Fachhochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Seminararbeit

Parallele Systeme

ROMIO

von
Jan Seidel
Patrice Kwemo (nicht bearbeitet)

Betreuer: Prof. Berrendorf

Gliederung

1. Einleitung

Diese Seminararbeit stellt das am Argonne National Laboratory entwickelte ROMIO vor. ROMIO ist eine Implementierung des vom MPI-Forum entwickelten MPI-IO-Standards zum parallelen I/O. ROMIO soll den MPI-IO-Standard komplett und effizient implementieren. Dazu nutzt es das so genannte ADIO, das „Abstract Device Interface for Parallel I/O“, welches ebenfalls in Argonne entwickelt wurde. Grundlage für die Entwicklung von ADIO war der, dass verschiedene zueinander inkompatible APIs (Application Programmer Interaces) zum parallelen I/O existierten und eine Applikation speziell für eine API geschrieben werden musste und damit nur für ein Dateisystem zur Verfügung stand. ADIO vereinfach diesen Vorgang so, dass eine an ADIO angebundene API auf jedes für ADIO implementierte Dateisystem zugreifen kann. Das Design von ROMIO und ADIO wird in den folgenden Kapiteln beschrieben.

Im ersten Teil dieser Arbeit soll gezeigt werden, wie der MPI-IO-Standard in ROMIO umgesetzt wurde und auf welchen Designkriterien ROMIO basiert. Dazu sollen die MPI-IO-Funktionen und ihre Umsetzung in ROMIO kurz erläutert werden. Der zweite Teil dieser Arbeit beschäftigt sich mit ADIO. ADIO ist eine Schnittstelle zwischen einer API zum parallelen I/O und einem Dateisystem. Der Hauptgesichtspunkt hier ist die Verwendung von ADIO zur Anbindung von ROMIO und damit MPI-IO an beliebige Dateisysteme. Auch hier sollen die Designkriterien erläutert und anhand von ADIO-Funktionen gezeigt werden. In diesem Kapitel zeigt ein Unterabschnitt beispielhaft an einem neuen Dateisystem, wie dieses in ADIO integriert werden kann, um danach MPI-Anwendungen zur Verfügung zu stehen. Ein weiteres Kapitel beschreibt den durch ADIO entstehenden Overhead gegenüber dem direkten Einsatz eines Dateisystems. Anhand von zwei Testprogrammen und einer reellen Applikation werden die Laufzeiten gemessen und verglichen.

2. ROMIO

(Patrice Kwemo)

3. ADIO

Nachdem im vorangegangen Kapitel der Zusammenhang zwischen dem MPI-IO-Standard sowie ROMIO verdeutlicht wurde, soll dieses Kapitel das Abstract Device Interface for parallel I/O, kurz ADIO, vorstellen.

3.1. Zusammenhang ROMIO und ADIO

ADIO selbst ist ein wichtiger Bestandteil von ROMIO. Es ist die Schnittstelle zwischen dem Application Programmers Interface (API) und den verschiedenen Dateisystemen, in ADIO Devices genannt. Die API ist in diesem Fall ROMIO, welches wie bereits beschrieben eine Implementierung des vom MPI-Forum definierten MPI-IO-Standards ist.

3.2. Design von ADIO

Das Design von ADIO wird von Thakur et al. in dem Paper „An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces“ [Thakur et al. 1996] aus dem Jahr 1996 beschrieben. Hauptaugenmerk bei der Entwicklung von ADIO wurde auf die beiden Punkte Portabilität und Effizienz gerichtet. ADIO soll als Schnittstelle zwischen jeder API für parallelen I/O und jedem Dateisystem dienen. Ein hohes Maß an Portabilität wird dadurch erreicht, dass ein Dateisystem nur einmal an die ADIO-Schnittstelle angebunden werden muss und von da an jeder API zum parallelen I/O zur Verfügung steht, die ebenfalls an ADIO angebunden ist. ADIO ist damit ein Ansatz zur Lösung des immer noch aktuellen Problems, dass viele untereinander inkompatible APIs zum parallelen I/O existieren. Die Dateisysteme PIOFS und PFS verfügen beispielsweise jeweils über eine eigene API und sind untereinander inkompatibel. Da ADIO sowohl für PIOFS als auch für PFS implementiert ist, kann aus der PIOFS-API auf das PFS-Dateisystem zugegriffen werden und anders herum. ADIO ist also ein Vermittler zwischen Anfragen von APIs an Dateisysteme. ADIO ist selber keine API zum parallelen I/O sondern ein Ansatz andere APIs zu implementieren. ROMIO ist eine dieser APIs und setzt direkt auf ADIO auf. Das Grundkonzept von ADIO wird in Abbildung 1 dargestellt:

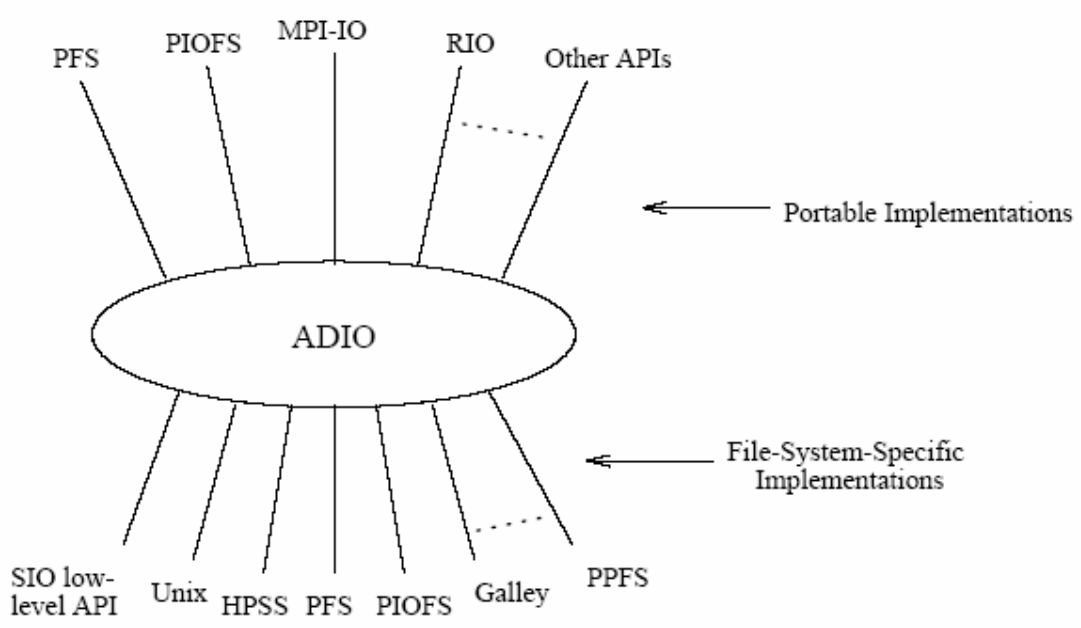

Abbildung 1: Grundkonzept ADIO, Quelle: [Thakur et al. 1996]

Da ROMIO eine Implementierung des MPI-IO-Standards ist und das Thema dieser Seminararbeit „ROMIO“ lautet, soll im Folgenden ausschließlich auf den Einsatz von ADIO für MPI-IO eingegangen werden.

Das Design von ADIO erlaubt eine einfache Portierung auf verschiedene Dateisysteme und erreicht trotzdem durch die Nähe zu dem jeweiligen Dateisystem eine hohe Performance. So erlauben einige Dateisysteme bspw. explizit nichtkontinuierliche Zugriffe auf Dateien, die durch ADIO direkt aufgerufen werden können. Um diese hohe Performance zu erreichen, muss beim Implementieren eines neuen Devices auf Eigenheiten des Dateisystems geachtet werden. Grundlegendes Wissen über das Dateisystem ist also erforderlich.

ADIO nutzt aus Portabilitäts- und Performance-Gründen überall wo möglich das Message Passing Interface MPI. So verwenden ADIO-Funktionen MPI-Datentypen und –Kommunikatoren. Diese enge Verbindung zwischen MPI und ADIO wird auch daran deutlich, dass einige Features von ADIO speziell zur Implementierung von MPI-IO hinzugefügt wurden. So unterstützt ADIO beispielsweise die von MPI-IO benötigten „etypes“ und „filetypes“, mit denen Sichten auf Dateien definiert werden. Die wichtigsten ADIO-Funktionen sollen im Folgenden beschrieben werden:

3.3. ADIO-Funktionen

Die ADIO-Funktionen zum parallelen I/O werden aus ROMIO aufgerufen. Der grobe Ablauf sieht dabei wie folgt aus: Ein Applikation ruft eine MPI-IO-Funktion auf, welche in ROMIO implementiert ist. ROMIO leitet diese Anfrage an ADIO weiter, welches die devicespezifische ADIO-Funktion aufruft. Diese wiederum greift direkt auf das jeweilige Dateisystem zu, auf dem die low-level I/O-Funktionen ausgeführt werden. Dieser Ablauf ist in Abbildung 2 für das Öffnen einer Datei mit MPI_File_Open(…) dargestellt. Der Ablauf bei weiteren I/O-Funktionen wie Lesen / Schreiben ist prinzipiell ähnlich, bis darauf, dass hier der Initialisierungsaufwand einer ADIO-Datei wegfällt.

Das zu verwendende Dateisystem wird beim Öffnen einer Datei angegeben, indem dem Dateinamen als Prefix der Name des Dateisystems sowie ein Doppelpunkt vorangestellt wird (Beispiel: filename = „ufs:work.txt“ für das UFS Dateisystem).

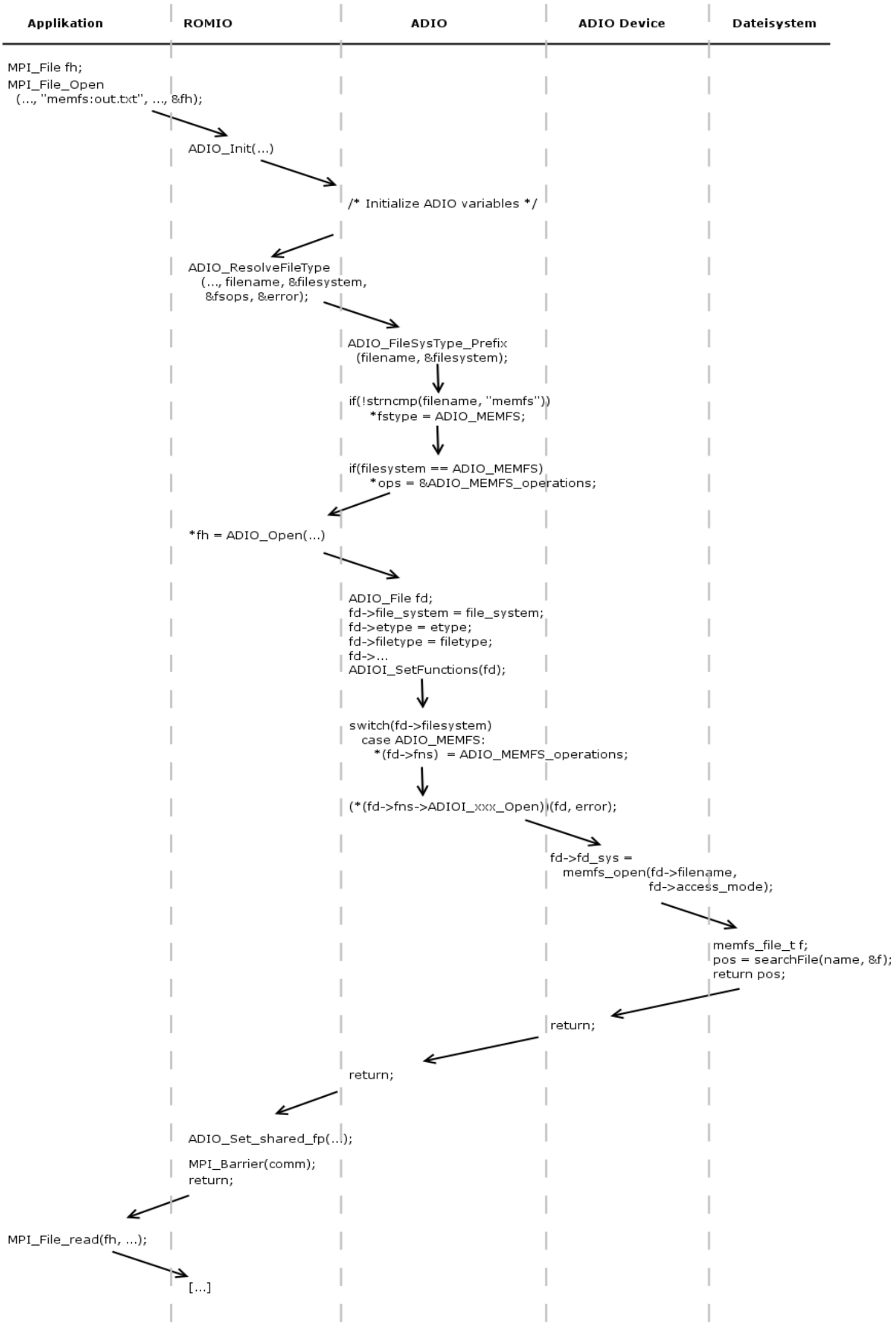

Abbildung 2: Aufrufreihenfolge für MPI_File_Open()

Im Folgenden soll nun kurz auf die jeweiligen ADIO-Funktionen aus Abbildung 2 eingegangen werden:

```
void ADIO_ResolveFileType(MPI_Comm comm., char *filename, int *fstype,
    ADIOI_Fns **ops, int *error_code)
```

ADIO_ResolveFileType setzt den Zeiger **ops auf eine struct vom Typ ADIOI_Fns in der die devicespezifischen ADIO-Funktionen gespeichert sind. Diese Funktionen sind in Tabelle 1 (siehe Anhang) anhand des Devices „MEMFS“ dargestellt. Bevor der Zeiger **ops auf die richtige struct gesetzt werden kann, muss erst mit der Funktion ADIO_FileSysType_Prefix(…) das zu verwendende Dateisystem identifiziert werden.

```
static void ADIO_FileSysType_Prefix(char *filename, int *fstype, int
*error_code)
```

Diese Funktion ermittelt anhand des Dateinamens das zu verwendende Dateisystem. Dazu wird der Dateiname bis zum ersten Doppelpunkt geparsed. Ein eindeutiger Identifikator für das Dateisystem wird in der Variablen *fstype zurückgegeben. Das ermittelte Dateisystem wird an die Funktion ADIO_Open(…) übergeben.

```
ADIO_File ADIO_Open(MPI_Comm orig_comm, MPI_Comm comm, char *filename,
    int file_system, int access_mode, ADIO_Offset disp, MPI_Datatype
    etype, MPI_Datatype filetype, int iomode, MPI_Info info, int perm,
    int *error_code)
```

Nachdem durch ADIO_ResolveFileType(…) das zu verwendende Dateisystem identifiziert wurde, kann nun das eigentliche Öffnen der Datei auf diesem Dateisystem erfolgen. Dazu wird zuerst ein Dateideskriptor vom Typen ADIO_File erstellt und mit Anfangswerten initialisiert. Die oben angeführten Übergabeparameter an ADIO_Open(…) werden in diesem Dateideskriptor abgelegt. Der Aufbau eines solchen Dateideskriptors ist in der Tabelle 2 (siehe Anhang) dargestellt. Als nächstes ruft ADIO_Open(…) die Funktion ADIOI_SetFunctions(…) auf (siehe unten). Daraufhin folgen einige weitere ADIO-Funktionen zur Verwaltung des Kommunikators, zum Anlegen eines Info-Objekts, über das bspw. Hints an ein Dateisystem übergeben werden können, etc. Diese sollen hier nicht alle im einzeln besprochen werden, da dies den Rahmen dieser Ausarbeitung sprengen würde. Nachdem diese Verwaltungsfunktionen ausgeführt wurden, wird die devicespezifische Funktion zum Öffnen einer Datei aufgerufen. Diese erhält den soeben erstellten Dateideskriptor als Übergabeparameter. Die devicespezifische Open-Funktion öffnet nun physikalisch eine Datei auf dem ausgewählten Dateisystem. Der Dateideskriptors des Dateisystems wird in der Variablen „fd_sys“ des ADIO-

Dateideskriptors gespeichert und steht damit für weitere I/O-Funktionen zur Verfügung. Die Entwicklung eines eigenen ADIO-Devices wird im folgenden Kapitel erläutert.

`ADIOI_SetFunctions(ADIO_File fd)`

Die Funktion ADIOI_SetFunctions(…) setzt den Zeiger "fns" des Dateideskriptors „fd" auf die struct der devicespezifischen ADIO-Funktionen. Die Funktion ist also in großen Teilen identisch zu ADIO_ResolveFileType(…) nur muss hier das Dateisystem nicht erst erkannt werden, sondern ist bereits in dem Dateideskriptor gespeichert.

3.4. Schritte zur Entwicklung eines eigenen ADIO-Devices

Die Entwicklung eines neuen ADIO-Devices ist dann erforderlich, wenn in einer API zum parallelen I/O ein Dateisystem eingesetzt werden soll, welches noch nicht an ADIO angebunden wurde. Dieses Kapitel beschreibt kurz die Entwicklung eines neuen Dateisystem und seine Anbindung an ADIO am Beispiel des an der Fachhochschule Bonn-Rhein-Sieg im VIOLA-Projekt (Vertically Integrated Optical Testbed for Large Application) entwickelten MEMFS-Devices. Das Hauptaugenmerk dieses Kapitels liegt dabei auf der Anbindung des neuen Dateisystems an ADIO, nicht auf dem Dateisystem selbst. Im Folgenden werden die vier dafür nötigen Schritte vorgestellt:

3.4.1. Entwicklung des Dateisystems

Das MEMFS-Dateisystem wurde im VIOLA-Projekt entwickelt, um aus MPI-IO-Funktionen auf entfernten Hauptspeicher zugreifen zu können. MEMFS steht dabei für „Memory Filesystem". In heutigen parallelen Anwendungen ist häufig der Zugriff auf Festspeicher wie Fileserver zur Speicherung von Zwischenresultaten ein performancekritischer Flaschenhals. Andererseits reicht der Speicher heutiger Parallelrechner in der Regel nicht aus, um das Zwischenspeichern von Resultaten zu umgehen. Der Ansatz von MEMFS ist nun, Zwischenergebnisse über das hochperformante VIOLA-Netz im Hauptspeicher anderer Parallelrechner abzulegen. Dadurch kann eine deutlich höhere Performance als beim Zugriff auf Fileserver erreicht werden. Der Zugriff auf andere Parallelrechner wird über ein so genanntes Tunnel Device ermöglicht, welches ADIO-Anfragen an entfernte Rechner weiterleitet. Das MEMFS-Device verfügt über die grundlegenden Operationen eines Dateisystems, wie Öffnen / Lesen / Schreiben / Schließen und Löschen von Dateien. Das genaue Design des MEMFS-Devices wird im August 2005 in einem VIOLA-Bericht veröffentlicht und soll nicht Thema dieser Seminarausarbeitung sein. Der Zugriff

auf die genannten Operationen durch MPI-IO geschieht über ein ADIO-Device, welches wie folgt implementiert wurde:

3.4.2. Entwicklung der ADIO-Device-Funktionen

Um ein neues Dateisystem – hier MEMFS – an ADIO anzupassen, muss ein neues ADIO-Device erstellt werden. Dazu müssen die in Tabelle 1 genannten Funktionen (siehe Anhang) implementiert werden. Diese Funktionen sind die Schnittstelle zwischen den APIs zum parallelen I/O und den Low-Level Hardware-I/O-Funktionen. Die wichtigsten dieser Funktionen sollen im Folgenden vorgestellt werden:

```
ADIO_MEMFS_Open(ADIO_File fd, int *error_code)
ADIO_MEMFS_Close(ADIO_File fd, int *error_code)
```

Die devicespezifischen Funktionen zum Öffnen und Schließen einer Datei sind sehr simpel. Aus dem übergebenen Dateideskriptor nimmt Open den Dateinamen heraus und versucht auf dem jeweiligen Dateisystem, hier MEMFS, eine Datei mit diesem Namen zu öffnen. Dazu wird die Dateisystemfunktion memfs_open(fd->filename, fd->access_mode) aufgerufen. Existiert die Datei nicht und als Access-Mode wurde nicht MPI_MODE_CREATE übergeben, so ist dieser Aufruf fehlerhaft. Anderenfalls wird in fd->fd_sys der Dateisystem-Filedeskriptor für spätere Zugriffe gespeichert. Close funktioniert hier ähnlich, es wird versucht, eine Datei im Dateisystem zu schließen. Dieser Aufruf ist fehlerhaft, falls die Datei nicht existiert oder nicht geöffnet ist. Wenn eine Datei korrekt geöffnet bzw. geschlossen wurde, wird „error_code“ auf „MPI_SUCCESS“ gesetzt, ansonsten auf einen Fehlercode.

```
ADIO_MEMFS_ReadContig(ADIO_File fd, void *buf, int count, MPI_Datatype
    datatype, int file_ptr_type, ADIO_Offset offset, ADIO_Status *status,
    int *error_code)
ADIO_MEMFS_WriteContig(…)
```

ReadContig liest aus der durch den Dateideskriptor „fd“ spezifizierten Datei eine durch „count“ angegebene Anzahl von Datenelementen vom Datentyp „datatype“ in den Lesebuffer „buf“. Durch die Variable „file_ptr_type“ kann unterschieden werden zwischen explizit angegebenem Offset durch MPI_File_read_at(…) oder durch Zugriff an der Stelle des individuellen Dateizeigers. Soll der gemeinsame Dateizeiger verwendet werden, findet die Umsetzung zwischen aktuellem Stand des gemeinsamen Dateizeigers auf expliziten Offset bereits in ADIO-Schicht vorher statt. Falls der individuelle Dateizeiger verwendet wurde, wird dieser nach der Leseoperation um die Anzahl gelesener Bytes

erhöht. Ist der Aufruf erfolgreich, so wird auch hier „error_code“ auf „MPI_SUCCESS“ gesetzt.

WriteContig funktioniert identisch, nur werden hier aus dem Schreibbufer „buf“ Daten in die durch den Dateideskriptor „fd“ spezifierte Datei geschrieben.

Bei allen ADIO-Devicefunktionen zu beachten ist, dass der eigentliche Dateizugriff durch Aufrufen von Dateisystemspezifischen Funktionen stattfindet, bei MEMFS bspw. durch memfs_read(fd->fd_sys, offset, buf, len) und memfs_write(fd->fd_sys, offset, buf, len). Das ADIO Device ist also nur die Schnittstelle zwischen ROMIO und dem Dateisystem.

`ADIO_MEMFS_Delete(char *filename, int *error_code)`

Das Löschen einer Datei funktioniert analog zum Öffnen einer Datei, indem der zu löschende Dateiname an das Dateisystem übergeben wird. Die Funktion ist fehlerhaft, wenn die Datei nicht existiert oder noch geöffnet ist. Ist dies nicht der Fall wird der Eintrag aus der Dateitabelle entfernt und der für die Datei reservierte Speicher freigegeben.

Weitere Funktionen:

Das Dateisystem muss nicht explizit jede der genannten Funktionen unterstützen. Ein Dateisystem, das beispielsweise nicht nativ kollektive Operationen unterstützt kann diese über Optimierungen wie „two-phase I/O“ simulieren. Beim two-phase I/O wird aus der Menge der bei der kollektiven Operation teilnehmenden Prozesse eine Untermenge gebildet, die so genannten I/O-Prozesse. Diese I/O-Prozesse sammeln die Anfragen aller Prozesse und fragen so große zusammenhängende Dateibereiche durch einen I/O-Aufruf ab. Dadurch lässt sich eine deutlich höhere Performance als durch mehrere einzelne Anfragen erreichen. Das Ergebnis wird daraufhin wieder entsprechend dem Zugriffsmuster auf die Prozesse verteilt. Bei der Entwicklung eines eigenen Devices kann weiterhin teilweise auf so genannte Generic-Funktionen zurückgegriffen werden, die ADIO zur Verfügung stellt. Diese Funktionen sind beispielsweise für Operationen verfügbar, für die nicht auf das eigentliche Dateisystem zugegriffen werden muss, wie Abfragen des Filepointers. Auch spezielle Funktionen wie Lesen / Schreiben nicht zusammenhängender Dateibereiche werden durch generische Funktionen ermöglicht. Hier existieren zwei verschiedene Ansätze: Zum einen kann ein nichtkontinuierlicher Zugriff durch mehrere kontinuierliche Zugriffe erreicht werden. Dadurch steigt die Anzahl der benötigten I/O-Zugriffe deutlich an. Bei plattenbasierten Dateisystemen führt dies zu drastischen Performanceverlusten. Dieses Problem ist bei MEMFS, das anstatt auf eine Platte für jeden I/O-Zugriff nur einen Speicherzugriff durchführen muss, zwar nicht so ausgeprägt, durch eine deutlich höhere Anzahl an Funktionsaufrufen ergibt sich aber auch bei MEMFS

eine Performanceverschlechterung. Der in den meisten Fällen bessere, da performantere Ansatz ist der des „data sieving“. Dabei wird ein großer Block aus der Datei in den Hauptspeicher gelesen und dort die angeforderten Daten herausgefiltert. Dieser Ansatz wird von ROMIO verfolgt und ist in dem Paper „Data Sieving and Collective I/O in ROMIO“ von [Thakur et al. 1999] ausführlich beschrieben. Zurzeit unterstützt MEMFS nur die einfachen Lese-/Schreiboperationen, nicht zusammenhängende Zugriffe werden durch die generische Funktion mit dem data sieving ausgeführt. Gerade für MEMFS sind hier aber Verbesserungen möglich, die in einem Folgenden Schritt der Entwicklung implementiert werden.

3.4.3. Anpassungen an ROMIO

Nachdem ein neues ADIO-Device entwickelt wurde, muss dieses noch ROMIO bekannt gemacht werden, damit aus MPI-IO-Anwendungen auf das Device zugegriffen werden kann. Diese Anpassungen sind je nach verwendeter MPI-Distribution unterschiedlich. Die Anpassung an MPICH ist leider nicht dokumentiert, durch ein „grep“ unter Linux auf den Namen eines bereits in ROMIO implementierten Dateisystems wie „testfs“ lassen sich aber schnell die Bereiche finden, an denen Anpassungen vorgenommen werden müssen. Die Angaben der anderen Dateisysteme lassen sich einfach übernehmen, nur die Namen müssen angepasst werden.

3.4.4. Anpassungen an Applikationen

Nachdem ein Dateisystem wie in den vorangegangen Abschnitten beschrieben in ROMIO und die MPI-Umgebung integriert wurde, können Applikationen darauf einfach zugreifen, indem der Name des Dateisystems dem Dateinamen beim Öffnen vorangestellt wird. Wird als Dateiname in unserem Fall bspw. „memfs:filename.txt“ verwendet, so wird das MEMFS-Device von ROMIO angesprochen. Weitere Anpassungen durch den Benutzer sind nicht erforderlich.

4. Performance

In dem Paper „An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces” von [Thakur et al. 1996] wird auch das Ergebnis einiger Performance-Tests vorgestellt. Die Tests fanden jeweils auf einem PIOFS- und einem PFS-Dateisystem statt. Dies sind Dateisysteme, die jeweils eine eigene API mit dem gleichen Namen verwenden.

Eine Anbindung für beide APIs existiert in ADIO, weshalb folgende Testszenarios gewählt wurden:

1. Eine PIOFS-API-Version eines Testprogramms wird direkt auf PIOFS ausgeführt. (PIOFS -> PIOFS)
2. Die PIOFS-API-Version wird durch ADIO auf PIOFS ausgeführt. (PIOFS -> ADIO -> PIOFS)
3. Die MPI-IO-Version eines Testprogramms wird durch ADIO auf PIOFS ausgeführt. (MPI-IO -> ADIO -> PIOFS)

Derselbe Testaufbau wurde noch einmal mit PFS statt PIOFS durchgeführt. In dem ersten Testprogramm greift jeder Prozess auf seine eigene Datei zu, schreibt 1 MB Daten in diese Datei und liest die Daten zurück. Dieses Lesen / Schreiben wird 10 Mal hintereinander ausgeführt. Das zweite Testprogramm ist identisch zum ersten bis darauf, dass alle Prozesse auf die gleiche Datei zugreifen und in Abhängigkeit ihres Rangs auf die Datei zugreifen. Die Ergebnisse beider Testprogramme für 16 Prozesse sind in der Tabelle 3 dargestellt:

PIOFS	PIOFS–ADIO		PFS–ADIO	
time	time	ovhd.	time	ovhd.
11.22	11.47	2.23%	11.68	4.10%

PFS	PFS–ADIO		PIOFS–ADIO	
time	time	ovhd.	time	ovhd.
22.28	22.78	2.24%	22.92	2.87%

Tabelle 1: Performance-Ergebnisse für ADIO, Quelle: [Thakur et al. 1996]

Der gemessene Overhead durch ADIO bewegt sich im Bereich zwischen 0 und 3 % gegenüber der Version, in der die API direkt auf das Dateisystem zugreift. In Hinblick auf die beschriebenen Vorteile, die sich durch ROMIO und ADIO ergeben, lässt sich dieser geringe Overhead in der absoluten Mehrzahl der Fälle wohl verkraften. Ein weiterer Test wurde mit einer Applikation durchgeführt, die eine wirkliche Anwendung der Universität Chicago zur Untersuchung des „gravitational collapes of self-gravitating gaseous clouds“ ist. Details dieser Anwendung sind in [Thakur et al. (2) 1996] beschrieben. Der hier gemessene Overhead bewegt sich hier im Bereich zwischen 0 und 4 %. Für diese Applikation existiert allerdings keine MPI-Implementierung, so dass hierfür keine Ergebnisse zur Verfügung stehen. Da ADIO stark an MPI-IO angelehnt ist, ließen sich hier wohl leicht bessere Resultate erzielen, wie auch Tabelle 3 vermuten lässt. Wie in Kapitel 3.3 gezeigt wurde, werden beim Einsatz von ADIO gegenüber dem direkten Einsatz auf einem Dateisystem viele ADIO-Funktionen nur bei der Initialisierung oder dem Öffnen einer Datei benötigt, der eigentliche I/O erzeugt nur wenig Overhead. Insofern ist bei I/O-intensiven Applikationen nicht von einem deutlichen Performanceverlust durch den

Einsatz von ADIO auszugehen, genaue Untersuchungen sind hier allerdings noch erforderlich.

5. Fazit

Als Fazit dieser Seminararbeit lassen sich die Vorteile, die sich durch ROMIO und ADIO ergeben, gegenüber nur geringen Performanceeinbußen, hervorheben. Da der erste Teil dieser Seminararbeit leider nicht bearbeitet wurde, beschränkt sich dieses Fazit auf ADIO. ADIO ist eine performante Schnittstelle zwischen APIs für parallelen I/O und beliebigen Dateisystemen. Jede API muss nur einmal an ADIO angepasst werden, um danach alle in ADIO implementierten Dateisysteme nutzen zu können. Ebenso muss jedes Dateisystem nur einmal für ADIO implementiert werden, um danach für jede an ADIO angepasste API verfügbar zu sein. Die nötigen Schritte zur Implementierung eines Dateisystems für ADIO wurden im Kapitel 3.4 vorgestellt und erweisen sich als nicht besonders aufwendig. Die Anpassung einer API an ADIO hätte am Beispiel der MPI-IO-Implementierung ROMIO gezeigt werden sollen, was leider nicht geschehen ist. Da ADIO ebenfalls zu einem großen Teil auf MPI-Datentypen setzt, ergibt sich gerade für die Beziehung ROMIO – ADIO eine gute Performance.

Literaturverzeichnis

[Thakur et al. 1996]: Thakur, R. et al.: „An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces“, 1996, Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, Argonne, IL, USA

[Thakur et al. 1999]: Thakur, R. et al.: „Data Sieving and Collective I/O in ROMIO“, 1999, Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, Argonne, IL, USA

[Thakur et al. (2) 1996]: Thakur, R. et al.: „An Experimental Evaluation of the Parallel I/O Systems of the IBM SP and Intel paragon Using a Production Application“, 1996, Proceedings of the 3rd International Conference of the Austrian Center for Parallel Computation (ACPC) with special emphasis on Parallel Databases and Parallel I/O

[Thakur et al. (2) 1999]: Thakur, R. et al.: “On Implementing MPI-IO Portably and with High Performance”, 1999, Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems

[Thakur et al. 2002]: Thakur, R. et al.: “Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation”, 2002, Argonne National Laboratory, Argonne, IL, USA

Anhang

A1: ADIO-Devicefunktionen für MEMFS

```
struct ADIOI_Fns_struct ADIO_MEMFS_operations = {
    ADIOI_MEMFS_Open, /* Open */
    ADIOI_MEMFS_ReadContig, /* ReadContig */
    ADIOI_MEMFS_WriteContig, /* WriteContig */
    ADIOI_MEMFS_ReadStridedColl, /* ReadStridedColl */
    ADIOI_MEMFS_WriteStridedColl, /* WriteStridedColl */
    ADIOI_MEMFS_SeekIndividual, /* SeekIndividual */
    ADIOI_MEMFS_Fcntl, /* Fcntl */
    ADIOI_MEMFS_SetInfo, /* SetInfo */
    ADIOI_MEMFS_ReadStrided, /* ReadStrided */
    ADIOI_MEMFS_WriteStrided, /* WriteStrided */
    ADIOI_MEMFS_Close, /* Close */
    ADIOI_MEMFS_IreadContig, /* IreadContig */
    ADIOI_MEMFS_IwriteContig, /* IwriteContig */
    ADIOI_MEMFS_ReadDone, /* ReadDone */
    ADIOI_MEMFS_WriteDone, /* WriteDone */
    ADIOI_MEMFS_ReadComplete, /* ReadComplete */
    ADIOI_MEMFS_WriteComplete, /* WriteComplete */
    ADIOI_MEMFS_IreadStrided, /* IreadStrided */
    ADIOI_MEMFS_IwriteStrided, /* IwriteStrided */
    ADIOI_MEMFS_Flush, /* Flush */
    ADIOI_MEMFS_Resize, /* Resize */
    ADIOI_MEMFS_Delete, /* Delete */
};
```

Tabelle 2: MEMFS Devicefunktionen

A2: Dateideskriptor ADIO_File

```
typedef struct ADIOI_FileD {
    int cookie;               /* for error checking */
    FDTYPE fd_sys;               /* system file descriptor */
#ifdef XFS
    int fd_direct;            /* On XFS, this is used for direct I/O;
                                 fd_sys is used for buffered I/O */
    int direct_read;          /* flag; 1 means use direct read */
    int direct_write;         /* flag; 1 means use direct write  */
    /* direct I/O attributes */
    unsigned d_mem;           /* data buffer memory alignment */
    unsigned d_miniosz;       /* min xfer size, xfer size multiple,
                                 and file seek offset alignment */
    unsigned d_maxiosz;       /* max xfer size */
#endif
    ADIO_Offset fp_ind;       /* individual file pointer in MPI-IO (in bytes)*/
    ADIO_Offset fp_sys_posn; /* current location of the system file-pointer
                                 in bytes */
    ADIOI_Fns *fns;           /* struct of I/O functions to use */
    MPI_Comm comm;            /* communicator indicating who called open */
    MPI_Comm agg_comm;       /* deferred open: aggregators who called open */
    int io_worker;           /* bool: if one proc should do io, is it me? */
    int is_open;      /* deferred open: 0: not open yet 1: is open */
    char *filename;
    int file_system;          /* type of file system */
```

```
    int access_mode;          /* Access mode (sequential, append, etc.) */
    ADIO_Offset disp;         /* reqd. for MPI-IO */
    MPI_Datatype etype;       /* reqd. for MPI-IO */
    MPI_Datatype filetype;    /* reqd. for MPI-IO */
    int etype_size;           /* in bytes */
    ADIOI_Hints *hints;       /* structure containing fs-indep. info values */
    MPI_Info info;

    /* The following support the split collective operations */
    int split_coll_count;     /* count of outstanding split coll. ops. */
    MPI_Status split_status; /* status used for split collectives */
    MPI_Datatype split_datatype; /* datatype used for split collectives */

    /* The following support the shared file operations */
    char *shared_fp_fname;    /* name of file containing shared file pointer */
    struct ADIOI_FileD *shared_fp_fd;  /* file handle of file
                                          containing shared fp */
    int async_count;          /* count of outstanding nonblocking operations */
    int perm;
    int atomicity;           /* true=atomic, false=nonatomic */
    int iomode;              /* reqd. to implement Intel PFS modes */
    MPI_Errhandler err_handler;
    void *fs_ptr;             /* file-system specific information */
} ADIOI_FileD;

typedef struct ADIOI_FileD *ADIO_File;
```

Tabelle 3: Dateideskriptor ADIO_File