uMMAP-IO: User-level Memory-mapped I/O for HPC

Nicolaus Johannes Leopold

Universität Hamburg

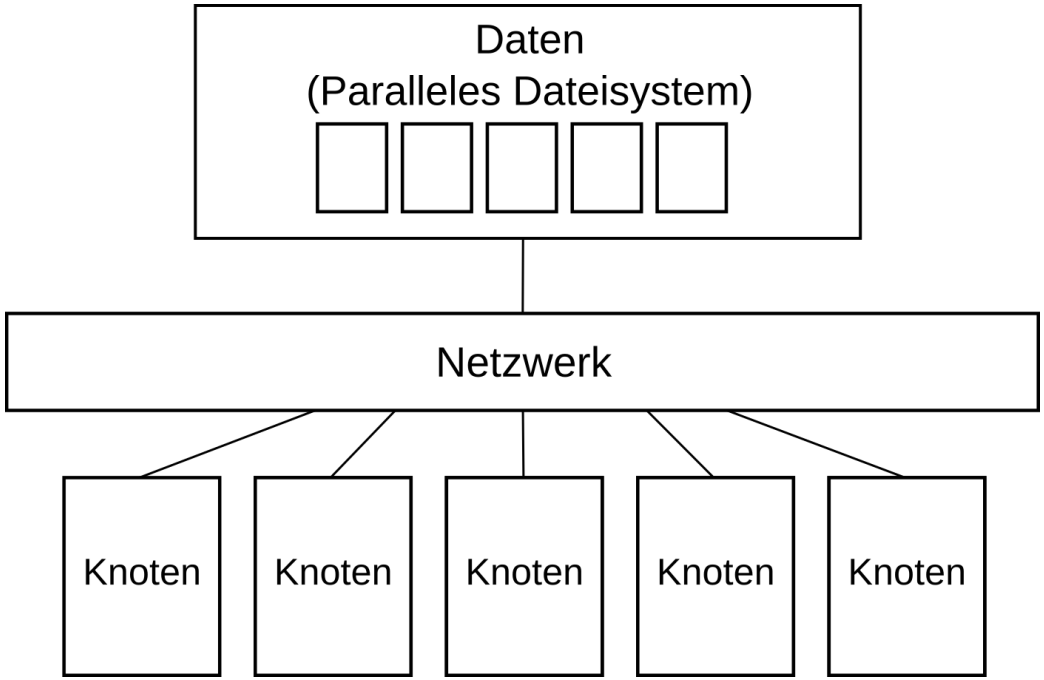
Seminar Wissenschaftliches Rechnen

30. November, 2021

Gliederung

Was ist die Problemstellung?

- Cluster von separaten Rechnerknoten für das Hochleistungsrechnen bzw. Wissenschaftliches Rechnen
- Ergebnisse müssen geladen/gespeichert und ggf. synchronisiert werden
- Stellt oft einen Flaschenhals für Performance dar

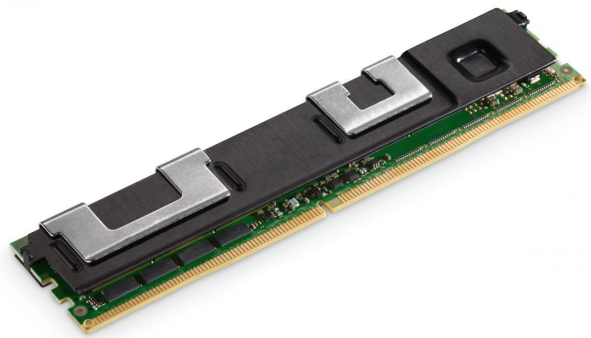


Mögliche Vorgehensweisen

- Separate nicht zentral gespeicherte Dateien
  - Keine Synchronisation / eigene Implementierung benötigt
  - Datenmengen eventuell zu groß
- Eigene(r) Rechnerknoten für Sammeln, Speichern und Synchronisieren von Daten
  - Vergleichsweise langsam, wegen Netzwerk
- Hardware: Prozessnahe Speichertechnologien, zum Zwischenspeichern
  - Leider oft Zugriff über andere Code-Interfaces
  - Synchronisation nach-wie-vor nötig, aber der Knoten kann sich anderem widmen

Vokabular: Burst Buffer

- Eine schnelle, lokale Zwischenspeicherebene
- Performante Brücke zwischen den Rechenknoten und dem parallelen Dateisystem
- Füllen die Lücke zwischen Rechen- und I/O-Geschwindigkeit
- Oft ein Array von schnellen Speichermedien wie SSDs oder NVRAM

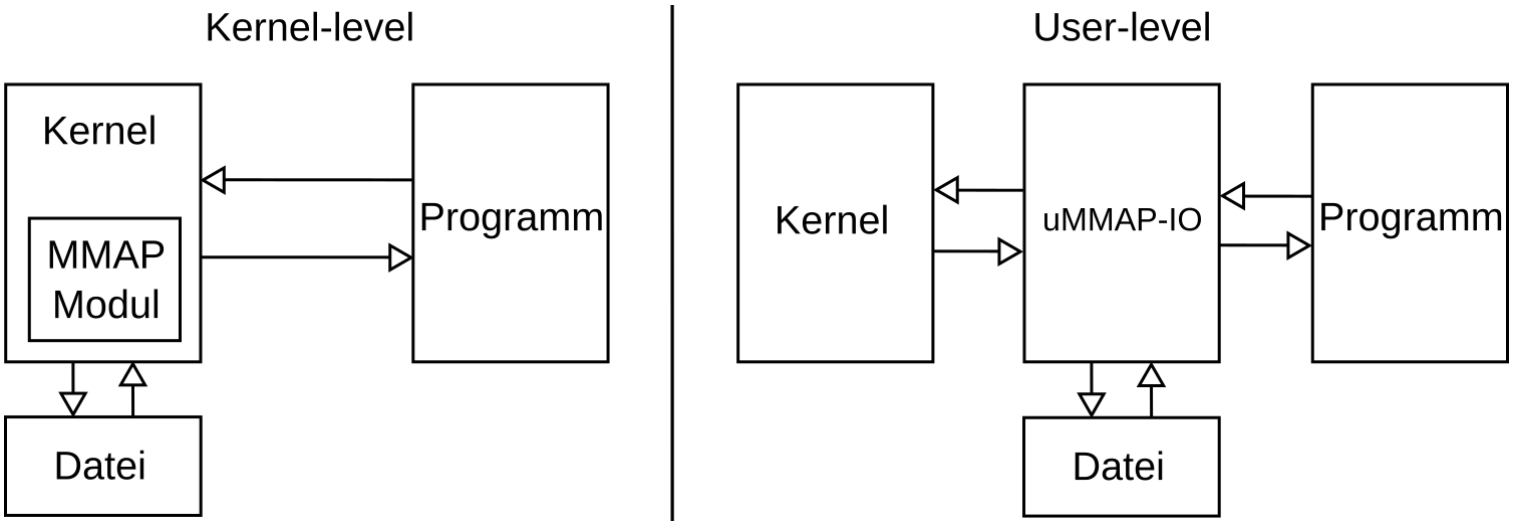


NVRAM (Intel® Optane™ Persistent Memory)

Quelle: http://www.nextgenio.eu/news/nextgenio-hardware-has-landed

Vokabular: User-level Memory-mapped I/O?

- Memory-mapped I/O (Input/Output):
  - Benutzt den gleichen Adressraum für I/O wie für RAM
  - Inhalt des Dateisystems über Speicherzugriffsoperationen les- und schreibbar
- User-level:
  - Nicht im Kernel des Betriebssystems, sondern auf Programmebene implementiert



Was ist uMMAP-IO?

- Bibliothek zur Erleichterung der Nutzung unterschiedlicher Speichertechnologien im Hochleistungsrechnen
- Abstrahiert diverse Speichermedien auf Nutzung wie Hauptspeicher (RAM)
- Ermöglicht Tausch des Speichermediums ohne große Code-Änderungen
- Flexibler als Memory Mapping des Betriebssystems
- Effiziente Synchronisation zwischen Hauptspeicher und persistenten Speicher

Vorteile von uMMAP-IO

- Laden / speichern mit minimalen Codeänderungen
- Fast optimal skalierbar bis zu 2048 Prozessen
- 20 - 50% degradierte Performance verglichen mit konventionellen Speicherallokationen (bis zu 8192 Prozessen)
- 5 - 10x bessere Performance als Standard OS Memory Mapping, 20x mit Burst Buffer
- Out-of-core Unterstützung (größere Datenmengen als vorhandener Speicher)

Warum User-level Memory-Mapping?

- Veränderbare Einstellungen für jede Allokation
  - Segmentgröße, Flush Interval, Adressen zu unterschiedlichen Dateien remappen
- Systemstabilität wird nicht beeinträchtigt.
- OS MMap-IO nicht lokal konfigurierbar, schwer änderbar auf Supercomputern
- Client-side buffering für parallele Dateisysteme
- Support für neuere Speichertechnologien

Library Design

- Abstraktion des persistenten Speichers zu laden/speichern
- Nutzbar über normale Speicherzugriffsmethoden (Array Notation, memset, etc.)
- Synchronisationsregeln für out-of-core Algorithmen
- Verschiedene Abschnitte einer Datei können von parallelen Prozessen bearbeitet werden
- Einfach erweiterbar auf neue Technologien durch einheitliche Schnittstelle
- Synchronisation von bereits geladenen Daten entweder explizit anforderbar oder implizit durch die Bibliothek

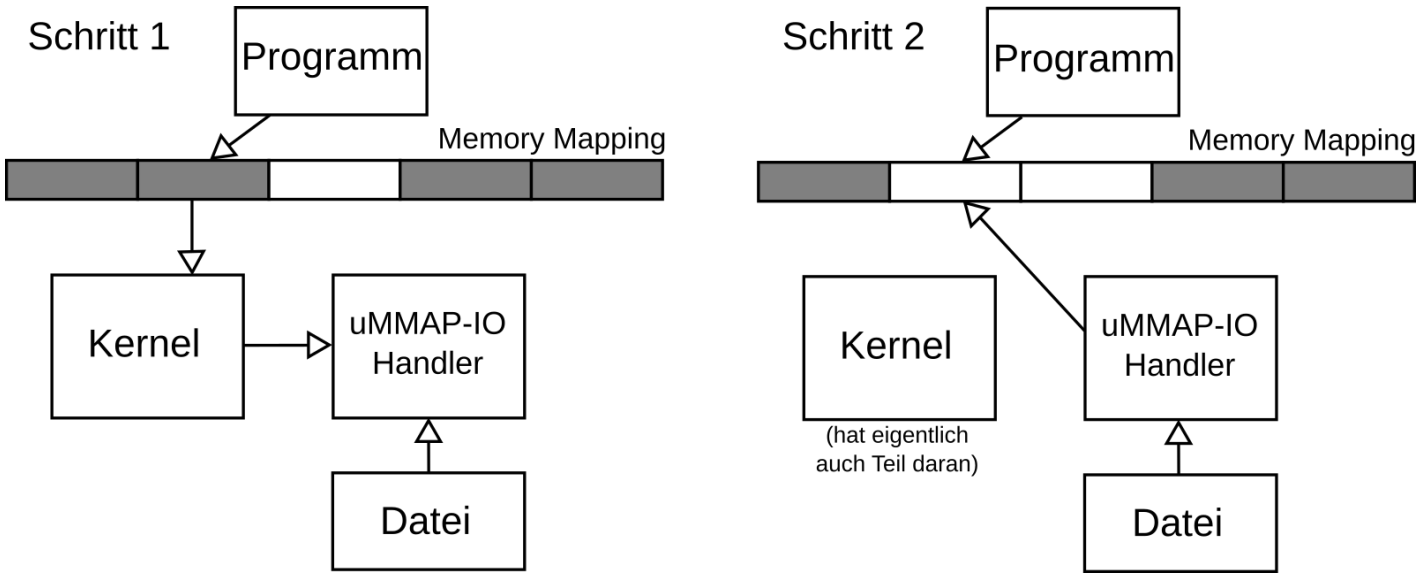
API

```
// Memory-mapped I/O-Allokation etablieren
int ummap(size_t size, size_t seg_size, int prot, int fd, off_t offset,
          int flush_interval, int read_file, int ptype, void **addr);

// Selektive Synchronisation von "dreckigen" Segmenten des übergebenen Mappings
int umsync(void *addr, int evict);

//Eine Memory-mapped Allokation umkonfigurieren
int umremap(void *old_addr, size_t size, int fd, off_t offset,
            int sync, void **new_addr);

//Memory-mapping wieder freigeben
int umunmap(void *addr, int sync);
```

Listing 1: Die uMMAP-IO API

Wie Funktioniert das Memory Mapping?

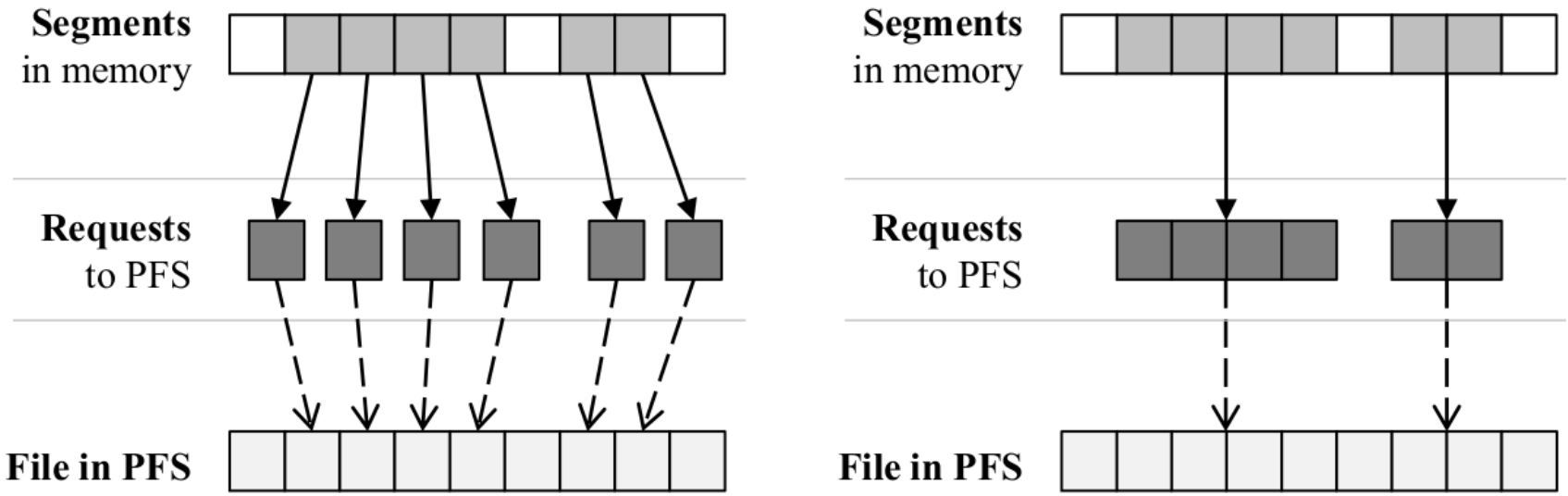
- Memory mapping wird von uMMAP-IO angefordert
- uMMAP-IO reserviert einen virtuellen Adressraum
- Teilt ihn nach Segmentgröße auf, markiert ihn als geschützt
- Registriert handler für Zugriff auf nicht geladenem Speicher
  - Dieser lädt den Inhalt aus der Datei in das angeforderte Segment



Wie Funktioniert das Memory Mapping?

- Auf Segmenten im Speicher wird bei weiterem Zugriff direkt gearbeitet
- uMMAP-IO gibt Segmente wieder aus dem Speicher frei, falls notwendig
  - Speichermangel, Überschreitung der konfigurierten Grenze für den Prozess
  - Algorithmus einstellbar:
    FIFO, LIFO, Least Recently Used, Pseudo-Zufall, Writes-In Reads-Out (FIFO oder LIFO)
- Synchronisiert mit Dateisystem
  - Entweder nach Anforderung (mit `umsync()`) oder in geregelten Zeitabschnitten
  - Individual oder Bulk (mehrere auf einmal), einstellbar

Synchronisation: Individual vs Bulk

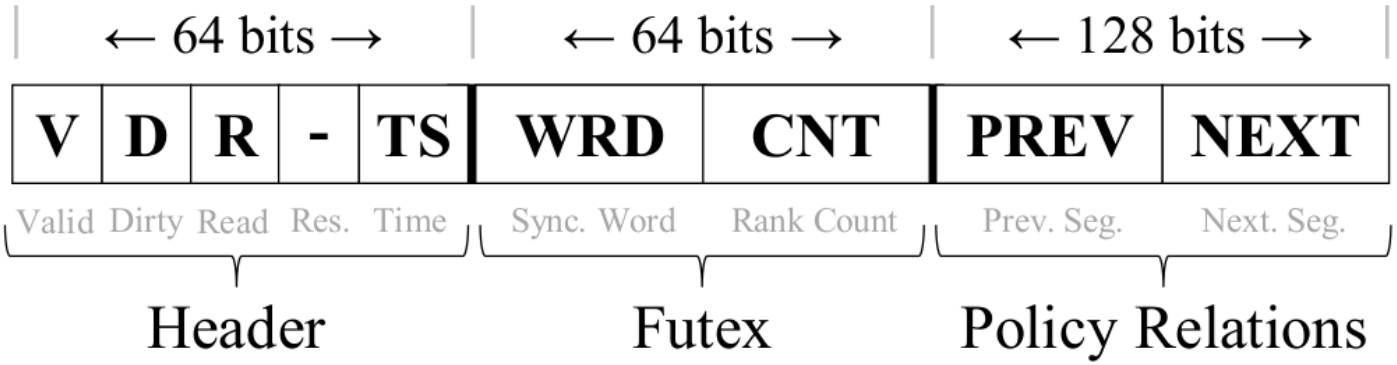


Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

- So oder so: Segment als Basiseinheit für Synchronisierung

Segmentstruktur

- Metadaten werden in einer Liste von Segmentstrukturen verwaltet:

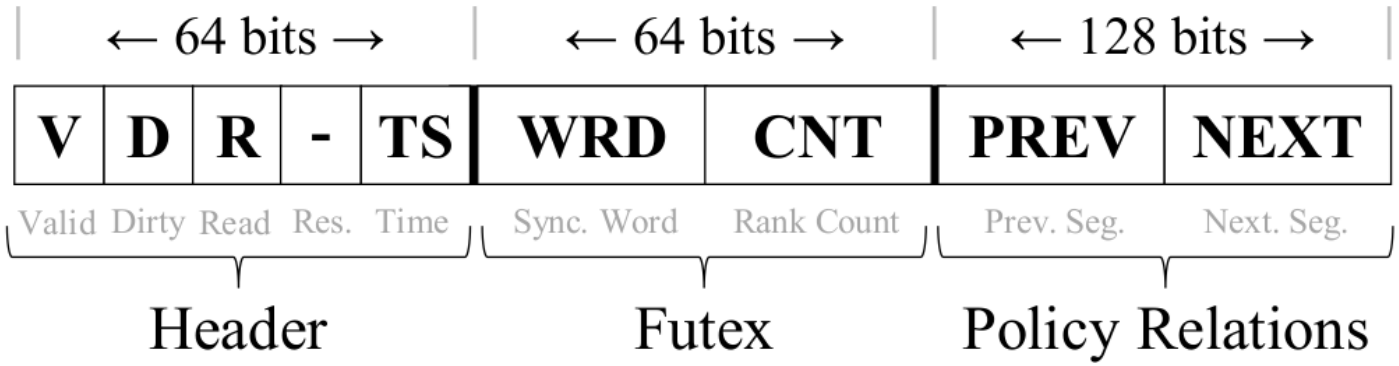


Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

- **V** - Valid - ist das Segment im Speicher?
- **D** - Dirty - nicht gespeicherte Änderungen?
- **R** - Read - Soll das Segment aus dem Speicher geladen werden?
- **TS** - Timestamp - Erste Änderung seit Synchronisation

Segmentstruktur

- Metadaten werden in einer Liste von Segmentstrukturen verwaltet:



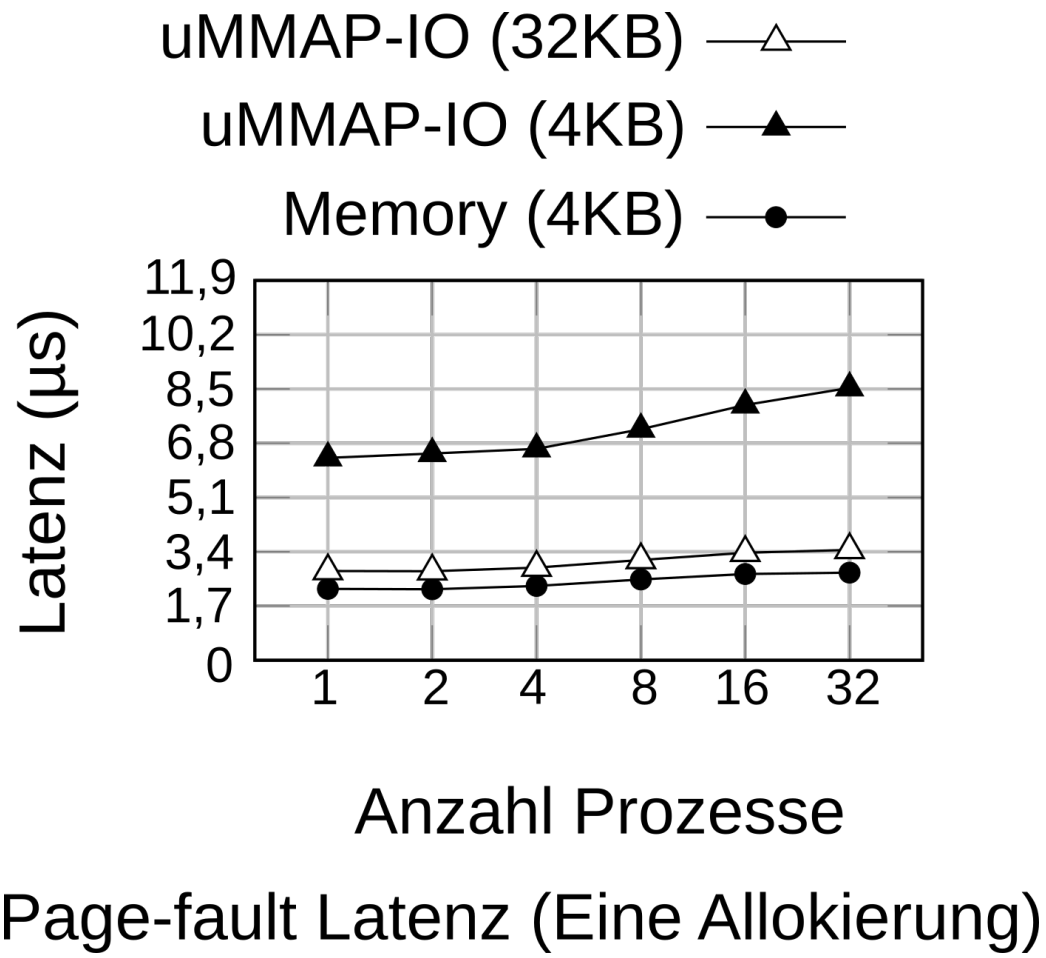
Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

- **Futex** - "Fast User-level Mutex" - Schließt Nebenläufigkeitsfehler aus
- **WRD** - Sync. Word - Ist der Futex frei?
- **CNT** - Rank-count - Wie viele Threads auf Freigabe des Futex warten
- **PREV, NEXT** - "Evict" Reihenfolge (als linked-list)

Benchmarks der Referenzimplementation ("Cori")

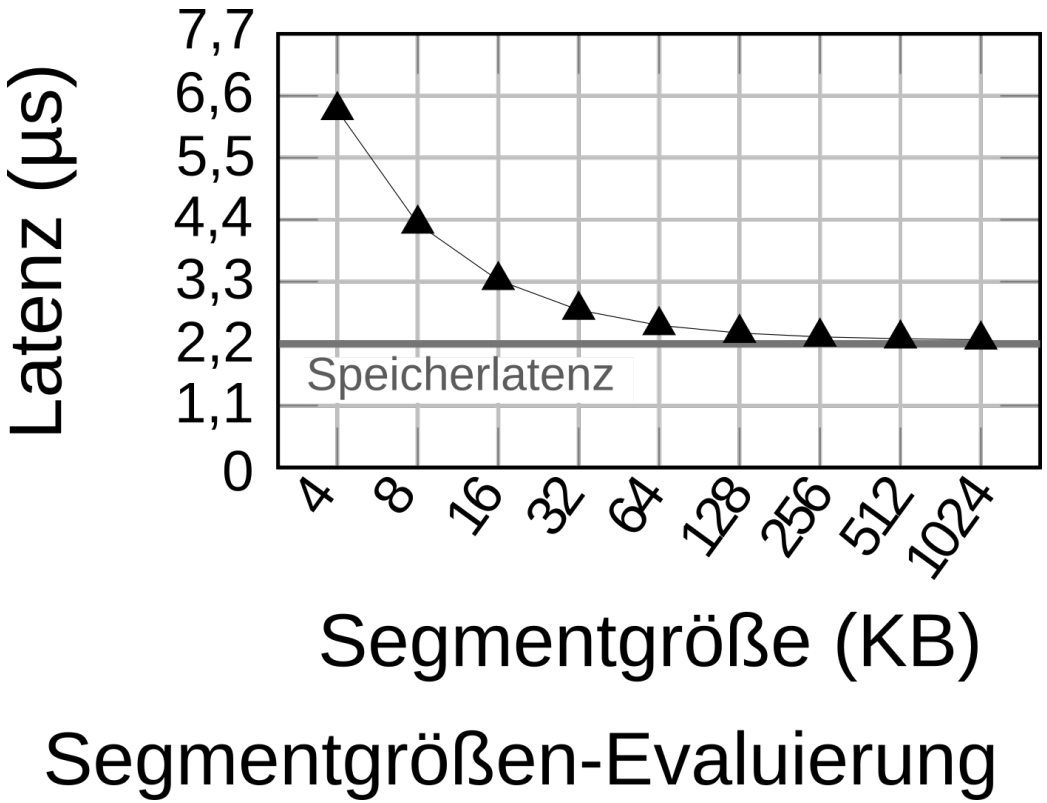
- https://github.com/sergiorg-kth/ummap-io
- National Energy Research Scientific Computing Center Supercomputer (**"Cori"**)
- Cray XC40 Supercomputer with 2388 nodes, dual 16-core Haswell E5-2698 v3 2.3GHz
- 128GB DRAM
- SSD als burst buffer
- Lustre Dateisystem

Benchmarks der Referenzimplementation (“Cori”)



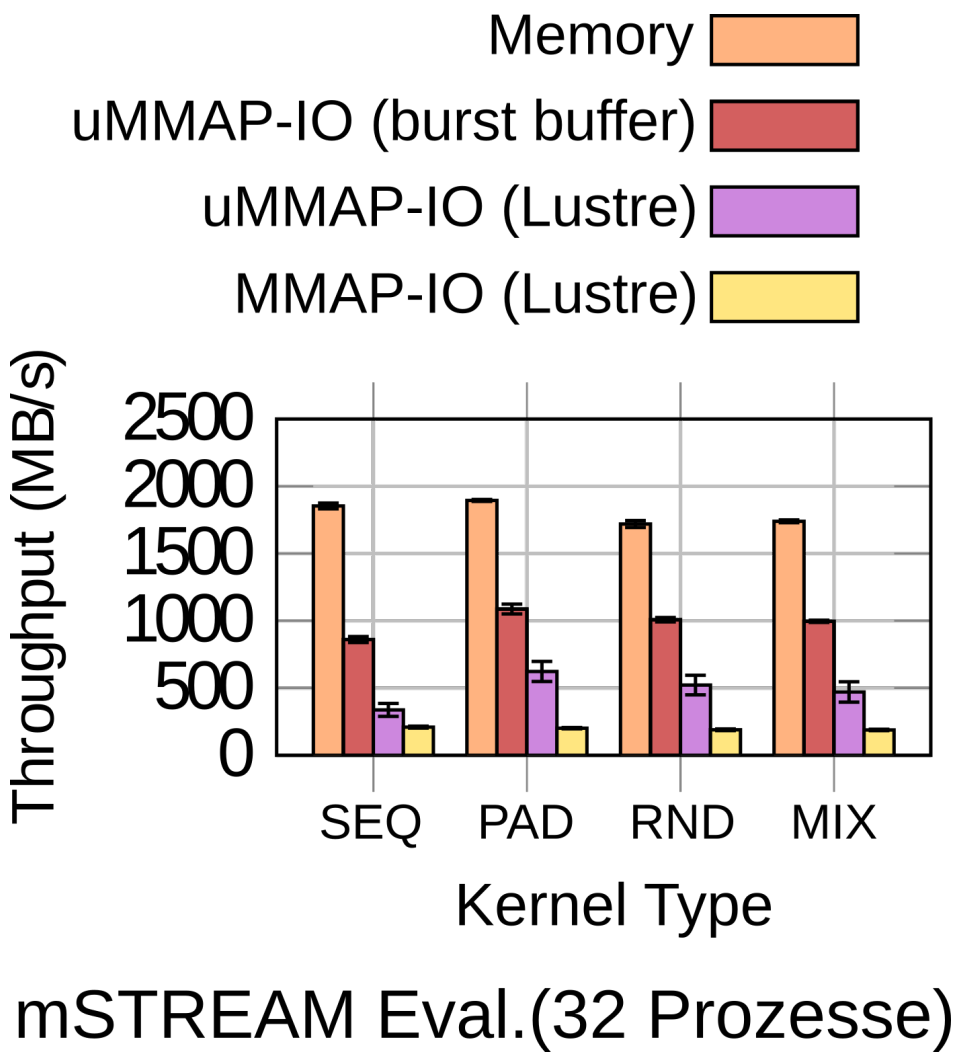
Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

Benchmarks der Referenzimplementation (“Cori”)



Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

Benchmarks der Referenzimplementation (“Cori”)

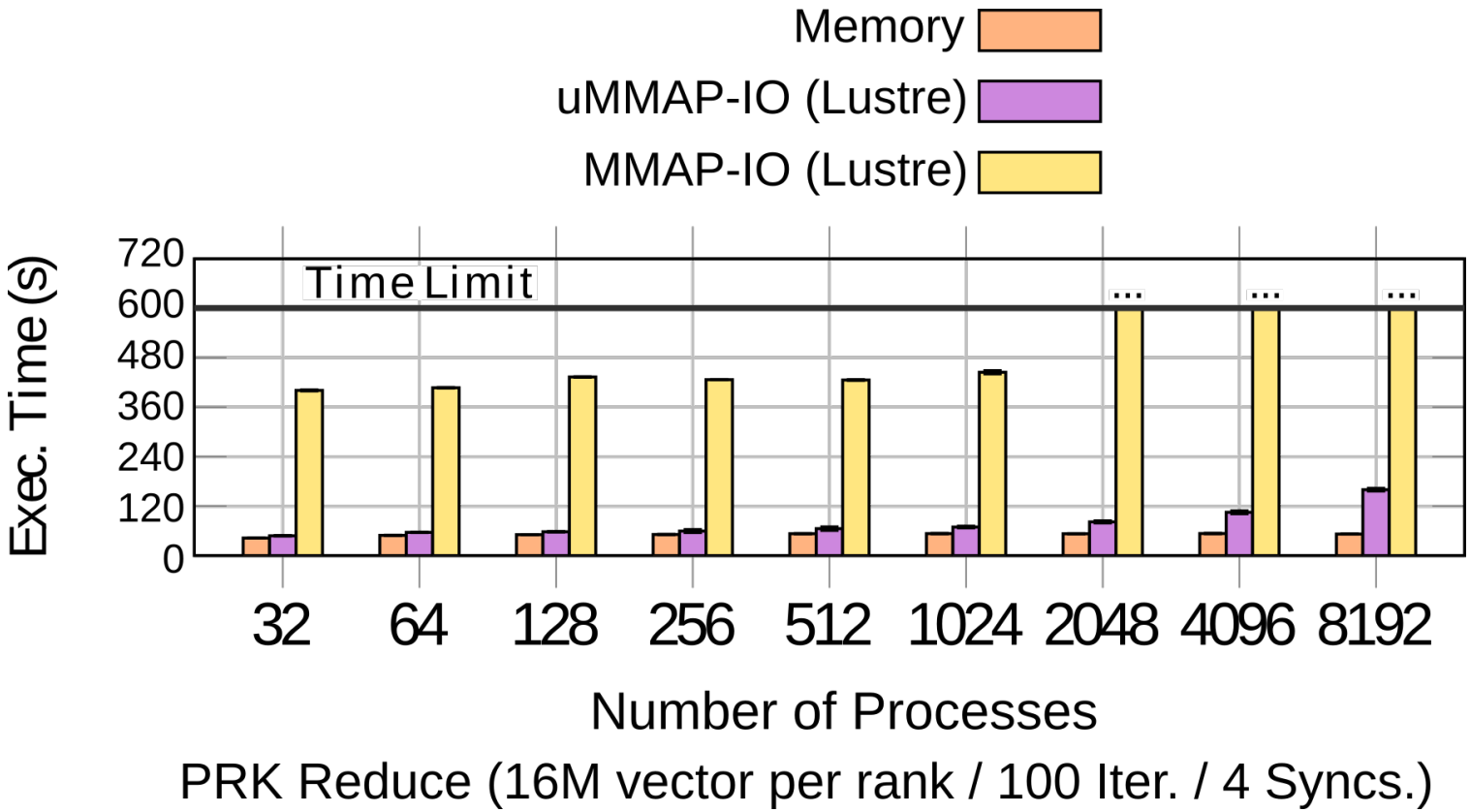


Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

Benchmarks der Referenzimplementation ("Beskow")

- https://github.com/sergiorg-kth/ummap-io
- KTH Royal Institute of Technology Supercomputer (**"Beskow"**)
- Cray XC40 Supercomputer with 1676 nodes, dual 16-core Haswell E5-2698v3 2.3GHz
- 64GB DRAM
- Ohne burst buffer
- Lustre Dateisystem

Benchmarks der Referenzimplementation ("Beskow")



Quelle: uMMAP-IO: User-level Memory-mapped I/O for HPC

Zusammenfassung

- uMMAP-IO ist eine Bibliothek für das Memory Mapping von verschiedenen Speichermedien
- Sie ist Performant und auf parallele Systeme ausgerichtet
- Bietet eine vereinigte Schnittstelle für verschiedene Arten von Speichermedien
- Ermöglicht dadurch einfaches testen und ändern von Systemkonfigurationen
- Konfigurierbar und Erweiterbar

Referenzen

2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)

uMMAP-IO: User-level Memory-mapped I/O for HPC

Github

uMMAP-IO: User-level Memory-mapped I/O for HPC

https://github.com/sergiorg-hpc/ummap-io

Wikipedia

Memory-mapped I/O

https://en.wikipedia.org/wiki/Memory-mapped_I/O

Wikipedia

Burst buffer

https://en.wikipedia.org/wiki/Burst_buffer

Referenzen

Wikipedia

Message Passing Interface

https://en.wikipedia.org/wiki/Message_Passing_Interface

Wikipedia

GPFS

https://en.wikipedia.org/wiki/GPFS

Wikipedia

Page fault

https://en.wikipedia.org/wiki/Page_fault

Wikipedia

External memory algorithm

https://en.wikipedia.org/wiki/External_memory_algorithm