Fachhochschule
Bonn-Rhein-Sieg

Fachbereich Informatik
Fachhochschule Bonn-Rhein-Sieg
Grantham-Allee 20
D-53757 Sankt Augustin

Seminararbeit

Seminar:
Parallel Systems
SS 05

Thema:
Remote I/O: Fast Access to Distant Storage

im
Studiengang
Master of Computer Science

von
Christian Leuschen
Christoph Schmitz

Betreuender Dozent: Prof. Dr. Rudolf Berrendorf

12. Juni 2005

Inhaltsverzeichnis

1. Einführung 3
2. Motivation für Remote I/O 3
3. RIO – Die Remote I/O Bibliothek 6
4. Experimente 9
   4.1. Die Testplattform 10
   4.2. Ergebnisse der speziell definierten Benchmarks 10
   4.3. Ergebnisse bei einer ausgesuchten Anwendung 13
5. Fazit und Ausblick 16
6. Literatur 17
7. Tabellenverzeichnis 17
8. Abbildungsverzeichnis 17

1. Einführung

Hochgeschwindigkeitsnetzwerke ermöglichen heutzutage eine effektive Nutzung von verteilten Ressourcen, d.h. häufig ist eine zentrale Datenhaltung nicht gegeben bzw. nicht notwendig. Anwendungen werden dabei auf entfernten Rechnern ausgeführt und so ausgelegt, dass sie Ressourcen von verschiedenen Orten nutzen. Bei solch einer „entfernten Ausführung“ sind Anwendung und Daten geographisch voneinander getrennt. Idealerweise sollte der Datenzugriff dabei unabhängig vom Speicherort sein, d.h. er sollte sich nicht von einer „lokalen Ausführung“ unterscheiden lassen. Doch gerade die Trennung von Anwendung und Daten stellt neue Herausforderungen an die Entwickler dar, wie z.B. den zusätzlichen Aufwand für die Kommunikation der Daten. Um einen vom Speicherort unabhängigen Datenzugriff zu gewährleisten, ist daher die Auseinandersetzung mit den Punkten Portabilität (zwischen verschiedenen Netzwerken und Dateisystemen), Performanz (in heterogenen Netzwerken) und Integration (in die Umgebung von verteilten Systemen) unabdingbar.

Ein trivialer Ansatz, der entfernten Dateizugriff ermöglicht, ist das so genannte Staging von Daten. Dabei werden die Daten an einen entfernten Rechner gesendet. Auf dem entfernten Rechner findet die Bearbeitung der Daten statt. Nach der Bearbeitung werden die Daten wieder an den Nutzer zurückgesendet. Dieser Ansatz ist deutlich vom Ziel des unabhängigen Datenzugriffes entfernt. Das Staging von Daten ist umständlich. Es verhindert die Überlappung von Kommunikation und Berechnung der Daten, da zunächst alle Daten an den entfernten Rechner übertragen werden müssen, bevor dieser mit der Berechung beginnen kann. Es verursacht unnötig hohe Netzlast in Fällen, in denen nur ein Teil einer Datei benötigt wird, jedoch die ganze Datei aufgrund des Protokolls übertragen werden muss.

Der in dieser Arbeit vorgestellte Ansatz für entfernten Dateizugriff, die Remote I/O Library (RIO), versucht die Unzulänglichkeiten vom Staging zu umgehen. RIO ist eine I/O Bibliothek, die für schnelle, entfernte Dateizugriffe ausgelegt ist, wie sie vor allem in parallelen Anwendungen bei Zugriffen auf entfernte Dateisysteme nötig sind. Bevor diese Arbeit detailliert auf RIO eingeht, wird zunächst kurz der Nutzen von Remote I/O motiviert. Anschließend werden Konzept und Technik von RIO erläutert. Den Abschluss bilden eine Betrachtung und ein Fazit der Ergebnisse von RIO.

2. Motivation für Remote I/O

Die zuvor beschriebenen Nachteile beim Ansatz des Stagings verdeutlichen, dass Staging sich nicht für effiziente entfernte Ausführungen eignet. Insbesondere die Mängel bezüglich der

Performanz und Flexibilität machen es ungeeignet für anspruchsvolle, parallele Anwendungen. Remote I/O hilft diese Mängel zu beheben.
Remote I/O erreicht eine Steigerung der Performanz, d.h. die insgesamt benötigte Ausführungszeit wird reduziert, indem es die Überlappung von Datentransfer und Berechnung ermöglicht. Während beim Staging gewartet werden muss, bis alle Daten übertragen werden wurden, kann durch die Überlappung schon mit der Berechnung begonnen werden, auch wenn noch nicht alle Daten gesendet worden sind. Im optimalen Fall kann dadurch die Ausführungszeit um den Faktor 2 im Vergleich zum Staging gesenkt werden.
Remote I/O ist flexibler als Staging. Wenn z.B. die Identität zur Berechnung benötigter Daten erst während der Ausführung bestimmt werden kann, werden beim Staging häufig mehr Daten als nötig übertragen. Dadurch werden Ressourcen sowohl beim Speicherplatz als auch beim Netzwerk unnötig in Anspruch genommen. Mit Hilfe von Remote I/O ist es möglich, nur die benötigten Datenelemente zu übertragen. Das spart Speicherplatz und reduziert die Netzwerklast.
Remote I/O bedeutet mehr Komfort für den Entwickler. Beim Staging braucht der Entwickler häufig Kenntnis über das entfernte Dateisystem, die unterschiedlichen Verzeichnishierarchien und ggf. eine notwendige Konvertierung der Daten. Eine Remote I/O Bibliothek kann diese Fragenstellungen automatisieren, also z.B. die richtige Konvertierung der Daten veranlassen. MPI-IO unterstützt diese automatische Konvertierung. Dadurch können Fehler vermieden werden, die der Entwickler beim Staging begehen könnte.

Diese Vorteile erhält man nicht umsonst. Bei der Entwicklung einer Remote I/O Bibliothek gilt es u.a Fragestellungen bezüglich der Performanz, Heterogenität der Komponenten und Fehlertoleranz zu beachten.
Die Leistungscharakteristik einer Remote I/O Bibliothek wird entscheidend durch das verwendete Netzwerk bestimmt. Die Latenz bei einer Verbindung über ein „kontinentales“ Netzwerk, also z.B. das Internet, liegt bei ungefähr 100 msec. Das ist um die Größenordnung drei- bis viermal mehr als bei der Kommunikation innerhalb eines parallelen Systems (0,1 – 0,01 msec) und immerhin noch um eine Größenordnung langsamer als bei einem Speicherzugriff innerhalb eines parallelen Systems (10 msec). Auch wenn mit kommenden Generationen von Netzwerken die Unterschiede bei der Leistung zwischen der Kommunikation innerhalb und außerhalb eines parallelen Rechners geringer werden dürften, bleibt das verwendete Netzwerk ein limitierender Faktor.
Die Konfiguration eines Remote I/O System erfordert mehr Aufwand als in einem typischerweise homogenen parallelen System. Die häufig gegebene Heterogenität der Komponenten - die

Mischung aus verschiedener Hardware, Software und Protokollen – in einem Remote I/O System macht diese Aufgabe so schwierig. Damit verbunden ist eine hohe Variabilität der Dienstgüte (z.B. Durchschnittliche Bitrate oder Ausfallsicherheit) in einem solchen System. Hier gilt es geeignete Maßnahmen zu finden, um gegen diese Variabilität geschützt zu sein, also z.B. Pufferung einzusetzen bzw. lokale Kopien vorzuhalten, um Datenverlust durch ausgefallene Komponenten zu vermeiden.
Weitere wichtige Punkte für die korrekte Arbeitsweise eines Remote I/O Systems sind Sicherheitsaspekte (Zugriffsrechte, Integrität), Namensräume für Dateien (RIO nutzt einen URL-ähnlichen Ansatz) und Fehlertoleranz (z.B. Checkpoints in einer Anwendung bei fehlgeschlagenen Operationen).

An dieser Stelle werden kurz andere Ansätze, die entfernten Datenzugriff ermöglichen, erwähnt. Die Ansätze lassen sich in drei Gruppen aufteilen: Verteilte Dateisysteme, parallele Dateisysteme und „Remote Execution Systeme“.
Verteilte Dateisysteme, wie z.B das Andrew File System (AFS) oder das Distributed File System (DFS), bieten komfortable Schnittstellen für Remote I/O. Der Dateizugriff erfolgt über konventionelle Lese- und Schreiboperationen und ein globaler Dateinamensraum wird unterstützt. Ein Nachteil dieser Systeme ist, dass sie nicht für rechenintensive parallele Anwendungen ausgelegt sind und daher keine gute Performanz erreichen. So fehlen den meisten verteilten Dateisystemen z.B. Schnittstellen für kollektive Operationen und die Unterstützung für parallele Zugriffe.
Diese Nachteile finden sich bei parallelen Dateisystemen natürlich nicht. Diese Systeme erreichen eine gute Leistung für parallele Anwendungen durch explizite I/O Schnittstellen, die kollektive sowie asynchrone Operationen und spezielle Puffertechniken erlauben. Was parallele Dateisysteme für Remote Netzwerke ungeeignet macht, ist die Tatsache, dass sie nicht für den Einsatz in solchen Netzwerken entwickelt worden sind und die speziellen Anforderungen an sie nicht erfüllen können.
Die dritte Gruppe, die „Remote Execution Systeme“, sind für entfernten Zugriff und Ausführung von Dateien ausgelegt. Nachteil solcher Systeme, wie z.B. Condor bzw. WebOS ist jedoch, dass sie keinerlei Unterstützung für parallele I/O bieten.
Zusammenfassend lässt sich sagen, dass die existierenden Techniken entweder Ansätze für die entfernte Ausführung oder die Performanz im Parallelen bieten, jedoch nicht für beides. Hier setzt RIO an, indem es versucht die hohen Leistungsanforderungen an eine parallele I/O Schnittstelle zu unterstützen und gleichzeitig die entfernte Ausführung in einer Netzwerkumgebung zu ermöglichen.

3. RIO – Die Remote I/O Bibliothek

Ziel bei der Entwicklung von RIO war es, eine I/O Bibliothek zu realisieren, die flexibel einsetzbar ist, d.h. eine große Bandbreite von Anwendungen und Dateisystemen unterstützt. Anstatt eine komplett neue I/O Schnittstelle zu definieren, wurde deshalb auf bestehende und erprobte Komponenten zurückgegriffen. So setzt sich die Bibliothek RIO aus MPI-IO, ROMIO, Nexus und ADIO zusammen. Diese Komponenten werden benötigt, um einen RIO-Server aufzusetzen und die RIO Bibliothek mit entsprechenden Anwendungen zu verlinken. MPI-IO stellt dabei die prinzipielle Schnittstelle zu den Anwendungen dar. ROMIO ist die Implementierung von MPI-IO, die RIO nutzt. Somit wird jede Anwendung unterstützt, die ROMIO benutzt. Für die Kommunikation zwischen Client und Server setzt RIO die Bibliothek Nexus ein. Als Vermittlungsschicht zwischen unterschiedlichen Dateisystemen nutzt RIO die Eigenschaften von ADIO aus.

Die in der Arbeit schon angesprochene Heterogenität der Architektur in einem Remote Netzwerk stellt eine große Herausforderung an die Portabilität einer Remote I/O Bibliothek dar. RIO begegnet diesem Problem durch die Nutzung von ADIO. ADIO ist eine Schnittstelle zwischen APIs für parallele I/O und verschiedenen Dateisystemen. ADIO dient dabei als universelle Vermittlungsschicht, die auch inkompatible Dateisysteme miteinander kommunizieren lässt. Eine detaillierte Beschreibung zu ADIO findet sich in der Seminararbeit „ROMIO“ von Jan Seidel.
Die Verwendung von ADIO durch RIO ist in der folgenden Abbildung verdeutlicht. RIO nutzt ADIO auf Client- und Server-Seite jeweils unterschiedlich.

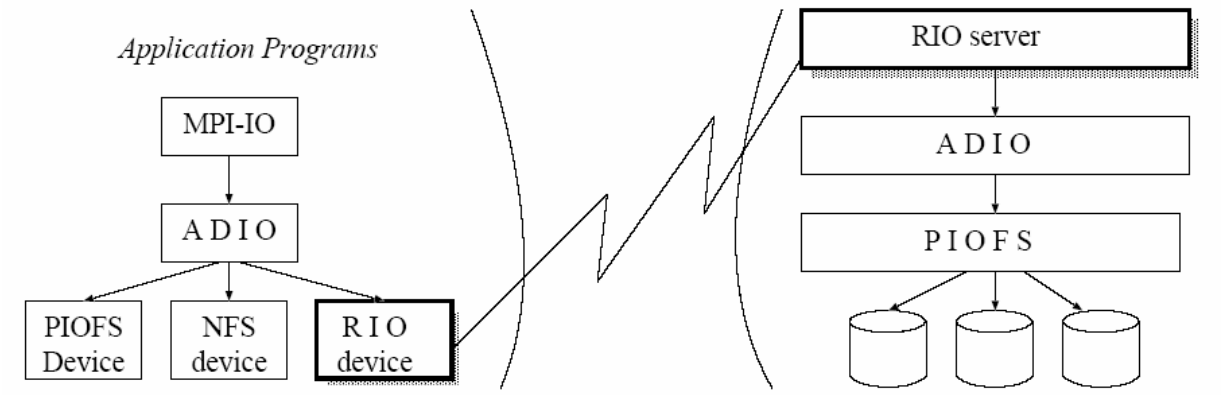

Abbildung 1: Architektur von RIO (links:Client - rechts:Server) [aus: Foster et al. 1997]

Auf der Client-Seite definiert RIO einen „Remote I/O Device“ für ADIO. Diese Komponente übersetzt Anfragen von ADIO in entsprechende Interaktionen mit einem entfernten I/O Server.

Auf der Server-Seite nimmt RIO die Anfragen eines entfernten RIO Clients entgegen und nutzt ADIO-Aufrufe, um auf das entfernte Dateisystem zuzugreifen. Der Ansatz RIO gleichzeitig oberhalb von ADIO (Server) und unterhalb von ADIO (Client) zu nutzen, zeichnet sich durch seine Transparenz aus. Der Client braucht sich keine Gedanken über das entfernte Dateisystem zu machen und der Server kann jedes Dateisystem nutzen, das von ADIO unterstützt wird.

Der folgende Abschnitt zeigt exemplarisch, wie RIO MPI-IO nutzt, also die Implementierung ROMIO, um parallel auf eine Datei zuzugreifen. Die Nutzung von ROMIO erlaubt es RIO alle Eigenschaften auszunutzen, die MPI-IO bietet, d.h. kollektive Operationen, nicht blockierende Aufrufe, individuelle Dateizeiger und verschiedene Zugriffsmuster.

```
 call MPI_Open(comm_solve, filenm, MPI_WRONLY+MPI_CREATE,
$              MPI_INFO_NULL, fp, ierr)

 call MPI_File_set_view(fp, 0, MPI_DOUBLE_PRECISION,
$                       MPI_DOUBLE_PRECISION, MPI_INFO_NULL, ierr)

 call initialise

 ...

 do step = 1, niter
   call adi
   if (mod(step, wr_interval) .eq. 0) then
     call MPI_File_write_at(fp, iseek, u(1,0,jio,kio,cio), count,
$                           MPI_DOUBLE_PRECISION, mstatus, ierr)
   endif
 enddo
 call MPI_Close(fp, ierr)
```

Das Öffnen einer Datei geschieht durch die Funktion *MPI_Open()*. Diese Funktion muss kollektiv von allen Prozessen aufgerufen werden, die durch den Kommunikator *comm_slove* gruppiert sind. Der Aufruf öffnet die Datei und liefert für jeden Prozess einen eindeutigen Dateizeiger (Parameter *fp*) zurück. Jeder dieser Dateizeiger zeigt zu Beginn an den Anfang der Datei. Durch die Funktion *MPI_File_set_view()* wird das Zugriffsmuster auf die Datei festgelegt. Dieser Aufruf wird ebenfalls kollektiv von allen Prozessen ausgeführt. Das Schreiben in die Datei wird durch die Funktion *MPI_File_write_at()* initiiert. Diese Funktion ruft jeder Prozess unabhängig von den anderen auf. Der Schreibzugriff auf die Datei wird mit dem Aufruf der Funktion *MPI_Close()* beendet.
Um einen globalen Namenraum für die Dateien in einem Remote Netzwerk zu schaffen, nutzt RIO eine URL-ähnliche Syntax.

```
MPI_Open(..., "x-rio://host:port-num/path", ...)
```

Das Kürzel „x-rio“ gibt dabei an, dass die Anfrage an einen RIO Server geht, die Werte „host“ und „prot-num“ identifizieren diesen Server eindeutig und „path“ enthält den Pfad für die Datei auf diesem Server.

Im nächsten Abschnitt wird beschrieben, welche Ansätze RIO nutzt, um die Performanz für Dateizugriffe in einem Remote Netzwerk zu optimieren.

Der erste Ansatzpunkt ist die direkte Kommunikation zwischen Client- und Serverprozessen in einem solchen Netzwerk. Ein Nachteil der direkten Kommunikation ist, dass ein einzelner Prozess zwei Kommunikationskanäle aufrechterhalten muss, MPI und TCP/IP, um mit dem Server zu interagieren. Diese gleichzeitige Nutzung von zwei Kommunikationsschnittstellen bedeutet einen signifikanten Overhead. Damit sich die Prozesse nicht um das Versenden über das Netzwerk kümmern brauchen, setzt RIO so genannte Forwarder-Knoten ein. Diese Forwarder-Knoten sind dedizierte Knoten, die allein für die Kommunikation zwischen Client- und Serverprozessen zuständig sind. Dadurch werden Client und Server entlastet. Eine weitere Funktion der Forwarder-Knoten ist die Regelung des Verkehrsaufkommens, um eine Überlastung des Netzwerkes zu verhindern.

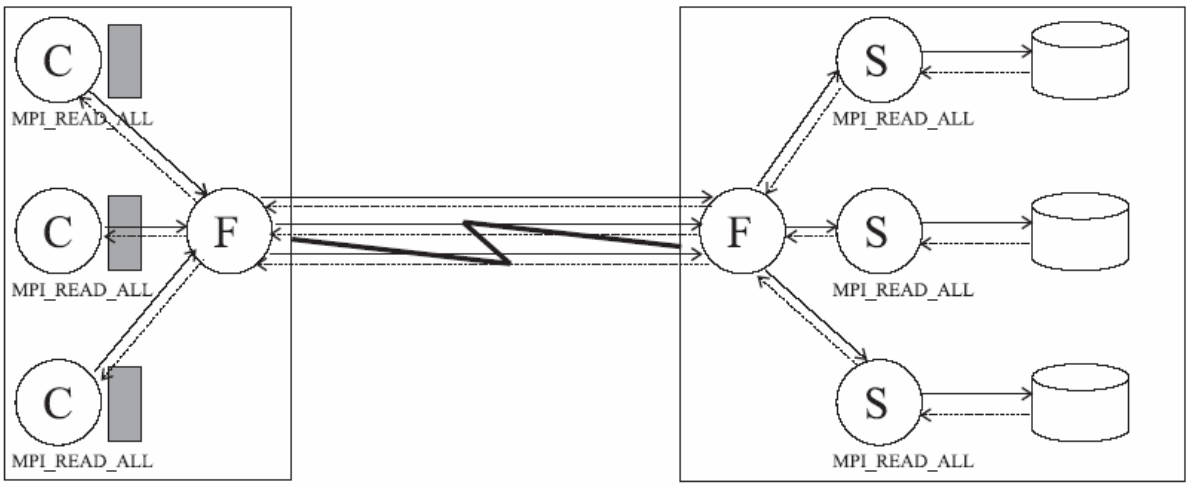

Abbildung 2: RIO's optimierte I/O Strategie - Kollektive Leseoperation mit Nutzung der Forwarder-Knoten [aus: Foster et al. 1997]

Ein weiterer Ansatzpunkt von RIO, die Performanz zu steigern, ist das Ausnutzen von kollektiven Operation und kollektiven Nachrichten. Hierbei spielen die Forwarder-Knoten ebenfalls eine Rolle. Bei der direkten Verbindung von Client- und Serverprozessen, kommuniziert jeder Clientprozess unabhängig mit dem Server, auch wenn die Clientprozesse an einer kollektiven Operation beteiligt sind. Das bedeutet, dass eine große Menge an Nachrichten

an den Server versendet und ebenfalls viele unabhängige I/O Operationen auf dem Server ausgeführt werden. Den dadurch entstehenden Overhead versucht RIO zu vermeiden, indem es mehrfache Nachrichten der Clients an einem Forwarder-Knoten bündelt und als eine einzige Nachricht an den Server sendet. Auf Server-Seite versucht RIO mit der gleichen Taktik mehrfache I/O Operationen zu vermeiden. Die Nachrichten der Clients werden mit speziellen Tags versehen, um zu kennzeichnen, ob sie zu einer kollektiven Operation gehören, und am Forwarder-Knoten des Servers gesammelt, bis sie in einer einzigen I/O Operation ausgeführt werden.
Die schon angesprochene Latenzzeit von 100 msec in einem Remote Netzwerk bedeutet einen limitierenden Faktor für die Performanz. Um hier eine möglichst das Optimum zu erreichen, setzt RIO auf asynchrone I/O Operationen. Mittels asynchroner Operationen wird die Überlappung von Berechnung und I/O in der Anwendung erreicht. Auch hier nutzt RIO die Forwarder-Knoten. Um eine asynchrone Operation zu initiieren, sendet der Clientprozess eine nicht blockierende Anfrage an den Forwarder-Knoten, so dass die Kontrolle sofort wieder zur Anwendung zurückkehrt. Der Forwarder-Knoten kümmert sich nun um den Rest, während der Client mit der Anwendung fortfahren kann. Die asynchrone Operation ist beendet, wenn der Forwarder-Knoten die Antwortnachricht an den Client sendet. Die Anwendung auf dem Client kann diese Antwortnachricht mittels der MPI-IO Aufrufe *test()* bzw. *wait()* abwarten.
Weitere Möglichkeiten zur Steigerung der Performanz, die bei der Entwicklung von RIO jedoch zunächst nicht umgesetzt wurden, sind clientseitige Pufferung, um die Anzahl der Nachrichten zu reduzieren, und die Nutzung des Speichers des parallelen Systems, auf dem die Anwendung läuft, als Cache, wenn die Anwendung auf bestimmte Daten mehrmals zugreifen muss.

Inwieweit diese Ansätze die Leistung von RIO verbessern können, wird nun im folgenden Kapitel erörtert.

4. Experimente

Die Autoren berichten über ihre Ergebnisse, die sie bei Experimenten entwickelt haben. Ihre Testumgebung haben sie so gewählt, dass sie die prinzipielle Performanz von RIO und eine vorläufige Abschätzung der Geschwindigkeit bei einer real existierenden Anwendung widerspiegeln. Der erste Teil wird von einem Benchmark abgedeckt, der ähnlich ist zu anderen existierenden, die benutzt werden, um die Geschwindigkeit von I/O-Libraries zu messen. Den zweiten Teil bildet eine einem anderen Benchmark entnommene Anwendung zur Berechnung von Flüssigkeitsströmungen.

4.1. Die Testplattform

Um einen Mittelweg zwischen der Messung von puren Remote-I/O-Werten und der Möglichkeit zur Messung des Einflusses verschiedener Parameter bzgl. der Geschwindigkeit, beschlossen die Autoren, ein IBM SP2 System in 2 Partitionen zu teilen. In jeweils einer davon läuft das Client- bzw. Serversystem. Die Kommunikation innerhalb jeder Partition verläuft über eine herstellereigene MPI-Implementierung während TCP/IP-Kommunikation über einen sog. High-Performance Switch zwischen beiden Partitionen zum Einsatz kommt. Die erreichten Geschwindigkeiten werden angegeben mit: 55 MB/s bei ca. 50 µsek. Latenz bzw. 22 MB/s bei ca. 320 µsek. Latenzzeit.

Die Experimente wurden auf dem SP2-System im Argonne National Laboratory durchgeführt. Die eingesetzte Version von PIOFS war 1.2 (PIOFS ist der Vorgänger von GPFS [Barrios et al. 1999]) unter dem Betriebssystem AIX 4.2. Es kamen vier PIOFS-Server zum Einsatz. Bei PIOFS besteht jede Datei aus einer Menge von sog. Zellen und jede Zelle wird auf einem Server-Knoten gespeichert. Eine Datei wird in Basic Striping Units (BSUs) aufgeteilt, die in einer Round-Robin Art auf die Zellen der Server-Knoten verteilt werden. Die Standardgröße der BSUs ist 32KB. Diese wurde bei den Experimenten benutzt, obwohl eine Veränderung der Größe Geschwindigkeitsveränderungen verursachen könnte, die anscheinend nicht betrachtet werden sollten. PIOFS ist z.B. empfindlich gegenüber Zugriffen auf Datenmengen < 8KB, größere Datenmengen am Stück erlauben eine höhere Schreib- bzw. Lesegeschwindigkeit.

4.2. Ergebnisse der speziell definierten Benchmarks

Die von den Autoren benutzten Benchmarks sollen die Lese- und Schreibgeschwindigkeit in Abhängigkeit der Anzahl der beteiligten Clients zeigen. Dabei führt jeder Benchmark wiederholt Lese- bzw. Schreiboperationen auf einer einzigen verteilten Datei durch, um zusammenhängende Datenstücke zu übertragen. Die Geschwindigkeit wird ermittelt durch das Teilen der Datenmenge durch die verstrichene Zeit (beides auf Seiten des Clients). Es wurden sowohl blockierende als auch nicht-blockierende Operationen verwendet, wobei bei letzteren die Zeit bis zum Beenden der letzten I/O-Operation gemessen wurde. Jede Messung repräsentiert den Mittelwert einer großen Anzahl an wiederholten Durchläufen der Datenübertragung (1000 und mehr). Die Anzahl der Wiederholungen wurde so bestimmt, dass ein Test mindestens eine Minute dauerte.

Bei Messungen mit viel Übertragungsvolumen gibt es Schwankungen, die auf konkurrierende Prozesse auf den PIOFS Server-Knoten zurückgeführt werden. Bei den Experimenten war kein alleiniger Zugriff auf die Ressourcen möglich, so dass andere Programme die Messungen stören

konnten. Anscheinend konnten die Autoren daran nichts ändern, denn eine Beseitigung dieses Missstandes wird nicht erwähnt.

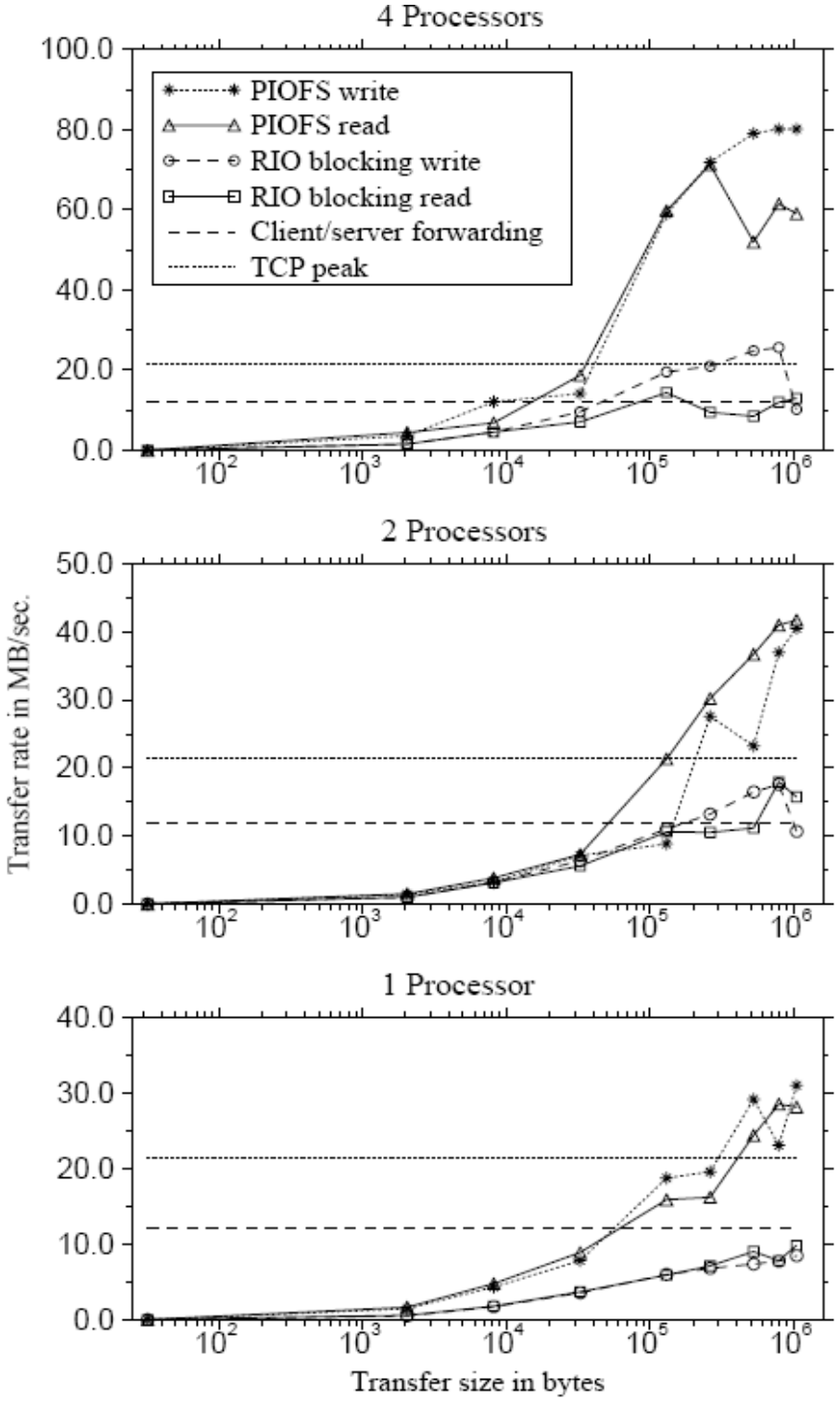

Abbildung 3: Transferrate bei 1, 2 und 4 Prozessen [aus: Foster et al. 1997]

In Abbildung 3 ist die Transferrate bei unterschiedlich vielen Prozessen dargestellt. Die gestrichelte Linie repräsentiert die Übertragungsrate, die bei der Kommunikation zwischen einem Client und einem Serverprozess gemessen wurde (ca. 12,0 MB/s). Dabei wurde eine große Nachricht – es wird nicht gesagt wie groß – zwischen den beiden Instanzen über einen Forwarder-Knoten hin und her geschickt. Dies soll die maximal mögliche Übertragungsrate zwischen einzelnen Client- und Serverprozessen bei synchronen Operationen im Versuchsnetzwerk darstellen (bei Kommunikation über einen Forwarder). Die gepunktete Linie (ca. 21,4 MB/s) stellt die Übertragungsgeschwindigkeit dar, die erreicht wird, wenn dieselbe Nachricht zwischen Client- und Serverprozess ausgetauscht wird, jedoch ohne einen Forwarder-Knoten dazwischen. Die unterschiedlichen Raten erklären sich dadurch, dass in erstem Fall insg. mehr Nachrichten ausgetauscht werden als im zweiten (Grund ist der Forwarder).
In Abbildung 3 ist erkennbar, dass die Übertragungsrate von PIOFS mit der Anzahl der beteiligten Prozesse ansteigt; bis auf ca. 80 MB/s. Hier sei angemerkt, dass die PIOFS-Server innerhalb der Partition stehen und die Daten somit nicht über den Flaschenhals (Verbindung zwischen den beiden Partitionen) übertragen werden. Die Transferrate von RIO mit jeweils einem Client- und Server Prozess bleibt immer unterhalb von 12 MB/s; zu erwarten, da dies ein ähnlicher Test-Aufbau ist wie beim Programm, das eine große Nachricht zwischen je einem Client- und Serverprozess über die Forwarder austauscht. Für eine höhere Anzahl beteiligter Prozesse (Abbildung 3 bei 4 CPUs zu sehen) kann RIO die 12 MB/s übertreffen, da dann Pipelining der Kommunikation genutzt werden kann. Trotzdem ist die Bandbreite durch die Verbindung zwischen den Forwardern begrenzt, so dass das Maximum mit 4 Clients bei ca. 26 MB/s erreicht wird.
Beim zweiten Experiment wurde verglichen, wie sich blockierende und nicht-blockierende Operationen mit RIO verhalten. Dabei wurde die Anzahl der Server-Knoten auf 2 festgesetzt. Wie zu erwarten, ist der Durchsatz bei nicht-blockierenden Operationen meist höher (die oben erwähnten 26 MB/s für Lesen bzw. 22 MB/s für Schreiben) als bei blockierenden. Die Autoren haben festgestellt, dass der höchstmögliche Durchsatz für nicht-blockierende Operationen schon mit 2 Servern erreicht und durch die Verbindung zwischen den Forwardern begrenzt wird. Sie ziehen daraus den Schluss, dass die optimale Anzahl der Server-Knoten in einem gegebenen Netzwerk (zwar abhängig von den technischen Gegebenheiten des Netzwerks und den I/O-Operationen der Anwendung, aber) klein sein wird.
Die Ergebnisse zeigen, dass RIO die Bandbreite der Verbindung zwischen den Partitionen für große Nachrichten ausreizen kann und dass damit der limitierende Faktor in ihrem Versuchsaufbau die Netzwerkgeschwindigkeit ist. Sie sehen den größten Vorteil von RIO in den

asynchronen Operationen und im damit entstehenden Pipeline-Effekt bei den I/O-Anfragen auf dem Weg zwischen Clients und Servern. Dies wird im folgenden Abschnitt umso deutlicher, nämlich in dem Fall, dass die Anwendung die Überlappung von Kommunikation und Berechnungen zulässt.

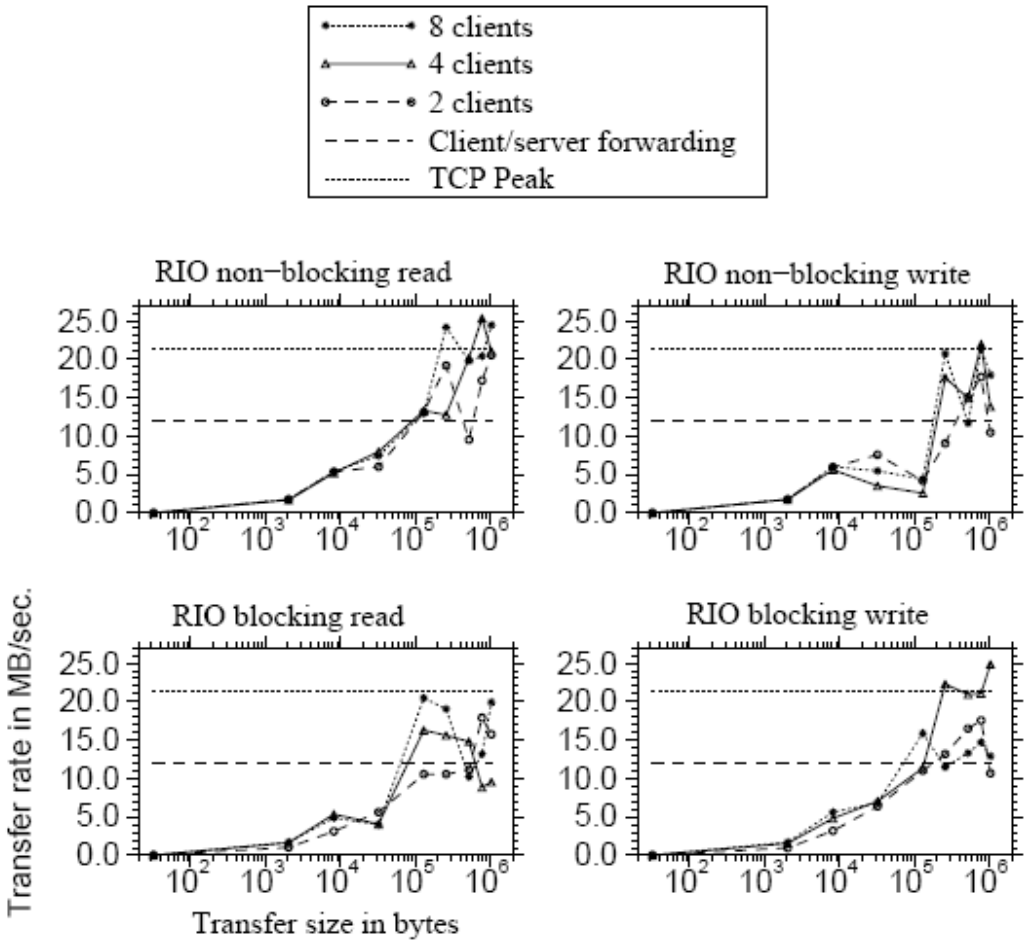

Abbildung 4: Transferrate bei 2, 4 und 8 Client Prozessen und einer festen Zahl (=2) von Server-Prozessen [aus: Foster et al. 1997]

4.3. Ergebnisse bei einer ausgesuchten Anwendung

Die von den Autoren verwendete Anwendung (`BTIO-simple-mpiio` aus dem NAS I/O Benchmark) zur Simulation des I/O-Verhaltens eines Programms zur Berechnung von Strömungsverhalten muss alle k Iterationen einen dreidimensionalen Lösungsvektor der Größe N^3 auf Festplatte schreiben. Insgesamt werden I Iterationen vollzogen. Die Anwendung ist in Fortran geschrieben, benutzt die MPI-IO Schnittstelle und liest keinerlei Daten sondern schreibt nur in eine einzige Datei. Die Problemgrößen bei den Tests waren N=32, N=64 und N=80, so dass sich die zu schreibenden Daten zu 52 MB, 420 MB und 1 GB ergaben. Die Laufzeiten

wurden so gemessen, dass die komplette Zeit der Berechnung und I/O Aktivität gemessen wurde. Dadurch ergeben sich selbstverständlich andere Durchsatzzahlen als in den obigen Benchmarks, da dort keine signifikanten Berechnungen durchgeführt werden mussten.

Die Schreiboperationen von `BTIO-simple-mpiio` haben dabei kein reguläres Muster und es sind nur kleine Datenmengen, die geschrieben werden. Dies zieht die Geschwindigkeit bei PIOFS durch den großen PIOFS-Overhead bei vergleichsweise kleinen Datenmengen runter. Die Autoren haben daher eine modifizierte Version von `BTIO-simple-mpiio` geschrieben, bei der vor der kollektiven Schreiboperation der Lösungsvektor jeder Client die Daten der anderen sammelt (dann hält jeder Client den gesamten Lösungsvektor in einem temporären Datenfeld vorrätig) und danach einen einzigen zusammenhängenden Block Daten schreibt. Die Autoren kritisieren dabei, dass die MPI-IO Implementierung diese Optimierung beim kollektiven Schreiben automatisch durchführen sollte, dies aber von der bei den Tests eingesetzten Version nicht geleistet wurde.

Bei den Tests mit beiden Versionen wurden vier Clients für die Berechnung verwendet. Bei Tests mit der originalen Version wurden auch vier Server-Knoten verwendet, während bei der optimierten Version nur einer verwendet wurde – da in diesem Fall die Größe der zu schreibenden Blöcke das Netzwerk zum limitierenden Faktor machten. Die Autoren beobachteten für andere als die gerade genannten Anzahlen für Client- und Server-Knoten ähnliche Resultate und gaben daher nur die nun folgenden Ergebnisse an.

Es wurden die Gesamtzeiten (in Sekunden) für vier verschiedene Konfigurationen gemessen (s. Tabelle 1):

1. Zugriff über PIOFS lokal (ohne Einbeziehung von RIO), wonach die Daten noch nicht an ihrem Zielort (der 2. Partition) liegen.
2. Wie 1., jedoch werden die Ergebnisdaten per FTP an ihren Zielort verschoben.
3. Zugriff durch RIO (blockierende Operationen), um die Ergebnisdaten an ihren Zielort (RIO-Server in der 2. Partition) zu senden. Dabei führt der RIO-Server dann lokal PIOFS-Schreiboperationen aus.
4. wie 4., jedoch mit nicht-blockierenden Operationen. Damit wird der Vorteil der Überlappung von Berechnung und Kommunikation ausgenutzt.

Version	N	I	k	Compute	Local PIOFS	PIOFS+ftp	RIO (blocking)	RIO (nonblock)	Improvement
Original	32	200	5	86.84	206.12	211.97	199.98	196.06	8.1%
Original	64	200	5	624.52	837.60	883.26	987.18	941.98	-6.2%
Original	80	50	1	304.04	857.75	968.86	1080.66	959.21	1.0%
Optimized	32	200	5	86.84	100.06	105.91	107.52	89.42	18.4%
Optimized	64	200	5	624.52	668.84	722.30	676.78	648.40	11.4%
Optimized	80	50	1	304.04	424.71	535.82	487.98	442.65	21.0%

Tabelle 1: Gemessene Ausführungszeiten für die originale und optimierte BTIO Version in Sekunden [aus: Foster et al. 1997]

Weiterhin sei anzumerken, dass die Spalte „Compute“ die alleinige Berechnungszeit in Sekunden angibt – ohne die Zeit für anfallende Kommunikation. Die Zeile „Improvement“ gibt die max. Verbesserung bei Benutzung von RIO gegenüber der Benutzung von PIOFS + FTP. Bei nicht-blockierenden RIO-Operationen in der Original-Version können bis zu 64 I/O Operationen ausstehen; bei mehr als 64 muss auf die Beendigung mindestens einer dieser gewartet werden, was das schlechte Abschneiden in der Originalversion erklären dürfte. Bei nicht-blockierenden RIO-Operationen in der optimierten Version kann sofort zur Berechnung zurückgekehrt und die großen Blöcke können asynchron geschrieben werden, da dies aus dem temporären Datenfeld geschieht. Ein Warten auf die Beendigung der Operation tritt nur vor dem Start der nächsten Iteration auf.

In Tabelle 1 und Tabelle 2 sind die gemessenen Zeiten bzw. daraus resultierende Transferraten angegeben. Generell liefert die optimierte Version wegen der reduzierten Schreiboperationen eine höhere Geschwindigkeit als die originale Version von `BTIO-simple-mpiio`. Nicht-blockierende RIO-Operationen steigern in allen Fällen die Geschwindigkeit im Vergleich zu blockierenden Operationen, denn in letzterem Fall ist der Austausch der bzw. das Warten auf Synchronisierungsnachrichten zw. Clients und Server die Quelle großen Overheads. Bei nicht-blockierenden Operationen reichen die Werte fast an die von PIOFS heran, da Berechnung und Warten auf Synchronisierungsnachrichten überlappen. Da PIOFS keine nicht-blockierenden Operationen unterstützt, ist RIO (nicht-blockierend) in zwei der drei Fälle bei der originalen Version schneller als PIOFS+FTP.

Version	N	I	k	Local PIOFS	PIOFS+ftp	RIO (blocking)	RIO (nonblock)
Original	32	200	5	0.2543	0.2473	0.2622	0.2674
Original	64	200	5	0.5008	0.4749	0.4249	0.4453
Original	80	50	1	1.1938	1.0569	0.9476	1.0676
Optimized	32	200	5	0.5240	0.4950	0.4876	0.5863
Optimized	64	200	5	0.6271	0.5807	0.6197	0.6469
Original	80	50	1	2.4111	1.9110	2.0984	2.3133

Tabelle 2: Durchschnittliche I/O Transferrate für originale und optimierte BTIO Version in MB/s [aus: Foster et al. 1997]

Die Gesamtausführungszeit bei der Benutzung von RIO ist fast immer kleiner als die bei PIOFS+FTP. Im besten Fall (optimierte Version, N=80, RIO nicht-blockierende Operationen) ist RIO 21% schneller fertig. Dies zeigt, dass RIO ein Ansatz sein kann, die Ausführungszeiten im Vergleich zu traditionellen Techniken für entfernten Datenzugriff zu verbessern und eine geeignete Schnittstelle anzubieten.

5. Fazit und Ausblick

Die Autoren haben gezeigt, dass mit ihrem Ansatz und dem Prototyp für RIO (einer Bibliothek für remote I/O) bei geeigneten parallelen Applikationen ein Geschwindigkeitszuwachs durch die Überlappung von Kommunikation und Berechnung möglich ist. Dazu müssen noch Dinge geklärt werden wie: Sicherheit, Verbesserung der Netzwerkleistung und die Komplexität der Konfiguration. Ihre Experimente zeigen, dass RIO wenig Overhead aufweist und die Gesamtausführungszeit beschleunigen kann. Sie räumen ein, dass dies erst der erste Schritt war und Tests folgen sollten, bei denen weiter entfernte Client- und Server-Knoten miteinbezogen werden sollten, um den Einfluss der höheren Latenz sowie evtl. geringeren Bandbreite zu untersuchen.
Außerdem wollten die Autoren den Einfluss der Quality of Service (QoS) auf remote I/O erforschen, da sie dies bei ihren Experimenten nicht mitberücksichtigt hatten. Dabei wollten sie herausfinden, ob die existierenden Maßnahmen zur Sicherung der QoS ausreichend für remote I/O sind bzw. ob es möglich ist, angepasste Techniken in RIO oder der Anwendung zu integrieren, um die Ausführungsgeschwindigkeit robuster angesichts der Variabilität der Netzwerkleistung zu machen.

6. Literatur

- Barrios, M.; Jones, T.; Kinnane, S.; Landzettel, M.; Al-Safran, S.; Stevens, J.; Stone, C.; Thomas, C.; Troppens, U.: Sizing and Tuning GPFS, http://www.redbooks.ibm.com/redbooks/pdfs/sg245610.pdf, Poughkeepsie (NY) 1999, online: 10.06.2005.
- Foster, I.; Kohr jr., D.; Krishnaiyer, R.; Mogill, J.: Remote I/O: Fast Access to Distant Storage. Workshop on I/O in Parallel and Distributed Systems, S. 14-25, San Jose (CA) 1997.
- Maui High Performance Computing Center: IBM SP Hardware/Software Overview. http://www.mhpcc.edu/training/workshop/ibmhwsw/MAIN.html, Maui (HI) 2003, online: 15.06.2005.
- University of Chicago: RIO: Remote I/O for Metasystems, http://www-fp.globus.org/details/rio.html, Chicago (Il) 2000, online: 24.06.2005.

7. Tabellenverzeichnis

Tabelle 1: Gemessene Ausführungszeiten für die originale und optimierte BTIO Version in Sekunden [aus: Foster et al. 1997] 15

Tabelle 2: Durchschnittliche I/O Transferrate für originale und optimierte BTIO Version in MB/s [aus: Foster et al. 1997] 15

8. Abbildungsverzeichnis

Abbildung 1: Architektur von RIO (links:Client - rechts:Server) [aus: Foster et al. 1997] 6

Abbildung 2: RIO's optimierte I/O Strategie - Kollektive Leseoperation mit Nutzung der Forwarder-Knoten [aus: Foster et al. 1997] 8

Abbildung 3: Transferrate bei 1, 2 und 4 Prozessen [aus: Foster et al. 1997] 11

Abbildung 4: Transferrate bei 2, 4 und 8 Client Prozessen und einer festen Zahl (=2) von Server-Prozessen [aus: Foster et al. 1997] 13