Parallele Dateisysteme

Michael Kuhn

Einleitung

Die weltweit produzierte Datenmenge verdoppelt sich momentan ungefähr alle zwei Jahre. Diese exponentiell wachsende Datenflut muss gespeichert, analysiert und weiterverarbeitet werden. Traditionelle Dateisysteme können die wachsenden Anforderungen an Speicherkapazität und -geschwindigkeit nicht erfüllen, weswegen parallele Dateisysteme bereits in vielen datenintensiven Bereichen wie z. B. der Hochenergiephysik oder den Klimawissenschaften Verwendung finden.

Das Hochleistungsrechnen hat sich dabei als ein nützliches und mittlerweile unverzichtbares Werkzeug für viele Wissenschaftsdisziplinen etabliert. Durch die dadurch möglichen Analysen und Simulationen kann der wissenschaftliche Erkenntnisgewinn in vielen Bereichen deutlich gesteigert werden. In den vergangenen Jahrzehnten konnte die Rechenleistung der in der TOP500-Liste vertretenen Hochleistungsrechner durchschnittlich alle 14 bis 15 Monate verdoppelt werden [6]. Obwohl seit 2015 ein Abflachen der Steigerungsraten zu beobachten ist, führt dieser weiterhin exponentielle Anstieg zu einer rasanten Zunahme der produzierten Daten.

Dateisystemgrundlagen

Dateisysteme ermöglichen einen komfortablen Zugriff auf Speichergeräte. Sie stellen eine Abstraktionsschicht zwischen Anwendungen und der tatsächlichen Speicherhardware dar, sodass sich Anwendungsentwickler nicht mit den Eigenheiten der darunterliegenden Speicherhardware auseinandersetzen müssen. Darüber hinaus stellen Dateisysteme üblicherweise standardisierte Zugriffsschnittstellen bereit, die die Portabilität der Ein-/Ausgabe (E/A) ermöglichen.

Die Speicherhardware setzt sich heutzutage typischerweise aus Festplatten und Solid-State-Drives zusammen, die unterschiedliche Leistungscharakeristika und Kosten aufweisen. Solche Speichergeräte geben keine Organisationsstruktur vor und erlauben einen block- oder byteweisen Zugriff auf den bereitgestellten Speicher.

Die beiden grundlegenden Datenstrukturen, die sich in fast allen Dateisystemen wiederfinden, sind Dateien und Verzeichnisse. Dateien enthalten die eigentlichen Daten, während Verzeichnisse zur Strukturierung des Dateisystemnamensraumes dienen. So können Verzeichnisse üblicherweise sowohl Dateien als auch weitere Verzeichnisse enthalten, wodurch ein hierarchischer Namensraum entsteht. Der Zugriff auf Dateien und Verzeichnisse findet normalerweise über Namen statt, wobei ein voll qualifizierter Name auch Pfad genannt wird.

Dateien und Verzeichnisse stellen hierbei nur das Minimum an Dateisystemfunktionalität dar. Viele Dateisysteme bieten zusätzliche Funktionalitäten wie z. B. Named Pipes [4]. Durch die immer weiter steigenden Datenmengen integrieren moderne Dateisysteme außerdem zunehmend Funktionen zur Volumenverwaltung, Kompression und Sicherstellung der Datenintegrität [7].

Obwohl sich die Funktionsweise von Dateisystemen auf unterschiedlichen Betriebssystemen

https://doi.org/10.1007/s00287-019-01209-7

Michael Kuhn
Universität Hamburg, Hamburg
E-Mail: michael.kuhn@informatik.uni-hamburg.de

nicht grundlegend unterscheidet, werden sich die Erklärungen in diesem Artikel auf Linux und seine durch POSIX (Portable Operating System Interface) standardisierten Schnittstellen beschränken. Insbesondere im Hochleistungsrechnen stellt Linux einen De-facto-Standard dar und so finden sich in der aktuellen TOP500-Liste ausschließlich Hochleistungsrechner mit einem auf Linux basierenden Betriebssystem. Die E/A-Schnittstelle des POSIX-Standards wurde primär für lokale Dateisysteme entwickelt. Eine erste formale Spezifikation erfolgte im Rahmen von POSIX.1 im Jahr 1988, wobei diese für asynchrone und synchrone E/A im Jahr 1993 durch POSIX.1b erweitert wurde. Zusätzlich zur Syntax der E/A-Schnittstellen wird auch deren Semantik, d. h. deren Verhalten, durch POSIX spezifiziert. So wird beispielsweise festgelegt, dass Leseoperation nach dem Zurückkehren einer Schreiboperation sofort die neuen Daten zurückliefern müssen, was Auswirkungen auf das Cachingverhalten haben kann [5]. Fast alle unter Linux verfügbaren Dateisysteme sind POSIX-konform, wodurch eine hohe Portabilität gewährleistet werden kann. Insbesondere bieten auch die meisten parallelen Dateisysteme eine POSIX-konforme Schnittstelle an, sodass Anwendungen ohne größere Anpassungen auch auf Hochleistungsrechnern lauffähig sind.

Ein weiteres wichtiges Konzept sind die Metadaten. Dateien und Verzeichnisse bestehen jeweils sowohl aus Daten als auch aus Metadaten. Im Fall einer Datei bezeichnen die Daten den eigentlichen Inhalt der Datei, während bei einem Verzeichnis die Verzeichniseinträge gemeint sind. Die Metadaten beschreiben in beiden Fällen weitergehende Informationen wie z. B. Zugriffsberechtigungen oder Eigentümer (siehe Abb. 1). Im Fall von POSIX-konformen Dateisystemen werden die Metadaten als sogenannte Inodes verwaltet. Während die Größe der Daten stark variieren kann, nehmen die Metadaten üblicherweise nur sehr wenig Platz ein. Im Fall des traditionellen Linux-Dateisystems ext4 haben Inodes eine Standardgröße von 256 Byte. Insbesondere enthalten die Inodes auch Verweise auf die eigentlichen Daten. Das darunterliegende Speichergerät wird meist in Blöcke gleicher Größe aufgeteilt, die dann im Inode referenziert werden. So kann der Benutzer komfortabel über Namen auf Dateien und Verzeichnisse zugreifen, während sich das Dateisystem intern um die Allokation und Verwaltung des notwendigen Speicherplatzes kümmert.

Feldgröße	Inhalt
2 Byte	Berechtigungen
2 Byte	Benutzer-ID (Untere Bits)
4 Byte	Dateigröße (Untere Bits)
4 Byte	Zugriffszeit (Sekunden)
⋮	⋮
4 Byte	Versionsnummer (Obere Bits)
100 Byte	Freier Speicher

Abb. 1 ext4-Inode [1]: Inodes sind in feste Felder und einen freien Bereich am Ende aufgeteilt. Aus Gründen der Rückwärtskompatibilität sind unter anderem die Felder für die IDs, die Größe und die Zeitstempel in jeweils zwei Felder aufgeteilt. Der freie Bereich kann für erweiterte Attribute oder spätere Erweiterungen der Inode-Datenstruktur genutzt werden.

Parallele Dateisysteme

Um die Anforderungen moderner datenintensiver Anwendungen erfüllen zu können, bieten parallele Dateisysteme sowohl einen effizienten parallelen Zugriff von mehreren Anwendungen sowie eine Verteilung der Daten über mehrere Speichergeräte und Dateisystemserver hinweg. Einerseits erlaubt es der parallele Zugriff verteilten Anwendungen gleichzeitig mit einem gemeinsamen Datensatz zu arbeiten, andererseits können durch die Verteilung der Daten die Kapazität und der Durchsatz vieler Speichergeräte genutzt werden (siehe Abb. 2). Dadurch sind aktuell massiv parallele Dateisysteme mit Speicherkapazitäten im dreistelligen Petabytebereich und Durchsätzen im einstelligen Terabytebereich realisierbar. Durch die vom Dateisystem bereitgestellte Abstraktion kann die Verteilung der Daten vollstän-

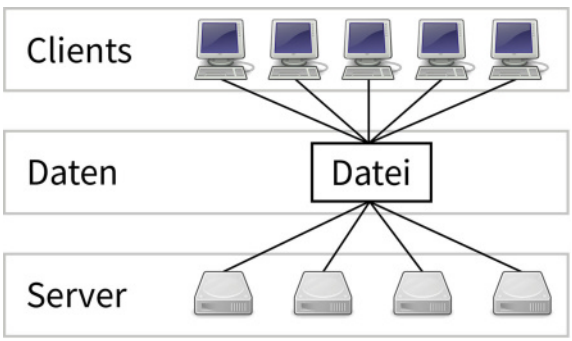

Abb. 2 Paralleler Zugriff und Datenverteilung: Mehrere Anwendungen können gleichzeitig auf eine gemeinsame Datei zugreifen, die wiederum über mehrere Speichergeräte und Dateisystemserver verteilt wird.

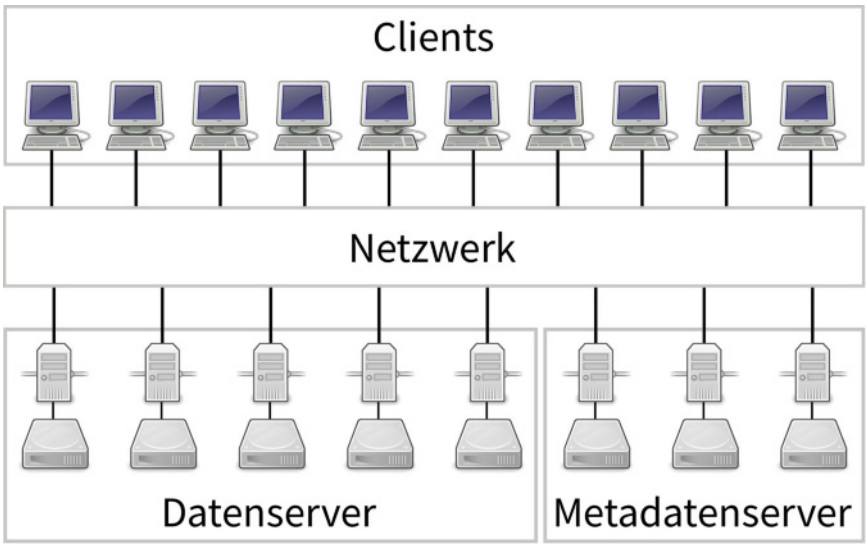

Abb. 3 Paralleles Dateisystem: Clients kommunizieren über ein Netzwerk mit den Daten- und Metadatenservern. Auf den Clients ist das Dateisystem häufig als POSIX-konformes Kernelmodul in das Betriebssystem eingebunden.

dig transparent für die Anwender geschehen. Drei der am weitesten verbreiteten parallelen Dateisysteme sind Lustre von DDN, Spectrum Scale (ehemals GPFS) von IBM und BeeGFS von ThinkParQ und Fraunhofer. Da im Bereich paralleler Dateisysteme keine einheitliche Nomenklatur existiert, werden diese häufig auch als verteilte Dateisysteme oder seltener Cluster-Dateisysteme bezeichnet.

Ein paralleles Dateisystem wird üblicherweise in Clients und Server getrennt (siehe Abb. 3). Dies erlaubt einerseits eine Spezialisierung auf die jeweilige Funktionalität und schließt andererseits eine gegenseitige Beeinflussung aus. Beispielsweise können die Clients so mit vielen Kernen für Berechnungsaufgaben ausgestattet werden, während die Server entsprechende Speichergeräte und ausreichend Arbeitsspeicher für das Caching enthalten. Darüber hinaus werden die Server häufig in Daten- und Metadatenserver aufgeteilt, da auf Daten meist große Zugriffe erfolgen, wohingegen Metadatenzugriffe häufig klein und zufällig verteilt sind. So können für die Datenserver optimalerweise Festplatten mit höherer Kapazität eingesetzt werden, während für die Metadatenserver kleinere Festplatten mit höherer Umdrehungszahl oder Solid-State-Drives genutzt werden.

Durch diese Aufteilung müssen die Clients über das Netzwerk mit den Servern kommunizieren, was zusätzliche Latenz verursacht. Eine typische Architektur verlagert den Großteil der Dateisystemlogik in die Clients, sodass diese selbstständig entscheiden können, welche Server kontaktiert werden müssen. In diesem Fall müssen die Server nicht untereinander kommunizieren und können als relativ einfache Datenspeicher fungieren.

Die eigentliche Verteilung der Daten und Metadaten wird dabei von Verteilungsfunktionen übernommen. Die häufigste Verteilungsfunktion für Daten ist ein simpler Round-Robin-Ansatz, der die Daten in gleichgroße Blöcke zerlegt und in Form von Streifen über die Datenserver verteilt (siehe Abb. 4). Etwaige Ungleichgewichte bei der Verteilung der Daten können aufgrund der zufällig gewählten Startparameter und der großen Dateigrößen für gewöhnlich vernachlässigt wer-

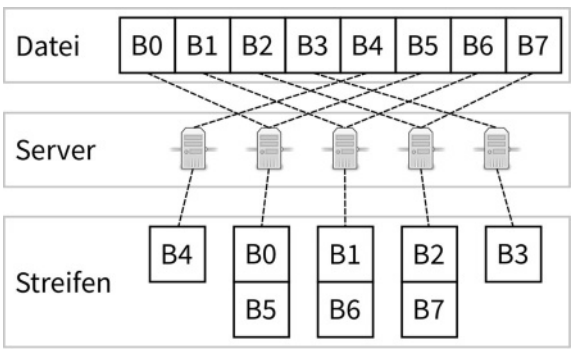

Abb. 4 Round-Robin-Verteilungsfunktion: Die Datei wird in acht gleichgroße Blöcke zerlegt und über fünf Datenserver verteilt. Der Startserver wird zufällig gewählt, um eine gleichmäßige Lastverteilung mit mehreren Dateien zu gewährleisten. Sobald der letzte Datenserver erreicht ist, wird mit dem ersten fortgefahren.

den. Da Metadaten häufig zu klein für eine solche Verteilung sind, werden die Metadaten einer einzelnen Datei oder eines einzelnen Verzeichnisses durch einen einzigen Metadatenserver verwaltet. Eine Ausnahme bilden sehr große Verzeichnisse, die zur Leistungssteigerung über mehrere Metadatenserver verteilt werden können. Zur Bestimmung des zuständigen Metadatenservers wird dabei häufig auf Hashing zurückgegriffen, wobei kryptographische Hashfunktionen den Vorteil einer gleichmäßigen Verteilung haben. So kann der zuständige Metadatenserver beispielsweise durch das Hashen des Dateinamens oder des vollen Pfads bestimmt werden.

Die grundlegende Funktionsweise soll an einem einfachen Beispiel erläutert werden: Ein Client will 42 Byte an Offset 4242 in die bereits existierende Datei `/meine/datei` schreiben. Dafür sind mehrere Schritte notwendig:

1. Zuerst muss das Wurzelverzeichnis (`/`) gelesen werden. Der Ort des Wurzelverzeichnisses ist daher meist statisch festgelegt, sodass keine weiteren Informationen für die Suche notwendig sind. Der Client kommuniziert mit dem zuständigen Metadatenserver und ruft die Verzeichnisinformationen ab. Daraufhin werden die Zugriffsrechte überprüft und bei Bedarf ein Fehler zurückgeben. Darf der Client auf das Wurzelverzeichnis zugreifen, werden die Einträge gelesen und nach dem Eintrag `meine` durchsucht.
2. Im nächsten Schritt muss das darauf folgende Verzeichnis (`meine`) gelesen werden. Der Ort und somit der zuständige Metadatenserver kann aus dem Verzeichniseintrag im Wurzelverzeichnis bestimmt werden, sodass nun der vorherige Schritt für diese Pfadkomponente wiederholt wird.
3. Als Nächstes müssen die Metadaten der eigentlichen Datei (`datei`) abgerufen werden. Der zuständige Metadatenserver kann aus dem im vorherigen Schritt gesuchten Verzeichniseintrag bestimmt werden. Daraufhin kommuniziert der Client mit dem Metadatenserver und ruft alle notwendigen Informationen ab. Es werden wieder die Zugriffsrechte geprüft und bei Bedarf ein Fehler zurückgeben.
4. Abschließend müssen die Daten geschrieben werden. Die im vorherigen Schritt abgerufenen Metadaten enthalten auch Informationen über die Verteilung der Daten. Auf deren Basis wird der zuständige Datenserver für das Offset 4242 bestimmt, auf dem wiederum die Schreiboperation durchgeführt wird. Sollte die Datei gleichzeitig von einem anderen Client geöffnet worden sein, müssen außerdem Sperren für den betroffenen Bereich angefordert werden, um das durch POSIX festgelegte Verhalten sicherzustellen [3].

Ausblick

Auf Grundlage der vorgestellten Architektur lassen sich parallele Dateisysteme in Betrieb nehmen, die einige Tausend Server enthalten, auf die wiederum mehrere Zehntausend Clients zugreifen. Trotz ihrer verteilten und skalierbaren Architektur stoßen parallele Dateisysteme allerdings immer häufiger an die Grenzen ihrer Leistungsfähigkeit. Während die mittleren Zugriffszeiten handelsüblicher Festplatten im zweistelligen Millisekundenbereich liegen, erreichen Solid-State-Drives Latenzen im zwei- bis dreistelligen Mikrosekundenbereich. In der Entwicklung befindliche Speichertechnologien wie beispielsweise NVRAM werden noch geringere Zugriffszeiten aufweisen. Um die Leistungsfähigkeit dieser neuen Speichergeräte ausnutzen zu können, dürfen die durch die Softwareschichten des Dateisystems verursachten Latenzen nicht überhand nehmen. Einen signifikanten Anteil am Dateisystem-Overhead hat dabei der Betriebssystemkernel, da zur Ausführung von E/A-Operationen mit dem Kernel kommuniziert werden muss. Die notwendigen Modus- und Kontextwechsel können je nach durchgeführter Operation und Hardwarearchitektur bis zu einigen Mikrosekunden dauern. Ein möglicher Ansatz ist die Umgehung des Kernels, wie dies bereits für diverse Netzwerktechnologien wie z. B. InfiniBand umgesetzt wird. Die strengen Kohärenz- und Konsistenzanforderungen des POSIX-Standards stellen ein weiteres Problem für die Skalierbarkeit paralleler Dateisysteme dar. Um diesen Anforderungen gerecht zu werden, entstehen aktuell neue Speichersystemkonzepte wie z. B. der Distributed Application Object Storage (DAOS), der vollständig im Benutzermodus läuft und eine transaktionsbasierte E/A-Schnittstelle bietet [2].

In Zeiten der rechen- und datenintensiven Forschung ist es notwendig, die Bemühungen um leistungsfähige und skalierbare Speichersysteme zu intensivieren. Auch grundlegende Dateisystemkonzepte müssen überdacht und gegebenenfalls verbessert werden, um den steigenden Anforderungen gerecht werden zu können.

Literatur

1. djwong. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout, last access: 30.8.2019
2. Liu J, Koziol Q, Butler GF, Fortner N, Chaarawi M, Tang H, Byna S, Lockwood GK, Cheema R, Kallback-Rose KA, Hazen D, Prabhat (2018) Evaluation of HPC application I/O on object storage systems. In: 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW-DISCS@SC 2018. Dallas, TX, USA, November 12, 2018, pp 24–34
3. Moore M, Farrell P, Cernohous B (2018) Lustre lockahead: Early experience and performance using optimized locking. Concurr Comp-Pract E 30(1):1–14
4. The Linux man-pages project. fifo(7). http://man7.org/linux/man-pages/man7/fifo.7.html, last access: 30.8.2019
5. The Linux man-pages project. write(2). http://man7.org/linux/man-pages/man2/write.2.html, last access: 30.8.2019
6. TOP500.org. Performance Development. https://www.top500.org/statistics/perfdevel/, last access: 30.8.2019
7. Zhang Y, Rajimwale A, Arpaci-Dusseau AC, RH Arpaci-Dusseau RH (2010) End-to-end data integrity for file systems: A ZFS case study. In: 8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23–26, 2010, pp 29–42