MPI-IO

Hochleistungs-Ein-/Ausgabe

Michael Kuhn

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

2017-05-19

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

E/A-Schichten

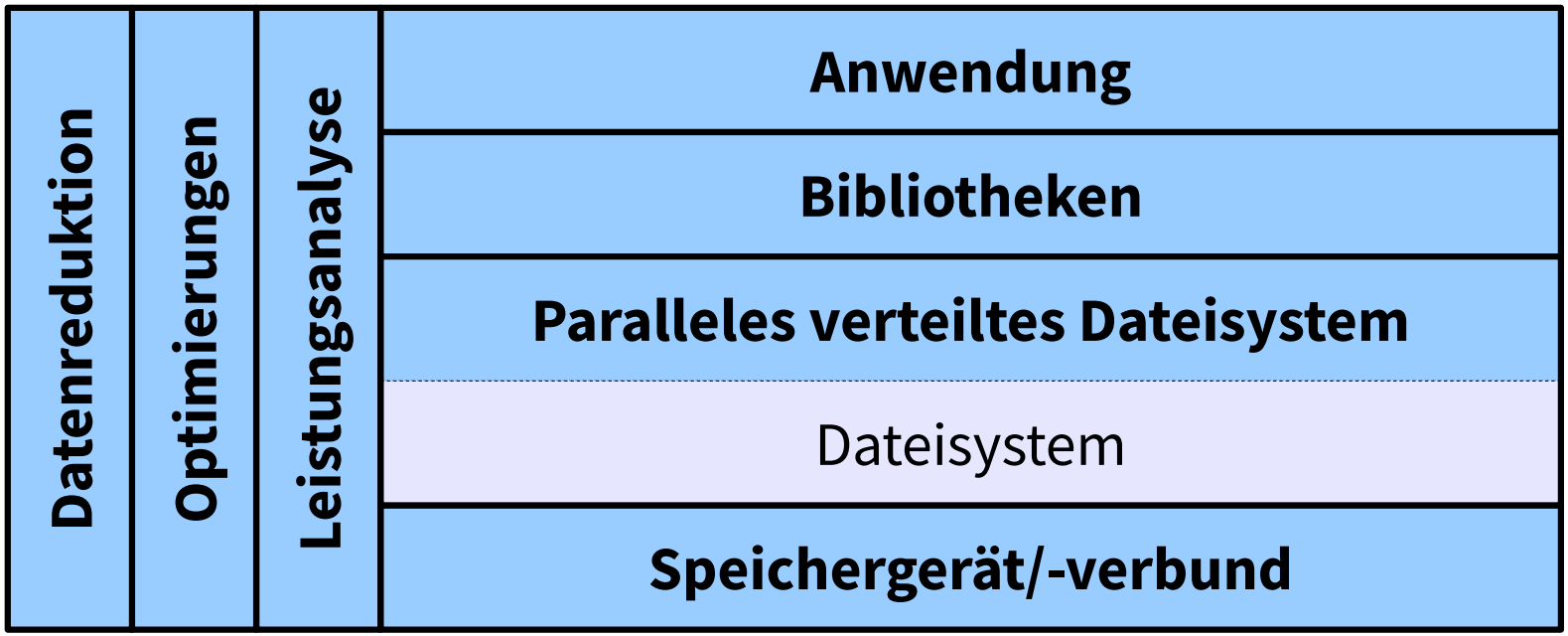

Abbildung: E/A-Schichten

Überblick

- Parallele Anwendungen benötigen häufig Unterstützung für parallele Ein-/Ausgabe
  - Serielle E/A ist ein Flaschenhals
  - Normalerweise kann kein Prozess alle Daten halten
- Häufige Szenarien
  - Lesen der Eingabedaten
    - Startbedingungen, größere Datensätze
  - Schreiben von Ausgabedaten
    - Ergebnisdaten, Checkpoints

- 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.

Überblick…

- MPI-IO bezeichnet den E/A-Teil von MPI
  - Wurde mit MPI 2.0 eingeführt (1997)
  - Anwendungen benutzen üblicherweise sowieso MPI
- Ist eine sogenannte Middleware
- Populärste Implementierung: ROMIO
  - Wird als Teil von MPICH vertrieben
  - Unter anderem in OpenMPI und MPICH-Derivaten
  - 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

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
- Gibt höheren Schichten Zugriff auf effiziente Algorithmen und Implementierungen für parallele E/A
  - 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…

- Dateisystem-spezifische Module in ADIO
  - Portabilität und höchstmögliche Leistung
    - Z. B. durch Zugriff auf Dateiverteilungsinformationen
  - 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

Grundlagen...

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

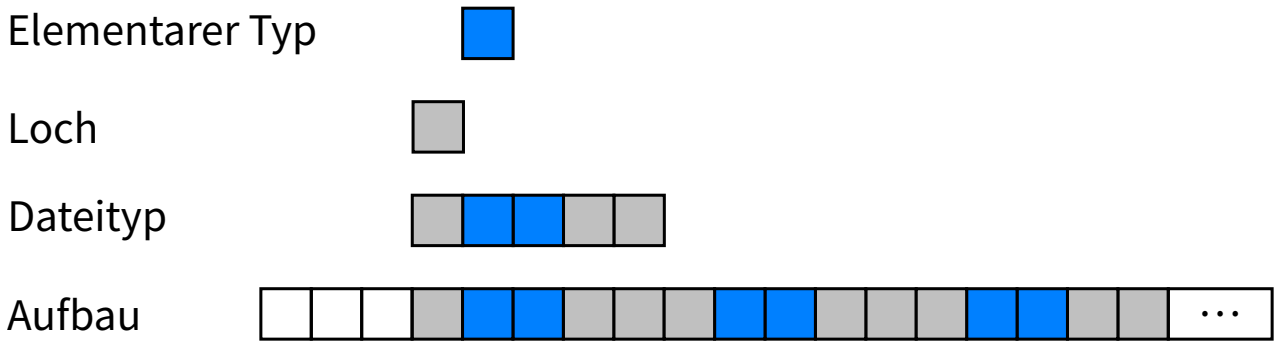

Abbildung: Dateityp

Grundlagen...

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

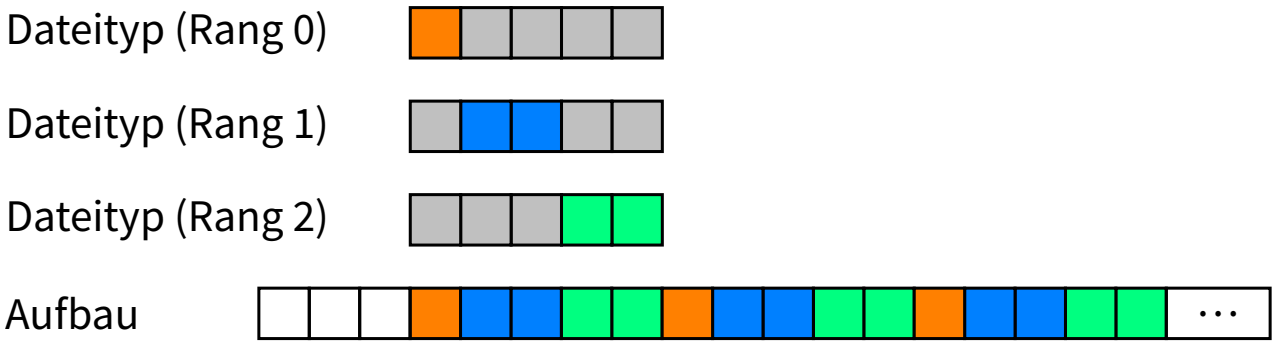

Abbildung: Dateisicht

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

- Öffnen einer Datei
  - `int MPI_File_open (MPI_Comm comm, char* filename, int amode, MPI_Info info, MPI_File* fh)`
  - Liefert Datei-Handle zurück
- Individuellen Dateizeiger setzen
  - `int MPI_File_seek (MPI_File fh, MPI_Offset offset, int whence)`
  - Analog zu `lseek`

Funktionen...

- Lesen und Schreiben von Daten
  - `int MPI_File_read (MPI_File fh, void* buf, int count, MPI_Datatype datatype, MPI_Status* status)`
  - `int MPI_File_write (MPI_File fh, void* buf, int count, MPI_Datatype datatype, MPI_Status* status)`
  - Analog zu `read` und `write`
- Schließen der Datei
  - `int MPI_File_close (MPI_File* fh)`

Code-Beispiel

```
 1  MPI_File fh;
 2  MPI_Offset size;
 3  MPI_Status status;
 4  char data[42];
 5  int nbytes;
 6
 7  memset(data, 23, sizeof(data));
 8  MPI_File_open(MPI_COMM_WORLD, "/tmp/mpi-io", MPI_MODE_RDWR
        ↪ | MPI_MODE_CREATE | MPI_MODE_DELETE_ON_CLOSE,
        ↪ MPI_INFO_NULL, &fh);
 9  MPI_File_write_at(fh, 0, data, sizeof(data), MPI_BYTE,
        ↪ &status);
10  MPI_Get_count(&status, MPI_BYTE, &nbytes);
11  MPI_File_get_size(fh, &size);
12  printf("File size is %" PRIdMAX " bytes.\n",
        ↪ (uintmax_t)size);
13  MPI_File_close(&fh);
```

MPI_File_open

- `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

MPI_File_open...

- `MPI_File_open` bietet mehrere Zugriffsmodi (1/2)
  - `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 Datei erstellt werden soll, die bereits existiert

MPI_File_open...

- `MPI_File_open` bietet mehrere Zugriffsmodi (2/2)
  - `MPI_MODE_DELETE_ON_CLOSE`: Datei beim Schließen löschen
  - `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

Positionierung

- Es gibt drei Arten der Positionierung
  - Individuelle Dateizeiger
  - Gemeinsame Dateizeiger
  - Expliziter Versatz
- Individuelle Dateizeiger
  - Prozess-lokaler Dateizeiger wird bei jedem Aufruf verändert
  - Analog zu `read` und `write`

Positionierung...

- Gemeinsame Dateizeiger
  - Globaler Dateizeiger wird bei jedem Aufruf verändert
  - Syntax: `MPI_..._shared` und `MPI_..._ordered`
- Expliziter Versatz
  - Versatz wird bei jedem Aufruf angegeben
  - Syntax: `MPI_..._at`
  - Analog zu `pread` und `pwrite`

MPI_File_seek und MPI_File_seek_shared

- `MPI_File_seek` und `MPI_File_seek_shared` erlauben das Setzen des Dateizeigers
- Beide Funktionen unterstützten drei Positionierungsmodi
  - `MPI_SEEK_SET`: Dateizeiger wird auf Versatz gesetzt
  - `MPI_SEEK_CUR`: Dateizeiger wird um Versatz erhöht
  - `MPI_SEEK_END`: Dateizeiger wird auf das Ende der Datei plus Versatz gesetzt
- Der Versatz kann auch negativ sein

Metadatenoperationen

- MPI-IO bietet wenige explizite Metadatenoperationen
  - Keine Verzeichnisoperationen
  - Erstellen nur über `MPI_File_open`
- Vergrößern und Verkleinern einer Datei
  - `MPI_File_set_size` und `MPI_File_preallocate`
- Kein Äquivalent zu `stat`
  - Nur `MPI_File_get_size`

Nicht-zusammenhängende Datentypen

- MPI-IO unterstützt nicht-zusammenhängende Datentypen
  - Zugriff mit einem einzigen E/A-Aufruf
  - Komfortfunktion für Entwickler
  - Erlaubt aber auch zusätzliche Optimierungen
- Grundsätzlich auch manuell umsetzbar
  - 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...

- Vektordatentyp unterstützt eine Schrittweite
  - `int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype* newtype)`
- 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 1: Nicht-zusammenhängender Vektordatentyp

Nicht-zusammenhängende Datentypen...

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

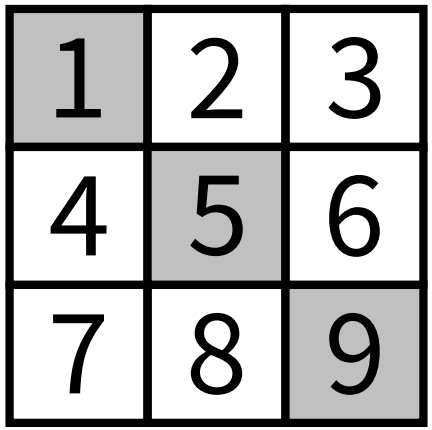

Abbildung: Nicht-zusammenhängender Vektordatentyp

Kollektive Operationen

- MPI-IO unterstützt kollektive E/A
  - Alle Prozesse führen ihre Zugriffe gleichzeitig durch
  - Syntax: `MPI_..._all`
  - Zusätzliche Informationen für eventuelle Optimierungen
- Beispiel: Kleine nicht-zusammenhängende Zugriffe
  - Jeder Prozess greift nur auf einen kleinen Bereich zu
  - Alle Prozesse zusammen aber auf die gesamte Datei

Nicht-blockierende Operationen

- MPI-IO unterstützt nicht-blockierende E/A-Operationen
  - Überlappung von E/A und Berechnung
  - Analog zu nicht-blockierendem Nachrichtenaustausch
  - Syntax: `MPI_..._i...`
- Statusüberprüfung mit den Standard-MPI-Funktionen
  - Z. B. `MPI_Wait` und `MPI_Test`

Nicht-blockierende Operationen...

- Split Collectives für nicht-blockierende kollektive E/A
  - Syntax: `MPI_..._begin` und `MPI_..._end`
  - Aufteilung dient der Optimierung und besseren Implementierbarkeit
- 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

Gemeinsame Dateizeiger

- Gemeinsame Dateizeiger für koordinierten Zugriff
  - Alle Prozesse nutzen denselben Dateizeiger
  - Zugriffe ändern den Dateizeiger für alle anderen Prozesse
- Problematisch effizient zu implementieren
  - Benötigt irgendeine Form von Sperren
  - Schwierig zu skalieren bei sehr vielen Prozessen
  - Nicht von jedem Dateisystem unterstützt

- OrangeFS unterstützt beispielsweise keine gemeinsamen Dateizeiger, da dafür Sperren benötigt würden.

Gemeinsame Dateizeiger...

- `MPI_..._shared` für nicht-kollektive Operationen
- `MPI_..._ordered` für kollektive Operationen
  - Wird entsprechend des Ranges ausgeführt
- Mögliche Anwendungsfälle
  - Gemeinsame Protokolldatei
  - Daten in Berechnungsreihenfolge in Datei schreiben

Hinweise

- Hinweise geben der Implementierung zusätzliche Informationen
  - Üblicherweise für Optimierungen
- 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

Datenrepräsentationen

- MPI-IO unterstützt mehrere Datenrepräsentationen
  - Portabilität der Daten ein wichtiger Faktor
- 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 maßgeblich für die erreichbare Leistung verantwortlich
  - Zusammenhängend vs. nicht-zusammenhängend
  - Individuell vs. kollektiv
- Beispiel
  - 3x3-Matrix wird von drei Prozessen gelesen
  - Jeder Prozess ist für eine Spalte zuständig

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 2: 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 3: Level 1

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

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 4: Level 2

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

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 5: Level 3

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

Übersicht

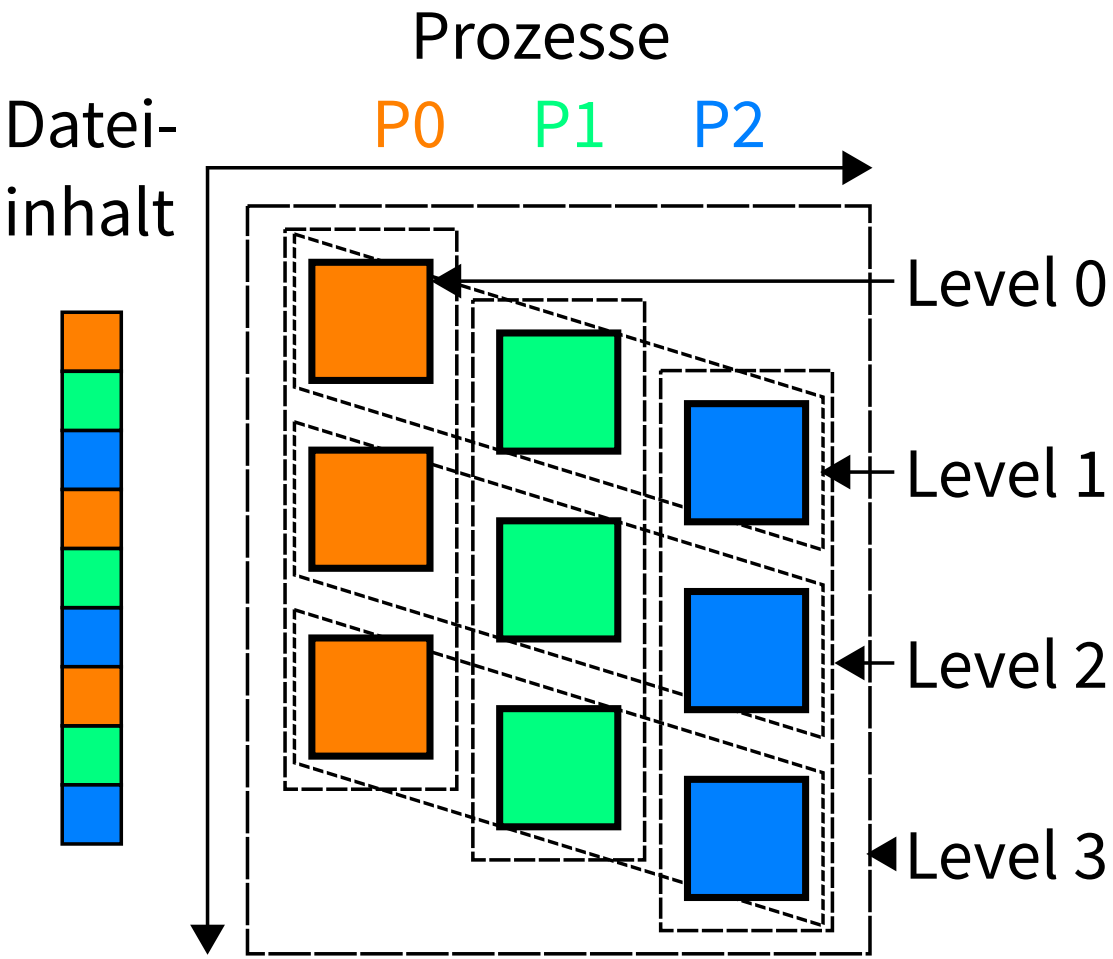

Abbildung: Visualisierung der unterschiedlichen Zugriffslevel [1]

POSIX

- 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
- Erlaubt bessere Skalierbarkeit
  - Änderungen müssen nicht sofort global sichtbar sein
  - Dadurch weniger Aufwand durch Sperren

Manuelle Synchronisation

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

Listing 6: Sync-Barrier-Sync-Konstrukt

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

Atomic-Modus

- Atomic-Modus stellt strikere Garantien bereit
  - `MPI_File_set_atomicity`
- Alle Änderungen sind sofort für alle Prozesse im Kommunikator sichtbar
  - 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

- Der Atomic-Modus ist immer noch weniger strikt als POSIX, da Änderungen nur für Prozesse im gleichen Kommunikator sichtbar sein müssen.

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
- Die Positionierung ist explizit, mit individuellen Dateizeigern und mit gemeinsamen Dateizeigern möglich
- Nicht-zusammenhängende und kollektive Operationen können die Effizienz erhöhen

Quellen I

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