ERSATZMODELLTRAINING FÜR TURBOMASCHINENOPTIMIERUNGEN: ENTWICKLUNG EINES VERTEILTEN SYSTEMS ZUR AUSLAGERUNG RECHENINTENSIVER ALGORITHMEN AUF GPUS

F. Küppers, A. Schmitz
Deutsches Zentrum für Luft- und Raumfahrt e.V., Köln, Deutschland

Zusammenfassung

Eine große Rolle in der Entwicklung und dem Design neuer Konzepte für Turbomaschinen spielen moderne Optimierungs- und Simulationsverfahren. Das Institut für Antriebstechnik des DLR verwendet in diesem Rahmen Multifidelity- und Gradient-Enhanced-Kriging-Ersatzmodelle, welche ein aufwändiges Training benötigen. In solch einem Training sind verschiedene Matrizenoperationen der Komplexität $O(n^3)$ notwendig. Bei aufwändigen Problemstellungen kann das Kriging-Training allerdings zum Flaschenhals einer Optimierung werden. Mittelfristiges Ziel ist es, solch ein Training auch für sehr große Matrizen (~20.000^2) in annehmbarer Zeit umzusetzen.

Für Optimierungen wird im Institut für Antriebstechnik des DLR meistens ein eigenes Rechencluster mit zahlreichen Knoten verwendet. Das Kriging-Training lässt sich aber nur teilweise auf mehrere Knoten verteilen. Die höchste Rechenleistung bei gleichzeitig bestem Preis-/Leistungsverhältnis bieten derzeit GPGPUs (General Purpose Computation on Graphics Processing Unit, nachfolgend GPU). Diese sind allerdings auf dem Rechencluster derzeit nicht vorhanden. Aus diesem Grund wurde für das Kriging-Training ein externer Server zur Verfügung gestellt, der solch eine GPU verwendet. Zur Auslagerung der wichtigsten Matrix-operationen in einer 1:1 Client-Server-Verbindung war darüber hinaus die Entwicklung einer geeigneten Netz-werkschnittstelle notwendig. Bei derzeitiger Infrastruktur konnte so ein Speedup um den Faktor 2 erreicht werden.

Die gleiche Netzwerkschnittstelle wurde in einem zweiten Schritt dazu verwendet, um die parallelisierbaren Berechnungen des Kriging-Trainings unabhängig von der Verwendung einer GPU auf verschiedene Cluster-knoten zu verteilen. Mit solch einer Verteilung konnte ein Speedup erreicht werden, der in begrenztem Maße beinahe linear mit der Anzahl an Clusterknoten wächst.

1. EINLEITUNG

Der Optimierungsprozess, welcher im Institut für Antriebstechnik des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR) eingesetzt wird, besteht aus einem Masterprozess und zahlreichen Slaveprozessen. Die Slaveprozesse bestehen aus einer Prozesskette zahlreicher Programme, wie z.B. Strömungslöser, FEM, etc. Diese erhalten vom Master einen vielversprechenden Parametersatz (z.B. Geometrieparameter) und berechnen daraus die zu minimierende Zielfunktion (bspw. Lärmbelastung, Kraftstoffverbrauch, etc.) und verschiedene Nebenbedingungen. Der Master verwendet zur Berechnung dieser Parametersätze Kriging-Ersatzmodelle (vgl. [1]), welche im Rahmen dieses Prozesses trainiert werden. Dieses Ersatzmodelltraining enthält symmetrische und positiv definite Kovarianzmatrizen der Größe $n\,x\,n$. Für die nachfolgend verwendeten Matrizen R gilt allgemein:

(1) $\vec{v} * R * \vec{v} > 0 \;\forall\, \vec{v} \neq 0$

(2) $R = R^T$

Die hier verwendeten Krigings Ersatzmodelle verwenden zahlreiche Hyperparameter, welche die Vorhersagegenauigkeit des Modells maßgeblich beeinflussen. Um die bestmögliche Vorhersagegenauigkeit zu erreichen, muss ein rechenintensives Training durchgeführt werden, welches den optimalen Satz an Hyperparametern finden soll. Hierfür wird die Maximum Likelihood Methode angewandt (vgl. [1], S. 40 ff.). Der Ablauf der Maximum Likelihood Methode gestaltet sich wie folgt:

1) Mit einem initialen Hyperparametersatz wird der Likelihood N bestimmt. Im Rahmen dieses Prozesses wird eine Cholesky-Zerlegung durchgeführt, welche mit einem Aufwand von $\frac{1}{6}n^3$ vor allem für große Matrizen sehr rechenintensiv ist
2) Anschließend wird für alle Hyperparameter $\vec{\theta} \in \mathbb{R}^p$ die Ableitung des Likelihoods nach den Hyperparametern in der Form $\frac{\partial N}{\partial \theta_i}$ für alle p Hyperparameter bestimmt. Hierfür ist eine Rückwärtsdifferenzierung der Cholesky-Zerlegung notwendig (vgl. [2]). Diese Berechnung muss für die Bestimmung aller Ableitungen allerdings nur 1x durchgeführt werden und ist vom Aufwand $\frac{1}{3}n^3$. Anschließend kann für jeden Hyperparameter unabhängig die partielle Ableitung bestimmt werden. Dies erfordert noch einmal Matrizenoperationen mit einem Aufwand von etwa $p * n^2$.
3) Mithilfe dieser Informationen wird ein iteratives Gradienten basiertes Minimierungsverfahren verwendet, um die optimalen Hyperparameter zu bestimmen.

©2016

Da die verwendeten Matrizen bei wachsender Problemgröße schnell sehr groß werden (Matrizen mit einer Größe um ~20.000² können durchaus vorkommen), können diese Berechnungen zum Flaschenhals für den gesamten Optimierungsprozess darstellen. Zudem wird bei gängigen Optimierungen eine große Anzahl an Hyperparametern verwendet, sodass einerseits die Cholesky-Zerlegung und andererseits die Bestimmung der partiellen Ableitungen sehr rechenintensiv werden können.

Ziel ist es, diesen gesamten Prozess zu beschleunigen. Dies soll einerseits durch die Verwendung leistungsstarker Graphics Processing Units (GPUs) erreicht werden, auch wenn diese nicht auf dem Rechencluster, sondern extern auf einem anderen Rechner zur Verfügung stehen. Dies ist vor allem bei Problemstellungen mit großen Matrizen interessant, da GPUs im Vergleich zu herkömmlichen Central Processing Units (CPUs) in der Regel deutlich mehr Kerne und somit eine höher Rechenleistung haben. Andererseits sollen Programmteile, die unabhängig voneinander parallel ausgeführt werden können, wie die Bestimmung der partiellen Ableitungen, in einem Netzwerk mit beliebig vielen Servern parallel berechnet werden. Dies ist bei einer großen Anzahl an Hyperparametern und moderaten Matrizengrößen der Fall.

2. MESSAGING-MIDDLEWARE ZEROMQ (ZMQ)

Da in der Regel nicht jeder Rechner mit einer leistungsstarken GPU ausgerüstet ist, müssen rechenintensive Probleme, die solch eine verwenden sollen, auf einem externen Rechner bearbeitet werden, der mit einer GPU ausgestattet ist. Als geeignete Middleware zum schnellen Versand von Daten hat sich dabei die API ZeroMQ (auch ØMQ) herausgestellt. Diese ist als Open-Source-Projekt im Internet zu finden. Sie bietet ein breites und plattformunabhängiges Interface zum Aufbau so genannter Sockets zur Kommunikation mit anderen Sockets (sowohl thread- als auch rechnerübergreifend) an und ist einfach in der Handhabung. Auch die in [3] durchgeführten Geschwindigkeitsvergleiche zu anderer Middleware sprechen für ZeroMQ. Im Zusammenhang mit dem Ersatzmodelltraining kann diese nun verwendet werden, um einerseits eine 1:1 Verbindung zu einem GPU-Server bei großen Rechenoperationen oder andererseits eine 1:n Verbindung zu n Servern bei vielen kleineren, aber parallelen Operationen herzustellen.

3. BERECHNUNGEN AUF DER GPU

Das Programm für das Kriging-Verfahren ist in der Programmiersprache C++ geschrieben. Um nun eine beliebige Funktion auf einer GPU ausführen zu können, wird hierzu eine geeignete Schnittstelle benötigt. Nvidia® bietet für eigene Grafikkarten verschiedene Bibliotheken an, wie z.B. cuBLAS oder cuSOLVER, die in dem so genannten CUDA Toolkit (nachfolgend CUDA) zusammenfasst sind. Nach erfolgreicher Installation können diese im eigenen Programm eingebunden werden. Generell stehen dem Entwickler für Matrizenoperationen an dieser Stelle zwei verschiedene Möglichkeiten zur Auswahl:

- Verwendung von BLAS-Routinen (Basic Linear Algebra Subprograms), die in CUDA enthalten sind und auf der GPU ausgeführt werden
- Eigene Funktionen mithilfe der Programmiersprache CUDA erstellen, die auf der GPU ausgeführt werden

Im Zusammenhang mit dem Kriging-Verfahren und den darin verwendeten Matrizenoperationen bietet sich an dieser Stelle der Gebrauch der BLAS-Routinen an, da diese von Haus aus hoch optimiert und auf die Eigenschaften der verwendeten Hardware zugeschnitten sind. So kann bspw. eine Cholesky-Zerlegung mit nur einer BLAS-Routine erfolgen, deren Rückwärtsdifferenzierung bedarf in dem hier verwendeten Fall insgesamt drei verschiedene Aufrufe (generell wurde das Verfahren zur Verwendung von BLAS-Routinen bei der Rückwärtsdifferenzierung des Cholesky-Algorithmus in Anlehnung an [4] in leicht abgewandelter Form entwickelt und implementiert. In [4] wird die mathematische Herleitung der Rückwärtsdifferenzierung und die Möglichkeit zur Verwendung von BLAS-Routinen in diesem Zusammenhang näher erläutert). Dazu muss zuerst GPU-Speicher allokiert und die entsprechende Matrix anschließend auf diesen kopiert werden. Dies geschieht analog zur Allokation von einfachem Hauptspeicher mit dem Befehl *cudaMalloc*(…). Anschließend kann die Matrix byteweise mit dem CUDA-Pendant zu der C++-Funktion *memcpy*(…) mit *cudaMemcpy*(…) auf den GPU-Speicher transferiert werden. Nun können alle benötigten Operationen auf der GPU mit der Matrix durchgeführt werden.

Ein Problem, welches sich im Laufe der Entwicklung auftat, war die unterschiedliche Interpretation der Lage der Matrizen im Speicher zwischen den CUDA-Bibliotheken und der Programmklasse, die die Matrizen des Ersatzmodells verwaltet. Während das Ersatzmodell mit Matrizen arbeitet, die reihenbasiert indiziert werden (row major; erster Index entspricht der Reihe), erwarten die BLAS-Routinen von CUDA spaltenbasierte Matrizen (column major; erster Index entspricht der Spalte). Glücklicherweise sind aber die Matrizen, die für die Cholesky-Zerlegung verwendet werden, symmetrisch, sodass dies an dieser Stelle unproblematisch ist. Auch die Rückwärtsdifferenzierung der Cholesky zerlegten Matrix ist kein Problem, da auch diese symmetrisch ist. Lediglich bei der Matrizenmultiplikation muss dieser Umstand berücksichtigt werden, da im Zuge des Krigings auch nicht-symmetrische Matrizen miteinander multipliziert werden.

Wird eine Matrix, die eigentlich mit reihenbasierter Indizierung erzeugt wurde, als spaltenbasiert interpretiert, so kann diese durch das Programm als transponiert betrachtet werden. Der umgekehrte Fall gilt hier ebenso. Dieser Umstand lässt sich für die Matrizenmultiplikation zweier Matrizen A und B unter Betrachtung folgender Gleichung ausnutzen:

$$(3) \quad (A * B)^T = B^T * A^T$$

In diesem Fall reicht es also aus, bei dem Funktionsaufruf der CUDA BLAS-Routine zur Matrixmultiplikation lediglich die Parametrisierung der Matrizen A und B umzudrehen. Das so (transponierte) Ergebnis wird von dem Programm anschließend automatisch korrekt behandelt. Die Aufrufe der im Kriging eingesetzten BLAS-Routinen zu Cholesky-Zerlegung und deren Rückwärtsdifferenzierung sowie Matrizenmultiplikation sind in [6] genau dokumentiert.

Vor allem für Problemstellungen mit großen Matrizen und einer eher moderaten Anzahl an Hyperparametern ist solch eine Auslagerung auf eine externe GPU interessant.

4. VERTEILUNG AN VIELE SERVER

Neben der Auslagerung rechenintensiver und sequentieller Matrizenoperationen auf eine GPU bei Anwendungsfällen mit großen Matrizen wurde zudem ein Algorithmus zur Verteilung parallel ausführbarerer Quellcodeabschnitte im Rahmen des Krigings implementiert. Konkret wurde die Berechnung der partiellen Ableitungen des Likelihoods nach allen Hyperparametern auf verschiedene Server in einem Netzwerk verteilt. Diese Verteilung eignet sich besonders für Problemstellungen mit zahlreichen Hyperparametern.

Diese Verteilung findet auf einem Rechencluster intern statt. Da dort die Netzwerkanbindung der einzelnen Knoten untereinander über InfiniBand extrem hoch ist, können die Zeiten für den Datentransfer in diesem Fall als vernachlässigbar gering betrachtet werden. Bei Verteilungen an Server mit unterschiedlichen Rechenleistung und Netzwerkgeschwindigkeiten müssten diese Kennwerte in aufwendigen Berechnungen mit einbezogen werden. Da aber alle Clusterknoten die gleiche Rechenleistung besitzen, müssen solch aufwendige Betrachtungen nicht durchgeführt werden. Hier reicht es, wenn jeder Clusterknoten bzw. jeder Server etwa gleich große Datenpakete möglichst zur gleichen Zeit erhält, sodass alle Rechnungen zur gleichen Zeit abgeschlossen sind und Teilergebnisse zusammengeführt werden können.

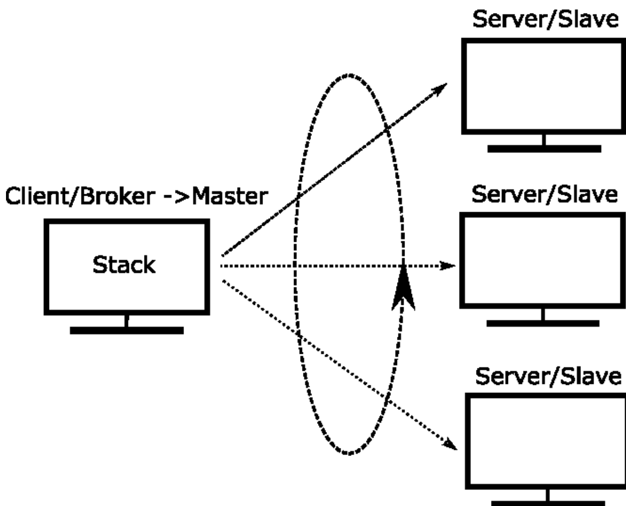

BILD 1. Prinzip des hier eingesetzten Round-Robin-Verfahrens

Die Umsetzung erfolgte mittels des so genannten Rundlaufverfahrens, oder auch Round Robin (vgl. Abbildung 1). Dazu wurden die Berechnungen der partiellen Ableitungen des Likelihoods für alle Hyperparameter in gleich große Serveranfragen bzw. Pakete so eingeteilt, dass jeder Server nach Möglichkeit nur eine Anfrage enthält und alle Anfragen etwa gleich groß sind. Diese Anfragen werden in einem Stack gespeichert, der nach dem Prinzip „last in first out" (LIFO) abgearbeitet wird. Anschließend werden alle für die Rechnungen benötigten Daten zeitgleich per Multithreading an alle Server versendet und anschließend die Anfragen verteilt. Sollte ein Server abstürzen bzw. eine Anfrage fehlschlagen, so wird diese wieder zurück auf den Stack gelegt und von einem anderen Server bearbeitet.

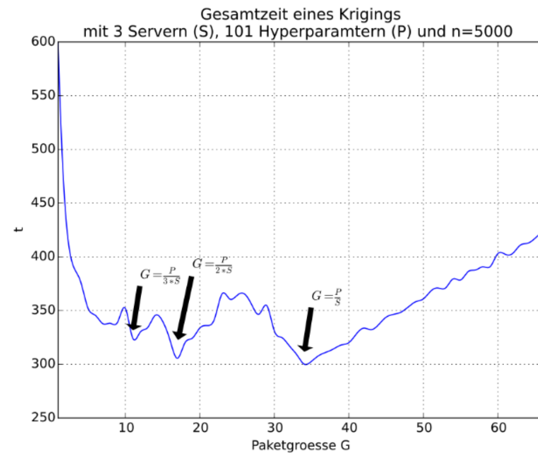

BILD 2. Gesamtzeit eines Krigings, welches an drei Server verteilt wurde, über der jeweiligen Paketgröße

Abbildung 2 stellt die benötigte Zeit eines Krigings mit einer Matrizengröße von 5000² und 101 Hyperparameter in Abhängigkeit von der Paketgröße dar, die zur Verteilung der Daten verwendet wurde. Die Operationen des Krigings wurden dabei an drei Server verteilt. Es lässt sich erkennen, dass die Geschwindigkeit für die Paketgrößen minimal wird, wenn alle Server möglichst gleich ausgelastet werden.

5. NETZWERKINTERFACE

Zur Einbindung aller relevanten Komponenten von ZeroMQ und CUDA wird ein geeignetes Netzwerkinterface benötigt, mit welchem das eigentliche Kriging interagieren kann. Zudem müssen Programmobjekte, die in einem Netzwerk verteilt werden sollen, zuerst für den Datentransport serialisiert und anschließend deserialisiert und wiederaufgebaut werden.

Zur Abstraktion der Zugriffe auf die Middleware ZeroMQ wurde zusätzlich eine neue Klasse *ZMQConnection* geschaffen. Diese fasst die gängigen Bibliotheksaufrufe zum Senden, Empfangen, Initialisierung und Schließen von Sockets in einfache Funktionen zusammen, sodass die Wartbarkeit des Netzwerkinterfaces deutlich verbessert werden konnte.

5.1. Netzwerkobjekte

Netzwerkobjekte sind im Folgenden Programmobjekte, die in einem Netzwerk verteilt und an beliebiger Stelle wieder in ihrem ursprünglichen Zustand aufgebaut werden können. Um dies zu erreichen, wurde daher eine abstrakte Superklasse *SaveableOnServer* geschaffen, die rein virtuelle Methoden bereitstellt. Die Implementierung der Methoden erfolgt anschließend in der Klasse des jeweiligen Objektes, welches von dieser Superklasse erbt. So kann eine einheitliche Struktur des Netzwerkinterfaces gewährleistet werden. An dieser Stelle wurde bewusst auf die Verwendung von anderen Konzepten (bspw. JSON, ProtoBuf, etc.) verzichtet, da die Datenstruktur der im Kriging verwendeten Objekte eine recht einfache und effiziente Datenserialisierung/-deserialisierung ohne großen Mehraufwand erlaubt. Wichtige Methoden, die bei Verwendung dieser Superklasse implementiert werden, sind:

- *getSerializedData(int& size): void**
- *setSerializedData(void* data): void*
- *getServerType(): string*
- *freeMem(): void*

So ist es beispielsweise möglich, eine gesamte Matrix samt Inhalt mithilfe der von dieser Superklasse bereitgestellten Methoden bei ausreichender Bandbreite zu serialisieren und auf einem entfernten Rechner im Netzwerk wieder aufzubauen. Allerdings können bei großen Matrizen enorme Datenmengen auftreten, deren Übertragungszeiten einen möglichen Geschwindigkeitsvorteil bei entfernt ausgeführten mathematischen Operationen zunichtemachen können. Daher ist es zudem möglich, auch komplexere Objekte zu versenden, die anschließend auf dem entfernten Server mit deutlich reduzierter Datenmenge die Matrizen wieder aufbauen können. Dies bringt vor allem bei geringen Netzwerkgeschwindigkeiten einen großen Geschwindigkeitsvorteil.

5.2. Client

Der Client wird im Quellcode des Krigings selbst verwendet und ist einerseits für die Datenhaltung von Netzwerkobjekten und andererseits für die geeignete Verteilung eben dieser zuständig. Er verwaltet das ihm bekannte Netzwerk und entscheidet, ob bei großen Matrizen eine 1:1 Verbindung zu einem GPU-Server oder bei großer Anzahl an Hyperparametern eine 1:n Verbindung zu n Servern infrage kommt. Der Client verwendet für die Kommunikation mit den Servern die Funktionsaufrufe der Middleware ZeroMQ und abstrahiert diese von dem eigentlichen Kriging. Im Zusammenhang mit verteilten Systemen sorgt dieser so für Orts-, Zugriffs-, Skalierungs- und Fehlertransparenz bzgl. des Krigings (Definitionen der Transparenzbegriffe für verteilte Systeme vgl. [5], S. 14 ff.).

Befehle, die an den Server gesendet werden, werden als Datenstring formuliert und serialisiert versendet. Einzelne Parameter werden mit einem Unterstrich getrennt. Es handelt sich hierbei also um nachrichtenbasierte Kommunikation.

5.3. Server

Der Server öffnet zur Kommunikation einen Socket mit ZeroMQ und empfängt über diesen Daten, die von dem Client gesendet werden. Er kann eine große Bandbreite an Operationen mit den ihm gesendeten Netzwerkobjekten durchführen. Der Server ist dabei stets der passive Part der Kommunikation. Er empfängt lediglich Befehle und Daten und sendet nur welche auf Anfrage an den Client zurück. Um Konflikte in der Datenhaltung und Ressourcenverwaltung zu vermeiden, wird allerdings pro Server maximal ein Client zugelassen.

5.4. Ausfallsicherheit

Damit der Client erkennen kann, welche Server aktuell im Netzwerk vorhanden sind, existiert neben jeder Hauptverbindung eine zusätzliche Ping-Verbindung zu jedem bekannten Server. Diese wiederum öffnen analog zum Client auch einen zusätzlichen Socket, der lediglich für die Verwaltung des periodischen Pingsignals zuständig ist. Der Ping besteht dabei aus der Signatur des Clients, anhand derer der Server erkennen kann, mit welchem Client dieser sich verbunden hat. Die Antwort des Servers wiederum besteht aus der Signatur des Clients, mit welchem er aktuell gekoppelt ist. Sollte ein zweiter Client eine Anfrage an einen bereits gekoppelten Server senden, kann dieser anhand der zurückgesendeten Signatur feststellen, dass der Server bereits anderweitig vergeben ist. Auch Ausfälle eines Servers während der Programmlaufzeit können so vom Client erkannt werden.

6. GESCHWINDIGKEITSVERGLEICH

Die zuvor beschriebenen Aspekte wurden in das Kriging integriert und anhand von Geschwindigkeitsmessungen bzgl. verschiedener Aspekte überprüft. Es wird dabei wie bisher zwischen der 1:1 Verbindung, die für große Matrizen mit einer geringen Anzahl an Hyperparametern verwendet wird, und der 1:n Verbindung zu n Servern, die für kleinere Matrizen mit einer großen Anzahl an Hyperparametern verwendet wird, unterschieden. Für die Geschwindigkeitsvergleiche wurden unterschiedliche Prozessoren verwendet:

- Lokale CPU: 2x Intel® Xeon® E5-2650 v3 mit 2,3 GHz → ca. 2x 368 Gflop/s bei doppelter Genauigkeit
- Lokale GPU: Nvidia® Quadro K6000 mit 601 MHz → ca. 1,1 Tflop/s bei doppelter Genauigkeit
- Cluster CPU: 2x Intel® Xeon® E5-2695 v2 mit 2,4 GHz → ca. 2x 230 Gflop/s bei doppelter Genauigkeit

Für alle Matrizenoperationen wurden BLAS-Routinen verwendet. Während bei der GPU die Bibliothek Nvidia® CUDA zum Einsatz kam, wurde für die Rechnungen auf den CPUs die Bibliothek Intel® Math Kernel Library (MKL) verwendet.

6.1. Vergleich 1:1 Verbindung

Die 1:1 Verbindung wird dazu verwendet, um eine leistungsstarke GPU in den Prozess zu integrieren. Insbesondere die Durchführungsgeschwindigkeit der Cholesky-Zerlegung profitiert von dem Einsatz einer GPU. Da die Verbindungsgeschwindigkeit bei dieser Verbindungsart als relativ langsam angenommen wird (Cluster ⇔ Abteilungsnetzwerk), werden hierbei nicht ganze Matrizen versendet (bei n=20.000 und doppelter Genauigkeit ergibt sich eine Datenmenge von 2,98 GiB), sondern die Instanzen der Klassen, die diese Matrizen enthalten (Größe: wenige Kilobyte). Anschließend werden diese Matrizen entfernt wieder aufgebaut.

6.1.1. Zeitverlust durch Kommunikation

Der Zeitverlust, der durch die Kommunikation entsteht, wird nachfolgend näher analysiert. Dazu wurde zuerst ein Kriging lokal unter Verwendung der GPU gestartet und die Zeit ermittelt. Anschließend wurde das gleiche Training von einem Clusterknoten aus gestartet und der Rechner mit der GPU als Server verwendet.

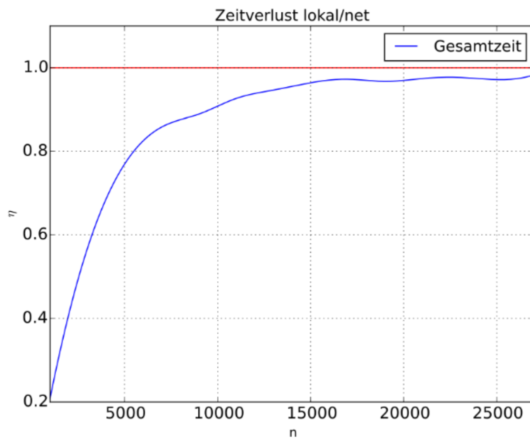

BILD 3. Speedup bzw. Zeitverlust η_s der Netzwerkkommunikation vom Cluster zum Abteilungsnetz im Vergleich zur lokalen Ausführung (über Matrixgröße n - 20 Hyperparameter)

Wie sich in Abbildung 3 erkennen lässt, ist der Zeitverlust besonders bei großen Problemen sehr gering. Dies liegt daran, dass die zu sendende Datenmenge bei wachsender Problemgröße nur sehr geringfügig um einige Kibibyte wächst. Daher wird der Zeitverlust geringer, je aufwändiger das Ersatzmodelltraining bzw. Kriging wird.

6.1.2. Speedup zu herkömmlichen Rechnungen

Nachdem nun der Zeitverlust ermittelt wurde, der bei der Netzwerkkommunikation entsteht, wurde anschließend der Speedup gemessen, der durch den Einsatz einer GPU in den Gesamtprozess erreicht wurde. Dazu wurde zuerst der Speedup gemessen, der durch den Einsatz einer GPU im Vergleich zur lokalen CPU bei einem Ersatzmodelltraining erreicht werden konnte. Dies ist in Abbildung 4 dargestellt. Es lässt sich hier erkennen, dass vor allem bei der Cholesky-Zerlegung ein Speedup zwischen 1,3 bis 1,5 erreicht werden konnte. Dies entspricht in etwa dem theoretischen Leistungsverhältnis von GPU zu CPU. Der Speedup der Gesamtzeit liegt leicht darunter, da einige Programmteile des Krigings weiterhin auf der CPU ausgeführt werden.

Abbildung 5 stellt den Speedup dar, welcher für ein Kriging auf einem Clusterknoten unter Einbeziehung einer externen GPU in einer 1:1 Verbindung erreicht werden konnte. Dies ist im Optimierungsprozess der gängigste Fall. Bei aktueller Infrastruktur konnte hier durch den Einsatz eines GPU-Servers ein Speedup der Gesamtdauer um den Faktor 2 erreicht werden.

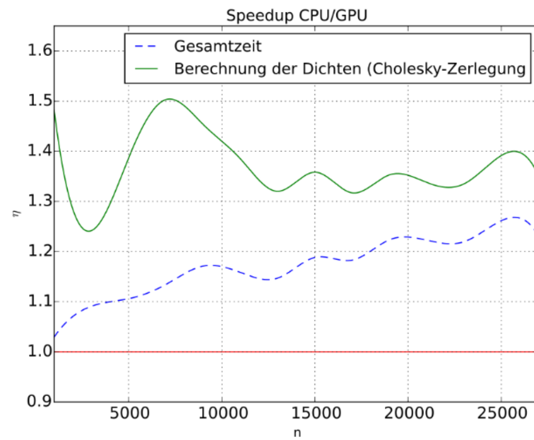

BILD 4. Speedup η_s der Gesamtzeit des Krigings und der Cholesky-Zerlegung vom Cluster zum Abteilungsnetz im Vergleich zur lokalen Ausführung (über Matrixgröße n – 20 Hyperparameter)

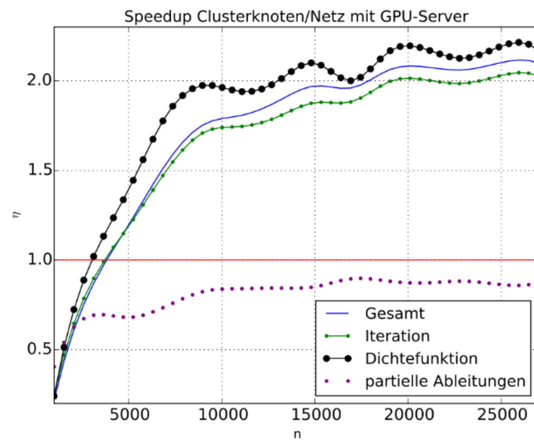

BILD 5. Speedup η_s einer 1:1 Client-Server-Verbindung im Vergleich zur herkömmlichen Ausführung eines Krigings auf dem Cluster (über Matrixgröße n – 20 Hyperparameter)

6.2. Vergleich 1:n Verbindung

Nachdem der Speedup zu einem Server in einer 1:1 Beziehung ermittelt wurde, ist daneben noch der Zeitgewinn bei der Verteilung der Berechnungen der partiellen Ableitungen interessant. Dazu wurde auf dem Cluster eine Rechnung auf einem Knoten gestartet und zurerst lokal berechnet (Server 0) und anschließend an eine variable Anzahl an Server verteilt.

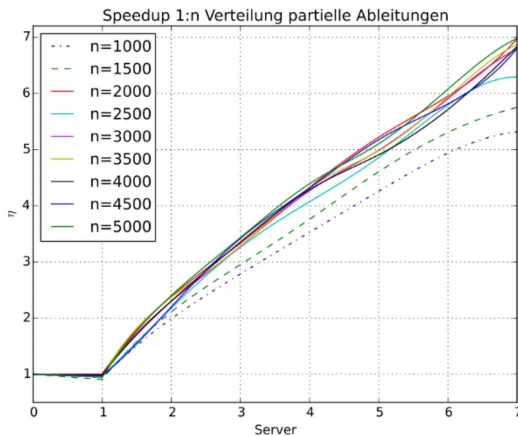

BILD 6. Speedup η_s für die Berechnung aller partiellen Ableitungen in einer 1:n Client-Server-Verbindung im Vergleich zur herkömmlichen Ausführung eines Krigings auf dem Cluster (über Anzahl der Server – 500 Hyperparameter)

Abbildung 6 zeigt den Speedup der Berechnung über der Anzahl an verwendeten Servern für verschiedene Matrixgrößen. Es lässt sich erkennen, dass der Speedup fast linear mit der Anzahl an Servern skaliert und dass es aufgrund der hohen Netzwerkgeschwindigkeit beinahe irrelevant ist, ob die Berechnungen lokal auf dem Knoten oder auf einem einzigen Server durchgeführt werden. Der Gesamtspeedup (Abbildung 7) liegt leicht darunter.

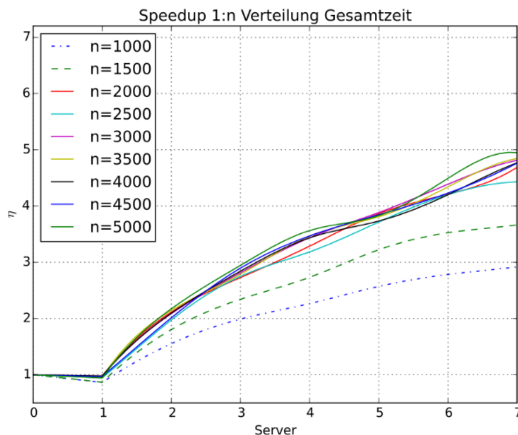

BILD 7. Speedup η_s für die Gesamtzeit in einer 1:n Client-Server-Verbindung im Vergleich zur herkömmlichen Ausführung des Krigings auf dem Cluster (über Anzahl der Server – 500 Hyperparameter)

7. ZUSAMMENFASSUNG & AUSBLICK

Das Kriging-Verfahren, welches im Rahmen des Ersatzmodelltrainings im Optimierungsprozess eingesetzt wird, wurde auf rechenintensive und parallel ausführbare Codesegmente hin untersucht und in einem Netzwerk an entweder einen Server (1:1 Beziehung) oder mehrere Server (1:n Beziehung) verteilt.

Durch die Integration einer GPU in den Prozess des Kriging-Verfahrens konnte bei einer 1:1 Verbindung bei aktueller Infrastruktur ein Speedup um den Faktor 2 im Vergleich zum herkömmlichen Verfahren erzielt werden. Wird für die Berechnung der partiellen Ableitungen auf ein 1:n Netzwerk zurückgegriffen, so konnte der Speedup in diesem Fall in Abhängigkeit von der Serveranzahl gesteigert werden.

Die Verwendung einer GPU zu Rechenzwecken ist daher vor allem für Probleme mit großen Datenmengen interessant. Bei geeigneter Infrastruktur wäre in Zukunft zudem eine Kombination beider Verfahren (1:1 Verbindung zu einer GPU für Matrizenoperationen und 1:n Verbindung für zahlreiche parallele Prozesse) lohnenswert, um einen noch höheren Speedup erzielen zu können. Da eine GPU auch parallele Prozesse sehr gut bewältigen kann, könnte solch eine Verteilung auch auf der GPU direkt stattfinden.

8. REFERENZEN

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

[2] 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

[3] Dworak, A.; Ehm, F.; Sliwinski, M.; Sobczak, M.: *Middleware Trends and Market Leaders 2011*, Online unter: http://zeromq.wdfiles.com/local--files/intro%Aread-the-manual/Middleware Trends and Market Leaders 2011.pdf, CERN, Geneva, Switzerland, 2011

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

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

[6] Nvidia® Corporation: *CUDA Toolkit Documentation*, Online unter: http://docs.nvidia.com/cuda/, August 2016