zusammenfassung und Ausblick

Ausgangslage dieser Arbeit war das Kriging-Verfahren im Rahmen des Optimierungsprozess in der Abteilung AT-FUV, welches durch Auslagerung und Verteilung von rechenintensiven Operationen in einem Netzwerk beschleunigt werden sollte. Als Messaging-Middleware wurde auf die Bibliothek ZeroMQ zurückgegriffen. Die bereits bestehenden Klassen und Funktionen wurden so modifiziert, dass diese nun auch verteilt in einem Netzwerk genutzt werden können. Für die Serialisierung und Deserialisierung der zugrunde liegenden Daten wurden jeweils eigene Lösungen entwickelt, da die Datenstrukturen vergleichsweise einfach serialisiert und deserialisiert werden können und der Einsatz weiterer Software demnach unnötigen Mehraufwand sowohl in der Entwicklung als auch in der Datenhaltung selbst mit sich gebracht hätte.

Es wurde ein Netzwerkinterface des Programms bestehend aus vier verschiedenen Klassen auf Grundlage des in Abschnitt 1.4 dargestellten Konzepts weiterentwickelt und vollständig in das Programm integriert. Es wurden aufbauend auf dem mathematischen Hintergrund des Verfahrens verschiedene Möglichkeiten zur Beschleunigung des Programms in Netzwerken mit unterschiedlichen Topologien untersucht. Dabei wurden zwei wesentliche Anwendungsszenarien anhand der möglichen Ersatzmodelle (Gradient-Enhanced- und Multifidelity-Kriging-Ersatzmodelle) mit unterschiedlichen Lösungen identifiziert:

1. große Matrixgröße, wenige Hyperparameter
2. kleine Matrixgröße, viele Hyperparameter

Es hat sich gezeigt, dass sich für den ersten Anwendungsfall eine Auslagerung der Berechnung der Dichtefunktionen, die auf einer Cholesky-Zerlegung aufbauen, auf einen einzigen Server (1:1 Beziehung) mit einer leistungsstarken GPU anbietet. Dadurch konnte mit der derzeitigen Infrastruktur bei ausreichender zugrunde liegender Matrixgröße ein Speedup der Gesamtdauer um den Faktor 2 erreicht werden.
Messungen im Rahmen des zweiten Szenarios haben gezeigt, dass sich für diesen Anwendungsfall eine Auslagerung der Berechnung der partiellen Ableitungen der Dichtefunktionen für alle Hyperparameter auf mehrere Server (1:n Beziehung) lohnt. Da jede partielle Ableitung

unabhängig von den anderen berechnet werden kann, lässt sich diese Operation besonders gut auf mehrere Server verteilen. So konnte für diese Operation bei n Servern fast ein Speedup von n erreicht werden, der der Gesamtausführungszeit liegt etwas darunter.
Für die gängigen Anwendungsfälle konnte so die Gesamtdauer eines Krigings teils deutlich reduziert werden.

Ein dritter Anwendungsfall (große Matrixgröße, viele Hyperparameter) kann durchaus vorkommen, tritt aber eher selten auf. Für diesen wäre eine Mischform der Netzwerkbeziehungen 1:1 und 1:n für verschiedene Operationen zwar durchaus denkbar - der zu erwartende Nutzen ist aufgrund der aktuellen Infrastruktur der Abteilung AT-FUV allerdings eher gering. Da dieser Anwendungsfall zudem eher selten auftritt, wurde dieser im Rahmen dieser Arbeit nicht weiter betrachtet.

Die Möglichkeit zur Integration eines Brokers/Balancers (1:1:n Beziehung) wurde ebenfalls vorerst nicht berücksichtigt. Langfristig wäre jedoch auch eine Umsetzung dieser Prinzipien bei geeigneter Infrastruktur - bspw. durch höhere Netzwerkgeschwindigkeiten oder auch Integration von GPUs auf dem Cluster - möglich. Dies könnte die Gesamtdauer des Kriging-Verfahrens noch einmal reduzieren.
Darüber hinaus wurde die Verteilung eines Krigings bisher nur separat und vom restlichen Optimierungsprozess losgelöst getestet. Hier gilt es, noch weitere Untersuchungen anzustellen, um zu ermitteln, wie groß der Speedup für den gesamten Optimierungsprozess ist.

Literaturverzeichnis

[1] Aainsqatsi, K.: *Turbofan_operation*. Online unter https://commons.wikimedia.org/wiki/File:Turbofan_operation_lbp.svg, 2008

[2] Deutsches Zentrum für Luft- und Raumfahrt e.V. - Institut für Antriebstechnik: *Adjungierter Strömungslöser*. Online unter http://www.dlr.de/at/desktopdefault.aspx/tabid-7167/11896_read-28084/, Stand: 2016

[3] Küppers, F.: *Auslagerung rechenintensiver Matrizenoperationen auf die GPU*. Praxisbericht, April 2016. – Duale Hochschule Baden-Württemberg Mannheim bzw. Deutsches Zentrum für Luft- und Raumfahrt e.V. - Institut für Antriebstechnik

[4] Schmitz, A.: *Entwicklung eines objektorientierten und parallelisierten Gradient Enhanced Kriging Ersatzmodells*, Fernuniversität Hagen, Masterarbeit, 2013

[5] Murray, I.: *Differentiation of the Cholesky decomposition*. Online unter http://homepages.inf.ed.ac.uk/imurray2/pub/16choldiff/choldiff.pdf, Februar 2016

[6] Smith, S.P.: *A Tutorial on Simplicity and computational Differentation for Statisticians*. Online unter http://www.johnrodgerssmith.com/StephenPapers/nonad3.pdf : UC Davis Physics Department, 2000

[7] Press, W.H. ; Teukolsky, S.A. ; Vetterling, W.T. ; Flannery, B.P.: *Numerical Recipes - The Art of Scientific Computing*. Cambridge University Press, 2007

[8] Williams, Anthony ; Escriba, Vicente J. B.: *Boost Thread*. Online unter http://www.boost.org/doc/libs/1_55_0/doc/html/thread.html, Oktober 2013

[9] Kornmayer, H.: *Parallele Programmierung*. 2015. – Vorlesung Paralleles Programmieren - unveröffentlicht

[10] Bengel, G.: *Grundkurs Verteilte Systeme*. Vieweg, 2004

[11] Colgen, R.: *Grundlagen Verteilte Systeme*. 2016. – Vorlesung Verteilte Systeme - unveröffentlicht

[12] GDB Developers: *GDB: The GNU Project Debugger*. Online unter https://www.gnu.org/software/gdb/, August 2016

[13] VALGRIND™ DEVELOPERS: *About Valgrind*. Online unter http://www.valgrind.org/info/about.html, August 2016

[14] GRIEWANK, A. ; WALTHER, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008