Fachhochschule Bonn-Rhein-Sieg

Fachbereich Informatik

Verteilte und Parallele Systeme WS 2001/2002

MPI I/O

Die Software-Schnittstelle zum parallelen I/O in MPI-2

Stefan Seichter

30-12-2001

Inhalt **Seite**

1 Einleitung

Der gemeinsame Zugriff mehrerer Prozesse auf einen Datenbestand stellt ein Problem bei Systemen ohne gemeinsamen Speicher dar. Die Zugriffe verschiedener Prozesse müssen auf den gemeinsamen Daten koordiniert werden, ohne die absolute Speicheradresse zu kennen. Der Standard MPI 2 bietet eine Möglichkeit auf hoher Abstraktionsebene einen parallelen Zugriff auf gemeinsam genutzte Dateien zu realisieren.

2 MPI – Ein Überblick

MPI – Message Passing Interface – ist ein Quasi-Standard zur nachrichtenbasierten Kommunikation in der parallelen Datenverarbeitung. Er wurde 1992 vom Message Passing Forum (MPF), einem Konsortium aus Industrie, Wirtschaft und Forschung, in der Version MPI 1 verabschiedet. 1995 und 1996 folgten die Standards MPI 1.1 und MPI 1.2.

MPI bietet eine einheitliche API zur parallelen Programmierung. Es soll Implementierungs- und Plattformunabhängigkeit sicherstellen und somit diesen Nachteilen vieler proprietärer Implementierungen entgegenwirken. Programme, die in MPI geschrieben wurden, sollen sourcecode-kompatibel sein, dass heißt sie sollen unverändert auf allen MPI-Implementierungen kompilierbar und lauffähig sein.

MPI ist keine Standardimplementierung sondern ein Interface Framework, das eine Betriebssystem- und Programmiersprachenunabhängigkeit sicherstellen soll. Daher schreibt MPI nicht alle Implementierungsaspekte vor. Einige Punkte sind als implementierungsabhängig in MPI vorgesehen, gefährden aber nicht die Funktionalität des Frameworks.

1997 wurde MPI 2 verabschiedet. Es enthält neben den Funktionalitäten von MPI 1.x Funktionen zur Prozessverwaltung, Verwaltung von shared memory. Funktionen zur parallelen Ein-/Ausgabe sind ebenfalls Bestandteil des MPI-Standards.

2.1 Prozessgruppen und Kommunikation

Alle Prozesse in MPI werden innerhalb von Prozessgruppen verwaltet. Das statische Prozessmodell von MPI schreibt vor, dass alle Prozesse zu Beginn einer Verarbeitung existieren und gleichzeitig in die MPI-Laufzeitumgebung eintreten. Die Kommunikation zwischen Prozessen einer Gruppe wird durch Kommunikatoren geregelt.

Die Laufzeitumgebung von MPI muss explizit gestartet und beendet werden (MPI_Init, MPI_Finalize). Nach Eintritt in die Laufzeitumgebung kann jeder Prozess Informationen über diese abrufen. Von der Laufzeitumgebung kann der Prozess erfragen, welche anderen Prozesse in der gleichen Kommunikationsumgebung angemeldet sind und welcher gerade aktiv ist. Alle Prozesse sind über den generischen Kommunikationskanal MPI_COMM_WORLD erreichbar.

Nachrichten werden vom MPI in message-envelopes ausgetauscht. Ein message-envelope enthält Informationen über Sender, Empfänger, einem Tag und den Kommunikator. Die versendeten Daten können sowohl primitive MPI-Datentypen(MPI_INTEGER, MPI_CHAR,…), als auch komplexe Datentypen sein.

Die Kommunikation zwischen zwei Prozessen erfolgt Point-to-Point. Sie kann blockierend oder nichtblockierend, gepuffert oder ungepuffert sein. Ein Prozess übernimmt explizit die Rolle des Senders, der andere die Rolle des Empfängers.

3 Problemstellung

Ein gemeinsamer paralleler Dateizugriff stellt ein System vor einige Probleme. Es muss sichergestellt werden, dass alle Prozesse Zugriff auf eine Datei erhalten, die sie bearbeiten. Weiterhin muss ein allen bekannter Speicherort und allen bekannte Speicheradresse existieren. Die Speicherung von Daten eines jeden Prozesses muss sichergestellt werden können, ohne, dass diese die Daten von anderen Prozessen beeinflusst oder stört. Zuletzt sollte ein gemeinsamer Dateizugriff asynchron und blockierungsfrei durchgeführt werden können.

3.1 MPI I/O

MPI I/O ist eine Bibliothek, die auf die Basisfunktionen der MPI Bibliotheken zurückgreift. Mit einer festgelegten Zahl von Zugriffsmethoden sorgt MPI I/O als High-Level-Interface für eine flexible und mächtige Schnittstelle zum parallelen I/O. Die Hauptmerkmale von MPI I/O sind:

- Partitionierung der Daten einer Datei für verschiedene Prozesse
- Ein Interface zum Transfer von Dateien und globalen Datenstrukturen zwischen Prozessspeichern und Dateien
- Ermöglichung von asynchronem I/O und gemeinsamen parallelen Dateizugriff mehrerer Prozesse
- Kontrolle über physikalisches File-Layout auf Speichermedien

Im Gegensatz zur Synchronisierung in parallelen Systemen (broadcast, reduction, scatter, gather), verwendet MPI eigene zusammengesetzte Datentypen zur Speicherung.

4 Begriffe

MPI definiert verschiedene Begriffe, die im Zusammenhang mit I/O benutzt werden. Sie bezeichnen die Basiselemente des Interfaces.

4.1 File

Eine File (Datei) im Sinn von MPI ist eine geordnete Sammlung von Daten, die von einer Gruppe von Prozessen benutzt wird. Dabei benutzen alle Prozesse der Gruppe die File zusammen. Die File wird daher von allen Prozessen geöffnet und alle Prozesse führen gemeinsam I/O Operationen auf ihr aus. Der Zugriff kann sequentiell oder wahlfrei sein.

4.2 Displacement

Ist die Adresse, an dem eine View beginnt. Sie wird in der absoluten Anzahl von Bytes vom Beginn einer Datei angegeben.

4.3 Etype

Ein Etype ist ein elementarer Datentyp in MPI I/O. Etypes bestehen entweder aus elementaren MPI Datentypen oder aus zusammengesetzten Datentypen. Der Datenzugriff, lesend und schreibend, erfolgt immer über den Zugriff auf Etypes. Ein Etype beschreibt abhängig vom Kontext:

- Einen expliziten MPI-Datentyp
- Die Daten dieses Datentypes
- Die Größe des Datentyps

Ein Etype ist immer nur für einen Prozess lesbar. Prozesse in der gleichen Gruppe, die ebenfalls auf dieselbe File zugreifen, sehen die Etypes der anderen Prozesse nur als sogenannte Holes, vom Ausmaß eines Etypes.

4.4 Filetype

Der Filetype ist die Basis der gemeinsamen Benutzung einer Datei, er partitioniert die Daten einer File für die einzelnen Prozesse, bildet also Datei-Abschnitte, die nach einem festgelegten Muster von den unterschiedlichen Prozessen genutzt werden.

4.5 View

Unter View versteht man die Sicht eines Prozesses auf die gerade geöffnete und benutzte Datei. Sie besteht aus:

- Displacement
- Etype/s
- Filetype

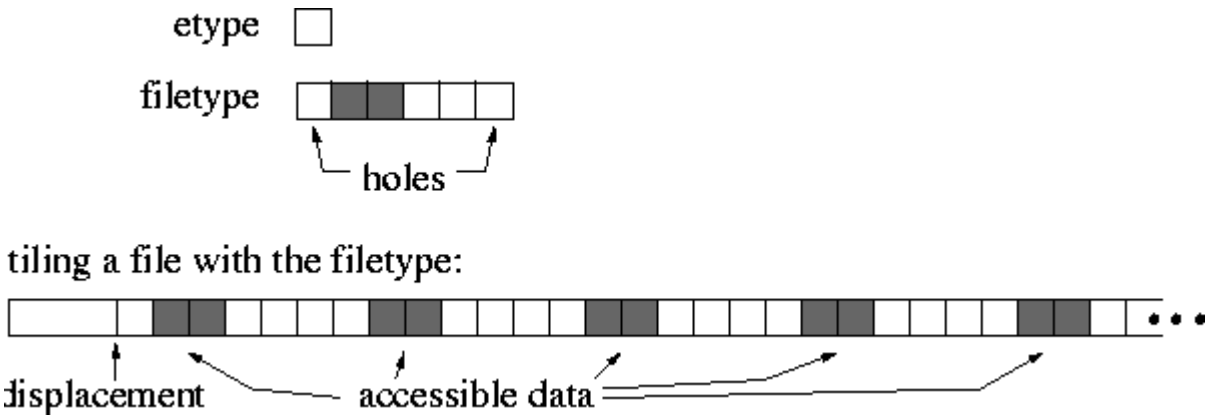

Abbildung 1: etype, filetype, view.

Quelle: MPI 2 Forum: MPI-2: Extensions to the Message-Passing Interface (Literaturverzeichnis [3])

Die Grafik (Abbildung 1) verdeutlicht die View eines Prozesses auf eine geöffnete Datei. Als Basiselement der Datei steht der Etype. Die Datei ist in atomare Datenelemente vom Typ Etype partitioniert.

Der Filetype ist eine Folge von Etypes; eine Sequenz von Daten, die wiederholt in der File vorkommt. Sie ist quasi der Etype der File und beschreibt ein atomares Datenobjekt einer View. Jeder Prozess kann aber nur auf seine ihm zugewiesenen Etypes zugreifen, alle Etypes der anderen Prozesse sind nicht verfügbar. Jeder Prozess hat seine eigene Sicht auf die Filetypes der File.

Die View eines Prozesses beginnt an der Adresse mit dem Wert des Displacements, der absoluten Adresse vom Dateianfang. Sie wird in verschiedenen Filetypes gleichen Aufbaus unterteilt. Die Grenzen der unterschiedlichen Filetypes sind allen Prozessen bekannt. Innerhalb der Filetypes sind die benutzbaren Daten eines Prozesses immer an der gleichen Stelle zu finden. Alle Daten anderer Prozesse werden als Holes dargestellt. Views können während der Ausführung eines Programms geändert werden.

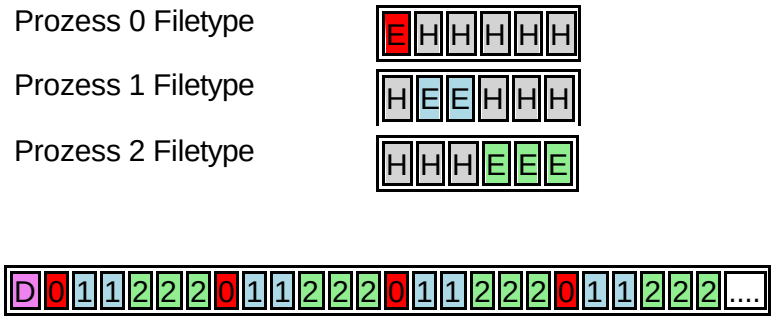

Abbildung 2: Partitionierung einer File

Die Aufteilung einer File kann man sich wie in Abbildung 2 dargestellt vorstellen. In diesem Beispiel greifen 3 Prozesse auf die geöffnete Datei zu. Während Prozess 0 nur einen Etype pro Filetype verwendet, benutz Prozess 1 - 2 Etypes und Prozess 2 - 3 Etypes.

Wird kein View zum Programmstart angegeben, verwendet MPI eine Default-View. Es handelt sich hierbei um einen linearen Byte-Stream, mit Displacement 0 und Etype = Filetype = MPI_Byte.

4.6 Offset

Offset ist die Anzahl von Etypes, die in einer View sichtbar sind. Offset 0 würde den ersten Etype, Offset n den n-ten bezeichnen. Holes werden bei der Berechnung des Offsets nicht berücksichtigt.

4.7 Filesize

In MPI wird die Größe einer File in Bytes vom Anfang der Datei (erstes Byte) errechnet. Eine neue File hat die Größe 0.

4.8 EndofFile

End-of-File (EoF) ist die Byte-Position, die für alle Views einer File das Ende der Datei bedeutet. EoF ist definiert als der Offset des ersten verfügbaren Etypes einer View, hinter dem letzten Byte der Datei.

4.9 Filepointer

Filepointer sind Zeiger auf konkrete Offsets einer geöffneten MPI-File. Die MPI Laufzeitumgebung verwaltet die Filepointer. MPI unterscheidet zwischen Individual und Shared Filepointern. Individual ist ein Filepointer, wenn er nur von einem Prozess, Shared, wenn er von einer Gruppe von Prozessen genutzt wird.

4.10 Filehandle

Wird eine MPI-File geöffnet (MPI_FILE_OPEN), so erstellt die MPI-Laufzeitumgebung ein Filehandle Objekt. Es ist völlig transparent für alle Prozesse und wird vollständig von der Laufzeitumgebung verwaltet. Alle Zugriffe auf eine File werden über den Filehandle realisiert. Beim Schließen einer Datei (MPI_FILE_CLOSE) wird das Filehandle-Objekt dereferenziert.

5 File Manipulation

Neben den üblichen Operationen wie Öffnen, Schließen und Löschen einer Datei stehen in MPI auch Funktionen wie resizing und space preallocating zur Verfügung. Außerdem können Informationen über die Datei abgefragt werden wie die Größe und die File-Parameter der Datei. Es kann auch ein File-Info-Objekt abgefragt werden.

5.1 Öffnen einer Datei

MPI_FILE_OPEN(comm, filename, amode, info, fh)

IN comm	communicator (handle)
IN filename	name of file to open (string)
IN amode	file access mode (integer)
IN info	info object (handle)
OUT fh	new file handle (handle)

Eine Datei wird in MPI durch Ihren Dateiname (ein String) repräsentiert. Beim Öffnen der Datei kann eine Gruppe von Prozessen über die Referenz auf den gemeinsamen Kommunikator, auf die Datei, zugreifen.

Als Rückgabewert der Funktion MPI_FILE_OPEN, wird eine Referenz auf den neuen generischen Filehandle an das Programm gegeben. Über ihn werden alle zukünftigen Zugriffe bis zum Schließen der Datei durchgeführt.

Da das Öffnen einer Datei eine kollektive Aktion ist, müssen die Werte aller Prozesse für den Dateinamen und die Zugriffsart(access mode) gleich sein. Der Kommunikator muss ein Intrakommunikator sein, also einer, der eine Gruppe von Prozessen verbindet.

Ein einzelner Prozess kann unabhängig von anderen Prozessen durch MPI_COMM_SELF, seinen eigenen Kommunikator eine Datei, öffnen. Der zurückgegebene Filehandle kann nachträglich genutzt werden, um auf die Datei zuzugreifen, bis diese geschlossen wird.

5.1.1 Access Modes

MPI sieht die folgenden Zugriffsmethoden vor:

• MPI_MODE_RDONLY	lesend
• MPI_MODE_RDWR	lesend und schreibend
• MPI_MODE_WRONLY	nur schreibend
• MPI_MODE_CREATE	erstellt Datei, wenn noch nicht vorhanden
• MPI_MODE_EXCL	Fehler beim erstellen einer schon vorhandenen Datei
• MPI_MODE_DELETE_ON_CLOSE	löscht Datei beim Schließen
• MPI_MODE_UNIQUE_OPEN	Datei kann nirgendwo nochmals geöffnet werden
• MPI_MODE_SEQUENTIAL	Datei wird nur sequentiell benutzt
• MPI_MODE_APPEND	setzt den Initialen Wert aller Filepointer auf das Ende der Datei

5.1.2 Beachtenswert

Problematisch kann der Dateiname auf heterogenen Systemen sein, da hier unterschiedliche Dateisperatoren verwendet werden könnten.

Der Benutzer hat sicherzustellen, dass vor Beenden der Laufzeitumgebung (mit MPI_FINALIZE) alle Dateien mit MPI_CLOSE geschlossen werden.

5.2 Schließen einer Datei

MPI_FILE_CLOSE(fh)
INOUT fh file handle (handle)

Eine Datei wird durch den Aufruf MPI_FILE_CLOSE geschlossen, indem die Datei zuerst synchronisiert und anschließend über den mit der Datei assoziierte Filehandle geschlossen wird. Anschließend wird der Filehandle freigegeben und auf MPI_FILE_NULL gesetzt.

Der Benutzer ist dafür verantwortlich, dass alle nichtblockierenden Aufrufe vor dem Schließen der Datei beendet sind.

5.3 Löschen einer Datei

MPI_FILE_DELETE(filename, info)
IN filename name of file to delete (string)
IN info info object (handle)

Eine Datei wird gelöscht, indem MPI_FILE_DELETE aufgerufen wird. Der Dateiname und ein optionales Infoobjekt können übergeben werden. Es ist implementierungsabhängig, wie sich das System verhält, wenn zu löschende Dateien noch geöffnet sind, oder noch Aktionen auf den Dateien ausgeführt werden.

5.4 Größenänderung einer Datei

MPI_FILE_SET_SIZE(fh, size)
INOUT fh file handle (handle)
IN size size to truncate or expand file (integer)

Die Größenänderung einer Datei ist ein kollektiver Zugriff. Alle Prozesse müssen die gleiche Größe der Datei übermitteln.

Die Größe einer Datei wird über den Filehandle abgefragt und gesetzt. Sie wird in Bytes vom Dateianfang gemessen. Wird eine Größe gesetzt, die kleiner als die aktuelle Größe der Datei ist, wird die Datei an dieser Stelle abgeschnitten. Was mit den frei werdenden Bytes passiert ist abhängig von der Implementierung. Wird ein Wert größer als der aktuelle gewählt, wird die Datei vergrößert.

Von der Vergrößerung unbeeinflusst bleiben alle Filepointer und die bisher geschriebenen Daten. Vor der Größenänderung müssen alle nichtblockierenden Aufrufe abgeschlossen sein.

Der Standard weist darauf hin, dass das Ändern der Dateigröße prinzipiell fehleranfällig ist, da es zu Konflikten bei Operationen kommen kann, wenn Daten referenziert werden, deren Displacement zwischen der neuen und alten Größe liegt.

5.5 Speicher reservieren

MPI_FILE_PREALLOCATE(fh, size)
INOUT fh file handle (handle)

IN size size to preallocate file (integer)

Die Allokation von Speicherplatz vor dem Speichern einer Datei stellt sicher, dass auch tatsächlich genügend Platz auf dem Speichermedium vorhanden ist. Alle Prozesse müssen den gleichen Speicherplatz reservieren. Wurde nicht genügend Speicherplatz reserviert, wird beim Abspeichern der Datei die tatsächliche Größe auf die aktuelle Größe verändert.

5.6 Information über eine Datei abfragen

MPI_FILE_GET_SIZE(fh, size)
IN fh file handle (handle)
OUT size size of the file in bytes (integer)

MPI_FILE_GET_AMODE(fh, amode)
IN fh file handle (handle)
OUT amode file access mode used to open the file (integer)

MPI_FILE_GET_GROUP(fh, group)
IN fh file handle (handle)
OUT group group which opened the file (handle)

Über den Filehandle könne Eigenschaften wie die Größe, die Zugriffsart und die beteiligten Kommunikatoren abgerufen werden. Die Funktionen geben jeweils die Wert der Größe bzw. die Zugriffsmethode als Integer bzw. eine Kopie der Kommunikatorgruppe zurück.

MPI_FILE_SET_INFO(fh, info)
INOUT fh file handle (handle)
IN info info object (handle)

MPI_FILE_GET_INFO(fh, info_used)
IN fh file handle (handle)
OUT info_used new info object (handle)

Weitergehende Informationen zu einer Datei können in einem speziellen Info-Objekt an die Datei übergeben werden. In ihm können spezielle Zugriffsmuster und spezifische Eigenschaften des Dateisystems festgeschrieben sein. Der Nachteil an den Info-Objekten ist, dass sich Implementierungen nicht zwingend um ihren Inhalt kümmern müssen; sie können ignoriert werden.

MPI reserviert File Hints. Das sind Schlüsselwörter-Wert-Paare, die zur Beschreibung von Dateieigenschaften in Info-Objekten genutzt werden können. Eine konkrete Implementierung muss diese Schlüsselworte nicht interpretieren, es muss aber die beschriebene Funktionalität unterstützen.

Beispiele für File Hints sind: access_style, collective_buffering, cb_block_size, filename.

6 File View

6.1 Setzen einer File View

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

INOUT fh	file handle (handle)
IN disp	displacement (integer)
IN etype	elementary datatype (handle)
IN filetype	filetype (handle)
IN datarep	data representation (string)
IN info	info object (handle)

Um die View eines Prozesses zu verändern, wird die Methode MPI_FILE_SET_VIEW aufgerufen. Der Beginn eines View wird auf den Wert des displacemets gesetzt. Die Filepointer werden auf den Wert 0 gesetzt. Da das Setzen einer View eine kollektive Aktion ist, müssen die Werte für die Größe der Etypes und Datenrepräsentation für alle Prozesse gleich sein. Es ist sicherzustellen, dass die verschiedenen Views eines Prozesses sich nicht überschneiden.

6.2 Erhalten einer View

MPI_FILE_GET_VIEW(fh, disp, etype, filetype, datarep)

IN fh	file handle (handle)
OUT disp	displacement (integer)
OUT etype	elementary datatype (handle)
OUT filetype	filetype (handle)
OUT datarep	data representation (string)

Die View eines Prozesses kann mit MPI_FILE_GET_VIEW über den Filehandle erfragt werden.

7 Datenzugriff

In MPI werden Daten zwischen Prozessen und Dateien durch Read- und Write-Methoden ausgetauscht. Während Read-Methoden die Daten über den Filehandle aus einer Datei in den Speicher eines Prozesses schreiben, schreiben Write-Methoden die Daten aus dem Speicher in eine Datei. Alle Daten eines Datentyps, die zwischen Puffer und Datei ausgetauscht werden, werden gezählt. So soll dem Verlust von Daten vorgebeugt werden. Die Zugriffsmethoden geben nur solche Datentypen zurück, die auch von ihnen angefordert wurden.

Im Speicher des Prozesses werden die Daten über das Tripple: Buffer, Größe, Datentype charakterisiert, in einer Datei über ihren Offset.

Nach Beendigung des Dateizugriffs durch eine MPI Zugriffsmethode wird immer der Status zurückgegeben.

Der Dateizugriff in MPI erfolgt unter drei Aspekten:

- Position der gewünschten Daten
- Synchronisation der Daten
- Koordination

Die folgende Tabelle bietet einen Überblick über die im Standard vorgesehenen Zugriffsmethoden und deren Eigenschaften.

positioning	**synchronism**	**2c| coordination**	
3-4		***noncollective***	***collective***
explicit	*blocking*	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
offsets		MPI_FILE_WRITE_AT	MPI_FILE_WRITE_AT_ALL
2-4	*nonblocking &*	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN
	split collective		MPI_FILE_READ_AT_ALL_END
		MPI_FILE_IWRITE_AT	MPI_FILE_WRITE_AT_ALL_BEGIN
			MPI_FILE_WRITE_AT_ALL_END
individual	*blocking*	MPI_FILE_READ	MPI_FILE_READ_ALL
file pointers		MPI_FILE_WRITE	MPI_FILE_WRITE_ALL
2-4	*nonblocking &*	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN
	split collective		MPI_FILE_READ_ALL_END
		MPI_FILE_IWRITE	MPI_FILE_WRITE_ALL_BEGIN
			MPI_FILE_WRITE_ALL_END
shared	*blocking*	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
file pointer		MPI_FILE_WRITE_SHARED	MPI_FILE_WRITE_ORDERED
2-4	*nonblocking &*	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN
	split collective		MPI_FILE_READ_ORDERED_END
		MPI_FILE_IWRITE_SHARED	MPI_FILE_WRITE_ORDERED_BEGIN
			MPI_FILE_WRITE_ORDERED_END

Abbildung 3: Überblick der Datenzugriffsmethoden

Quelle: MPI 2 Forum: MPI-2: Extensions to the Message-Passing Interface (Literaturverzeichnis [3])

7.1 Positionierung

Die Position der Daten in MPI Dateien kann mit drei Methoden bestimmt werden:

- Explizit - durch Angabe eines Offsets
- Implizit - durch die Verwendung von Individual Filepointern
- Implizit - durch die Verwendung von Shared Filepointern

Alle Methoden können in einem Programm kombiniert eingesetzt werden und behindern sich nicht gegenseitig. File Pointer werden immer von der Laufzeitumgebung aktualisiert, bevor der aufrufende Prozess die Aktion abschließt. Der aktualisierte Wert des Filepointers ist in der Regel das dem zuletzt genutzten Datum folgende Datenobjekt.

7.1.1 Datenzugriff mit Explizitem Offset

Methoden mit explizitem Offset haben die Endung _AT im Namen. Sie verwenden direkte Speicheradressen als Argument zum Auffinden der Daten. Es müssen keine Filepointer bearbeitet, benutzt oder aktualisiert werden. Folgende Methoden stehen zur Verfügung:

- MPI_FILE_READ_AT blockierendes Lesen
- MPI_FILE_READ_AT_ALL blockierendes kollektives Lesen
- MPI_FILE_WRITE_AT blockierendes Schreiben

- MPI_FILE_IREAD_AT nicht blockierendes Lesen
- MPI_FILE_IWRITE_AT nicht blockierendes Schreiben

7.1.2 Datenzugriff mit Individual Filepointer

Für jeden Prozess und jeden Filehandle verwaltet MPI I/O einen Individual Filepointer. Die Datenzugriffsmethoden in MPI benutzen und aktualisieren den Filepointer. Die Berechnung der Adressen, auf die der Pointer zeigt, übernimmt die MPI-Laufzeitumgebung. Nach der Benutzung eines Individual Pointers zeigt er auf den Nachfolger des zuletzt benutzten Etype. Alle Methoden entsprechen denen des Expliziten Offsets, ihnen wird anstelle der direkten Speicheradresse der Fileponter übergeben.

Blockierendes Lesen einer Datei:

MPI_FILE_READ(fh, buf, count, datatype, status)
INOUT fh file handle (handle)
OUT buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

Kollektivers blockierendes Lesen einer Datei:

MPI_FILE_READ_ALL(fh, buf, count, datatype, status)
INOUT fh file handle (handle)
OUT buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

Blockierendes Schreiben einer Datei:

MPI_FILE_WRITE(fh, buf, count, datatype, status)
INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

Blockierendes kollektives Schreiben einer Datei

MPI_FILE_WRITE_ALL(fh, buf, count, datatype, status)
INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT status status object (Status)

Nichtblockierendes MPI_FILE_WRITE:

MPI_FILE_WRITE(fh, buf, count, datatype, request)
INOUT fh file handle (handle)
IN buf initial address of buffer (choice)
IN count number of elements in buffer (integer)
IN datatype datatype of each buffer element (handle)
OUT request request object (handle)

Neben den Lese- und Schreibmethoden sind Methoden zur Aktualisierung der Individual Filepointer enthalten:

- MPI_FILE_SEEK Setzt den Pointer auf eine bestimmte Position
- MPI_FILE_GET_POSITION Gibt den Offset des Pointers zurück
- MPI_FILE_GET_BYTE_OFFSET Gibt den relativen Offset einer View als absolu ten Byte-Offset zurück

7.1.3 Datenzugriff mit Shared Filepointer

Shared Filepointers sind Zeiger, die zur gemeinsamen Nutzung einer Datei einer Gruppe von Prozessen in einer Kommunikationsgruppe zur Verfügung stehen. Pro Kommunikationsgruppe existiert genau ein Filepointer. Der Wert des Filepointers wird nur von den nachfolgenden Methoden benutzt und aktualisiert. Dazu werden keine Individual Filepointers benutzt oder aktualisiert.

Die Zugriffsmethoden der Shared Filepointers entsprechen denen des Zugriffs mit Explizitem Offset. Mit einigen Ausnahmen:

- Statt des Offsets wird der von MPI-System verwaltete Shared Filepointer verwendet.
- Gleichzeitige Aufrufe des Shared Filepointers werden behandelt, als ob sie serialisierte Aufrufe wären
- Die Benutzung des Shared Filepointers ist fehleranfällig, solange nicht alle Prozesse die gleiche View haben
- Nichtkollektive Zugriffe haben kein definiertes Ergebnis – der Nutzer muss für Synchronisation der Daten sorgen

7.1.3.1 Nichtkollektive Operationen

MPI_FILE_READ_SHARED Lesender Dateizugriff
MPI_FILE_WRITE_SHARED Schreibender Dateizugriff
MPI_FILE_IREAD_SHARED Nicht blockierender lesender Dateizugriff
MPI_FILE_IWRITE_SHARED Nicht blockierender schreibender Dateizugriff

7.1.3.2 Kollektive Operationen

Der gemeinsame Zugriff auf eine Datei wird über die Ranks (Wertigkeit) der Prozesse innerhalb der Prozessgruppe geregelt. Ein Prozess kann erst auf Daten zugreifen, wenn alle niederwertigen Prozesse ihre Datenzugriffe abgeschlossen haben. Nach Abschluss des Zugriffs zeigt der Filepointer auf das nächste verfügbare Etype einer View, das nach dem zuletzt verwendeten liegt. Folgende Methoden sind vorgesehen:

MPI_FILE_READ_ORDERED	Lesender Dateizugriff
MPI_FILE_WRITE_ORDERED	Schreibender Dateizugriff

7.1.3.3 Positionssuche

Folgende Methoden stehen zur Bestimmung des Offsets eines Shared Filepointers zur Verfügung:

MPI_FILE_SEEK_SHARED	Setzt den Pointer auf eine bestimmte Position
MPI_FILE_GET_POSITION_SHARED	Gibt den Offset des Pointers zurück

7.2 Synchronisation

MPI unterstützt blockierende und nichtblockierende I/O Routinen. Blockierende Aufrufe verhindern den weiteren Programmablauf, solange bis sie vollständig abgearbeitet sind. Nichtblockierende Funktionen blockieren die Datei nicht. Man erkennt sie am Dateinamen mit dem Prefix MPI_FILE_IXXX.

Nichtblockierende Aufrufe warten nicht auf die vollständige Abarbeitung der Methoden. In der Regel sind nichtblockierende Zugriffe nicht als kollektive Zugriffe realisiert. Es muss durch den Programmierer sichergestellt werden, dass keine Inkonsistenzen entstehen, notfalls durch explizite Funktionsaufrufe. Lokale Puffer sollten nicht als Quelle oder Ziel zwischen Initiierung und Abschluss einer nichtblockierenden Dateioperation benutzt werden, da ihre Verwendung fehleranfällig ist.

7.2.1 Split Collective Routines

Nichtblockierende kollektive I/O Funktionen werden eingeschränkt durch Split Collective Routines bereitgestellt. Diese teilen den Dateizugriff in eine BEGIN-Methode und in eine END-Methode mit den folgenden Eigenschaften:

- Es darf höchstens eine Split Collective Routine auf einem Filehandle ausgeführt werden
- BEGIN-Methoden sind kollektiv und öffnen die Datei gemeinsam für alle Prozesse
- Die Zugriffsreihenfolge für die Prozesse einer Gruppe ist wie bei blockierenden Aufrufen
- END-Methoden sind kollektiv und beenden genau die BEGIN-Methode, die ihre Partnermethode ist
- Zur Implementierung können die blockierenden kollektiven Methoden verwendet werden
- Split Collective Routines sind nicht gleich den blockierenden kollektiven Methoden
- Split Collective Routines müssen einen Buffer angeben um Datenverluste zu vermeiden
- Kollektive I/O-Operationen dürfen zur gleichen Zeit auf einem Filehandle ausgeführt werden, wenn Split Collective Routines ausgeführt werden.
- Eine Multithreadimplementierung muss sicherstellen, dass die BEGIN- und END-Methode vom gleichen Thread ausgeführt werden.

Folgende Methoden sind vorgesehen:

MPI_FILE_READ_AT_ALL_BEGIN
MPI_FILE_READ_AT_ALL_END
MPI_FILE_WRITE_AT_ALL_BEGIN
MPI_FILE_WRITE_AT_ALL_END

MPI_FILE_READ_ALL_BEGIN
MPI_FILE_READ_ALL_END
MPI_FILE_WRITE_ALL_BEGIN
MPI_FILE_WRITE_ALL_END
MPI_FILE_READ_ORDERED_BEGIN
MPI_FILE_READ_ORDERED_END
MPI_FILE_WRITE_ORDERED_BEGIN
MPI_FILE_WRITE_ORDERED_END

8 Interoperabilität

MPI I/O stellt sicher, dass ein vollständig kompatibler und transparenter Datenaustausch gewährleistet ist. Egal welcher MPI-Prozess eine Datei auf welchem Dateisystem, Betriebsystem oder unter welcher MPI-Laufzeitumgebung gespeichert hat, ein anderer MPI-Prozess in einer anders implementierten MPI-Laufzeitumgebung kann auf diese Dateien zugreifen.

Drei Implementierungsaspekte sind besonders wichtig:

- Übertragung der Bits
- Konvertierung zwischen verschiedenen Dateistrukturen
- Konvertierung zwischen verschiedenen Maschinen-Repräsentationen

8.1 Datenrepräsentationen

Folgende Datenrepräsentationen werden unterstützt:

- Native
- Internal
- External32
- Benutzerdefinierte Datenrepräsentation

MPI stellt nicht sicher, dass die Datenrepräsentation einer Datei passend zur Anwendung gewählt wird, vielmehr muss die Anwendung selber auf die Verwendung des geeigneten Repräsentationstyps beim Erstellen einer Datei achten.

8.1.1 Native Datenrepräsentation

„Native“ speichert die Daten in einer Datei so wie sie im Speicher vorgehalten werden. Eine Hohe I/O-Performance ist der Vorteil. Der Nachteil ist die Systemabhängigkeit und der Verlust flexibler Interoperabilität in heterogenen Systemen. Daher sollte dieser Typ nur in homogenen Systemen Verwendung finden.

8.1.2 Internal Datenrepräsentation

Diese Implementierung kann, wenn notwendig, die Daten in das Internal Format konvertieren und ist daher für homo- und heterogene Systeme geeignet. Die Daten können in jedem Format gespeichert werden, das die Implementierung verlangt. Die Implementierung muss allerdings sicherstellen, dass die Widerverwertbarkeit der Datei gewährleistet und dokumentiert ist.

8.1.3 External32 Datenrepräsentation

Alle Daten werden hier in das Format External32 umgewandelt. Das External32 Format definiert MPI-Datengrundtypen (MPI_BYTE; MPI_FLOAT;…) und einige optionale Datentypen fester Länge. Die Fließkommazahlen sind im IEEE Big-Endian-Format.

Der Vorteil der Verwendung dieser Repräsentation ist der hohe Grad an Abstraktion von konkreten Implementierungen. Es handelt sich bei den verwendeten Datentypen um Standardtypen, die in jedem Fall interpretierbar in plattformunabhängig sind. Sie sind daher uneingeschränkt portierbar und von jeder Implementierung verwendbar.

Der Nachteil ist ein möglicher Genauigkeitsverlust (z.B. kleinere Speicherbereiche für Floating Point Zahlen) und eine schlechtere I/O Performance.

8.2 Datentypen

Da von den verschiedenen Repräsentationsklassen unterschiedliche Datentypen verwendet werden können, kann die Funktion MPI_FILE_GET_TYPE_EXTENT genutzt werden um die Größe der verwendeten Datentypen zu ermitteln.

Wenn ein Benutzer schreibend auf eine Datei zugreifen möchte, deren Repräsentation in der aktuellen Implementierung unbekannt ist, oder er möchte eine Datei lesen, die in einer unbekannten Repräsentation geschrieben wurde, ist dies von den vorgesehenen Datenrepräsentationen nicht abgedeckt.

Mit der Funktion MPI_REGISTER_DATAREP kann eine benutzerdefinierte Datenrepräsentation an der Laufzeitumgebung angemeldet werden. Ihr können Konvertierungsfilter mitgegeben werden, die eine korrekte Nutzung sicherstellen.

8.3 Kompartibilität

Der Benutzer ist für die Kompatibilität der gespeicherten Daten verantwortlich. Die Daten, die von einer Datenrepräsentation gelesen werden, müssen kompatibel zu denen sein, die geschrieben werden.

Nur wenn unterschiedliche Namen für unterschiedliche Datenrepräsentationen gewählt werden, ist eine Kompatibilität sichergestellt. Es wird empfohlen, die Repräsentation External32 zu verwenden, da diese implementierungsunabhängig auf allen MPI-Systemen vorhanden ist. Benutzerdefinierte Datentypen können eine Kompatibilität bei Native oder Internal Repräsentation sicherstellen.

9 Konsistenz

Die Referenzierung einer Datei über einen Filehandle, der bei einer gemeinsamen Öffnen-Operation von der Laufzeitumgebung erzeugt wird, sorgt für Datenkonsistenz in MPI. Alle Dateioperationen laufen über diesen Filehandle, der alle Zugriffsadressen für die Datei berechnet und somit Inkonsistenzen direkter Adressierung vorbeugt. Drei Level der Konsistenz werde unterstützt:

- Sequentiell zwischen allen Zugriffen über einen Filehandle
- Sequentiell zwischen allen Zugriffen über Filehandles, die durch eine kollektive Öffne-Operation im „atomic mode“ erzeugt wurden
- Benutzerverwaltete Konsistenz

Unter sequentieller Konsistenz versteht man in MPI, das eine Befehlsfolge so nacheinander abgearbeitet wird, wie sie vom Programmablauf erforderlich ist. Im „atomic mode“, der mit der Funktion MPI_FILE_SET_ATOMICITY, gesetzt wird, werden alle Operationen grundsätzlich seriell abgearbeitet. Bei der benutzerverwalteten Konsistenz sorgt der Anwender selbst für die korrekte Reihenfolge der Befehlsausführung und die Synchronisation der Datei.

Auf Dateien kann sequentiell oder wahlfrei zugegriffen werden. Sequentieller Zugriff ist sinnvoll bei Datenträgern wie Magnetbändern. Zu beachten ist, dass bei sequentiellen Zugriffen manche Funktionen wie der Größenänderung die Datei auf einem Band gelöscht werden muss, um anschließend in neuer Größe wieder geschrieben zu werden.

Wenn eine Dateioperation komplett abgeschlossen wurde, ist sie sicher. Alle Objekte, die der Methode übergeben wurden, sollten dereferenziert werden. Datenpuffer bei nichtblockierenden Funktionen sollten erst wieder verwendet werden, wenn deren Bearbeitung abgeschlossen ist.

MPI gewährleistet nur die logische Sicht auf Dateien, nicht die physikalische. Wie die Daten endgültig gespeichert werden, ist Teil der Implementierung und abhängig von den eingesetzten Systemen.

Nicht alle Operationen, die die Dateigröße ändern, haben tatsächlich eine Größenänderung zur Folge. Wird zum Beispiel mit MPI_FILE_PREALLOCATE mit einer zu geringen Größe aufgerufen, hat das keine Auswirkung auf die Datei.

10 I/O Fehlerbehandlung

Mit jedem Filehandle wird in MPI ein Errorhandler assoziiert. Beim Erstellen eines Filehandels wird mit ihm ein „default file error handler“ erstellt. Falls ein Fehler während der Ausführung von I/O-Operationen auftritt, wird dieser als Fehlercode zusammen mit dem Filehandle an den Errorhandler übergeben.

Folgende Fehlerklassen sind vorgesehen, in die alle auftretenden Fehler einsortiert werden sollten:

MPI_ERR_FILE	Invalid file handle
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file which supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to

	MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function.
MPI_ERR_IO	Other I/O error

Abbildung 4: Fehlerklassen in MPI I/O

Quelle: MPI 2 Forum: MPI-2: Extensions to the Message-Passing Interface (Literaturverzeichnis [3])

11 Resümee

MPI 2 enthält ein umfassendes Regelwerk, das paralleles I/O auf hoher Abstraktionsstufe sicherstellt. Durch eine überschaubare Zahl von Funktionen, die alle Dateioperationen abdecken, ist ein schmales Interface sichergestellt, das auch implementierbar ist.

Die Qualität der Implementierung hängt stark von der Umsetzung ab. Viele Realisierungen obliegen den Entwicklern einer MPI-Bibliothek und liegen in ihrer Verantwortlichkeit.

Definierte Funktionalität wird in MPI sichergestellt. Durch die optionale Umsetzung einiger Funktionen sind sowohl kleinere MPI Realisierungen möglich, als auch umfangreiche I/O Bibliotheken.

Die Möglichkeit, benutzerdefinierte Datentypen zu realisieren, schafft Flexibilität in der Entwicklung speziell angepasster Implementierungen für eigene Anwendungen.

Anhang A

A.1 Literaturverzeichnis

[1] Greiml, Alexander:
Message Passing Interface,
http://www.syssoft.uni-trier.de/systemsoftware/Download/Seminare/Middleware/middleware.3.book.html, 2001

[2] Manfred Eiling:
Programmierung mit dem Message Passing Interface, IX 3/97, Seite 158 ff, Hannover 1997

[3] MPI 2 Forum:
MPI-2: Extensions to the Message-Passing Interface,
http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm#Node0, 1997

[4] N.N.:
N.N, Chapert Five – Low-Level I/O Interfaces, S. 156 ff.
(Literatur zur Seminararbeit lag als Kopie ohne Titel und Autor vor)

[5] National Energy Research Scientific Computing Center, N.N:
Introducing to MPI I/O,
http://hpcf.nersc.gov/software/libs/io/mpiio.html, 2001

[6] Rainer Kowallik:
Standardschnittstelle für nachrichtengesteuerte Parallelisierung auf Multiprozessorsystemen, IX 7/95, Seite 138 ff, Hannover 1995

A.2 Abbildungsverzeichnis

Abbildung Seite