Seminararbeit

„RFS – Remote File Access for MPI-IO“

Betreuer

Prof. Dr. Rudolf Berrendorf

Fachbereich Informatik, FH-Bonn-Rhein-Sieg

Bearbeiter

Tom Vedder

Köln, den 25. Juni 2005

Inhaltsverzeichnis

Kapitel 1: Einführung

Die Entwicklung immer schnellerer Prozessoren bzw. breitbandiger Netzwerkverbindungen mit geringen Latenzzeiten hat auf dem Gebiet des High Performance Computing (HPC) zu einem neuen Trend geführt. Anstatt teurer und relativ unflexibler Supercomputer erfreuen sich Cluster einer immer größer werdenden Beliebtheit. Ihr sehr gutes Preis-/Leistungsverhältnis macht sie insbesondere für den akademischen Einsatz interessant.

1.1 Problemstellung

Wissenschaftliche Anwendungen bzw. Simulationen sind typischerweise I/O-lastig. Häufig werden Berechnungsergebnisse periodisch abgespeichert, um sie zu einem späteren Zeitpunkt nachbearbeiten zu können. Bereits hier jedoch treten erste Performance-Probleme auf. Die Rechenleistung heutiger Prozessoren und die damit verbundene, erforderliche Datenbandbreite zum Speichern der Rechenergebnisse ist um ein vielfaches höher als die momentan verfügbare Speicherbandbreite lokaler Festplattensysteme, selbst wenn Daten parallel (RAID) abgespeichert werden. Noch schwerer jedoch wiegt der Geschwindigkeits- bzw. Zeitverlust, der durch das verteilte Speichern der Daten an räumlich entfernt gelegenen Orten entsteht. Bei diesen Szenarios kommt zu dem Problem der Datenspeicherung noch das der Netzwerkübertragung bzw. -geschwindigkeit hinzu. Denn üblicherweise ist die Netzbandbreite um einiges geringer als die Datenspeicherbandbreite.
Der traditionelle Ansatz, diese Problem zu lösen, beruht auf dem Prinzip des verteilten Bereitstellens von Simulations- bzw. Applikationsdaten („staging“). Hierdurch wird es ermöglicht, den gleichen Datensatz an verteilten Standorten anzusehen, zu bearbeiten und gegebenenfalls zurückzuschreiben.
Die damit verbundenen Probleme sind jedoch zahlreich und schwerwiegend. An jedem Arbeitsplatz muss genauso viel lokaler Datenspeicher vorhanden sein wie beim bereitstellenden Server; ein erstes, schwerwiegendes Problem, da wissenschaftliche Simulationen häufig mehrere Terabytes an Daten bearbeiten müssen. Selbst wenn eine Workstation über derart viel Speicherkapazität verfügen würde, müssen die Daten erst einmal übertragen werden. Hierdurch aber wäre die Anwendung wieder sehr stark von

der Leistungsfähigkeit der Netzwerkübertragung abhängig, eine Beziehung, die ja gerade durch „staging“ verhindert werden sollte! Sind die Daten an mehreren Orten lokal verfügbar, ergibt sich aber zwangsläufig das Problem der Dateninkonsistenz. Benutzer des Systems nehmen unterschiedliche Modifikationen der Daten vor, die dann mittels einer enorm netzwerkbelastenden Datensynchronisation auf dem Server abgeglichen werden müssten. Werden nicht Berechnungsdaten, sondern Applikationen lokal bereitgestellt, stellt sich das Problem der Portabilität. Denn es ist nicht unüblich, daß für Berechnung und Visualisierung von Simulationen unterschiedliche Hardwarearchitekturen verwendet werden.

1.2 Zielsetzung

Um die oben beschriebenen Defizite zu beheben, gilt es, einen Mechanismus zu finden, um effizient, netzübergreifend und portabel auf entfernt gespeicherte wissenschaftliche Daten zuzugreifen.
Das in dieser Arbeit vorgestellte RFS – Remote File System – soll einerseits eine automatische Datenmigration zwischen entfernt aufgestellten Rechnern ermöglichen, andererseits aber unnötigen bzw. performance-schädlichen Datenverkehr minimieren. Diese Funktionalitäten laufen weitgehend benutzertransparent ab, lediglich zur Optimierung der Performance können Hinweise an den Benutzer weitergegeben werden. Da wissenschaftliche Simulationen häufig schreibintensiv sind und nur relativ wenig Leseoperationen durchführen, ist der Mechanismus schreiboptimiert. Als zusätzliche Optimierung wird ABT („Active Buffering with Threads“) integriert, ein Mechanismus, der mittels Überschneidung von Rechenzeit mit Datentransferzeit die tatsächlich anfallenden Zeit- und Rechenkosten verteilter Schreibeoperationen verdeckt. Weitere Verbesserungen beinhalten temporäres Zwischenspeichern von Datensätzen sowie ein Algorithmus, der ermittelt, wann der nächste Datensatz zur Zwischenspeicherung bereitsteht bzw. wie groß er sein wird. Um größtmögliche Portabilität herzustellen, benutzt RFS ROMIO, eine verbreitete Implementierung von MPI-IO, dem de facto Standard im Bereich paralleler I/O [MPI, 97].

Kapitel 2: Forschungshintergrund

Weithin bekannte, verteilte Dateisysteme wie NFS oder AFS stellen komfortable Schnittstellen dar, die eine Integration entfernt liegender Speichersysteme leicht machen. Ihre universelle Einsetzbarkeit stellt jedoch im Bereich rechenintensiver, wissenschaftlicher Anwendungen keinen Vorteil dar.

2.1 RIO

Ein früher Optimierungsansatz auf diesem Gebiet ist Remote I/O [Foster, 97]. RIO basiert auf ROMIO und dessen Abstraktionsschichten-Modell und verwendet eine klassische Client-Server Architektur. Hierdurch wird eine weitgehende Portabilität erreicht, dateisystemspezifische Implementierungen können somit unberücksichtigt bleiben. Dem steht jedoch die Anforderung nach dezidierten „forwarder nodes“ entgegen. Diese spezielle Knoten benötigt RIO, um Netzwerknachrichten zu sammeln bzw. asynchronen Daten-I/O durchzuführen. Häufig jedoch ist es nicht möglich, Rechenleistung bzw. Rechenknoten nur für diese Aufgabe bereitzustellen. RIO stellt zudem besondere Anforderungen an die Zahl der gleichzeitig kommunizierenden bzw. interagierenden Server- und Clientprozesse. Ausserdem beruht es auf dem Globus Toolkit, das die Verwendung anderer Kommunikationsprotokolle wie beispielsweise TCP/IP nicht zulässt.
Weitere Ansätze auf diesem Forschungsgebiet wie etwa GASS (Global Access to Secondary Storage) [GASS, 99] oder GridFTP [GridFTP, 02] sollen für diese Arbeit unberücksichtigt bleiben, stellen sie doch lediglich Methoden dar, Daten schnell zu transferieren.

2.2 ABT

Ein frühes Modell des „Active Buffering“ [Seamonns, 95] sieht vor, dass dedizierte I/O-Prozessoren ihren freien Speicher hierarchisch in Buffer aufteilen, um möglichst zu jeder Zeit Rechenergebnisse zwischenspeichern zu können. Die Ergebnisdaten besorgen sie sich mittels MPI von den reinen Rechenprozessoren, die ihrerseits alle zu

schreibenden Daten im Speicher puffern, dann aber sofort zum Berechnungsprozess zurückkehren. Somit wird die Zeit zum Speichern minimiert und die Rechenzeit bzw. -leistung erhöht. Synthetische Benchmarks haben gezeigt, dass hierdurch der ersichtliche Schreibedurchsatz ca. 70% der vorhandenen Speicherbandbreite erreichen kann [Ma, 03].

Im Gegensatz zu diesen Entwicklungen weißt ROMIO keine Bedingung bezüglich spezieller I/O-Prozessoren auf. Hier wird das Konzept der ABT verwendet, bei dem anstatt eines eigenen Prozessors ein Prozess für Buffering und Background-I/O zuständig ist. Zwar wird hierdurch das optimale gleichzeitige Ausführen von Simulationsberechnungen und I/O-Operationen etwas gemindert (ein Prozessor verwendet mindestens 2 Prozesse, um Berechnung und I/O-Operationen ausführen zu können), die Portabilität jedoch erhöht (zusammen mit Preis, Wartung, etc.).

Die Idee hinter der Verwendung von I/O-Threads zur Steigerung der Schreib-/Leseperformance ist es, den Prozessor möglich nie unbeschäftigt zu lassen. Werden mehrere Threads verwendet, muss nicht immer erst die Beendigung eines Prozesses abgewartet werden, um die nächste Operation auszuführen, sondern es können mehrere Schreibeoperationen gleichzeitig durchgeführt werden.

Mit ABT werden ROMIO's Schreibprozesse unterbrochen und stattdessen mittels active buffering aufgehalten bzw. verzögert. Der ABT Hauptprozess alloziert für jede Schreibanfrage Pufferspeicher und schreibt den Inhalt der Anfrage hinein. Ein im Hintergrund laufender Thread ruft die Daten des Puffers ab und sendet sie an das darunterliegende Dateisystem. Hauptprozess und Hintergrundprozess agieren in einer klassischen Produzierer-Konsument Abfolge. Die von den Prozessen verwendete Queue wird ABS („Available Buffer Size“) genannt und je nachdem, ob sie gefüllt oder ausgelesen wird, verkleinert bzw. vergrößert. Für eine optimale Performance von ABT ist es essentiell, eine geeignete Größe für ABS zu finden. Wenn ABS kleiner ist als die Speichermenge, die benötigt wird, um eine Schreibeanfrage zu puffern, wird der ABT Hauptprozess solange blockiert, wie der Hintergrundprozess an Zeit braucht, um ABS auszulesen. Dadurch würde ein Teil der (Zeit-) Kosten, die sonst im Hintergrund verborgen blieben, für den Hauptprozess ersichtlich. Andererseits aber, wenn ABS deutlich größer angelegt ist als die typischerweise erforderliche Speichermenge zum Zwischenpuffern, entsteht ein erhöhter Synchronisations-aufwand. Das Ein- bzw. Auslesen vieler kleiner Datenmengen mit Hilfe der Queue bringt hohe Unkosten mit sich, die sich ebenfalls negativ auf die Gesamtleistung auswirken. Um dies zu verhindern, fasst ROMIO nicht zusammenhängende oder

kleine Datenfragmente mittels „Data Sieving“ zusammen [Thakur, 99]. Dabei wird für mehrere, auf unterschiedliche Speicherstellen zugreifende Leseprozesse ein einzelner Datenblock gespeichert, in dem sich alle angefragten Speicherstellen befinden. Diese werden dann einzeln aus dem Block entnommen und dem entsprechenden Benutzerprozess übergeben. Bei Schreibzugriffen wird der komplette Data Sieving Puffer eingelesen, mit den zu schreibenden Daten aktualisiert und auf permanenten Speicher geschrieben. Um den Aufwand zu minimieren, wird für alle I/O-Operationen lediglich ein Prozess verwendet, der als Daemon im Hintergrund agiert. Dies hat mehrere Vorteile:

- Es minimiert den Aufwand für Prozess Scheduling bzw Switching
- Es minimiert den Synchronisationsaufwand, der entstehen würde, wenn mehrere Prozesse auf gemeinsam benutzte Datenstrukturen wie die Puffer der Queue bzw. ABS selbst zugreifen würden.
- Indem die von ROMIO schon „voroptimierte“ Reihenfolge sequenzieller Schreibprozesse durch die Verwendung lediglich eines I/O-Threads nicht gestört sondern eingehalten wird, wird die tatsächliche IO Performance sogar noch erhöht.

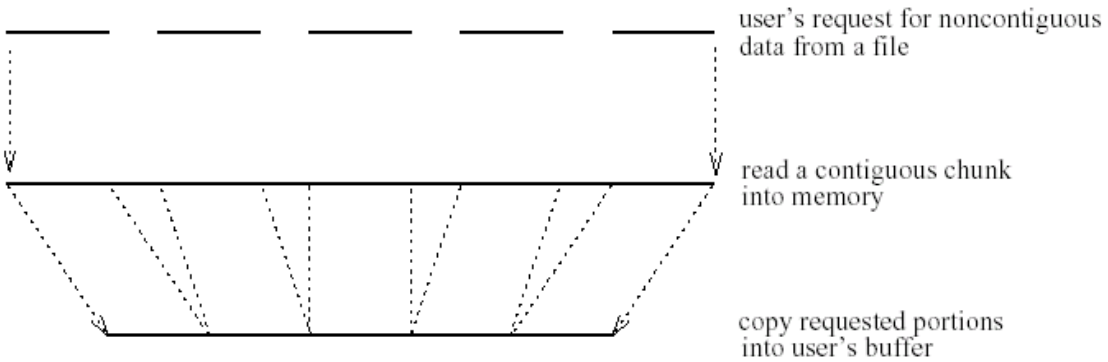

Abbildung 1: Data Sieving

Wissenschaftliche Berechnungen bzw. Simulationen, die kontinuierlich Ergebnisdaten produzieren, es aber nicht erfordern, diese Daten gleich wieder einzulesen, profitieren insbesondere von ABT. Die flexible Art und Weise, wie unbenutzter Speicher zum Verdecken von I/O-Kosten bzw. zum Überlappen mit Berechnungsoperationen verwendet wird, macht ABT besonders interessant für derlei Anwendungen.

Kapitel 3: Das RFS Design

Wie bereits zuvor erwähnt, verwendet RFS die innerhalb von ROMIO liegende Zwischenschicht ADIO. Innerhalb von ADIO sind einige grundlegende I/O-Schnittstellen definiert, die zur Entwicklung komplexerer, höherschichtiger Schnittstellen wie z.B. bei MPI-IO verwendet werden können. Dateisysteme, die ADIO verwenden, müssen eben diese generischen Schnittstellen implementieren, die dann zusammengefasst als „Modul“ dateisystemspezifisch sind.
RFS besteht aus zwei Komponenten, einem Client RFS Modul und einem als Server implementierten sogenannten „Request Handler“. Wenn auf Clientseite I/O-Funktionalität angefordert wird, kommuniziert das RFS Client Modul mit dem Request Handler, um die angefragten I/O-Operationen durchzuführen.

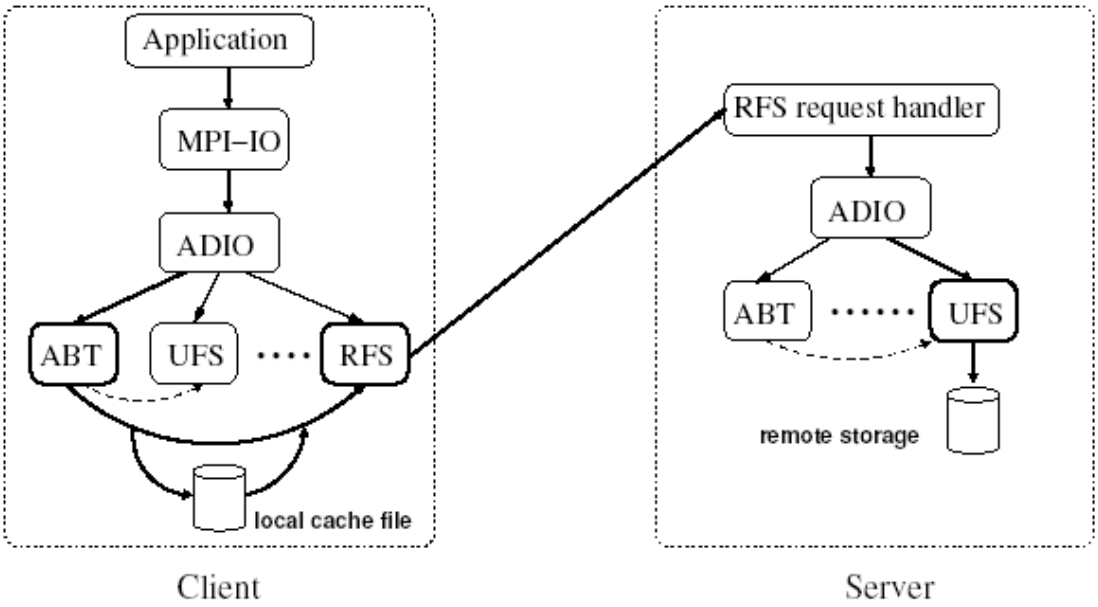

Abbildung 2: RFS Architektur

3.1 RFS ADIO Modul

Auf der Clientseite sind einige grundlegende ADIO Funktionen in RFS implementiert. Diese umfassen vorallem Schreibe- bzw. Leseoperationen von zusammenhängenden bzw. nicht zusammenhängenden Datenblöcken.
Für I/O-Zugriffe auf nicht zusammenhängende Daten verwendet ROMIO das bereits vorgestellte „Data Sieving“. Bei Lesezugriffen wird zuerst der ganze Datenblock –

vom Beginn des ersten bis zum Ende des letzten angefragten Datenstücks – eingelesen und dann die angefragten Daten ausgewählt. Bei Schreibzugriffen wird der zu schreibende Datenblock in den Speicher geschrieben, durch die neuen Daten aktualisiert und dann wieder komplett zurückgeschrieben.
Dem hierdurch entstandenen Vorteil – Verringerung des Overheads durch ständige Datenzugriffe – steht aber auch ein klarer Nachteil entgegen. Auf Netzwerkebene ist es nicht nur wichtig, die Anzahl der I/O-Operationen zu minimieren, sondern auch, I/O-Datentransfer möglichst gering zu halten. Dies gilt insbesondere für Schreibzugriffe, da hierbei ja erst ein großer Datenblock über das Netzwerk transportiert werden muss, dann aktualisiert und wieder per Netzwerkverbindung zurückgeschrieben werden muss.
RFS bietet hierfür eine spezielle MPI Operation an. MPI_Pack wird auf Clientseite verwendet, um zu schreibende Daten zu komprimieren. Diese sind dann vom Typ MPI_PACKED und werden so zum RFS Server gesendet.
Für die Kommunikation zwischen Client RFS Modul und dem Server verlangt RFS keinerlei spezifische Protokolle. Stattdessen beinhaltet RFS vier sehr einfach gehaltene Funktionen, um verbindungsorientierte Datenströme verwirklichen zu können.

```
RFS_handle RFS_Make_connection(char *host, int port);
int RFS_Writen(RFS_handle handle, char *buf, int count);
int RFS_Readn(RFS_handle handle, char *buf, int count);
int RFS_Close_connection(RFS_handle handle);
```

Abbildung 3: RFS Kommunikationsfunktionalität

Solange es für diese Funktionen eine protokollspezifische Implementierung gibt, kann das entsprechende Protokoll auch verwendet werden. Im Moment wird als Standard TCP/IP verwendet.

3.2 Integration des ABT Moduls

Das bereits ausführlich beschriebene ABT wird wie die Clientkomponente von RFS als ADIO Modul implementiert. Wenn ABT aktiviert ist, unterbricht es alle Lese- bzw. Schreibzugriffe auf das Dateisystem, puffert die zu schreibenden Daten mitsamt dem entsprechenden I/O-Aufruf im Speicher und gibt seine Prozessorressourcen

wieder frei. Gleichzeitig führt ein ABT Hintergrundprozess den I/O-Aufruf auf den gespeicherten Daten aus. Der Performance-Vorteil hierbei ist recht eindeutig: während das (Simulations-)Programm schon wieder Berechnungen anstellen kann, wird parallel im Hintergrund der (zeitraubende) Daten-I/O vorgenommen. Aufgrund seiner Struktur als ADIO Modul kann ABT direkt mit RFS interagieren.

Trotz der genannten Vorteile gibt es für zwei Konstellationen immer noch Optimierungspotential im Zusammenspiel von ABT und RFS. Im ersten Fall müsste ABT auf die Freigabe von hinreichend vorhandenem Speicher zur Pufferung von I/O-Daten warten, falls dieser nicht sofort verfügbar ist. Im Zusammenspiel mit RFS aber, insbesondere bei weitverzweigten Kommunikationsstrukturen, könnte dies einen enormen Performance-Verlust bedeuten. Um dies zu umgehen, schreibt ABT sofort nach dem Abfangen die Daten in einen lokalen Cache des Client-Dateisystems. Lediglich die Beschreibung der abgefangenen I/O-Operationen sowie einige Metadaten werden im Speicher gehalten. Der Hintergrundprozess überprüft nun immer zuerst den lokalen Cache und alloziert bei einem positiven Check genug Speicher für die Daten. Sobald diese im Speicher verfügbar sind, wird die entsprechende I/O-Operation ausgeführt. Die Behebung dieses ersten Falls wird „Foreground Staging“ genannt.

Im zweiten Fall sollen die Kosten, die durch Daten-I/O entstehen, sogar noch weiter vermindert werden. Hierfür werden Teile der im Speicher gehaltenen I/O-Daten auf lokalen Festspeicher geschrieben, um sicherzustellen, dass für den nächsten I/O-Aufruf der Simulationssoftware genug Speicher vorhanden ist. Dafür ist es natürlich erforderlich, zu wissen, wieviel Zeit bis zum nächsten I/O-Aufruf vergeht bzw. wie groß die zu speichernde Datenmenge ist. Typischerweise schreiben Simulationsberechnungen immer die gleiche Menge an Ergebnisdaten, meist auch in ungefähr gleichen Zeitabständen. Hierfür also wäre das sogenannte „Background Staging“ optimal. Falls aber Unregelmäßigkeiten auftreten, besteht immer noch die Möglichkeit, die Simulation und ihr „typisches“ Verhalten zu analysieren bzw. dem Benutzer des Systems die Möglichkeit einzuräumen, Angaben über Datenfolge bzw. Datenmenge zu machen. Background Staging sollte allerdings nur in Ausnahmefällen eingesetzt werden, da dem Gewinn an Speicherplatz im RAM der Nachteil entgegensteht, bereitgestellte Daten immer erst von Platte einlesen zu müssen, bevor sie zum Server gesendet werden können.

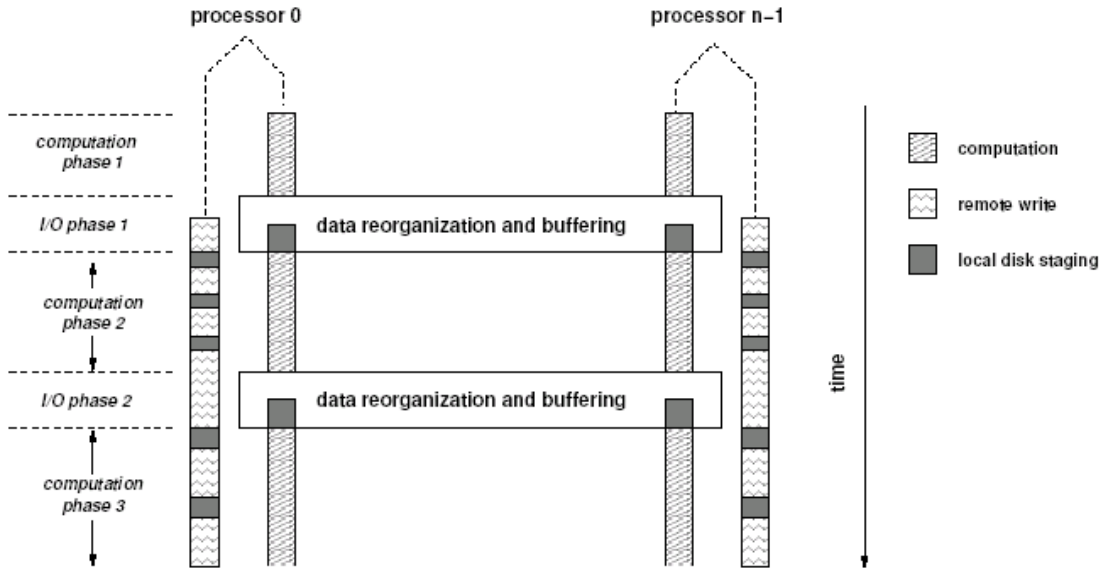

Abbildung 4: RFS und ABT

3.3 RFS Request Handler

Der Server ist als Systemdaemon auf dem Datenserver des Simulationssystems implementiert und kann mehrere Prozessoren benutzen (SMP-fähig).
Im Vergleich zum RFS Client Modul ist der Aufbau des Request Handlers recht simpel. Wann immer ein RFS Client I/O-Anfragen an den Server schickt, werden diese lokal ausgeführt und der Client erhält eine Bestätigung in Form von „Errorcodes“ bzw. im Falle eines Lesezugriffs die entsprechenden Daten. Kommunizieren mehrere Clients gleichzeitig mit dem Request Handler, erzeugt dieser für jede Anfrage einen eigenen Prozess und bearbeitet die Anfragen gleichzeitig. Für den Fall, dass mehrere Schreibzugriffe auf einen einzelnen Datenblock erfolgen, muss RFS keinerlei zusätzliche Funktionalität bereitstellen, denn MPI verlangt grundsätzlich vom Benutzer eines Systems, bei mehrfachem Zugriff auf einen einzelnen Datenblock für die Konsistenz der Daten zu sorgen.

Kapitel 4: Tests

Die Testumgebung befindet sich im Argonne National Laboratory und besteht aus einem Linux Cluster, – „Chiba City“ genannte – der 512 Prozessoren umfasst. Der

Datenserver befindet sich an der Universität Illinois und ist ein 1,4 GHz 1 Prozessor System („Elephant“). Sowohl die Clusterknoten untereinander als auch Cluster und Serversystem sind über eine 100 Mbit/s Fast Ethernet Leitung miteinander verbunden. Grundsätzlich verfolgt der Test die Verwirklichung zweier Ziele. Zum einen soll gezeigt werden, wie hoch der mittels RFS erziele Datendurchsatz gegenüber dem durch den typischen I/O-Flaschenhals entstandenen Durchsatz ist. Zum anderen soll ermittelt werden, in wie weit RFS im Zusammenspiel mit ABT die tatsächlich anfallenden Kosten des I/O-Transfers durch Überlappung dieser mit Berechnungsoperationen verdecken kann.

Als Testapplikation wird ein paralleles 3D Jacobi Relaxation Programm verwendet. Jacobi berechnet bei jeder Iteration den Wert einer Zelle, indem der Durchschnittswert aller benachbarter Zellen aus der vorangegangenen Iteration ermittelt wird. Zwischenergebnisse werden in einer vom Benutzer zu spezifizierenden Anzahl von Iterationen auf Festplatte gespeichert. Um durch RFS erzielte Ergebnisse mit Standard I/O-Werten vergleichen zu können, werden insgesamt 6 unterschiedliche Dateisystemkonfigurationen definiert.

1. PEAK

Hierbei wird der theoretisch maximal mögliche Schreibdurchsatz ermittelt, allerdings ohne wirklich Daten auf Datenträgern zu speichern (quasi ein unendlich schneller Festspeicher). Hierfür wird ROMIO's TESTFS ADIO Modul verwendet.

2. LOCAL

Hiermit wird der Schreibdurchsatz der lokal auf den einzelnen Clusterknoten installierten Festplatten ermittelt.

3. RFS

Nur RFS Funktionalitäten werden verwendet, keinerlei Optimierungen mittels ABT. Somit sind alle anfallenden Datentransferkosten sichtbar.

4. RFS + ABT -large -long

RFS und ABT werden im Zusammenspiel getestet. “Large” bedeutet, dass auf jedem eingebundenen Prozessor der verfügbare Bufferspeicher gleich groß oder größer als der maximal benötigte Speicher je I/O-Operation ist. “long” besagt, dass die Zeit zur Berechnung eines Datensatzes lange genug ist, um den im Hintergrund ablaufenden Schreibprozess vor der Simulation zu verdecken. Dementsprechend wird kein “local staging” verwendet.

5. RFS + ABT -large -short

Im Unterschied zur vorgehenden Konfiguration ist die Berechnungszeit kürzer

("short") als die für den Schreibprozesse benötigte Zeit. Folglich kann es erforderlich sein, "background staging" einzusetzen, um genug Bufferspeicher für die nächste I/O-Operation bereitstellen zu können.

6. RFS + ABT -small -long

Hierbei ist zwar die Berechnungszeit lange genug, der zur Verfügung stehende Bufferspeicher jedoch ist zu klein, um die Daten einer einzelnen I/O-Operation zwischenzuspeichern ("small"). Hierdurch wird "foreground staging" verursacht, dessen Kosten für die Gesamtperformance ersichtlich werden.

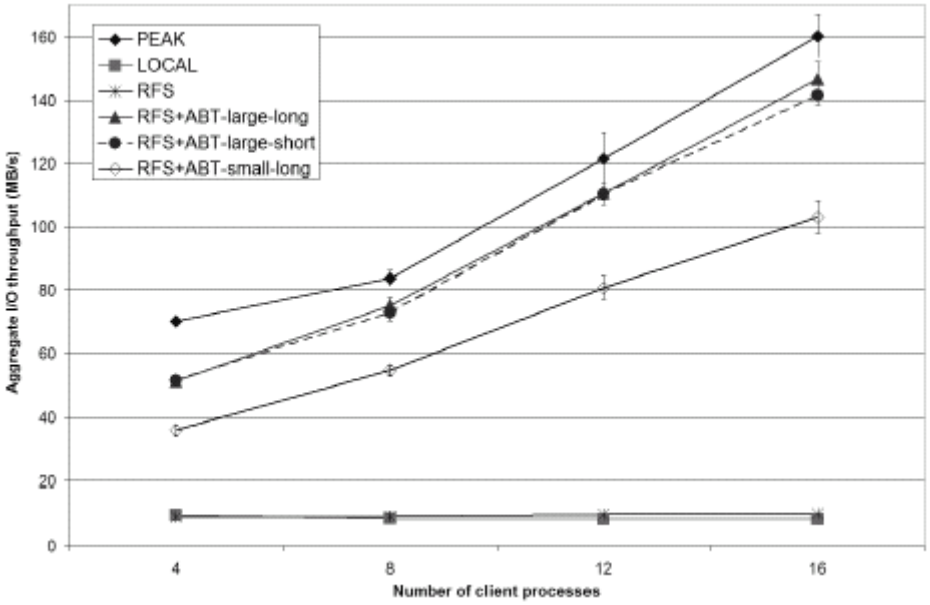

Abbildung 5: Test Ergebnisse

Abbildung 5 zeigt das Verhältnis von angesammeltem I/O-Datendurchsatz zur Anzahl der verwendeten Prozessoren für jede einzelne der oben beschriebenen Dateisystemkonfigurationen. Die dabei anfallenden I/O-Kosten sind jedoch unterschiedlicher Natur. Für LOCAL und RFS fallen hierbei die Zeit für das Erreichen der I/O-Daten vom Client zum Dateisystem sowie die Zeit für das Versenden etweiliger Fehlermeldungen zusammen. Für die Konfigurationen mit ABT sind I/O-Kosten als solche Kosten spezifiziert, die durch den Zeitaufwand entstehen, den lokales Speicherpuffern bzw. "foreground staging" verursachen.

Für den Durchsatz von PEAK ist die Anzahl der verwendeten Prozessoren signifikant. Werden 16 Prozessoren eingesetzt, wird ein maximaler Durchsatz von 160,2 MB/s erreicht. Da für die Messung von PEAK keinerlei lokaler Plattenspeicher verwendet wird, limitiert das bei Chiba City verwendete Fast Ethernet die maximal erreichbare

Bandbreite (16 x 100 Mbit/s). LOCAL hingegen ist durch den maximal verfügbaren Durchsatz (9,8 MB/s) der verwendeten Festplatte (IDE-ATA Standard) gesetzt. Wird nur RFS eingesetzt, so ist die zwischen Cluster und Server verfügbare Netzwerkbandbreite (100 Mbit/s) der limitierende Faktor. Dementsprechend erreicht RFS ca. 10,1 MB/s und damit 86% der verfügbaren Bandbreite. Die übrigen 14% Leistung gehen dadurch verloren, dass von RFS gesteuerte Schreibvorgänge die durch das Ansprechen von Festplatten entstehenden Verzögerungen sowie die für den Transport der Fehlermeldungen benötigten Zeit mit einbeziehen. Generell ist anzumerken, dass die Einbeziehung von RFS keinen negativen Einfluss auf den sonst verfügbaren lokalen Datendurchsatz hat. Wird nun ABT eingesetzt, erhöht sich der maximal verfügbare Schreibdurchsatz erheblich. Bei "RFS+ABT+large+long" wird ein Maximum von 146,7 MB/s erreicht, fast 92% der theoretisch verfügbaren Datendurchsatzrate oder auch 14,5 mal mehr als bei reiner Verwendung von RFS. Die fehlenden 8% zum absoluten Maximum erklären sich dadurch, dass die Kopiervorgänge der I/O-Daten in die "active buffers" sowie die parallel stattfindenden Prozesse für "foreground buffering" und "background remote I/O" Rechen- bzw. Zeitkosten verursachen. Wenn wie bei "RFS+ABT+large+short" die Berechnungsphase nicht lange genug ist, um eine komplette "remote I/O" Operation durchführen zu können, ist der erzielte Datendurchsatz immer noch annähernd so gut wie bei längeren Berechnungsphasen. Nie gehen mehr als 4% an Durchsatz verloren, vorausgesetzt, es steht immer genug Speicher für das Zwischenspeichern der nachfolgenden I/O-Daten zur Verfügung. Wenn wie in der letzten Konfiguration RFS "foreground staging" durchführen muss, werden die dadurch entstehenden I/O-Kosten offensichtlicher. Mit 103,1 MB/s wird aber immer noch ein 12.4fach höherer Datendurchsatz als bei LOCAL erzielt.

Um abschätzen zu können, in wie weit die IO Hintergrundprozesse von ABT die Ausführung des Jacobi Tests beeinflusst haben, wird eine zweite Messung vorgenommen. Abbildung 6 stellt den durchschnittlichen Performance-Verlust der Berechnungsphase im Zusammenhang mit zwei ABT-spezifischen Konfigurationen dar.

	No. Procs	4	8	12	16
Chiba to Elephant	RFS+ABT-large-long	2.33%	0.68%	0.53%	1.41%
	RFS+ABT-large-short	6.24%	3.62%	1.97%	2.46%

Abbildung 6: RFS + ABT Unkosten

Aus den Werten wird deutlich, dass die maximal auftretende Verlangsamung der Berechnungsleistung höchstens 6,24% beträgt. Dieser Verlust wird jedoch durch den Gewinn, der aus der Überlappung von Berechnungsoperationen und I/O-Operationen erzielt wurde, mehr als wett gemacht.

Kapitel 5: Diskussion

RFS in Zusammenarbeit mit ABT ist auf schreibintensive wissenschaftliche Anwendungen hin entwickelt bzw. optimiert worden. Lesezugriffe treten bei Simulationsberechnungen normalerweise nicht häufig auf und sind somit bei der Entwicklung vernachlässigt worden. Dennoch kann es wichtig sein, auch Leseoperationen im Bereich paralleler Berechnungen zu optimieren. Der traditionelle Ansatz hierbei ist es, angefragte Daten schon zuvor abzurufen („prefetching“) und gegebenenfalls diese Daten für wiederholten Zugriff zwischenzuspeichern („caching“). RFS bedient sich diesen Ansatzes und fügt entsprechende Funktionalitäten wie z.B. einen lokalen Zwischenspeicher fürs caching bei ABT hinzu. Umgesetzt wird dies, indem eine eigene Schnittstelle spezifiziert wird, die es den Hintergrundprozessen ermöglicht, unmittelbar nach der Leseanfrage auf einen Datensatz diesen mittels prefetching bereitzustellen. Wenn nun die Leseoperation ausgeführt wird, wird zuerst der bereits lokal abrufbare Teil der Daten gelesen und – wenn erforderlich – der Rest über das Netzwerk transferiert. Für das Caching verhält es sich ähnlich, zuerst die lokal und damit schnell verfügbaren Daten lesen und gegebenenfalls den Rest vom entfernten Server abholen.
In der aktuellen Entwicklungsstufe von RFS ist es dem Benutzer des Systems möglich, I/O-Anfragen für Schreib- oder Lesezugriffe bzw. open()- und close()-Aufrufe zu verzögern, um den Netzwerkverkehr zu minimieren bzw. Antwortzeiten zu verbessern. Eine Folge hiervon kann es jedoch sein, dass Systeminformationen, die einen Fehler bei der Ausführung der entsprechenden I/O-Operation melden, verzögert beim Benutzer ankommen können. Wenn also die unmittelbare Benachrichtigung über Systemfehler Priorität hat vor Geschwindigkeitsgewinnen, sollte RFS in der aktuellen Entwicklungsstufe nicht verwendet werden.

Kapitel 6: Schlussfolgerung

RFS stellt eine effektive Lösung auf dem Gebiet verteilter, paralleler I/O-Operationen im Bereich MPI-IO dar. Seine einfach gehaltene und hinreichend flexible Architektur macht es möglich, auf eine effektive Art und Weise Zugriffe auf entfernt gespeicherte, zusammenhängende bzw. nicht zusammenhängende Daten zu ermöglichen. Im Zusammenspiel mit ABT wird durch offensives Zwischenspeichern von zu schreibenden Datensätzen bzw. durch die Verminderung von Kommunikations-unkosten eine Überschneidung von Berechnungs- und I/O-Operationen erreicht, die zu einer gesteigerten Gesamtperformance maßgeblich beitragen. Weiterhin können mittels „data staging“ Latenzzeiten minimiert bzw. für die Berechnung unsichtbar gemacht werden. Die gemachten Tests zeigen, dass RFS ohne ABT nicht zu einer verminderten Gesamtperformance führt, sie liegt sogar leicht über der der langsamsten im System eingesetzten Komponente, dem Festplattensystem. Wird aber ABT aktiviert, erreicht RFS bis zu 92% der theoretisch überhaupt erzielbaren Performance. Selbst wenn die einzelnen Berechnungsphasen der Simulation zu kurz sind, um sämtliche im Hintergrund stattfindenden I/O-Prozesse abzuschließen, ist der Performancegewinn beträchtlich. Die durch den Einsatz von RFS und ABT hervorgerufene Zusatzkosten bleiben relativ gering und schrumpfen im Vergleich zum erzielten Gewinn auf eine unbedeutende Größe zusammen.

Bibliographie

[MPI, 97] – Message Passing Interface Forum. *MPI-2: Extensions to theMessage-Passing Standard*. 1997.

[Foster, 97] - I. Foster, D. Kohr, Jr., R. Krishnaiyer, and J. Mogill. Remote I/O: Fast access to distant storage. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1997.

[GASS, 99] – J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[GridFTP, 02] – B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnal, and S. Tuecke. Data management and transfer in high performance computational grid environments. *Parallel Computing Journal*, 28(5):749–771, 2002. 2001.

[Seamonns, 95] – K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, November 1995.

[Ma, 03] – X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPIIO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[Thakur, 99] – R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.