



## Semantik – POSIX I/O, MPI-IO

Seminar Parallele Dateisysteme  
WS 07/08  
Universität Heidelberg

06.11.2007

Referent:           Stefan Becker  
Betreuer:           Julian M. Kunkel

# Gliederung

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Einführung – Semantik
- Semantik der gemeinsamen Nutzung von Daten
- POSIX I/O Semantik
- MPI-IO Semantik
- Zugriffssemantiken
- Zusammenfassung
- Quellen

# Einführung – Semantik

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Allgemein: Bedeutungslehre
  - Beziehung zwischen Worten in einem Text
  - Bedeutung einer Informationsfolge
- Semantik in der Informatik
  - Formalisierung der Bedeutung von Computerprogrammen und Spezifikationen
  - Korrektheit von Programmen prüfen
- Semantik = Vorschrift für System und Programmierer
- Beispiel: Identische Funktionsprototypen aber unterschiedliche Auswirkung (Semantik)
- Semantik schränkt ein!
  - Auswirkung von Semantik auf Performance, Portabilität, Benutzbarkeit, Sicherheit etc.
  - Semantik schreibt zum Teil die Implementierung vor
  - Legt fest, wie ein System sich gegenüber der Anwendung verhält

Der Begriff Semantik kann unterschiedlich aufgefasst werden. Allgemein versteht man darunter die Bedeutungslehre. In einem Text beschreibt Semantik die Beziehung zwischen Worten. Des Weiteren zeigt Semantik den Bedeutungswandel eines Wortes auf.

Ein kleines Beispiel: „Bedeutung einer Informationsfolge“: Eine reine Zufallsreihenfolge hat keine Semantik, da die Reihenfolge der Informationen willkürlich ist und ohne weitere Bedeutung erstellt wurde.

Der Begriff Semantik in der Informatik beschäftigt sich mit der Formalisierung der Bedeutung von Computerprogrammen und Spezifikationen. Dies wird zum Beispiel für den Nachweis der Korrektheit von Computerprogrammen gebraucht (Verifikation).

Die Semantik einer Programmiersprache schränkt den Programmierer ein. Es ist darunter eine Art Vorschrift zu verstehen, wie das System und der Programmierer im Umgang mit einer Sprache zu verhalten haben. ☺ Eine Programmiersprache legt gewisse Integrationsmerkmale fest (vereinfacht: definiert ein Verhalten) und macht z.B. ein Dateisystem langsam, erschwert die Portabilität, Benutzbarkeit für den Programmierer oder verhindert die Gewährleistung von sicherem Datenaustausch.

Zum Beispiel haben zwei APIs identische Funktionen aber ihre Semantik ist unterschiedlich: Die eine Sprache verwendet automatisch Caching bei der Read-Funktion; die andere Sprache nicht. Oder: Die eine Sprache blockiert während eines Aufruf der Write-Funktion, die andere Sprache dagegen nicht!

Aufgrund von Hardwarebeschränkungen, Performance (Skalierbarkeit erhöhen), Portabilität, Benutzbarkeit, Sicherheit gibt es verschiedene Semantiken.

Semantiken für die gemeinsame Nutzung von Dateien möchte ich Ihnen auf den folgenden Folien näher bringen.

## Semantik der gemeinsamen Nutzung von Daten (1)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Situation:
  - Mehrere Prozesse greifen auf gleiche Datei innerhalb eines verteilten Dateisystems zu.
- Zugriffssemantik: „Welche Bedeutung hat ein (ändernder) Dateizugriff eines Prozesses für andere Prozesse?“
- Fragen:
  - Sind Änderungen erlaubt. Sehen die anderen Prozesse diese und wenn ja, wann?
  - Welche Bedeutung hat ein ändernder Dateizugriff eines Prozesses für andere Prozesse? (Zugriffssemantik)
- Probleme:
  - Unterschied zw. Einzelrechner und verteiltes Dateisystem
  - Nebenläufigkeitsprobleme
  - Konflikt, wenn mindestens ein Schreibzugriff daran beteiligt ist

Greifen mehrere Prozesse auf eine Datei gemeinsam zu, entstehen Nebenläufigkeitsprobleme. Dabei treten Fragen auf wie: „Sind Änderungen an Dateien erlaubt?“, oder „Wann sehen die anderen Prozesse diese Änderungen?“.

Auf einem Einzelrechner ist es durch die zentrale Auslegung des Systems leicht möglich Lese- und Schreibzugriffe stets sequentiell geordnet zu halten z.B. durch einen einzelnen Dateipositionszeiger. Dies kann auf ein verteiltes Dateisystem übertragen werden, wenn auch hier eine zentrale Koordinierung durchgeführt wird. In den meisten verteilten Dateisystemen wird aber wegen Performance mit Caching oder Replikation gearbeitet, dann funktioniert diese Methode nicht. Eine (schlechte) Lösung ist es, generelles Sperren von Dateien durchzuführen: Greift ein Client auf eine Datei zu, so sperrt er sie für den Zugriff durch Andere. Nachteil: Diese Methode verhindert jegliche Parallelität des Zugriffs. Andere Semantiken werden auf den folgenden Folien nun vorgestellt.

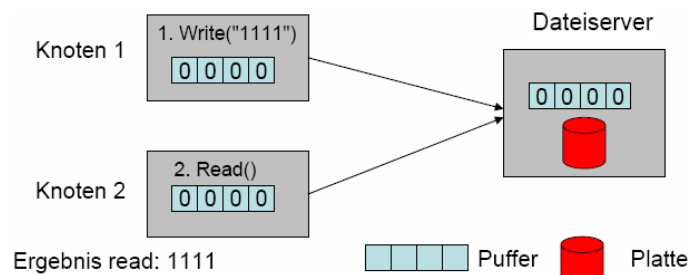
## Semantik der gemeinsamen Nutzung von Daten (2)

### Semantik

1. Einführung
- 2. Semantik der gemeinsam. Nutzung
3. POSIX I/O Semantik
4. MPI-IO Semantik
5. Zugriffssemantik
6. Zusammenfassung

### ■ UNIX-Semantik

- Sequentieller Zugriff
- Änderungen sofort sichtbar
- Keine Kopien der Dateien
- Entspricht strikter Konsistenz
- Ineffizient: Hoher Aufwand für Konsistenzwahrung (write-through) bei verteiltem System



© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

5

**UNIX-Semantik (Einzelkopie-Semantik):** Jede Datei-Operationen wird von jedem Prozess sofort gesehen, d.h. wenn ein Prozess eine Datei schreibt, dann bekommt das jeder andere Prozess bei einem Read() unmittelbar mit.

**Architektur:** Der Dateiserver kann stateless (zustandslos) und verbindungsunabhängig sein. Dadurch ist das Netz durch seinen Absturz nicht sehr gefährdet.

**Problematik:** Wenn die Clients (Klienten, Knoten) Dateien cachen, dann muss das Caching so implementiert werden, dass Veränderungen einer Cachekopie sofort bei allen anderen Kopien sichtbar werden (write-through), ansonsten kommt es zu Dateninkonsistenzen.

**Fazit:** Die UNIX-Semantik für verteilte Dateisysteme ist durch ihre zentrale Implementierung bzw. durch den Aufwand zur Konsistenzwahrung (write-through) sehr ineffizient, aufwendig. Für lokale Dateisysteme jedoch leicht zu implementieren, da alle Operationen einen gemeinsamen Puffer verwenden.

Quelle der Grafik: <http://www.ipd.uni-karlsruhe.de/Tichy/uploads/fohlen/126/Cluster12Dateisysteme.pdf>

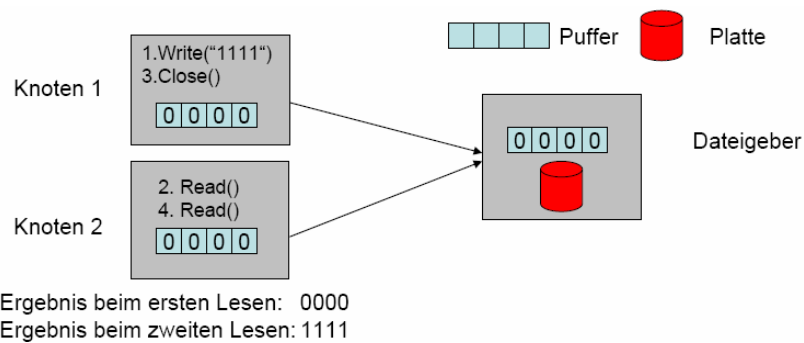
## Semantik der gemeinsamen Nutzung von Daten (3)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

### ■ Sitzungs-Semantik

- Kopie von Datei
- Änderungen für andere Knoten erst sichtbar nach Schließen der Datei
- Verlust von Zwischenergebnissen
- Evtl. Versionsverwaltung nötig



© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

6

**Sitzungs-Semantik (Session-Semantik):** Beim Öffnen einer Datei erhält der Klient eine eigene Kopie. Mit dieser Kopie arbeitet er bis zum Schließen der Datei. Dann wird die Datei als Ganzes zum Dateiserver zurück geschrieben. Dadurch sind Dateiänderungen für andere Klienten erst nach dem Schließen sichtbar.

**Problematik:** Probleme treten auf, wenn eine Datei bei mehreren Klienten gleichzeitig geändert wird.

**Lösung:** Versionsverwaltung um das Verwerfen von mehreren zeitgleichen Änderungen zu verhindern.

Die Idee der Versionsführung ist:

- Mit dem letzten Schließen werden alle vorherigen Versionen beim Server überschrieben.
- Die Klienten müssen zwischen den verschiedenen Versionen einen Abstimmungsprozess durchführen.
- Unterschiedliche Versionen werden von dem Server geeignet gemischt. Dazu muss der Anwendungskontext bekannt sein.
- Die Versionen werden getrennt voneinander weitergeführt, etwa durch spezielle Namensgebung für die Versionen.

Quelle: [http://www.fbi.h-da.de/~a.schuette/Vorlesungen/VerteilteSysteme/Skript/6\\_VerteilteDateisysteme/VerteilteDateisysteme.pdf](http://www.fbi.h-da.de/~a.schuette/Vorlesungen/VerteilteSysteme/Skript/6_VerteilteDateisysteme/VerteilteDateisysteme.pdf)

Quelle der Grafik: <http://www.ipd.uni-karlsruhe.de/Tichy/uploads/fohlen/126/Cluster12Dateisysteme.pdf>

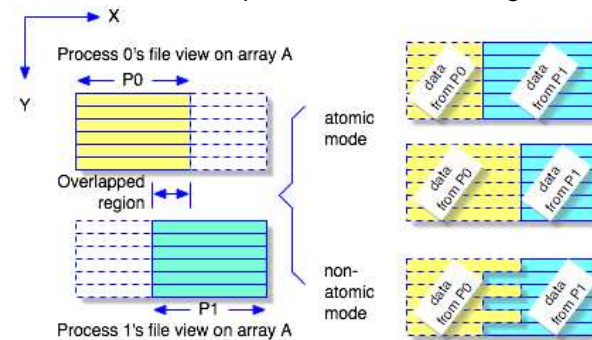
## Semantik der gemeinsamen Nutzung von Daten (4)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffsemantik
- 6. Zusammenfassung

### Atomare Transaktion:

- Sperren für gemeinsame Daten(-bereiche)
- Änderungen werden vollständig oder gar nicht ausgeführt (Kontrollierende Instanz)
- Performance: keine Parallelität, bei überlappenden Bereichen. Sonst parallele Verarbeitung



© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

7

Atomare Transaktion: Änderungen werden vollständig oder gar nicht ausgeführt. Eine kontrollierende Instanz bestimmt darüber, welcher Aufruf ausgeführt wird und welcher nicht. Wenn z.B. zwei Prozesse Daten gleichzeitig schreiben möchten, dann kann das Ergebnis bei einer nicht atomaren Transaktion gemischt sein. Bei einer atomaren Transaktion ist das Ergebnis nicht gemischt, da nur ein Aufruf zu einer Zeit ausgeführt wird. In der obigen Grafik lesen zwei Prozesse eine gemeinsame Datei. Ihr Lesebereich überlappt sich. Nun wollen beide Prozesse ihre Daten an den Server schicken. Dieser schreibt bei einem atomaren Verfahren zuerst die Daten von P0 oder von P1 (Die Rechtecke rechts-oben und rechts-mitte). Bei einem nicht atomaren Verfahren kann dagegen ein Mischergebnis zustande kommen. Dies wird in der Grafik rechts unten dargestellt. Der überlappende Bereich wurde durcheinander geschrieben. Atomarität wird durch das Sperren von überlappenden Dateibereichen gewährleistet.

Architektur: Standard-Dateisystem-Schnittstellen unterstützen diese Semantik nicht. D.h. es ist eine Erweiterung der Dateisystem-Schnittstelle nötig.

Problematik: Sperren der Dateibereiche notwendig, die von zwei oder mehreren Prozessen gelesen werden. Überlappende Bereiche können nur sequentiell bearbeitet werden und schließen somit eine Parallelität aus.

Quelle der Grafik: <http://cucis.ece.northwestern.edu/projects/MPIIO/>

## Semantik der gemeinsamen Nutzung von Daten (5)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

### ■ Datenbank-Semantik

- Transaktion
  - Folge von Operationen mit fester Reihenfolge
  - Nebenläufigkeit erlaubt
  - BOT, EOT, Commit, Rollback
  - Arbeitet auf ganzer Gruppe von Dateien
- Transaktionssystem
  - inhaltliche Regelung (Zugriffsemantik)
  - Kontrollierende Instanz
  - Garantie des ACID Prinzips
  - Aufspalten der Transaktionen in ihre Operationen
- Probleme: Deadlocks, kaskadierendes Zurücksetzen
- Verwaltungsaufwand für transaktionsbezogene Dateizugriffe ist sehr hoch und damit für verteilte Dateisysteme auch nicht sehr effizient.
- Semantik zwingend wenn Dateien DB-Ersatz

Datenbank-Semantik: Operationen werden bei dieser Semantik in einer festen Reihenfolge zu Transaktionen zusammengefasst. Ein Beispiel für eine Transaktion von Wikipedia:

*In einer Bücherei wird ein Karteikarten-System zur Verwaltung des Bestandes an Büchern verwendet. Eine Transaktion könnte hier lauten: „Leihe das Buch ‚Die Schatzinsel‘ an den Benutzer Peter Müller aus.“ Diese Transaktion könnte in der formalen Darstellung so aussehen:*

begin of transaction

lies das Feld "Vorbestellung" der Karte

schreibe "Peter Müller" in das Feld "ausgeliehen an"

schreibe "29. Juli 2001" in das Feld "Rückgabe am"

end of transaction

Transaktionen werden von Transaktionssystemen verarbeitet. Bei der Ausführung von Transaktionen muss das Transaktionssystem die ACID-Eigenschaften garantieren: **Atomarität** (*atomicity*), **Konsistenz** (*consistency*), **Isoliertheit** (*isolation*) und **Dauerhaftigkeit** (*durability*)!

Performance: Optimierung durch Nebenläufigkeit erlaubt und gewollt. Serielle Ausführung zwar einfacher zu realisieren aber nicht performant genug.

spalten daher Transaktionen in ihre Operationen auf und setzen diese zu Historien zusammen, wobei selbstverständlich die ACID-Eigenschaften bewahrt bleiben müssen. Bewertung: Der Verwaltungsaufwand für transaktionsbezogene Dateizugriffe ist sehr hoch und damit für verteilte Dateisysteme auch nicht sehr effizient. Wenn eine Transaktion aufgrund einer anderen Transaktion nicht ausgeführt werden kann, spricht man von einer Blockierung. Wird die erste Transaktion durch die zweite und gleichzeitig die zweite durch die erste blockiert, so spricht man von einem Deadlock (Verklemmung). Es kann beim Zurücksetzen einer Transaktion vorkommen, dass das Zurücksetzen einer anderen Transaktion notwendig wird, was zur Bildung regelrechter Ketten von Zurücksetzungen führen kann; dies wird als kaskadierendes Rücksetzen bezeichnet und ist ein wenig erwünschter Effekt.

Aber: Falls Dateien als Datenbankersatz verwendet werden sollen, ist diese Semantik zwingend.

# Metadaten bei parallelem I/O

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Daten, die Informationen über andere Daten enthalten.
  - Länge, Diverse Zeitstempel, Schreibschutz etc.
- Extended Attributes:
  - ACLs (Zugriffskontrolllisten)
  - Z.B. für Indexdienste
- Zu klären:
  - Welche Metadaten werden erhoben?
  - Wann sehen Clients Modifikationen?
- Problem
  - Metadaten kaum effizient parallelisierbar bzw. verteilbar
  - → zerstört Skalierbarkeit des Cluster-Dateisystems
- Beispiel: Metadatei wird zentral gehalten
  - Viele Prozesse schreiben in eine Datei und Zeitstempel ändert sich
- Lösungen
  - Weglassen von Metadaten → Extended Attributes
  - Weglassen der Zugriffsrechte
  - Aktualisierung in größerem Intervall
  - Replikation von Metadaten

© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

9

Als **Metadaten** oder **Metainformationen** bezeichnet man allgemein [Daten](#), die Informationen über andere Daten enthalten. Bei den beschriebenen Daten handelt es sich oft um größere Datensammlungen ([Dokumente](#)) wie [Bücher](#), [Datenbanken](#) oder [Dateien](#). So werden auch Angaben von [Eigenschaften](#) eines Objektes (beispielsweise [Personennamen](#)) als Metadaten bezeichnet. (Quelle: Wikipedia.de)

Enthalten diverse Zeitstempel (letzte Modifikation, Erstellungsdatum etc.) aber auch Zugriffsrechte, Zugriffskontrolllisten und stellen erweiterte Attribute für Suchmaschinen und Indexdienste zur Verfügung (Websuchmaschinen, Desktopsuchmaschine...).

[Access Control Lists](#) (Zugriffskontrolllisten), werden bei Betriebssystemen zum Kontrollieren der Zugriffsberechtigungen auf diverse Ressourcen wie Dateien und Programme, verwendet. In der [Unixwelt](#) versteht man unter Access Control List eine Erweiterung der klassischen Zugriffssteuerung auf Ebene des Benutzer-Gruppe-Welt-Modells. Bei Access Control Lists lassen sich Zugriffsrechte spezifisch für einzelne Benutzer zuteilen oder verbieten. Unter [Microsoft Windows](#) (NT, 2000, XP, 2003 Server) wird jedem Betriebssystemobjekt (Dateien, Prozesse etc.) ein [Security Descriptor](#) zugeordnet, der eine ACL enthalten kann. Ist keine ACL vorhanden, so erhält jeder Benutzer Vollzugriff auf das Objekt. Ist die ACL vorhanden, aber leer, so erhält kein Benutzer Zugriff. Eine ACL besteht aus einem Header und maximal 1.820 Access Control Entries (ACE).

Quelle: Wikipedia.de

## Semantik der gemeinsamen Nutzung von Daten (6)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffs-semantik
- 6. Zusammenfassung

- Unveränderlichkeits-Semantik
  - Dateien sind nicht veränderbar!
  - Änderung erzwingt Neuanlegung der Datei
- Problem: Versionsproblematik (siehe Sitzungs-Semantik)
- Pro:
  - Ermöglicht effizientes Caching und dadurch effiziente gemeinsame Nutzung der Daten
- Contra:
  - Ineffizient für Änderungen an großen Dateien
  - Anwendung ist für Konsistenz verantwortlich! → Informieren der beteiligten Prozesse über Dateierneuerung.

**Unveränderlichkeits-Semantik:** Bei diesem Ansatz sind Dateien nicht veränderbar. Dateien können immer nur neu erstellt oder gelesen werden. Änderungen erzwingen stets

ein Neuanlegen der Datei. Greifen mehrere Klienten (lokal) schreibend auf eine gerade mehrfach gelesene Datei zu, entstehen mehrere Versionen. D.h. es ergibt sich eine ähnliche Versionsproblematik wie bei der Sitzungs-Semantik.

Positiv an der Unveränderlichkeits-Semantik ist, dass sie effizientes Caching ermöglicht und dadurch eine effiziente gemeinsame Nutzung der Daten, da man sich nicht mehr um die Aktualisierung der Daten kümmern muss.

Problematisch ist Semantik bei der Änderung von großen Dateien, da diese komplett kopiert, geändert und neu geschrieben werden müssen. Des Weiteren ist die Anwendung für die Konsistenz der Daten verantwortlich. Die beteiligten Prozesse müssen über Dateierneuerungen informiert werden, sonst kommt es zum Datenverlust. Da evtl. gleichzeitige Änderungen stattfinden ist eine Versionsverwaltung durch die Anwendung notwendig. Dies kostet Zeit und Leistung.

# POSIX – Portable Operation System Interface + X

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Definiert von der IEEE Gruppe zur Definition einem klaren, unmissverständlichen und konsistenten Set an Standards für Betriebssysteme
- <http://standards.ieee.org/regauth/posix/>
- Standard „DIN/EN/ISO/IEC 9945“ ☺
- Stellt Schnittstelle zwischen Anwendung und Betriebssystem dar
- Basiert auf der Sprache C
- Integrierbar in UNIX und Nicht-UNIX Systemen
- Die Funktionen von POSIX stellen unter anderem
  - Ein- und Ausgabe für Dateien, Terminals und Netzwerkdienste
  - Erzeugung und Kontrolle von Prozessen
  - sowie Benutzer- und Gruppenverwaltung zur Verfügung

© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

11

POSIX (**P**ortable **O**perating **S**ystem Interface + **X**) ist ein gemeinsam von der IEEE und der Open Group für Unix entwickeltes standardisiertes, klar definiertes Applikationsebeneninterface, das die Schnittstelle zwischen Applikation und dem Betriebssystem darstellt. Der (inter-)nationale Standard trägt die Bezeichnung DIN/EN/ISO/IEC 9945.

Die Spezifikation der Benutzer- und Software-Schnittstelle des Betriebssystems ist in vier Teile unterteilt.

Zusätzlich existieren noch Erweiterungen im Echtzeit-Bereich:

- Basis-Definitionen (POSIX.1/IEEE 1003.1-2001): Eine Liste der im Standard benutzten Konventionen und Definitionen, zusätzlich noch eine Liste der bereitzustellenden [C-Headerdateien](#)
- Kommandozeileninterpreter und Hilfsprogramme (POSIX.2/IEEE 1003.1-2001): Eine Liste der Hilfsprogramme und der [Kommandozeileninterpreter](#)
- Echtzeit-Erweiterungen (POSIX.4/IEEE 1003.1b-1993/IEEE 1003.1d-1999)
- Thread-Erweiterungen (POSIX.4a/IEEE 1003.1c-1994)
- System-Schnittstelle: Eine Liste der C-[Systemaufrufe](#), die unterstützt werden müssen
- Erklärungen: Erläuterungen zum Standard

Betriebssysteme können vollständig oder teilweise POSIX-konform sein – dies hängt davon ab, ob sie die POSIX-Standards gänzlich oder nur zum Teil einhalten. Zertifizierte Produkte werden auf der [POSIX Certification](#) Website der [IEEE](#) genannt. Windows POSIX-konform?

<http://www.microsoft.com/technet/archive/ntwrkstn/reskit/poscomp.msp?mfr=true>

Diese auf C-basierende POSIX APIs definieren gewisse Festlegungen, wie ein System sich gegenüber der Anwendung verhält, wenn es den diversen POSIX-Standards entspricht.

Quelle: <http://de.wikipedia.org/wiki/Posix>

## POSIX I/O API

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- POSIX-I/O stellt die Standardschnittstelle für den Dateizugriff in Unix-Systemen dar.
- Spezifiziert im POSIX-Standard 1003.1,1996
- Sehr starke Verbreitung (Linux, Windows)
- Ursprünglich:
  - Für sequentielle Anwendungen
  - Synchrone sequentielle Datenzugriffe
  - kein Ausdruck für Parallelität!
- Später erweitert für:
  - Parallele Anwendungen
  - Asynchrone Datenzugriffe (AIO)
  - Nichtsequentielle Zugriffe

POSIX-I/O stellt die Standardschnittstelle für den Dateizugriff in Unix-Systemen dar. POSIX-I/O wurde im POSIX-Standard 1003.1,1996 spezifiziert und ist Teil des klaren, unmissverständlichen und konsistenten Set an Standards für Betriebssysteme. Zu Beginn war UNIX nicht POSIX-konform. Die Datei-API von UNIX wurde im nachhinein darum erweitert.

Die POSIX I/O API ist mittlerweile sehr stark verbreitet. Viele Linux-Distributionen und Windows NT halten sich an den Großteil des POSIX Standards. Deswegen ist POSIX die meistgenutzte API von Anwendungen und Programmbibliotheken für Dateizugriffe.

Ursprünglich war POSIX nur für sequentielle Anwendungen, die zusammenhängende Datenblöcke bearbeiteten, entwickelt. Der Zugriff war nur synchron, d.h. jeder Lese- oder Schreibvorgang verursachte eine Synchronisation. Durch diese Prozesssynchronisation kommt es zu Wartezustände, in denen der wartende Prozess nicht weiterarbeiten kann. Daher ist dieses Verfahren nicht für parallele Anwendungen geeignet. Später wurde POSIX für parallele Anwendungen erweitert und an deren Bedürfnisse angepasst. Zum Beispiel durch die Unterstützung von „Asynchronen -“ und Nichtsequentiellen Zugriffen“. Durch das Lesen von nicht zusammenhängenden Dateibereichen war das parallele Arbeiten an gemeinsamen Dateien/ Dateibereichen möglich. Wie geeignet POSIX für parallele Anwendungen mittlerweile ist, wird auf den folgenden Folien vorgestellt.

Quelle: [http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5\\_16\\_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg](http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5_16_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg)

## POSIX I/O Semantik (1)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- POSIX wurde ursprünglich für einen Prozess entworfen
- Änderungen sofort sichtbar von allen Prozessen (Konsistenzmodell)
- Konkurrierende Schreibzugriffe → Nur ein Zugriff ist erfolgreich zu einem Zeitpunkt (Atomaritätsmodell)
- Obige Modelle schränken Implementierung stark ein!
  - Schreibender Prozess benötigt immer exklusiven Zugriff auf Dateibereich (Unwissenheit der Prozesse)
  - Kein abgeschwächter Konsistenzmodus verfügbar

POSIX hat ein strenges Konsistenz- und Atomaritätsmodell, die festlegen, dass ein Schreibvorgang von einem Prozess sichtbar für alle anderen Prozesse ist, sobald die Write-Funktion zurückkehrt. Wenn konkurrierende Schreibvorgänge von zwei oder mehreren Prozessen in einer Datei überlappen, dann sind die geschriebenen Daten von einem der beiden Prozesse das Ergebnis. Entweder von Prozess 1 oder von 2 aber nicht von beiden. Diese Anforderungen schränken die Implementierung ein:

Ein Prozess muss exklusiven Zugriff für den Datenbereich erhalten (in Bytes) auch wenn kein anderer Prozess auf diesen Datenbereich zugreifen möchte. Dies ist nötig, da die Prozesse untereinander nicht wissen was die anderen Prozesse machen/ vorhaben. Die meisten wissenschaftlichen Anwendungen schreiben nicht konkurrierend in gleiche Datenbereiche. Würde ein optionales schwächeres Konsistenzmodus existieren, dann könnten Anwendungen diesen für höhere Performance auswählen.

Quelle: <http://pvs.informatik.uni-heidelberg.de/Teaching/HEAS-0708/heas-0607.pdf>

Quelle: [http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5\\_16\\_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg](http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5_16_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg)

## POSIX I/O Semantik (2)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffsemantik
- 6. Zusammenfassung

- Nichtsequentielle Bereichszugriffe in einer Datei
  - Benötigt von vielen parallelen Anwendungen
  - read, write Funktion von POSIX nur sequentiell und synchronisieren
    - Viele Aufrufe → sehr teuer aufgrund hoher I/O Latenz
  - readv, writev: nichtsequentielle Speicherzugriffe. (nicht Datei)
- POSIX-Dateizugriffs-Semantik = Sequentiell
  - POSIX kann nur sequentielle Konsistenz garantieren!
  - Operationen wirken in der Reihenfolge, in der sie aufgerufen werden
  - Nur Lesen/ Schreiben eines zusammenhängenden Datenbereichs zu einer Zeit möglich (atomar)

Viele parallele Anwendungen benötigen nichtsequentielle Bereichszugriffe in einer Datei, aufgrund dem Unterschied zwischen der Datenanordnung in der Datei und die Datenverteilung zwischen den Prozessen. Beispiel: Verteilter Array mit Zeilen-Ordnung. Verteilung findet aber per Block statt ( $x*y$ ).

Die Read- und Write-Funktionen von POSIX erlauben den Benutzern ein einzelnes zusammenhängendes Stück Daten zu einer Zeit zu lesen oder zu schreiben. Möchten die Benutzer nichtsequentielle (nicht zusammenhängende) Daten schreiben, dann müssen sie diese Funktionen mehrmals aufrufen. Diese Semantik ist wegen der hohen I/O Latenz sehr teuer. Hätte POSIX stattdessen eine ReadList- oder WriteList-Funktion, die es dem Benutzer erlauben würden ein nichtsequentielles Zugriffsmuster mit einer einzigen Funktion zu spezifizieren, dann könnten nichtsequentielle Anfragen mit Verfahren wie Data Sieving (wird im Kapitel MPI-IO erklärt) optimiert werden. Die POSIX Funktionen Readv und Writev erlauben zwar das Verwenden von nicht zusammenhängender Bereiche im Speicher aber leider nicht in der Datei!

POSIX kann nur sequentielle Konsistenz bei Dateizugriffen gewährleisten.

Quelle: <http://pvs.informatik.uni-heidelberg.de/Teaching/HEAS-0708/heas-0607.pdf>

Quelle: [http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5\\_16\\_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg](http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5_16_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg)

## POSIX I/O Semantik (3)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffsemantik
- 6. Zusammenfassung

#### ■ Gewünscht:

- read\_list, write\_list Funktionen zur Spezifikation eines nichtsequentiellen Datenzugriffsmuster mit einer einzigen Funktion und Optimierung des Zugriffs

#### ■ Funktion lio\_listio

- Erlaubt die Spezifikation einer Liste von Schreib- und Lesezugriffen und das Ausführen dieser mit einem Aufruf
- Aber: Jede einzelne Operation ist eine separate Anfrage → keine Optimierung möglich
- Operationen wirken in der Reihenfolge, in der sie aufgerufen wurden

- Problem: Momentan nutzen nur sehr wenige Betriebssysteme die asynchrone Funktionalität von POSIX aus.

Wie bereits besprochen wäre es optimal, wenn POSIX eine ReadList- oder WriteList-Funktion, die es dem Benutzer erlauben würden ein nichtsequentielles Zugriffsmuster mit einer einzigen Funktion zu spezifizieren, dann könnten nichtsequentielle Anfragen optimiert werden. In der Tat bietet POSIX eine solche asynchrone Funktion. Die Funktion lio\_listio erlaubt dem Benutzer eine Liste von Schreib- und Lesevorgänge zu spezifizieren und danach mit einem einzigen Aufruf zu starten. Jedoch wird jede Schreib- und Leseanfrage als eine einzelne Anfrage bearbeitet. Dies verhindert die Optimierung. Es können somit keine nicht zusammenhängende Zugriffe zusammengefasst und wenn möglich zu Zusammenhängenden gemacht werden. Momentan nutzen nur sehr wenige Betriebssysteme die asynchrone Funktionalität von POSIX aus. Auf der folgenden Folie wird ein Ansatz für eine breitere Unterstützung des asynchronen Ansatzes vorgestellt.

Quelle: [http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5\\_16\\_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg](http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5_16_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg)

# Asynchronous I/O (AIO) Support for Linux

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Asynchrone Erweiterung für POSIX
- Seit Linux Kernel 2.6 integriert
- Verkürzt die Wartezeit der Anwendung auf I/O Operationen
- Alle AIO Operationen werden durch eine Struktur ,
- AIO bietet an:
  - Asynchrones Lesen/Schreiben
  - das Abfragen der Stati von AIO Operationen
  - das Synchronisieren von AIO Operationen: Ermöglicht Konsistenz
  - das Abbrechen von AIO Operationen
  - Optimierung durch Angabe von Tuningparametern (Threads-, Anfragenanzahl)

© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

16

## AIO:

Der POSIX.1b Standard definiert eine neue Ansammlung an I/O Operationen (asynchrone Erweiterung für POSIX), die die Wartezeit auf I/O einer Anwendung beträchtlich reduzieren können. Diese neue Funktionen (integriert seit Kernel 2.6) erlauben einem Programm eine oder mehrere I/O Operationen zu initiieren und sofort danach mit der normalen Arbeit weiterzumachen während die I/O Operationen parallel dazu ausgeführt werden. Die Funktionen sind Teil der realtime Bibliothek `librt`.

Alle AIO Operationen arbeiten auf Dateien, die im Voraus geöffnet wurden. Die asynchrone I/O Operationen werden durch eine Struktur `aiocb` (*AIO control block*) kontrolliert.

AIO unterstützt:

- Asynchrones Lesen/Schreiben: `aioread`, `aiowrite`
- das Abfragen der Stati von AIO Operationen: `aioread_error` (0,1,EINPROGRESS), `aioread_return` (read,write,fsync res)
- das Synchronisieren von AIO Operationen: `aioread_fsync`, `aioread_suspend`
- das Abbrechen von AIO Operationen; `aioread_cancel`,
- Optimierung durch Funktion `aioread_init` und Struktur `aioread_t`

## Abfragen der Stati von AIO Operationen:

Gewährleistung von Konsistenz durch `aioread_fsync`, `aioread_suspend` (`aioread_error`). Der Umgang mit asynchronen Operationen setzt voraus, dass man einen konsistenten Status erreichen kann. Das würde bei AIO bedeuten, dass ein Prozess wissen möchte, ob eine bestimmte Anfrage oder eine Gruppe von Anfragen beendet wurde. Dies kann erreicht werden, indem man auf eine Benachrichtigung durch das System wartet, nachdem die Operation beendet wurde. Dies würde aber zu einer Ressourcenverschwendung führen. Stattdessen definiert POSIX.1b zwei Funktionen (`aioread_error`, `aioread_return`), die bei der Erhaltung der Konsistenz helfen.

## PAIOL: (nicht auf der Folie erwähnt)

Momentan wird auch an einer weiteren API gearbeitet, der POSIX Asynchronous I/O for Linux (PAIOL). Dies ist eine konforme API zur nativen AIO API des Linux Kernels 2.6 und setzt kein Patching des Kernels voraus. Das Ziel dieser API ist ein besseres Mapping der AIO auf POSIX Operationen. PAIOL besteht aus fünf Patches. Weitere Informationen finden Sie unter <http://www.bullopen-source.org/posix/>

Quelle: [http://www.gnu.org/software/libc/manual/html\\_node/Asynchronous-I\\_002fO.html#Asynchronous-I\\_002fO](http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html#Asynchronous-I_002fO)

Quelle: <http://lse.sourceforge.net/io/aio.html>

Quelle: <http://lwn.net/Articles/145365/>

Quelle: <http://www.bullopen-source.org/posix/>

## POSIX I/O Semantik (3)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

### ■ Metadaten

- Viele Funktionen (s. Notizteil)
- Schlüssel-Wert-Paare
- Werden mitkopiert beim Kopieren einer Datei
- Verknüpfung mit Verzeichnissen und Inodes

### ■ Dateien

- Explizites Anlegen und Löschen möglich

### ■ Verzeichnisse

- Explizites Anlegen und Löschen möglich

POSIX definiert erweiterte Attribute (Metadaten) für Dateien. Die Attribute sind Schlüssel-Wert Paare. Der Schlüssel ist ein String, der mit ‚\0‘ terminiert. Der Wert ist ein binärer mit fester Länge. Wenn eine Datei kopiert wird, dann werden auch die dazugehörigen Metadaten mitkopiert. Metadaten können auch an Verzeichnisse (ohne Vererbung) und Inodes (Indexeinträge, die alle Attribute einer Datei zusammenfasst) angehängt werden.

### Einige POSIX Metadateien – Funktionen:

- `ssize_t getxattr(const char *path, const char *key, void *value, size_t size);`
  - Gibt Schlüsselwert zurück
- `int setxattr(const char *path, const char *key, const void *value, size_t size, int flags);`
  - Setzt Schlüsselwert
- `ssize_t listxattr(const char *path, char *list, size_t size);`
  - Gibt Schlüssel als \0-terminated Strings zurück
- `int removexattr(const char path, const char *key);`
  - Löschen eines Schlüssels
- `size_t findfiles(const char *pattern, const char *query, char *list, size_t size);`
  - Gibt Liste der Dateien, die einem Suchmuster oder einer Schlüsselabfrage entsprechen zurück

POSIX erlaubt das explizite Anlegen und Löschen einer Datei und von Verzeichnissen ohne anschließende Operationen, die mit der Erzeugung oder dem Löschen in Zusammenhang stehen müssen. Der Programmierer kann somit eine Datei/ ein Verzeichnis gezielt mit einer Create-Funktion erstellen und nicht über einen Umweg (Programmierer schreibt etwas in die Datei. Datei wird dabei implizit erzeugt).

Quelle: <http://pvs.informatik.uni-heidelberg.de/Teaching/HEAS-0708/heas-0607.pdf>

Quelle: [http://lcg.web.cern.ch/LCG/PEB/arda/public\\_docs/MetadataGAG20040906.pdf](http://lcg.web.cern.ch/LCG/PEB/arda/public_docs/MetadataGAG20040906.pdf)

## POSIX I/O Semantik - Zusammenfassung

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- POSIX ist konzipiert für den Zugriff auf eine Datei durch einen Prozess
- Keine Möglichkeiten eines optimierten parallelen Zugriffs, da Funktionalität der I/O Operationen nicht ausreicht.
- Keine Optimierungen möglich, da keine Informationen über kooperierende Prozesse an POSIX weitergegeben werden.
- POSIX muss immer Datenbereiche sperren
- Schreib- und Lesezugriffe synchronisieren sofort
- Die Portabilität und Optimierung von paralleler I/O kann mit POSIX nicht erreicht werden.
- AIO ist eine sinnvolle Erweiterung aber noch nicht optimal

POSIX ist ursprünglich konzipiert für den Zugriff auf eine Datei durch einen Prozess erst später wurde POSIX für parallelen I/O erweitert. POSIX bietet keine Möglichkeiten eines optimierten parallelen Zugriffs, da Funktionalität der I/O Operationen nicht ausreicht. Die Portabilität und Optimierung von paralleler I/O kann mit POSIX nicht erreicht werden. Keine Optimierungen der Zugriffe möglich, da keine Informationen über kooperierende Prozesse an POSIX weitergegeben werden. Somit lassen sich keine Operationen zu einer gemeinsamen zusammenfassen. Z.B. ist kein gemeinsamer Lesevorgang möglich, der mit einem Aufruf alle benötigten Dateien liest und dann an die Prozesse verteilt. Stattdessen muss jeder Prozess seine Daten selber lesen/ schreiben. POSIX muss immer Datenbereiche sperren um Konsistenz garantieren zu können. Schreib- und Lesezugriffe synchronisieren sofort, d.h. nach dem Zugriff sind sofort die Änderungen für jeden Prozess sichtbar. Z.B: write() & read() funktioniert übers File-Locking (write-lock, read-lock,..). AIO ist eine sinnvolle Erweiterung aber noch nicht optimal, da nicht überall vorhanden und zum Teil werden die Funktionen noch ineffizient gemappt.

# MPI – Message Passing Interface

## Semantik

1. Einführung
2. Semantik der gemeinsam. Nutzung
3. POSIX I/O Semantik
- 4. MPI-IO Semantik
5. Zugriffssemantik
6. Zusammenfassung

- Vorangetrieben von MPI-Forum
- 1995 MPI Standard (nur Kommunikation)
- Stellt Programmierschnittstelle (API) dar
- Beschreibt Informationsaustausch zw. Prozessen
- Meistens in Parallelcomputern mit verteiltem Speicher
- Semantik ist sprachunabhängig
- MPI-2 (1997) ergänzt MPI um:
  - Explizite Operationen für gemeinsamen Speicher
  - Unterstützung durch das Betriebssystem für z.B. unterbrechungsgesteuerte Kommunikation
  - Explizite Unterstützung zur Prozessverwaltung
  - Parallele Ein-/Ausgabe

MPI – Message Passing Interface – ist ein Standard zur nachrichtenbasierten Kommunikation in der parallelen Datenverarbeitung. Er wurde 1992 vom Message Passing Forum (<http://www.mpi-forum.org>), 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 Quellcode-kompatibel sein, das 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 [Semantik ist sprachunabhängig] (Fortran77, C/C++, Java etc.) sicherstellen soll. Daher schreibt MPI nicht alle Implementierungsaspekte vor. Die Semantik ist sprachunabhängig (MPI Anwendungen können in unterschiedlichen Sprachen geschrieben sein). 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 verteiltem Speicher. Funktionen zur parallelen Ein-/Ausgabe sind ebenfalls Bestandteil des MPI-Standards. Ist ein Standard, der den Informationsaustausch zwischen Prozessen beschreibt, die gemeinschaftlich an der Lösung einer Aufgabe arbeiten.

### •MPI enthält:

- Punkt zu Punkt- (MPI\_Send, MPI\_Recv), Direkte- (Broadcasting) und Indirekte-Kommunikation (Gather, Scatter usw.)
- Kollektive Operationen
- Prozessgruppen
- Kommunikationskontexte
- Prozesstopologien
- Abfragefunktionen zur Programmumgebung
- Profiling-Schnittstelle

# MPI-IO

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffsemantik
- 6. Zusammenfassung

- Teil des MPI-2 Standards
- Ziel: Anwendungen mit hohem I/O Anteil portabel und effizient zu machen.
- Für den Zugriff durch mehrere Prozesse bei verteiltem Speicher konzipiert ist
- Ermöglichung von asynchronem I/O
- Semantik-Analogie:
  - Daten schreiben/ lesen  $\leftrightarrow$  Nachrichten senden/ empfangen
- Wichtigste Konzepte der MPI-IO
  - Dateizeiger
  - Dateisichten
  - Nichtsequentielle Zugriffe
  - Kollektive Aufrufe
  - Hints

Parallele I/O Performanz wurde lange durch den Mangel an passenden, standardisierten und portablen Schnittstellen für parallelen I/O behindert. Vor MPI-IO benutzen Programmierer die POSIX API, die nicht besonders gut für parallele Anwendungen geeignet ist. Aus diesem Grund wurde von dem MPI Forum eine Schnittstelle für parallelen I/O definiert, der Teil des MPI-2 Standards wurde. Diese Schnittstelle ist allgemein als MPI-IO bekannt. MPI-IO 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. Der Standard versteht darunter den Zugriff auf Sekundärspeicher. Das Ziel von MPI-IO ist Anwendungen mit hohem I/O Anteil portabel und effizienter zu machen. D.h. kürzere Wartezeiten bei I/O Operationen und bessere Skalierbarkeit. Während POSIX für den Dateizugriff eines Prozesses auf eine Datei konzipiert ist, stellt MPI-IO die Grundlagen für den Zugriff mehrerer Prozesse auf eine Datei dar. Da MPI-IO eine Zwischenschicht zwischen der verteilten Applikation und dem Dateisystem darstellt, können die I/O-Zugriffe optimiert werden, was den Programmablauf beschleunigen kann. Zudem ermöglicht es asynchronen I/O und gemeinsamen parallelen Dateizugriff mehrerer Prozesse. Die Semantik von MPI-IO ist analog zum Nachrichtenaustausch von Prozessen. Das Schreiben- und Lesen von Daten funktioniert analog zu dem Senden und Empfangen von Nachrichten bei MPI1.

Die Hauptmerkmale von MPI I/O sind:

- Dateizeiger: Wer greift wann auf welche Byte der Datei zu?
- Dateisichten: Partitionierung der Daten einer Datei für verschiedene Prozesse
- Nichtsequentielle (nicht zusammenhängende) Zugriffe [und Optimierung dieser!]
- Kollektive Aufrufe: von mehreren Prozesse gemeinsame Datenzugriffe
- Hints: Hinweise vom Benutzer an die MPI Implementierung
  - Kontrolle der physikalischen Lage der Daten
  - Es ist zudem möglich in MPI Anwendungen die POSIX-Schnittstelle für den Zugriff auf das Dateisystem zu verwenden. (vor MPI-2 die Regel)

## MPI-IO Semantik (1) - Dateikonsistenz

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

#### ■ Filehandle:

- erzeugt von Laufzeitumgebung bei gemeins. Öffnen
- von allen beteiligten Prozessen für Datei-Operationen verwendet
- berechnet alle Zugriffsadressen für die Datei (Konsistenz)

#### ■ Sequentiell:

- Befehlsfolge wird so nacheinander abgearbeitet, wie es für den Programmablauf erforderlich ist.
- Jeder Zugriff erscheint atomar, obwohl die exakte Abwicklung der Zugriffe unspezifiziert ist. Alle Operationen werden seriell, zeitlich nacheinander, abgearbeitet.

#### ■ MPI unterstützt drei Level der Konsistenz:

- Sequentielle Konsistenz (alle Zugriffe gleiches Filehandle)
- Sequentielle Konsistenz (alle Zugriffe Filehandles im „Atomic Mode“ erstellt wurden.
- Benutzerverwaltete Konsistenz: Anwender muss sich selbst um die korrekte Reihenfolge der Befehlsausführung und die Synchronisation der Datei kümmern. (MPI\_FILE\_SYNC)

Das Filehandle, das bei einer gemeinsamen Öffnen-Operation von der Laufzeitumgebung erzeugt wird, sorgt für Datenkonsistenz in MPI. Das Filehandle ist eine Referenz auf die geöffnete Datei und wird von allen beteiligten Prozessen für Datei-Operationen verwendet. Alle Dateioperationen laufen über diesen Filehandle, der alle Zugriffsadressen für die Datei berechnet und somit Inkonsistenzen direkter Adressierung vorbeugt.

Drei Level der Konsistenz werden unterstützt:

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

Unter sequentieller Konsistenz versteht man in MPI, dass 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 muss der Anwender selbst für die korrekte Reihenfolge der Befehlsausführung und die Synchronisation (`MPI_File_Sync`) der Datei sorgen. Auf Dateien kann sequentiell oder wahlfrei zugegriffen werden. Sequentieller Zugriff ist sinnvoll bei Datenträgern wie Magnetbändern. Wenn eine Dateioperation komplett abgeschlossen wurde, ist sie sicher. Datenpuffer bei nicht blockierenden Funktionen sollten erst wieder verwendet werden, wenn deren Bearbeitung abgeschlossen ist. <http://www.mpi-forum.org/docs/mpi-20-html/node206.htm#Node206>

## MPI-IO Semantik (2) - Dateikonsistenz

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Im Gegensatz zu POSIX kann die MPI Semantik für Zugriffe, die im Konflikt stehen, nicht sequentielle Konsistenz garantieren.
  - Lösungen:
    - Aktivieren von „Atomic mode“
    - Verwenden von kollektiver Funktion `MPI_FILE_SYNC`, bei konkurrierenden Schreibsequenzen.
    - Algorithmus umschreiben.
  - um sequentielle Konsistenz zu garantieren.
- Atomic mode ist kollektiv, d.h. jeder Prozess in der Gruppe muss identische Werte für `fh` und `flag` übergeben.

Im Gegensatz zu POSIX kann die MPI Semantik für Zugriffe, die im Konflikt stehen, nicht sequentielle Konsistenz garantieren. Um sequentielle Konsistenz zu gewährleisten muss der „**Atomic mode**“ aktiviert werden, der mit der Funktion `MPI_FILE_SET_ATOMICITY`, gesetzt wird. Bei aktiviertem „Atomic mode“ werden alle Operationen grundsätzlich seriell abgearbeitet. Der „Atomic mode“ ist kollektiv, d.h. jeder Prozess in der Gruppe muss identische Werte für `fh` und `flag` übergeben, wenn sie die gemeinsame Datei bearbeiten.

Eine andere Lösung wäre das Verwenden der kollektiven Funktion **MPI\_File\_sync** um Konsistenz bei konkurrierenden Schreibzugriffen zu gewährleisten. Die konkurrierenden Schreibzugriffe werden dann synchronisiert. Zudem könnte der Programmierer auch den **Algorithmus** des Programms umschreiben, so dass nur noch nichtsequentielle Zugriffe stattfinden.

## MPI-IO Semantik (3)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

### ■ Dateizugriff

- Spezifikation des Datenlayouts (selbstdef. Datentypen) im Speicher oder in Dateien
- Direkter Dateizugriff
  - Blockierende, nicht blockierende Funktionen
- Kollektive Zugriffe, Unabhängige Zugriffe
- Schwächere Konsistenz- und Atomaritätsmodelle als Standardvorgabe

### ■ Verzeichnisse

- Es existieren keine MPI Verzeichnis-Operationen.
- Nur implizites Anlegen und Löschen beim Erzeugen und Schließen

### ■ Metadaten

- Keine Operationen für Metadaten

MPI-IO erlaubt die Spezifikation des **Datenlayouts** im Speicher oder in Dateien. Der Benutzer kann somit eigene Datentypen definieren und kann zudem bestimmen, wie diese physikalisch verteilt werden sollen. Der **Dateizugriff** kann sequentiell oder wahlfrei stattfinden. Darunter versteht man das Bearbeiten von Dateien, die nacheinander auf einem Speichermedium liegen oder das Springen auf dem Speichermedium (das Lesen/ Schreiben von nicht zusammenhängenden Daten). Der Zugriff auf eine Datei kann blockierend oder nicht blockierend stattfinden. **Nicht blockierend**: Eine Prozedur ist nicht blockierend, wenn die Prozedur zurückkehren kann, bevor die Operation beendet wurde und der Benutzer die verwendeten Ressourcen (bspw. Puffer) wieder verwenden darf/ kann. Eine Operation ist beendet, wenn der Benutzer wieder die Ressourcen verwenden darf und alle Ausgabepuffer aktualisiert wurden. Eine Kommunikation ist beendet, wenn alle beteiligten Operationen beendet wurden. Verbesserte Effizienz durch Überlappung von Berechnung und Kommunikation! **Blockierend**: Eine Prozedur ist blockierend, wenn der Benutzer die in dem Aufruf verwendeten Ressourcen nach der Prozedurrückkehr wieder verwenden darf.

MPI-IO bietet zudem **kollektiven I/O** an. 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 Dateisicht (View), das nach dem zuletzt verwendeten liegt. MPI-IO bietet im Gegensatz zu POSIX schwächere Konsistenz- und **Atomaritätsmodelle** als Standardvorgabe an. D.h. diese sind die Standardmodelle in der MPI-IO Konfiguration. Der Benutzer kann jedoch bei Bedarf aber auch andere, z.B. stärkere (strengere) Modelle wählen! Gerade von parallelen Anwendungen wird ein schwächeres Konsistenz- und Atomaritätsmodell benötigt. MPI-IO hat noch weitere Eigenschaften, die es für parallele Anwendungen interessant machen. Zum Beispiel Unterstützung für nichtsequentielle Zugriffe und kollektivem I/O. MPI-IO ist mittlerweile die am meisten genutzten API für die Implementierung von ‚high-level‘ I/O-Bibliotheken, wie HDF-5 und PnetCDF. Viele Anwendungen benutzen MPI-IO auch direkt! MPI-IO unterstützt nicht das explizite Anlegen und Löschen einer Datei und von Verzeichnissen ohne anschließende Operationen, die mit der Erzeugung oder dem Löschen in Zusammenhang stehen. Der Programmierer kann somit nicht eine Datei/ ein Verzeichnis gezielt mit einer Create-Funktion erstellen.

Für Metadaten stehen keine Operationen zur Verfügung!



## MPI-IO – Dateizeiger

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Positionierung innerhalb der Datei
- Shared Filepointer
  - wird von allen Prozessen verwendet
  - Prozess verschiebt Dateizeiger für alle Prozesse
  - Kann durch Seek-Funktionen versetzt werden
- Individual Filepointer
  - wird für jeden Prozess separat verwaltet.
  - Wenn eine Sicht def. wurde, dann nur Zugriff auf markierte Bereiche der Datei
  - Kann durch Seek-Funktionen versetzt werden.
- Expliziter Offset
  - Zugriff auf eine beliebige Stelle in einer Datei unabhängig von Filepointers

**Dateizeiger:** Wer greift wann auf welche Byte der Datei zu?

Ein Differenzierungskriterium stellt die Positionierung innerhalb der Datei dar. In MPI wird zwischen zwei verschiedenen Filepointern unterschieden.

Der **Shared Filepointer** (gemeinsamer Dateizeiger) wird von allen Prozessen verwendet. Greift ein Prozess unter Verwendung dieses Dateizeigers auf die Datei zu, so verschiebt sich der Dateizeiger für alle Prozesse in der Prozessgruppe.

Der **Individual Filepointer** (prozessbezogener Dateizeiger) hingegen wird für jeden Prozess separat verwaltet. Durch die Wahl einer entsprechenden Sicht kann bewirkt werden, dass jeder Prozess fortlaufend in der Datei schreibt oder liest, dabei aber nie in den Bereich eines anderen Prozesses eindringt. MPI sorgt implizit dafür, dass Funktionen, die den individual Dateizeiger benutzen, nur auf die markierten Bereiche in der Datei zugreifen, wenn eine Dateisicht definiert wurde. Der Prozess sieht somit eine virtuelle Datei dar, die tatsächlich über eine reelle, existierende Datei verteilt sein kann. Beide Filepointer, sowohl gemeinsame als auch individuelle Dateizeiger können durch entsprechende Seek-Funktionen versetzt werden.

Die dritte Positionierungsmöglichkeit stellt ein **expliziter Offset** (Angabe einer Position in der Datei) dar. Er kann genutzt werden, um unabhängig von den Positionen der Dateizeiger auf eine beliebige Stelle der Datei zuzugreifen.

## MPI-IO – MPI\_Hints

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

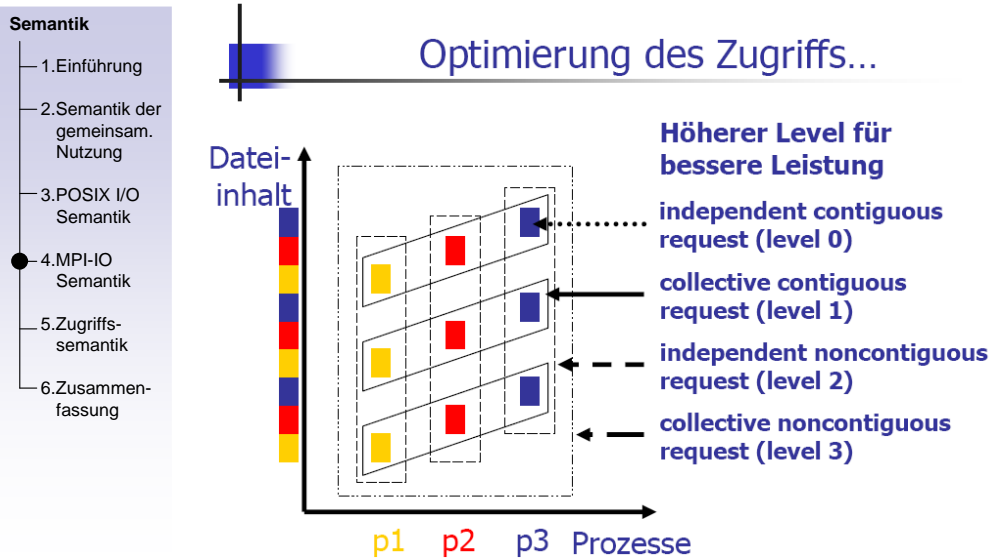
- Informationen an die MPI-Implementierung
- Beispiele zur Leistungssteigerung:
  - Puffergröße
  - Anzahl und Position der beteiligten I/O Prozesse
  - „Data Sieving“ und „Two Phase“ Techniken
  - Zugriffsart auf eine Datei
- Alle Optimierungsmechanismen werden durch Hints kontrolliert
  - data sieving, two-phase I/O, list I/O, datatype I/O
- Zusätzliche Hints erlauben ROMIO Anpassung an Zugriffsmuster wie:
  - nur kollektivem I/O, sequentieller vs. wahlfreier Zugriff, Abhängigkeiten zw. Dateien
- Hinweise sind immer optional, der Benutzer muss sie nicht angeben
- Die MPI Implementierung darf wiederum Hinweise beliebig ignorieren

### Hinweise (hints)

- Hinweise geben dem Nutzer die Möglichkeit, Informationen an die MPI-Implementierung durchzureichen
- Beispiele für Hinweise sind hier:
  - Anzahl der Festplatten, über die eine Datei verteilt werden soll (striping)
  - Breite der Streifen (stripsize)
  - Puffergröße
  - Anzahl und Position der beteiligten I/O Prozesse
  - „Data Sieving“ und „Two Phase“ Techniken
  - Zugriffsart auf eine Datei
- Alle Optimierungsmechanismen werden durch Hints kontrolliert (data sieving, two-phase I/O, list I/O, datatype I/O)
- Zusätzliche Hints erlauben ROMIO Anpassung an Zugriffsmuster wie: nur kollektivem I/O, sequentieller vs. wahlfreier Zugriff, Abhängigkeiten zw. Dateien
- Hinweise sind immer optional, der Benutzer muss sie nicht angeben
- Gleichzeitig darf eine Implementierung Hinweise beliebig ignorieren

Quelle: <http://pvs.informatik.uni-heidelberg.de/Teaching/HEAS-0708/heas-0607.pdf>

## MPI-IO Semantik – Optimierung des Zugriffs



© 2007 Stefan Becker – Universität Heidelberg – WS 07/08

27

Da MPI-IO eine Zwischenschicht zwischen der verteilten Applikation und dem Dateisystem darstellt, können die I/O-Zugriffe optimiert werden, was den Programmablauf beschleunigen kann.

Höherer Level = bessere Leistung per MPI

Level 0: unabhängiger zusammenhängender Zugriff

Level 1: kollektiver zusammenhängender Zugriff

Level 2: unabhängiger nicht zusammenhängender Zugriff

Level 3: kollektiver nicht zusammenhängender Zugriff

MPI ist in der Lage einen nicht zusammenhängenden Lese-/ Schreibzugriff zu optimieren.

## MPI-IO Semantik – Data Sieving

### Semantik

- 1. Einführung
- 2. Semantik der gemeins. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Optimiert nicht zusammenhängende Zugriffe (Level 2 und 3).
- Implementierung innerhalb der AIDO-Schicht.
- Vorgang:
  - Holen von großem zusammenhängendem Stück
  - Abbilden in Benutzeradressraum
  - davon angeforderte nicht zusammenhängende Bereiche lesen/schreiben
  - Verwerfen der nicht benötigten Teile beim Lesen. Beim Schreiben: Zurückschreiben des kompletten Puffers!
- Puffergröße kann per MPI\_Hints verändert werden!

**Data Sieving** ist eine Technik, mit der ROMIO die Bearbeitung von nicht zusammenhängenden Zugriffen (Level 2) optimiert. Dazu wird ein großes zusammenhängendes Stück, das alle einzelne Datenbereichsanforderungen enthält gelesen und anschließend die nicht benötigten Dateibereiche wieder verworfen. ROMIO deaktiviert Data Sieving bei zu großen und/oder vielen Löchern! Beim **Lesen** werden alle Bytes der Anfrage mit nicht zusammenhängenden Bereichen. Danach werden die nicht benötigten Teile verworfen. Gegebenenfalls mehrfach lesen und Data Sieving ausführen bis Anfrage abgewickelt ist. Beim **Schreiben** wird ein zusammenhängender Bereich von der Platte in den temporären Puffer gelesen. Anschließend werden die Benutzerdaten an passende Stellen kopiert. Danach wird der gesamte temporäre Puffer zurück geschrieben.

Probleme:

- Speicherlastig, da großes Stück im temporären Speicher
- Evtl. mehrfaches Lesen und Data Sieving, da temporäre Speichergröße nicht ausreichend für komplette Abfrage
- Sperren des kompletten Bereichs beim Schreiben!
  - Gesperrter Bereich zu groß → Behinderung anderer Prozesse
  - Gesperrter Bereich zu klein → Leistungsverluste

Data Sieving bei **ROMIO deaktiviert, wenn** es zu viele bzw. zu große Löcher sind. Entscheidung aufgrund heuristischer Werte.

## MPI-IO Semantik – Collective I/O

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffs-semantik
- 6. Zusammenfassung

- Optimierungen sind möglich, wenn Zugriffsmuster aller beteiligten Prozesse bekannt und wenn diese nicht zusammenhängend sind (Level 2 und 3).
- ROMIO
  - Ist Bibliothek und legt Wert auf hohe Portabilität
  - Deshalb: Realisierung Collective I/O auf Client-Ebene
- Integriert in Level 3 Anfragen (Kollektive nicht zusammenhängende Zugriffe)
- Kollektive zusammenhängende Zugriffe (Level 1) haben oft zu wenig Infos zur Optimierung und werden als Unabhängige zusammenhängende Zugriffe (Level 0) behandelt.

Wenn die Zugriffsmuster aller beteiligten Prozesse bekannt sind, dann sind weitere Optimierungen möglich. Das Konzept der Optimierung wird **Collective I/O** genannt.

Am besten können nichtsequentielle Zugriffe auf eine Datei von mehreren Prozessen optimiert werden. Die in ROMIO implementierte Variante des kollektiven I/O ist eine so genannte „Two Phase I/O“. Diese Variante wird auf der nächsten Folie vorgestellt. ROMIO benutzt kollektive I/O, wenn User einen Level 3 MPI-IO Zugriff startet.

## MPI-IO Semantik – Two Phase I/O

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Optimiert kollektive Aufrufe (Level 1 und 3)
- Situation: Daten eines Prozesses sind nicht zusammenhängend in Datei abgelegt.
- Problem: Alles einzeln zu lesen wäre ineffizient
- Prozess:
  - 1. Phase: Jeder Prozess liest Datenanteil
  - 2. Phase: Jeder Prozess sendet Daten an ihre Besitzer
- Im Gegensatz zu „Data Sieving“ kein Wegwerfen der nicht benötigten Daten!
- Zwei Parameter für Optimierungen
  - Anzahl der beteiligten I/O Prozesse
  - Maximale Größe des temp. Puffers für jeden Prozess

Die in ROMIO implementierte Variante des kollektiven I/O wird **Two Phase I/O** genannt. ROMIO benutzt kollektive I/O, wenn der Benutzer einen Level 3 MPI-IO Zugriff startet.

Folgende Situation wäre gegeben: Daten eines Prozesses sind nicht zusammenhängend in Datei abgelegt. Die Prozesse benötigen alle Daten aus der Datei. Alles einzeln zu lesen wäre sehr ineffizient, da jedes mal ein Lese-/Schreibvorgang gestartet werden müsste. **In der ersten Phase** liest jeder Prozess seinen entsprechenden Anteil der Daten von der Platte in den temporären Puffer. **In der zweiten Phase** werden die Daten mittels MPI-Kommunikation zu den richtigen Prozessen verteilt. Alle Prozesse senden Daten an ihre entsprechenden Besitzer. Das Verfahren ähnelt Data Sieving. Es werden aber keine Daten verworfen. ROMIO benutzt **zwei Parameter zur Optimierung von Two Phase I/O**, die vom Benutzer eingestellt werden können:

- Anzahl der beteiligten I/O Prozesse die auf eine Datei zugreifen können
- Wahl der maximalen Größe des temporären Puffers für jeden Prozess.

### Skalierbarkeit

- Wenn nicht alle Prozesse effizient I/O machen können, dann soll nur ein Teil der Prozesse I/O ausführen.
- An der Verteilung der Daten sind aber alle Prozesse beteiligt.

# MPI-IO Semantik Zusammenfassung

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Leistungsgewinn durch
  - Kollektive Aufrufe (Two Phase I/O)
  - Asynchronen I/O
  - Nichtsequentiellen Zugriffen (Data Sieving)
  - Spezifikation des Datenlayouts im Speicher/ in Dateien
  - Zugriff auf eine Datei mit mehreren Prozessen (versch. Sichten)
- Portabilität
  - ADIO
  - Automatische Konvertierung von Daten zw. heterogenen Systemen
- Benutzbarkeit
  - Einfacherer Zugriff durch abgeleitete Datentypen bei irregulären Daten
  - Automatische Konvertierung von Daten zw. heterogenen Systemen
  - Dateisichten zur Partitionierung und nichtseq. Zugriff
  - Dateizeiger (individueller/ gemeinsamer)
  - Kollektive Aufrufe und Nichtsequentielle Zugriffe
  - Hinweise

MPI 2 stellt parallele I/O auf hoher Abstraktionsstufe sicher. Durch eine überschaubare Zahl von Funktionen, die alle Dateioperationen abdecken, ist ein schmales Interface sichergestellt, das auch implementierbar ist. Die Möglichkeit, benutzerdefinierte Datentypen zu realisieren, schafft Flexibilität in der Entwicklung speziell angepasster Implementierungen für eigene Anwendungen. (Kontrolle über physikalisches File-Layout auf Speichermedien). Durch die Definition von Sichten (jeder Prozess hat seine eigene Dateisicht) kann gewährleistet werden, dass eine Datei unter den zugreifenden Prozessen aufgeteilt ist und jeder Prozess nur in seinem Bereich liest und schreibt, ohne dass dieser sich um die Zuordnung kümmern muss. Andere Funktionen ermöglichen den direkten Dateizugriff, so dass jeder Prozess auf jede Speicherposition zugreifen kann. Wir positionieren explizit, mit individuellen Dateizeigern oder einem gemeinsamen Dateizeiger. MPI-IO hält neben dem gewohnten Zugriff, auch Funktionen bereit, die die Zugriffsreihenfolge der Prozesse garantieren. Die Mechanismen "nichtsequentieller Datenzugriff" und "Kollektiver I/O" in MPI dienen dem Programmierkomfort und geben zusätzlich die Möglichkeit, der Implementierung Hinweise über Zusammenhänge zukommen zu lassen und somit die Leistung der parallelen Anwendungen deutlich zu verbessern. Der I/O ist in MPI2 analog zur Kommunikation in MP1 definiert.

# Zugriffssemantiken (1)

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- Zugriffssemantik: Bedeutung eines Dateizugriffs
- Dateizeiger
  - POSIX
    - Lesen/ Schreiben verschiebt Zeiger
    - Funktionen zur Zeigermanipulation vorhanden
    - Problem: Skalierbarkeit:
      - Aktionen eines Prozesses für alle wirksam
  - MPI-IO
    - Ein Zeiger für alle Prozesse
    - Ein Zeiger pro Prozess
- Dateisicht
  - POSIX
    - sieht alle Bytes einer Datei
  - MPI-IO
    - Prozess sieht alle Bytes (globale Sicht)
    - Prozesse sehen disjunkte Teile der Datei (Views)

Zugriffssemantik: Bedeutung eines ändernden Dateizugriffs eines Prozesses an einer gemeinsamen Datei für die anderen beteiligten Prozesse.

## Zugriffssemantiken (2)

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

### ■ Datenmenge

- POSIX
  - Verarbeitet einzelnen Block zusammenhängender Bytes
  - Verarbeitet mehrere Listen und Blöcke
- MPI-IO
  - Verarbeitet nicht zusammenhängende Bereiche
    - Definition von speziellen Datentypen
    - Optimierung im Cluster Dateisystem
  - Menge der Bereiche ist oft wieder zusammenhängend (kollektiver Aufruf)

### ■ Koordination der zugreifenden Prozesse

- POSIX
  - ???
- MPI-IO
  - unabhängig voneinander
  - kollektiver Zugriff

## Schlusswort

### Semantik

- 1. Einführung
- 2. Semantik der gemeinsam. Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

- MPI-2 enthält umfassendes Regelwerk, das parallelen I/O auf hoher Abstraktionsstufe sicherstellt
- POSIX ist für den Dateizugriff eines Prozesses auf eine Datei konzipiert. MPI-IO stellt dagegen die Grundlagen für den Zugriff mehrerer Prozesse auf eine Datei dar
- MPI-IO leistungsfähiger als POSIX bei parallelem I/O
- POSIX Problematik:
  - Parallelität
  - Skalierbarkeit (Metadaten)
- MPI-IO ist benutzerfreundlicher
- MPI-IO ist portabel
  - Sprachunabhängige Semantik
  - Automatische Datentypkonvertierung

MPI-IO kann mehr bei parallelen Anwendungen mehr leisten und verfolgt zudem ein anderes Ziel als POSIX. Während POSIX für den Dateizugriff eines Prozesses auf eine Datei konzipiert ist, stellt MPI-IO die Grundlagen für den Zugriff mehrerer Prozesse auf eine Datei dar. Da MPI-IO eine Zwischenschicht zwischen der verteilten Applikation und dem Dateisystem darstellt, können die I/O-Zugriffe optimiert werden, was den Programmablauf beschleunigen kann. Die POSIX Prozesse wissen nicht, was gerade der andere Prozess vorhat. Daher keine kollektive Optimierungen möglich. MPI-IO Anwendungen sind portabel, da sie sprachunabhängig sind. D.h. eine MPI-Anwendung die bspw. in Java geschrieben wurde kann ohne Probleme mit einer MPI-IO Anwendung in C kommunizieren. Der Datenaustausch zwischen heterogenen Systemen ist kein Problem, da eine Konvertierung der Daten automatisch passiert.

# Quellen

## Semantik

- 1. Einführung
- 2. Semantik der gemeinsamen Nutzung
- 3. POSIX I/O Semantik
- 4. MPI-IO Semantik
- 5. Zugriffssemantik
- 6. Zusammenfassung

Siehe Notizteil



### -Semantik:

-<http://www.bullhost.de/s/semantik.html>

### -Semantik der gemeinsamen Nutzung:

-[http://www.fbi.h-da.de/~a.schuetter/Vorlesungen/VerteilteSysteme/Skript/6\\_VerteilteDateisysteme/VerteilteDateisysteme.pdf](http://www.fbi.h-da.de/~a.schuetter/Vorlesungen/VerteilteSysteme/Skript/6_VerteilteDateisysteme/VerteilteDateisysteme.pdf)

-<http://www.ipd.uni-karlsruhe.de/Tichy/uploads/fohlen/126/Cluster12Dateisysteme.pdf>

### -POSIX:

-[http://www.gnu.org/software/libc/manual/html\\_node/Asynchronous-I\\_002fO.html](http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html)

-<http://lse.sourceforge.net/io/aio.html>

-<http://www.opengroup.org/onlinepubs/009695399/toc.htm>

-[http://lcg.web.cern.ch/LCG/PEB/arda/public\\_docs/MetadataGAG20040906.pdf](http://lcg.web.cern.ch/LCG/PEB/arda/public_docs/MetadataGAG20040906.pdf)

-[http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5\\_16\\_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg](http://www.iop.org/EJ/article/1742-6596/16/1/069/jpconf5_16_069.pdf?request-id=Wj4hG8x73BGTPAe-2wi7Kg)

-[http://www.gnu.org/software/libc/manual/html\\_node/Asynchronous-I\\_002fO.html#Asynchronous-I\\_002fO](http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html#Asynchronous-I_002fO)

-<http://lse.sourceforge.net/io/aio.html>

-<http://lwn.net/Articles/145365/>

•<http://www.bullopensource.org/posix/>

### -MPI:

-<http://pvs.informatik.uni-heidelberg.de/Theses/2004-sadleder-bsc.pdf>

-[http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/mpi\\_fkt\\_liste.html](http://www.tu-chemnitz.de/informatik/RA/projects/mpihelp/mpi_fkt_liste.html)

-<http://sdm.lbl.gov/sdmcenter/pub/SD.rob.sdm-sept-2002-all-hands-status.ppt>

-[http://www2.inf.fh-brs.de/~rberre2m/lehre/ws0102/vps2/Seminar\\_MPI\\_IO\\_Seichter.pdf](http://www2.inf.fh-brs.de/~rberre2m/lehre/ws0102/vps2/Seminar_MPI_IO_Seichter.pdf)

-[http://www2.inf.fh-brs.de/~rberre2m/lehre/ws0102/vps2/Seminar\\_Impl\\_IO\\_Koch.pdf](http://www2.inf.fh-brs.de/~rberre2m/lehre/ws0102/vps2/Seminar_Impl_IO_Koch.pdf)

-<http://www-fp.mcs.anl.gov/~toonon/Papers/mpi-io-atomic-ccgrid-2005.pdf>

-[http://www.imd.uni-rostock.de/ma/rb230/stud\\_essays/Diplomarbeit.pdf](http://www.imd.uni-rostock.de/ma/rb230/stud_essays/Diplomarbeit.pdf)

-<http://www.mpi-forum.org/docs/mpi-20-html/node206.htm#Node206>

### -Für alle Kapitel:

-<http://www.wikipedia.de>

-<http://pvs.informatik.uni-heidelberg.de/Teaching/DASY-07/nguyen.pdf>

-<http://pvs.informatik.uni-heidelberg.de/Teaching/HEAS-0708/heas-0607.pdf>