IMPLEMENTIERUNGSKONZEPTE FÜR PARALLELEN I/O IN MPICH/ROMIO

von

Alex Koch (ak@em-motion.com)

Im Fach „Verteilte und parallele Systeme“

Fachbereich Informatik
FH-Bonn-Rhein-Sieg

INHALTSVERZEICHNIS

EINLEITUNG

Für parallelen I/O gab es lange Zeit keine notwendigen Standards. Diese sind Voraussetzung für portable parallele Programmierung. Die meisten parallelen Dateisysteme haben unixähnliche APIs und sind dadurch nicht immer portabel. Die unixähnlichen API ist nicht für parallelen I/O geeignet. Viele Features, welche für parallele Programmierung unverzichtbar sind und große Performancegewinne mit sich bringen, wie z.B. nichtsequentielle oder kollektive Zugriffe, sind in dieser API nicht vorhanden. Um diese Einschränkung zu überwinden wurde von dem MPI Forum eine neue API für parallelen I/O als Teil des MPI-2 Standards definiert. Diese sogenannte MPI-IO ist eine umfassende API mit vielen Besonderheiten, die extra für parallele I/O ausgelegt sind und für Portabilität und hohe Performance sorgen. Es existieren schon viele MPI-IO - Implementierungen, sowohl portable, als auch hardware-spezifische. ROMIO ist eine freie MPI-IO Implementierung und wird z.B. in MPICH verwendet. Die folgende Seminararbeit beschreibt einige Implementierungskonzepte der parallelen I/O in ROMIO.

PORTABILITÄT UND PERFORMANCE

Um MPI-IO portabel zu implementieren, können verschiedene Wege gewählt werden. Es wäre möglich eine API zu bauen, die auf grundlegenden Unix I/O Funktionen, wie open, lseek, read, write, und close arbeitet. Da diese Funktionen auf jedem unixähnlichem Betriebssystem unterstützt werden, wird so eine API portabel gestaltet. Diese Lösung ist aber sowohl in Funktionalität, als auch von der Performanceseite begrenzt.

Die Standard I/O Funktionen eines unixähnlichen Betriebssystems haben folgenden Begrenzungen:

1. Unix I/O Funktionen sind blockierend. Viele bieten einen Set an nichtblockierenden I/O Funktionen. Diese sind aber nicht portabel. (Nichtblockierende Funktionen können Implementiert werden indem man blockierende I/O Funktionen in Threads aufruft.)

2. Bei vielen Dateisystemen sind die Standard I/O Routinen nur für eine Dateigröße von maximal 2Gbyte ausgelegt. Um größere Dateien ansprechen zu können müssen andere Routinen, die extra dafür Implementiert sind, benutzt werden. Diese Routinen sind ebenfalls nicht portabel. (Operationen für extragroße Dateien müssen in MPI-IO nicht implementiert werden. Es ist zu empfehlen solche Operationen zu implementieren.)

3. Einige Dateisysteme unterstützen zusätzliche, nicht portable Funktionen. (z.B. file preallocation und atomarer und nichtatomarer Dateizugriffsmodus)

Obengenante nichtportable Funktionen sind in manchen Fällen sehr nützlich. Sie wirken sich auf die Performance und die Gesamtleistung einer MPI-IO Implementierung aus. Es ist also sinnvoll eine MPI-IO Implementierung nicht nur auf Basis von grundlegenden Unix-I/O Funktionen aufzubauen, wenn diese auch auf vielen Dateisystemen unterstützt werden. Für jedes der unterschiedlichen Dateisysteme gibt es auf die Systeme abgestimmte I/O Funktionen. Es wird empfohlen, je nach Dateisystem, auf diesen aufzubauen:

- Auf dem Intel Paragon System werden die Funktionen cread und cwrite anstatt read und write empfohlen

- Auf SGI IRIX 6.5 System werden die Funktionen pread64 und pwrite64 anstatt read und write empfohlen

- Auf HP Maschinen mit dem SPPUX Betriebssystem werden die Funktionen pread64 und pwrite64 anstatt read und write empfohlen

Ein anderer Grund ist die Nutzung anderen APIs auf einigen Systemen im Forschungsbereich.

Alternativ kann eine portable parallele API auf dem POSIX I/O Interface aufgebaut werden. POSIX ist ein internationaler Standard mit erweiterter Funktionalität.

Obwohl POSIX ein Standard ist, wird dieses Interface nicht in jedem System implementiert. Weiterhin halten sich nicht alle Systeme strickt an dieses Standard. Es werden nur Teile des Standards eingebaut.

Fazit: Um eine MPI-IO Implementierung portabel und leistungsfähig zu Implementieren, sollten die besseren Eigenschaften des jeweiligen System ausgenutzt werden. Eine MPI-IO Implementierung darf nicht nur auf grundlegenden Unixfunktionen aufgebaut sein. Auch Aufbau auf POSIX I/O Interface wird nicht den hohen Leistungsanforderungen entsprechen. Ein im nächsten Kapitel beschriebene Konzept, bittet eine Möglichkeit dieses Problem zu lösen.

Die Entwickler von ADIO haben eine abstrakte Standardschnittstelle für parallele I/O Implementierungen definiert. Man kann verschiedene sehr effiziente parallele I/O Implementierungen entwickeln, indem man auf ADIO aufsetzt. Das ADIO muss dann für die verschiedensten Plattformen und Dateisysteme entwickelt werden. Die parallel I/O Implementierungen werden dadurch portabel.

Das Hauptziel von ADIO ist es die Programmierung der neuen APIs für existierende und neue Dateisysteme zu vereinfachen. Die Abbildung 1 veranschaulicht das ADIO – Konzept.

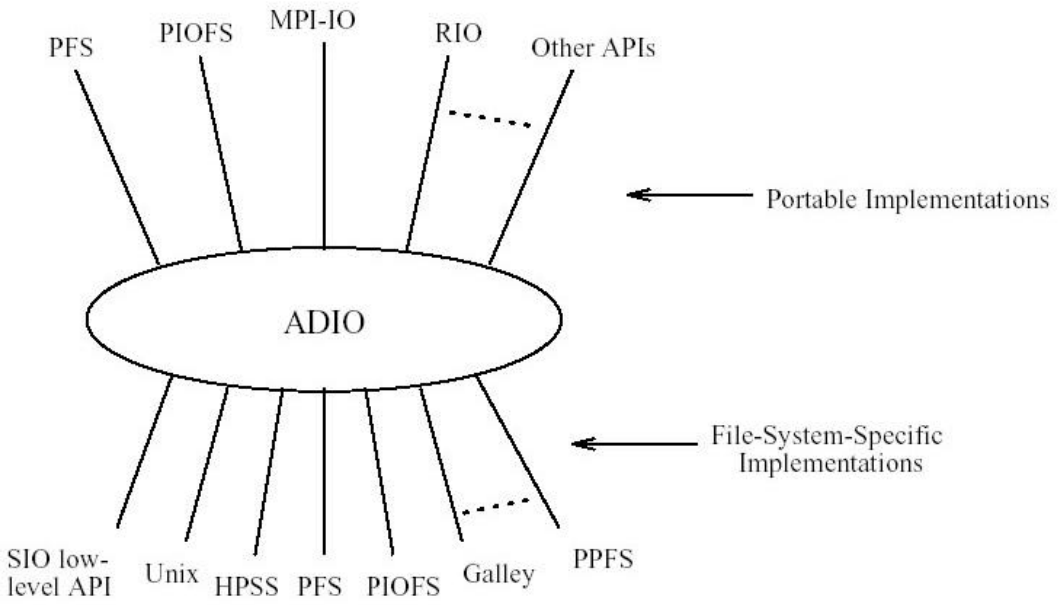

Abbildung 1: ADIO - Konzept

ADIO besteht aus einem kleinen Set von grundlegenden Funktionen zum Durchführen von parallelen I/O Zugriffen. Es muss für jedes Dateisystem, auf dem parallele I/O Implementierung eingesetzt werden soll, eine optimierte Implementierung von ADIO geben. Mit Hilfe des ADIO – Konzepts ist es möglich den hardwareunabhängigen Teil einer parallelen API vom hardwareabhängigen Teil zu trennen. Der hardwareabhängige Teil ist dann das ADIO selbst.Wird eine neue API auf dem ADIO aufgebaut, kann sie direkt auf allen Systemen benutzt werden, für die schon eine ADIO-Implementierung existiert. Wenn für ein neues Dateisystem eine ADIO-Implementierung entwickelt wird, kann dieses Dateisystem von allen Applikationen genutzt

werden, die eine parallele I/O API benutzen, welches auf ADIO aufgebaut wird. Diese Eigenschaft von ADIO soll das Ausführen der Userapplikationen auf vielen Plattformen ohne Rücksicht auf die eingesetzte API erlauben.

ADIO wurde so entworfen, dass alle hochperformanten Eigenschaften jedes parallelen Dateisystems ausgenutzt werden können. Vor dem Entwurf von ADIO haben Authoren die Schnittstellen und Funktionalitäten von vielen verschiedenen parallelen Dateisystemen und Programmbibliotheken studiert. Auf dieser Basis wurde entschieden, welche Funktionalitäten von ADIO unterstütz werden müssen.

MPI wird in ADIO vielfach eingesetzt. Dadurch wird ADIO portabel und leistungsfähig. Daraus ergibt sich, dass fast alle ADIO Routinen MPI – Datentypen und Kommunikatoren als Parameter bekommen.

In ADIO sind Routinen für folgende Operationen definiert:

- Zum Öffnen und Schließen einer Datei. Diese Operationen sind kollektiv und bekommen einen MPI Kommunikator als Parameter. (ADIO_Open, ADIO_Close)

- Zum sequentiellen Lesen bzw. Schreiben in eine Datei. (ADIO_ReadContig, ADIO_WriteContig)

- Zu nichtsequenziellen Lesen bzw. Schreiben in eine Datei. (ADIO_ReadStrided, ADIO_WriteStrided)

- Für nichtblockierende Lese- bzw. Schreibzugriffe. (ADIO_IreadContig, ADIO_IwriteContig, ADIO_IreadStrided, ADIO_IwriteStrided)

- Für Kollektive sequentielle und nicht sequentielle Lese- bzw. Schreibzugriffe. (ADIO_ReadContigColl, ADIO_ReadStridedColl, ADIO_IreadContigColl, ADIO_IreadStridedColl, ADIO_WriteContigColl, ADIO_WriteStridedColl, ADIO_IwriteContigColl, ADIO_IwriteStridedColl). Verschiedene Forschungen haben gezeigt, dass alle verschiedene Zugriffsmuster, die in den vorherigen Punkte beschrieben wurden, auch als kollektive Variante sehr sinnvoll und Performancesteigernd sind.

- Zur Suche bzw. freier Positionierung des Filepointers (ADIO_SeekIndividual)

- Zum Testen der nichtblockierenden Lese und Schreiboperationen und zum Warten auf diese Operationen (ADIO_ReadDone, ADIO_ReadIComplete, ADIO_ReadComplete, ADIO_WriteDone, ADIO_WriteIComplete, ADIO_WriteComplete)

- Zum Abfragen der Dateiinformationen wie z.B. Type der Datei oder Zugriffsmethode (ADIO_Fcntl)

- Verschiedene anderen z.B. zum Initialisieren oder Herunterfahren von ADIO (ADIO_Init, ADIO_End, ADIO_Delete, ADIO_Resize, ADIO_Flush)

Viele ADIO Funktionen können auf einigen Dateisystemen direkt auf passende Funktionen dieses Dateisystems abgebildet werden. Z.B. wird read auf cread. Die Abbildung der Funktionen kann in ADIO über Makros realisiert werden. Nur die Funktionen, die von einem Dateisystem nicht unterstützt werden, müssen explizit für dieses Dateisystem implementiert werden.Um eine nichtsequentielle Lese- bzw. Schreiboperation zu ermöglichen, müssen eine Reihe von sequentiellen Zugriffen, gesteuert durch Suchoperationen, durchgeführt werden. Eine Reihe von Studien hat gezeigt, dass eine ADIO Implementierung mit Makros besser ist, als eine mit Funktionsaufrufen.

Fazit: Über Abstract Device Interface for parallele I/O (ADIO) kann eine parallele I/O API portabel und mit geringem Aufwand implementiert werden.

Im nächsten Kapitel wird eine solche Implementierung am Beispiel von ROMIO vorgestellt. Es wird auf andere Implementierungsaspekte zur Steigerung der Performance eingegangen.

ROMIO ist eine frei verfügbare Implementierung von MPI-IO die auf ADIO aufbaut. In der aktuellsten Version 1.0.3 ist ROMIO für folgende Plattformen verfügbar: IBM SP, Intel Paragon, HP Exemplar, SGI Origin2000 und T3E, NEC SX-4 und andere Systeme, wie HP, SUN, SGI, DEC und IBM. ROMIO ist auf Workstation-cluster einsetzbar (SUN, SGI, HP, IBM, DEC, Linux und FreeBSD). Es werden folgende Dateisysteme unterstützt: IBM PIOFS, Intel PFS, HP HFS, SGI XFS, NEC SFS, NFS, und einige Unixdateisysteme. Bezogen auf I/O enthält ROMIO alle im MPI-2-Standard definierte Funktionen (Z.B. für kollektive I/O und Fehlerbehandlung). In MPICH, eine frei verfügbare MPI-Implementierung, HP MPI und SGI MPI wird ROMIO genutzt.

Um hohe Latenzzeiten zu vermeiden benutzt ROMIO ein Konzept namens „Data sieving“. Durch dieses Konzept sind nur einige wenige sequentielle Zugriffe auf Dateien notwendig, um einen nichtsequentiellen Request bearbeiten zu können. Ein anderes Konzept namens „Collective IO“ erlaubt es, Requests von verschiedenen Prozessen, die sich möglicherweise überlappen, zusammenzufassen und zu bearbeiten. Um den Einsatz der Konzepte zu verdeutlichen, ein Beispiel, welches Probleme bei nichtsequentiellen I/O Zugriffen erläutert.

„File view“ ist ein Konzept in MPI-IO, das einen geordneten Dateizugriff ermöglicht. Ein Prozess öffnet eine Datei. Der gesamte Inhalt der Datei ist für den Prozess sichtbar. Diese Sichtbarkeit kann von dem Prozess geändert werden, indem eine „file view“ mit Hilfe der Routine MPI_File_set_view und eines bestimmten MPI Datentyps gesetzt wird. Danach sind alle Lese bzw. Schreibzugriffe auf diese Datei nach diesem Datentyp geordnet. Nur die Bereiche der Datei können beschrieben bzw. gelesen werden die in diesem Datentyp definiert sind. Die dabei entstehenden Lücken in der sequentiellbeschriebenen Datei werden ignoriert.

Einige Studien haben gezeigt, dass in vielen parallelen Applikationen die Prozesse auf relativ kurze nichtsequentiell geordnete Dateiabschnitte zugreifen. Um die Leistung solcher Applikationen nicht unnötig zu verschlechtern, muss das I/O Interface solche Zugriffsmethoden optimieren.

Ein geeignetes Beispiel dafür ist das Einlesen eines verteilten Arrays aus einer Datei (Abbildung 2). Dieses Zugriffsmuster kommt in der Welt des parallelen Rechnens sehr oft vor.

Ein Zweidimensionaler Array wird von 16 Prozessen aus einer sequentiell Beschriebenen Datei eingelesen. Jeder Prozess muss sein eigenes zweidimensionales, lokales Subarray erstellen.

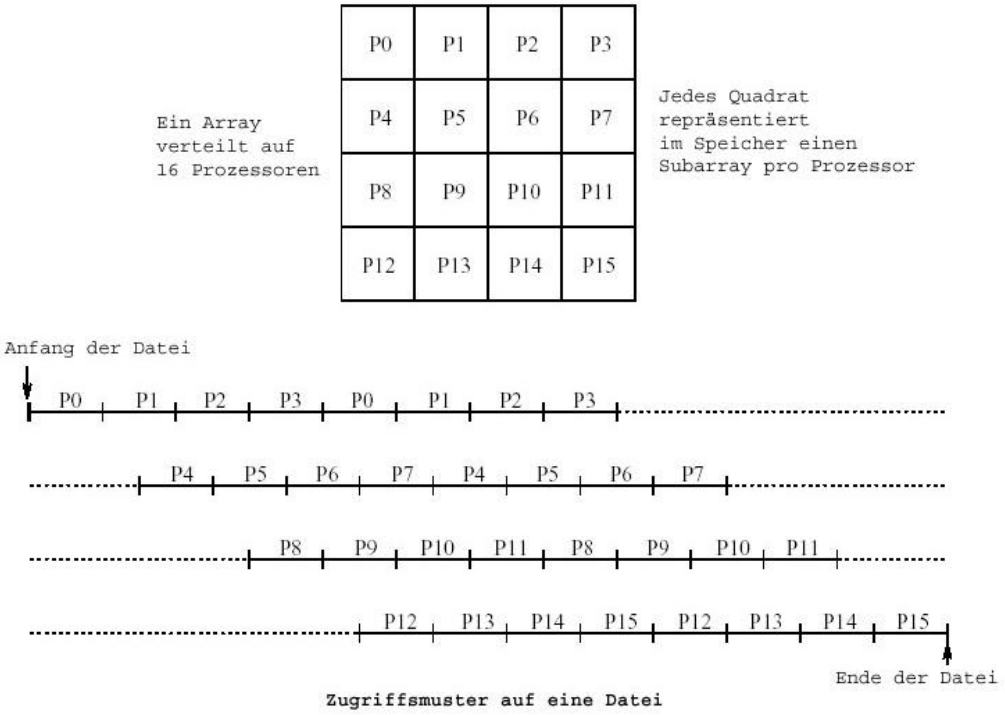

Abbildung 2: Einlesen eines verteilten Arrays

Die Daten sind in der Datei so Angeordnet, dass jeder Prozess zwei nichtzusammenhängende Blöcke einliest (noncontiguous access). In der Abbildung 3 wird gezeigt, auf welche 4 Arten ein User sein Programm schreiben kann damit die verschiedenen Prozesse ihre lokalen Arrays einlesen.

- Level 0: Jeder Prozess benutzt Standard Unix-operationen zum zeilenweisen Einlesen in seinen lokalen Array.
- Level 1: Wieder Standard Unix-operationen aber kollektiv.
- Level 2: Jeder Prozess benutzt ein nichtsequentielles Zugriffsmuster, definiert ein „file view“ und greift so auf die Daten zu.
- Level 3: Wieder Verwendung von nichtsequentiellen Zugriffsmuster aber auf kollektive Weise

```
MPI_File_open(..., "filename", ..., &fh)
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, ...)
   MPI_File_read(fh, row[i], ...)
}
MPI_File_close(&fh)
```

Level 0
(many independent, contiguous requests)

```
MPI_File_open(MPI_COMM_WORLD, "filename", ..., &fh)
for (i=0; i<n_local_rows; i++) {
   MPI_File_seek(fh, ...)
   MPI_File_read_all(fh, row[i], ...)
}
MPI_File_close(&fh)
```

Level 1
(many collective, contiguous requests)

```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(..., "filename", ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read(fh, local_array, ...)
MPI_File_close(&fh)
```

Level 2
(single independent, noncontiguous request)

```
MPI_Type_create_subarray(..., &subarray, ...)
MPI_Type_commit(&subarray)
MPI_File_open(MPI_COMM_WORLD, "filename", ..., &fh)
MPI_File_set_view(fh, ..., subarray, ...)
MPI_File_read_all(fh, local_array, ...)
MPI_File_close(&fh)
```

Level 3
(single collective, noncontiguous request)

Abbildung 3: Pseudocode für die 4 Zugriffsmöglichkeiten

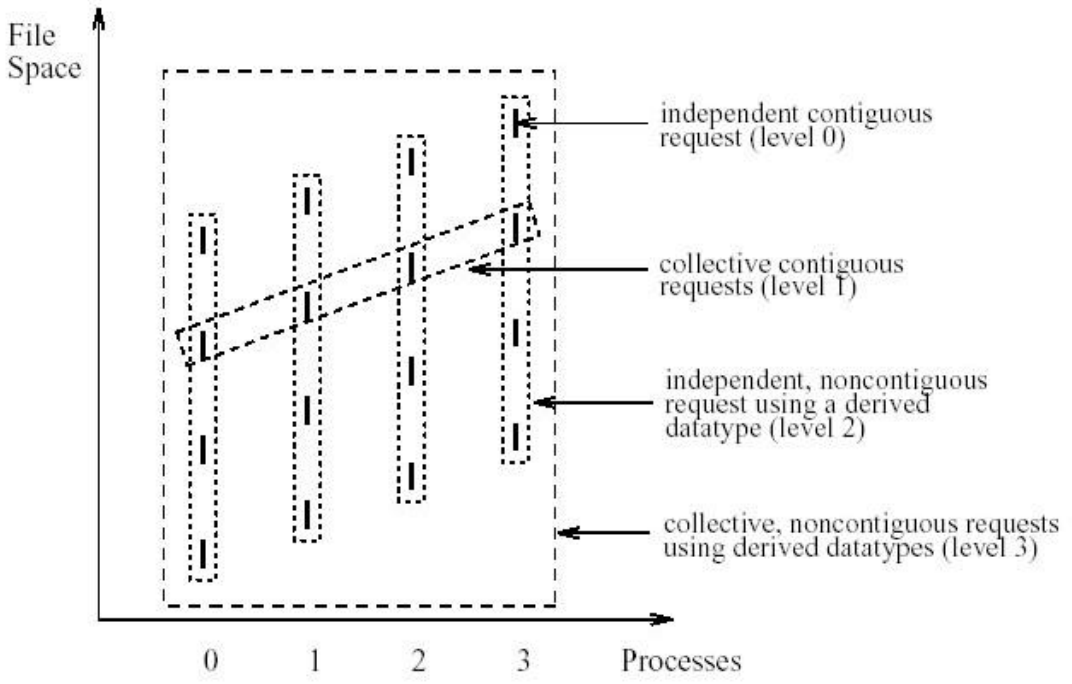

Abbildung 4: Verhalten der Datenmenge pro Request

Das Diagramm in Abbildung 4 veranschaulicht noch einmal die 4 Zugriffsarten und zeigt das Verhalten der Datenmenge pro I/O Request.

In welchen Applikationen und an welcher Stelle welche Zugriffsmethode eingesetzt werden, müssen die Entwickler der Applikation entscheiden. Es ist zu empfehlen die Level 3 Zugriffsmethoden einzusetzen.

„DATA SIEVING“

Um hohe Latenzzeiten zu vermeiden benutzt ROMIO ein Konzept namens „Data sieving“.

Diese Technik wird im ROMIO zur Bearbeitung von nichtsequentiellen Zugriffen eingesetzt. Der Nachteil dieser Technik ist der hohe Speicherverbrauch. In der Abbildung 5 ist die Idee dieser Technik dargestellt.

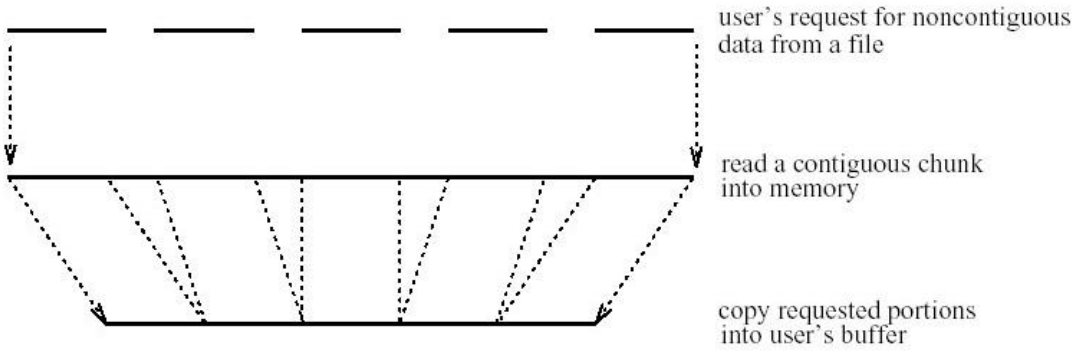

Abbildung 5: Data sieving

Ein User hat 5 nichtsequentielle Datenblöcke in einem Request angefordert. Die Datenblöcke werden nicht in 5 verschiedenen I/O Zugriffen gelesen. Über einen Zugriff liest ROMIO alle Datenblöcke, einschließlich der Lücken, in einen Buffer ein. Anschließend werden nur die Datenlöcke in einen Buffer des Users umkopiert.

„Data sieving“ kann auch bei Schreibzugriffen eingesetzt werden. Das Verfahren heißt „read-modify-write“. Bei Schreibzugriffen mit „data sieving“ liest ROMIO zuerst einen zusammenhängenden Bereich der Datei in den Speicher ein. Die Daten werden modifiziert und zurück in die Datei geschrieben. Während dieses Vorganges muss der eingelesene Bereich blockiert werden, damit andere Prozesse die Daten in dem Bereich nicht verändern können.

Wie man der Abbildung 5 entnehmen kann, belegen die Lücken zusätzlichen Speicher, was den Nachteil dieser Technik ausmacht. Die maximale Größe des Buffers kann in ROMIO per Parameter dynamisch definiert werden. Default Größe dieses temporären Buffers ist 4 MB. Wenn die Größe des Datenblocks, der eingelesen werden soll, größer als der temporäre Buffer ist, führt ROMIO „data sieving“ in Teilen aus. Anders betrachtet, wenn die Größe des Buffers zu groß

gewählt wird, führt das auch zu Beeinträchtigung der parallelen Abläufe der Prozesse durch blockierende Schreibzugriffe.

ROMIO benutzt „data sieving“, wenn der User nichtsequentielle, nichtkollektive Dateizugriffe ausführt. „Data sieving“ wird aber auch aus dem „Collective I/O“ Konzept bei nichtsequentiellen kollektiven Zugriffen eingesetzt.

Ein anderes Konzept wird im ROMIO eingesetzt um gemeinsame (von mehreren Prozessen) nichtsequentielle Zugriffe auf eine Datei zu optimieren.

Die in ROMIO implementierte Variante des kollektiven I/O ist eine sogenannte „client level two-phase I/O“. ROMIO benutzt kollektive I/O, wenn User einen Level 3 MPI-IO Zugriff startet.

Zwei Phasen I/O wurde zum ersten mal beim Lesen der verteilten Arrays aus einer Datei eingesetzt (siehe Abbildung 6). In dem Beispiel aus der Abbildung sind die Daten für ein lokales Array eines Prozesses nicht sequentiell in der Datei abgespeichert. Jeder Zeile eines lokalen Array eines Prozesses folgt eine Zeile eines lokalen Array eines anderen Prozesses. Es wäre aus den Performancegründen ungünstig, wenn jeder Prozess jede von ihm benötigte Zeile separat einlesen würde. Bei zwei Phasen I/O wird anders verfahren. Wenn das gesamte Zugriffsmuster der Applikation von Anfang an bekannt ist, kann jeder Prozess in der ersten Phase einen zusammenhängenden Zugriff auf eine größere Datenmenge machen. In der zweiten Phase werden die Daten an die anderen Prozesse verteilt, welche diese benötigen. Durch Reduktion der Anzahl der I/O Zugriffe auf kleinere Mengen von Daten wird die I/O Leistung drastisch gesteigert.

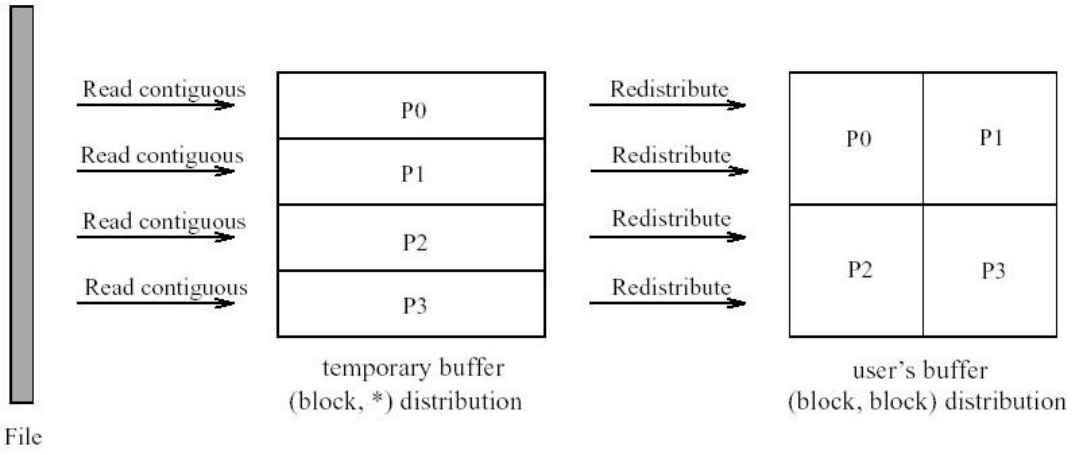

Abbildung 6: Zwei Phasen I/O

ROMIO baut auf einer verallgemeinerte Version von zwei Phasen I/O auf. Dabei benutzt ROMIO zwei Parameter die vom User eingestellt werden können.

- Anzahl der Prozesse die auf eine Datei Zugreifen können

- maximale größe des temporären Buffers für jeden Prozess.

Jeder Prozess analysiert in ROMIO zuerst seine eigene I/O Anfrage und erzeugt anschleißend eine Liste von Startpositionen und eine Liste mit Längen der einzulesenden Daten. Jeder Prozess rechnet die Startposition des ersten und des letzten Bytes aus und sendet diese per Broadcast an alle anderen Prozesse. Als Ergebnis dieser Aktion hat jeder Prozess die Start- und Endposition jedes Prozesses parat. Als nächstes versuchen die Prozesse zu bestimmen, ob die gegebenen Zugriffsmuster von Kollektiven I/O optimiert werden können (Überlappungen). Das heißt, wenn für zwei Prozesse mit fortlaufenden Nummern i und i+1 die Bedingung (start-offset i+1 < end-offset i) nicht erfüllt ist, dann starten die Prozesse keinen kollektiven I/O Zugriff. Wenn aber diese Bedingung erfüllt ist, werden den Prozessen ihre Zugriffsbereiche (File domains) in der Datei zugewiesen (Abbildung 7). Die Zuweisung der Zugriffsbereiche erfolgt auf folgende Art und Weise. Alle Prozesse bestimmen den minimalen Start-offset und den maximalen End-offset aller Prozesse. Die Differenz zwischen diesen zwei Offsets ist die Gesamtdatenmenge die eingelesen werden muss. Die Gesamtdatenmenge wird durch Anzahl der Prozesse geteilt und die dadurch entstandene Bereiche den Prozessen zugewiesen.

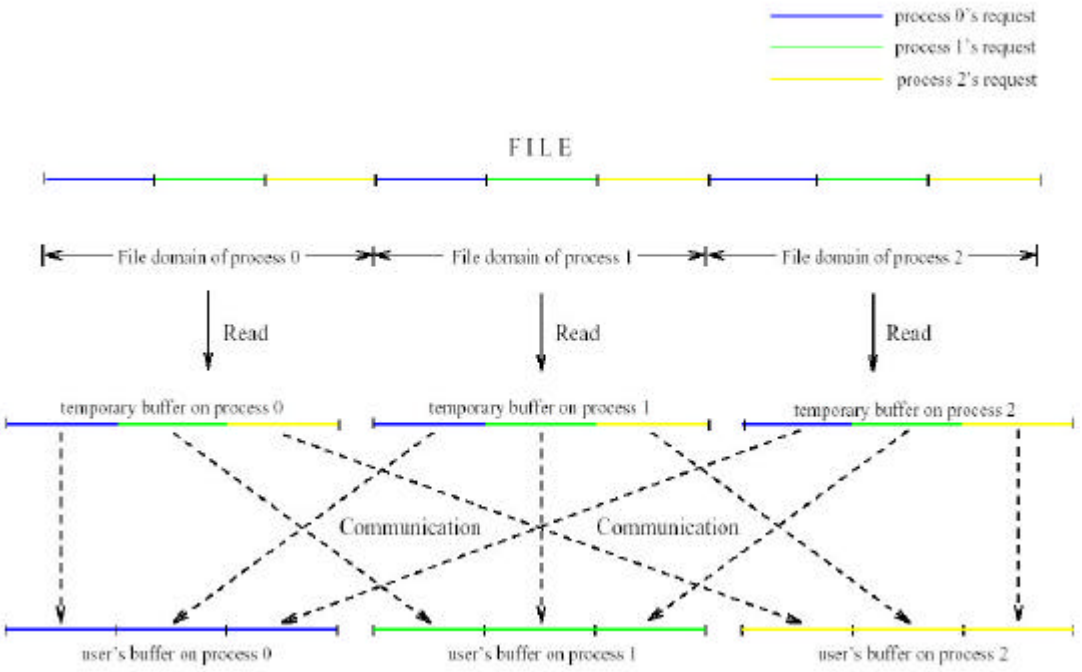

Abbildung 7: Kollektiver Lesezugriff in ROMIO

Beispiel: Es werden Daten vom Start-offset 200 bis End-offset 499 benötigt Ausserdem gibt es drei Prozesse die diese Daten anfordern. Die „file domain“ des Prozesses mit der laufenden

Nummer 0 repräsentiert den Dateibereich 200 bis 299, die Domain des Prozesses mit der Nummer 1 repräsentiert den Bereich 300 bis 399 und die Domain des Prozesses mit der Nummer 2 repräsentiert den Bereich 400 bis 499.

Wie im zwei Phasen I/O Konzept, list jeder Prozess in der ersten Phase die Daten mit einer einzigen Anfrage aus dem ihm zugewiesenen Bereich.

In der nächsten Phase müssen die Prozesse errechnen von welchen anderen Prozessen und aus welchen Bereichen sie Daten benötigen. Sie erstellen wieder Listen mit Startpositionen und Längen der benötigten Daten. Diese werden an die zuständigen Prozesse geschickt. Danach sendet jeder Prozess die angeforderten Daten an die entsprechenden Nachbarn.

Kollektives Schreiben ist in ROMIO auch möglich. Es ist dabei zu beachten, dass beim Schreiben die Kommunikationsphase die erste Phase und I/O die zweite Phase ist. Beim Schreiben wird wie in „data sieving“ das Konzept „read-modify-write“ eingesetzt.

An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces	Rajeev Thakur, William Gropp, Ewing Lusk Oktober 1996
On Implementing MPI-IO Portably and with Heigh Performance	Rajeev Thakur, William Gropp, Ewing Lusk May 1999
Optimizing Noncontiguous Access in MPI-IO	Rajeev Thakur, William Gropp, Ewing Lusk Parallel Computing, 2002
Data Sieving and Collective I/O in ROMIO	Rajeev Thakur, William Gropp, Ewing Lusk February 1999