MPI-IO

Hochleistungs-Ein-/Ausgabe

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Michael Kuhn

2019-04-30

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

MPI-IO

- Orientierung
- Einführung
- Konzepte und Funktionalität
- Leistungsbetrachtungen
- Zusammenfassung

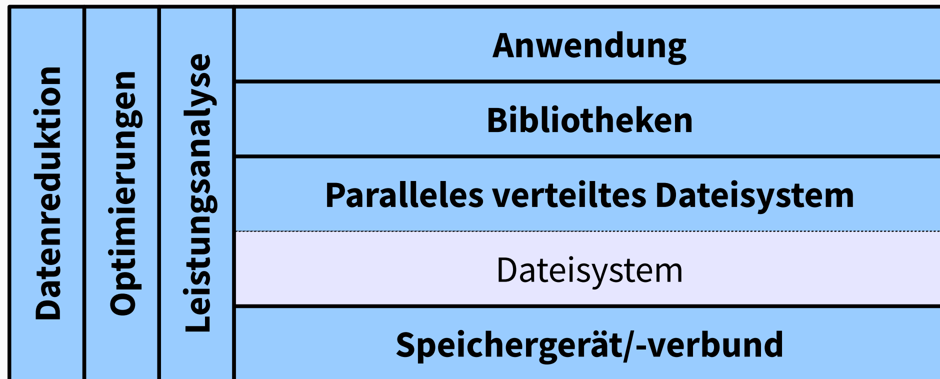


Abbildung 1: E/A-Schichten und orthogonale Themen

Überblick

- Parallele Anwendungen benötigen Unterstützung für effiziente E/A
  - Synchrone und serielle E/A sind Flaschenhälse
  - Normalerweise kann kein Prozess alle Daten halten
- Häufige Szenarien
  - Lesen der Eingabedaten
    - Startbedingungen, größere Datensätze
  - Schreiben von Ausgabedaten
    - Ergebnisdaten, Checkpoints

- Synchrone E/A führt dazu, dass alle Prozesse untätig warten, während die E/A stattfindet.
- Serielle E/A führt dazu, dass alle Daten zu einem ausgewählten Prozess gesendet werden müssen, der diese dann in das Dateisystem schreibt. Das Problem verschlimmert sich durch die immer weiter steigenden Prozesszahlen. Kann der E/A-Prozess nicht alle Daten auf einmal im Speicher halten, muss dieser Vorgang mehrfach wiederholt werden.

Überblick...

- MPI-IO bezeichnet den E/A-Teil von MPI
  - Wurde mit MPI 2.0 eingeführt (1997)
  - Parallele Anwendungen benutzen üblicherweise sowieso MPI
- Populärste Implementierung: ROMIO
  - Wird als Teil von MPICH entwickelt und vertrieben
  - Wird unter anderem in OpenMPI und MPICH-Derivaten genutzt
  - Nutzt das Abstract-Device Interface for I/O (ADIO)
- Alternative Implementierung: OMPIO in OpenMPI

Überblick...

- MPI-IO stellt element-orientierten Zugriff bereit
  - Im Gegensatz zum POSIX-Bytestrom
- Schnittstelle ist analog zum Nachrichtenaustausch definiert
  - Lesen und Schreiben wie Empfangen und Senden
  - Kollektive und nicht-blockierende Operationen
  - Abgeleitete Datentypen
- MPI-IO wird üblicherweise nicht direkt in Anwendungen genutzt
  - Indirekt durch höhere Schichten wie z. B. HDF5

Ausblick

- MPI-IO bildet die Basis vieler E/A-Bibliotheken
  - HDF und NetCDF nutzen MPI-IO für parallelen Zugriff auf gemeinsame Dateien
  - ADIOS unterstützt MPI-IO
- Effiziente Algorithmen und Implementierungen für parallele E/A
  - Müssen nicht in höheren Schichten implementiert werden
  - Bibliotheken können sich um ihre eigentliche Aufgabe kümmern

- Über die POSIX-Schnittstelle kann mit HDF/NetCDF nur serieller Zugriff realisiert werden.
- ADIOS erlaubt auch mit dem POSIX-Backend parallelen Zugriff, nutzt dann aber keine gemeinsame Datei.

ROMIO

- MPI-IO abstrahiert vom darunter liegenden Dateisystem
  - Stellt MPI-IO-Syntax und -Semantik bereit
- Unterstützte Architekturen
  - IBM SP, Intel Paragon, HP Exemplar, SGI Origin2000, Cray T3E, NEC SX-4 etc.
- Unterstützte Dateisysteme
  - IBM PIOFS, Intel PFS, HP/Convex HFS, SGI XFS, NEC SFS, PVFS, Lustre, NFS, NTFS, Unix-Dateisysteme (UFS) etc.

ROMIO...

- MPI-IO-Schnittstelle für Anwendungen und Bibliotheken
  - Portabilität über viele Dateisysteme und Architekturen hinweg
- Dateisystem-spezifische Module in ADIO
  - Erlaubt höchstmögliche Leistung
    - Z. B. durch Zugriff auf Dateiverteilungsinformationen
  - Verbirgt unterschiedliche Syntax und Semantik
- Generische Optimierungen für parallele E/A
  - Insbesondere bei vielen Prozessen notwendig
  - Später: Data Sieving und Two-Phase I/O

Grundlagen

- Datei (file)
  - Kollektives Öffnen durch Prozesse im Kommunikator
  - Sequentieller oder wahlfreier Zugriff
  - Sammlung typisierter Daten (Elemente)
- Dateizeiger (file pointer)
  - Zeiger innerhalb der Datei
  - Individuelle oder gemeinsame Dateizeiger möglich

Grundlagen...

- Elementarer Typ (etype)
  - Einheit mit der auf die Datei zugegriffen wird
  - Kann auch ein abgeleiteter Datentyp sein
- Versatz (displacement)
  - Position an der die Dateisicht beginnt
  - Byte-Position relativ zum Anfang der Datei
    - Z. B. für Header

- Dateityp (file type)
    - Schablone für den Aufbau der Datei
    - Besteht aus elementaren Typen und Löchern
    - Wiederholt sich regelmäßig

Elementarer Typ

Loch

Dateityp

Aufbau

Abbildung 2: Dateityp

- Dateisicht (file view)
  - Prozessbezogene Sicht auf die Datei
  - Festgelegt durch Versatz, elementaren Typ und Dateityp

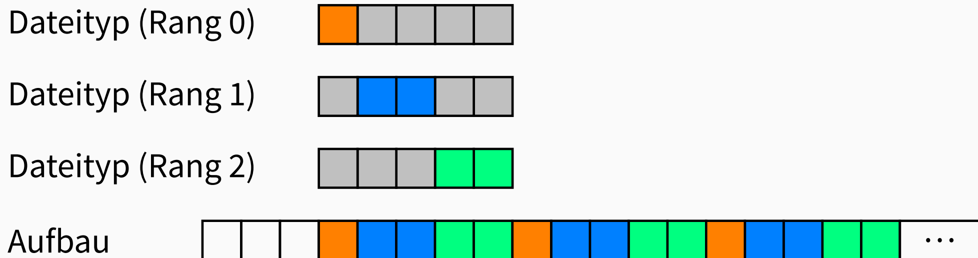


Abbildung 3: Dateisicht

Grundlagen...

- Versatz (offset)
  - Position in der Datei
  - Ausgedrückt in Anzahl elementarer Typen
  - Relativ zur aktuellen Dateisicht
- Dateigröße (file size)
  - Größe der Datei in Bytes

Grundlagen...

- Datei-Handle (file handle)
    - Analog zum Dateideskriptor
    - Wird für fast alle Operationen benötigt
- Hinweise (hints)
    - Zusätzliche Informationen für die Implementierung
    - Üblicherweise zur Leistungssteigerung

Funktionen

```c
int MPI_File_open (MPI_Comm comm, char* filename, int amode,
                   MPI_Info info, MPI_File* fh)
```

Listing 1: Öffnen einer Datei

- Liefert Datei-Handle zurück, das späteren Operationen übergeben wird

```c
int MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence)
```

Listing 2: Individuellen Dateizeiger setzen

- Setzt den Dateizeiger des aktuellen Prozesses (analog zu `lseek`)

```
1 int MPI_File_read (MPI_File fh, void* buf, int count,
2                    MPI_Datatype datatype, MPI_Status* status)
3 int MPI_File_write (MPI_File fh, void* buf, int count,
4                     MPI_Datatype datatype, MPI_Status* status)
```

Listing 3: Lesen und Schreiben von Daten

- Liest bzw. schreibt Datenelemente aus einer bzw. in eine Datei
- Ansonsten analog zu `read` und `write`

```
1 int MPI_File_close (MPI_File* fh)
```

Listing 4: Schließen der Datei

Code-Beispiel

```
1  MPI_File fh;
2  MPI_Offset size;
3  MPI_Status status;
4  int nbytes;
5
6  MPI_File_open(MPI_COMM_WORLD, "/tmp/mpi-io",
7                MPI_MODE_RDWR | MPI_MODE_CREATE | MPI_MODE_DELETE_ON_CLOSE,
8                MPI_INFO_NULL, &fh);
9  MPI_File_write(fh, data, sizeof(data), MPI_BYTE, &status);
10 MPI_Get_count(&status, MPI_BYTE, &nbytes);
11 MPI_File_get_size(fh, &size);
12 MPI_File_close(&fh);
```

- Grundsätzlicher Aufbau sehr ähnlich zu POSIX
  - Separater Aufruf für `MPI_Get_count` (Rückgabewert ist ein 32-Bit-Integer)
  - Metadatenzugriff mit `MPI_File_get_size` eingeschränkt

Erstellen und Öffnen von Dateien

- `MPI_File_open` ist eine kollektive Operation
  - Alle Prozesse müssen dieselbe Datei öffnen
  - Prozess-lokale Dateien durch `MPI_COMM_SELF`
- Dateiname ist implementierungsabhängig
  - Üblicherweise kann ADIO-Modul angegeben werden
  - Z. B. `pvfs2:/pvfs/path/to/file`
- Initiale Dateisicht ist ein Bytestrom
  - D. h. alle Prozesse haben Zugriff auf die gesamte Datei

Erstellen und Öffnen von Dateien...

- `MPI_File_open` bietet mehrere Zugriffsmodi
  - `MPI_MODE_RDONLY`: Nur lesen
  - `MPI_MODE_RDWR`: Lesen und schreiben
  - `MPI_MODE_WRONLY`: Nur schreiben
  - `MPI_MODE_CREATE`: Datei erstellen, wenn sie noch nicht existiert
  - `MPI_MODE_EXCL`: Fehler zurückgeben, wenn zu erstellende Datei bereits existiert
  - `MPI_MODE_DELETE_ON_CLOSE`: Datei beim Schließen löschen

Erstellen und Öffnen von Dateien...

- `MPI_File_open` bietet mehrere Zugriffsmodi...
  - `MPI_MODE_UNIQUE_OPEN`: Datei wird nicht parallel woanders geöffnet
  - `MPI_MODE_SEQUENTIAL`: Datei wird nur sequentiell zugegriffen
  - `MPI_MODE_APPEND`: Alle Dateizeiger initial ans Ende der Datei setzen
- Modi können teilweise auch kombiniert werden
- Einige Modi bieten zusätzliches Optimierungspotential
  - Caching bei `MPI_MODE_UNIQUE_OPEN`
  - Read Ahead bei `MPI_MODE_SEQUENTIAL`

Positionierung

- Es gibt drei Arten der Positionierung
  1. Individuelle Dateizeiger
  2. Gemeinsame Dateizeiger
  3. Expliziter Versatz

1. Individuelle Dateizeiger
   - Prozess-lokaler Dateizeiger wird bei jedem Aufruf verändert
     - Analog zu `read` und `write`
   - Zugriffe unterschiedlicher Prozesse führen u. U. zu Konflikten

Positionierung...

2. Gemeinsame Dateizeiger
   - Globaler Dateizeiger wird bei jedem Aufruf verändert
     - Syntax: `MPI_..._shared` und `MPI_..._ordered`
   - Stellt sicher, dass unterschiedliche Prozesse nicht auf die gleichen Daten zugreifen
   - Möglicherweise Einschränkungen je nach benutztem Dateisystem
3. Expliziter Versatz
   - Versatz wird bei jedem Aufruf angegeben
     - Analog zu `pread` und `pwrite`
     - Syntax: `MPI_..._at`
   - Sicherer Zugriff durch unterschiedliche Prozesse möglich
     - Dafür allerdings explizite Berechnung der Offsets notwendig

Positionierung...

- `MPI_File_seek` und `MPI_File_seek_shared` setzen den Dateizeiger
  - Beide Funktionen unterstützten drei Positionierungsmodi
    1. `MPI_SEEK_SET`: Dateizeiger wird auf Versatz gesetzt
    2. `MPI_SEEK_CUR`: Dateizeiger wird um Versatz erhöht
    3. `MPI_SEEK_END`: Dateizeiger wird auf das Ende der Datei plus Versatz gesetzt
  - Der Versatz kann auch negativ sein

Gemeinsame Dateizeiger

- Gemeinsame Dateizeiger können für koordinierten Zugriff genutzt werden
  - Alle Prozesse nutzen denselben Dateizeiger
    - Datenkonflikte können so vermieden werden
  - Zugriffe ändern den Dateizeiger für alle anderen Prozesse
- Problematisch effizient zu implementieren
  - Benötigt irgendeine Form von Sperren
    - Dateizeiger darf nur durch einen Prozess zur Zeit verändert werden
  - Schwierig zu skalieren bei sehr vielen Prozessen
    - Änderungen müssen anderen Prozessen mitgeteilt werden
  - Nicht von jedem Dateisystem unterstützt
    - OrangeFS unterstützt keine Sperren und somit keine gemeinsamen Dateizeiger

Gemeinsame Dateizeiger...

- `MPI_..._shared` für nicht-kollektive Operationen
  - Operationen können in beliebiger Reihenfolge durchgeführt werden
- `MPI_..._ordered` für kollektive Operationen
  - Operationen werden entsprechend des Ranges ausgeführt
- Mögliche Anwendungsfälle
  - Gemeinsame Protokolldatei
  - Daten in Aufteilungsreihenfolge in Datei schreiben

Metadatenoperationen

- MPI-IO bietet wenige explizite Metadatenoperationen
  - Schnittstelle entspricht eher einem Object Store
- Keine Verzeichnisoperationen
  - Dateiname muss bekannt sein
- Eingeschränkte Dateiverwaltung
  - Erstellen von Dateien nur über `MPI_File_open`
- Vergrößern und Verkleinern einer Datei
  - `MPI_File_set_size` und `MPI_File_preallocate`
- Nur wenige Metadaten abrufbar
  - Kein Äquivalent zu `stat`, Dateigröße mittels `MPI_File_get_size`

Nicht-zusammenhängende Datentypen

- MPI-IO unterstützt nicht-zusammenhängende Datentypen
  - Zugriff auf komplexe Strukturen mit einem einzigen E/A-Aufruf
  - Einerseits Komfortfunktion für Entwickler
  - Andererseits Möglichkeit für zusätzliche Optimierungen
- Grundsätzlich auch manuell umsetzbar
  - Allerdings mit Zusatzaufwand verbunden
  - Ähnlich zu `readv`, `writev`, `aio_read`, `aio_write` und `lio_listio`

- Mit `readv` und `writev` können nicht alle Fälle abgedeckt werden, da immer ein zusammenhängender Bereich in der Datei gelesen bzw. geschrieben wird.

Nicht-zusammenhängende Datentypen...

```
1 int MPI_Type_vector (int count, int blocklength, int stride,
2                      MPI_Datatype oldtype, MPI_Datatype* newtype)
```

- Vektordatentyp unterstützt eine Schrittweite (`stride`)
- Beispiel: Diagonale einer 3x3-Matrix

```
1 MPI_Type_vector(3, 1, 4, MPI_DOUBLE, &newtype);
2 MPI_Type_commit(&newtype);
3 MPI_File_write(fh, buffer, 1, newtype, &status);
```

Listing 5: Nicht-zusammenhängender Vektordatentyp

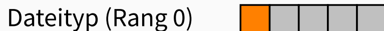
Nicht-zusammenhängende Datentypen...

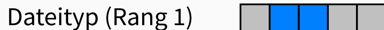
```
1 MPI_Type_vector(3, 1, 4, MPI_DOUBLE, &newtype);
```

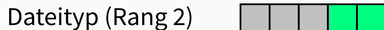
1	2	3
4	5	6
7	8	9

- Annahme: Matrix steht zeilen- oder spaltenweise abgerollt im Speicher
  - Bei einer 3×3-Matrix gibt es 3 Diagonalelemente
  - Jedes Diagonalelement besteht aus einem Double-Wert
  - Die Diagonalelemente haben einen Abstand von 4

- MPI-IO unterstützt kollektive E/A
  - Alle Prozesse führen ihre Zugriffe gleichzeitig und koordiniert durch
    - Syntax: `MPI_..._all`
  - Zusätzliche Informationen für eventuelle Optimierungen
    - Bei individueller E/A u. U. zufällige Reihenfolge
- Beispiel: Kleine nicht-zusammenhängende Zugriffe
  - Jeder Prozess greift auf mehrere kleine Bereiche zu
  - Alle Prozesse zusammen greifen auf die gesamte Datei zu







Nicht-blockierende Operationen

- MPI-IO unterstützt nicht-blockierende E/A-Operationen
  - Analog zu nicht-blockierendem Nachrichtenaustausch
    - Syntax: `MPI_..._i...`
  - Überlappung von E/A und Berechnung
    - Erlaubt es Anwendungen, die E/A-Zeit produktiv mit Berechnungen zu nutzen
    - Ermöglicht maximal zweifache Leistungssteigerung
- Statusüberprüfung mit den Standard-MPI-Funktionen
  - Z. B. `MPI_Wait` und `MPI_Test`
  - Auf Puffer darf vor Abschluss der Operationen nicht zugegriffen werden

Nicht-blockierende Operationen...

- Split Collectives für nicht-blockierende kollektive E/A
  - Aus Optimierungs- und Implementierbarkeitsgründen aufgeteilt
    - Syntax: `MPI_..._begin` und `MPI_..._end`
- Einige Einschränkungen
  - Pro Prozess und Datei nur ein laufender Aufruf
  - Nicht mit normalen kollektiven Operationen kombinierbar
  - Währenddessen keine anderen kollektiven E/A-Operationen erlaubt
  - Dürfen mit Hilfe der blockierenden Operationen implementiert werden

Hinweise

- Hinweise geben der Implementierung zusätzliche Informationen
  - Üblicherweise für Optimierungen
  - Können u. a. bei `MPI_File_open` angegeben werden
- Beispiele:
  - Anzahl der Geräte über die eine Datei verteilt werden soll
  - Größe der zu verteilenden Blöcke
  - Informationen über das Datenlayout
- Hinweise müssen nicht angegeben werden
  - Können aber auch beliebig durch die Implementierung ignoriert werden
  - Manche Implementierungen erlauben das Setzen über Umgebungsvariablen

Datenrepräsentationen

- MPI-IO unterstützt mehrere Datenrepräsentationen
  - Portabilität der Daten ein wichtiger Faktor
  - Häufig aber durch auf MPI-IO aufbauende Bibliotheken gehandhabt
- Drei mögliche Repräsentationen
  - native: Keine Umwandlung der Daten, Speicherung wie im Hauptspeicher
  - internal: Portabel zwischen allen Plattformen, die diese Implementierung unterstützt
  - external32: Portabel zwischen allen Implementierungen und Plattformen, möglicher Präzisions- und Leistungsverlust
- Zusätzlich benutzerdefinierte Repräsentationen

Allgemeines

- Verwendete Operationen sind für die erreichbare Leistung verantwortlich
  - Große Zugriffe häufig effizienter als kleine
  - Zusammenhängende Zugriffe häufig effizienter als zufällig verteilte
- MPI-IO bietet mehrere Möglichkeiten, die E/A durchzuführen
  - Zusammenhängend vs. nicht-zusammenhängend
  - Individuell vs. kollektiv
- Beispiel mit 3×3-Matrix:
  - Matrix ist zeilenweise im Speicher abgerollt
  - Matrix wird von drei Prozessen gelesen
  - Jeder Prozess ist für eine Spalte zuständig

Level 0: Individuelle zusammenhängende Zugriffe

```c
for (i = 0; i < 3; i++)
{
    MPI_File_seek(fh, ...);
    MPI_File_read(fh, ..., 1, MPI_DOUBLE, ...);
}
```

Listing 6: Level 0

- Jeder Prozess führt individuelle Zugriffe aus
- In jeder Iteration wird ein zusammenhängender Bereich gelesen

Level 0: Individuelle zusammenhängende Zugriffe

```
1 for (i = 0; i < 3; i++)
2 {
3     MPI_File_seek(fh, ...);
4     MPI_File_read(fh, ..., 1, MPI_DOUBLE, ...);
5 }
```

Listing 7: Level 0

- Jeder Prozess führt individuelle Zugriffe aus
  - Pro Iteration wird eine Zeile gelesen, allerdings in zufälliger Reihenfolge
- In jeder Iteration wird ein zusammenhängender Bereich gelesen

Level 1: Kollektive zusammenhängende Zugriffe

```
1 for (i = 0; i < 3; i++)
2 {
3     MPI_File_seek(fh, ...);
4     MPI_File_read_all(fh, ..., 1, MPI_DOUBLE, ...);
5 }
```

Listing 8: Level 1

- Prozesse führen koordiniert kollektive Zugriffe aus
- In jeder Iteration wird ein zusammenhängender Bereich gelesen

Level 1: Kollektive zusammenhängende Zugriffe

```
for (i = 0; i < 3; i++)
{
    MPI_File_seek(fh, ...);
    MPI_File_read_all(fh, ..., 1, MPI_DOUBLE, ...);
}
```

Listing 9: Level 1

- Prozesse führen koordiniert kollektive Zugriffe aus
  - Pro Iteration wird eine Zeile am Stück gelesen
- In jeder Iteration wird ein zusammenhängender Bereich gelesen
  - Bereich besteht aus nur einem Element

Level 2: Individuelle nicht-zusammenhängende Zugriffe

```
1 MPI_Type_vector(3, 1, 3, MPI_DOUBLE, &newtype);
2 MPI_Type_commit(&newtype);
3
4 MPI_File_seek(fh, ...);
5 MPI_File_read(fh, ..., 1, newtype, ...);
```

Listing 10: Level 2

- Jeder Prozess liest seine nicht-zusammenhängende Spalte
- Jeder Prozess führt individuelle Zugriffe aus

Level 2: Individuelle nicht-zusammenhängende Zugriffe

```
1 MPI_Type_vector(3, 1, 3, MPI_DOUBLE, &newtype);
2 MPI_Type_commit(&newtype);
3
4 MPI_File_seek(fh, ...);
5 MPI_File_read(fh, ..., 1, newtype, ...);
```

Listing 11: Level 2

- Jeder Prozess liest seine nicht-zusammenhängende Spalte
  - Führt zu kleinen, verteilten Zugriffen
- Jeder Prozess führt individuelle Zugriffe aus
  - Es werden alle Spalten gelesen, allerdings in zufälliger Reihenfolge

Level 3: Kollektive nicht-zusammenhängende Zugriffe

```
MPI_Type_vector(3, 1, 3, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);

MPI_File_seek(fh, ...);
MPI_File_read_all(fh, ..., 1, newtype, ...);
```

Listing 12: Level 3

- Jeder Prozess liest seine nicht-zusammenhängende Spalte
- Prozesse führen koordiniert kollektive Zugriffe aus

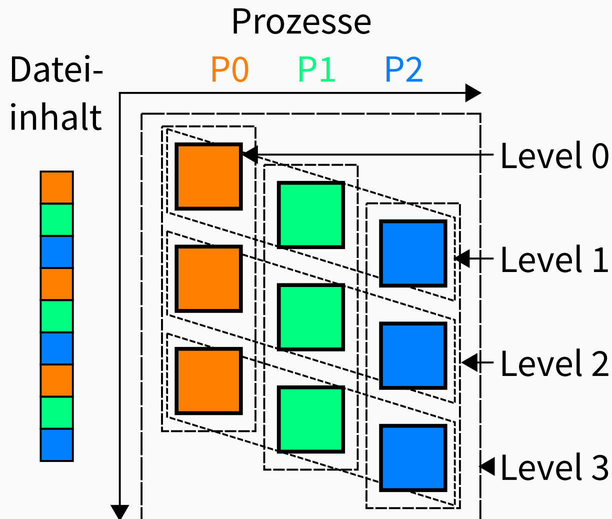
Level 3: Kollektive nicht-zusammenhängende Zugriffe

```
1 MPI_Type_vector(3, 1, 3, MPI_DOUBLE, &newtype);
2 MPI_Type_commit(&newtype);
3
4 MPI_File_seek(fh, ...);
5 MPI_File_read_all(fh, ..., 1, newtype, ...);
```

Listing 13: Level 3

- Jeder Prozess liest seine nicht-zusammenhängende Spalte
- Prozesse führen koordiniert kollektive Zugriffe aus
  - Es werden alle Spalten am Stück gelesen

Übersicht [1]



POSIX-Semantik

- Erinnerung: POSIX hat strenge Konsistenzanforderungen
  - Änderungen müssen nach `write` global sichtbar sein
  - E/A soll atomar geschehen
- Effiziente parallele E/A wird dadurch erschwert
  - Daten können nicht beliebig im Cache gehalten werden
  - Atomarität erfordert Sperren

Übersicht

- MPI-IO hat weniger strikte Anforderungen als POSIX
  - Änderungen sind nur im aktuellen Prozess sichtbar
  - Nicht-überlappende oder nicht-gleichzeitige Operationen werden korrekt gehandhabt
- Änderungen müssen nicht sofort global sichtbar sein
  - Dadurch weniger Aufwand durch Sperren
  - Erlaubt bessere Skalierbarkeit
- MPI-IO-Semantik meist ausreichend für wissenschaftliche Anwendungen
  - Z. B. nicht-überlappender Zugriff auf berechnete Daten

Manuelle Synchronisation

```
1 MPI_File_sync(fh);
2 MPI_Barrier(MPI_COMM_WORLD);
3 MPI_File_sync(fh);
```

Listing 14: Sync-Barrier-Sync-Konstrukt

1. Sync transferiert Änderungen ins Dateisystem
2. Barrier synchronisiert alle Prozesse
3. Sync macht Änderungen für alle Prozesse sichtbar

Atomic-Modus

- Atomic-Modus garantiert sequentielle Konsistenz
  - Kann mit `MPI_File_set_atomicity` dynamisch aktiviert werden
- Gleichzeitige und überlappende Zugriffe werden korrekt gehandhabt
  - Ähnlich der POSIX-Semantik
- Wird nicht überall unterstützt
  - ROMIO unterstützt den Modus, OMPIO nicht
  - Erfordert üblicherweise Sperren
    - Daher nicht in allen Dateisystemen verfügbar
    - Z. B. keine Unterstützung in OrangeFS

Zusammenfassung

Positionierung	Blockierung	Individuell	Kollektiv
Expliziter Versatz	Blockierend	read_at write_at	read_at_all write_at_all
	Nicht-blockierend & Split Collective	iread_at	read_at_all_begin read_at_all_end
		iwrite_at	write_at_all_begin write_at_all_end
Individuelle Dateizeiger	Blockierend	read write	read_all write_all
	Nicht-blockierend & Split Collective	iread	read_all_begin read_all_end
		iwrite	write_all_begin write_all_end
Gemeinsame Dateizeiger	Blockierend	read_shared write_shared	read_ordered write_ordered
	Nicht-blockierend & Split Collective	iread_shared	read_ordered_begin read_ordered_end
		iwrite_shared	write_ordered_begin write_ordered_end

Zusammenfassung…

- MPI-IO analog zu MPI-Kommunikation definiert
  - U. a. mit Unterstützung für abgeleitete Datentypen
- Dateien sind eine Abfolge von elementaren Datentypen
  - Jeder Prozess hat seine eigene Dateisicht
  - Mehrere Repräsentationen für Portabilität
- Unterschiedliche Modi zur Positionierung
  - Explizit, mit individuellen Dateizeigern und mit gemeinsamen Dateizeigern
- Unterschiedliche Zugriffsmodi zur Optimierung der E/A
  - Nicht-zusammenhängende, kollektive und nicht-blockierende Operationen

Quellen

[1] Rajeev Thakur, William Gropp, and Ewing Lusk. **Optimizing Noncontiguous Accesses in MPI-IO.** *Parallel Computing*, 28(1):83–105, January 2002.