

- 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.

- 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.

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 Setzen des Dateizeigers
- Beide Funktionen unterstützen 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);
```

1	2	3
4	5	6
7	8	9

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 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
- 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 3: 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

- 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

