Toward IOVerbs with the Memory-Centric Storage for Exascale (MCSE) Project

25.05.2023

HPC IODC Workshop

Table of contents

1 Introduction

2 Semantics

3 Conclusions

Goals

- Exploration of alternative storage APIs
- Development of IO-Verbs to express elementary parallel I/O

⇒ Uniform interface for memory and persistent storage

- Exploit different storage classes flexible in HPC-workflows (campaigns)
- Development of the Memory-Centric Storage System (MCS2)

⇒ Based upon IML and MCSE

- Migration path for existing applications

The Project

Key information

- 3-years funding by BMBF
- Funding Reference: 16ME0663K
- Start date: 10/2022

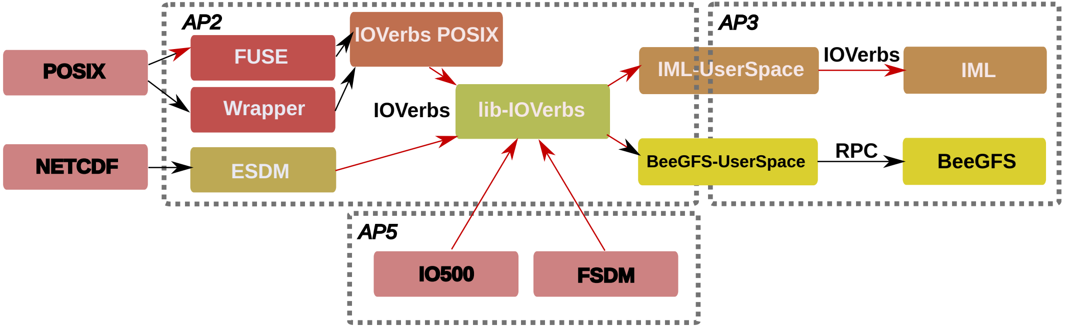
Partners

- University Göttingen (Coordinator)
- Fraunhofer ITWM
- Deutsches Zentrum für Luft- und Raumfahrt e.V.
- ThinkParQ

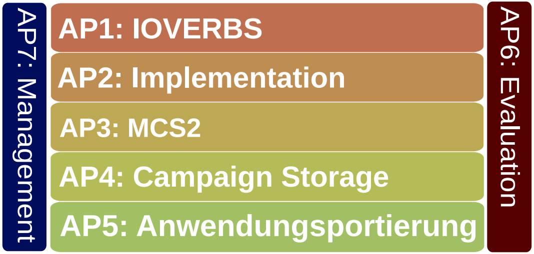
What are IOVerbs?

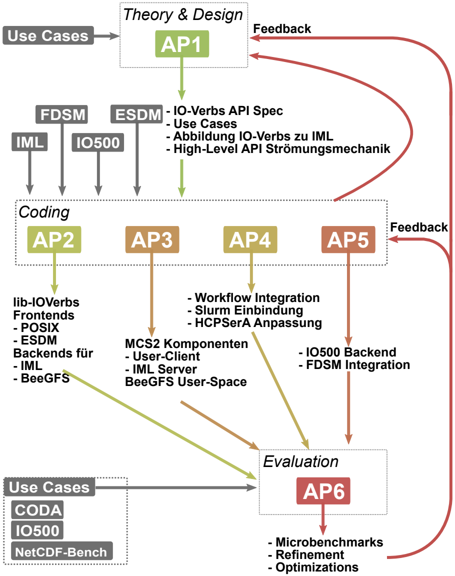
- Analogon to IBVerbs but for I/O
- Describe I/O on the low-level and enabling parallel I/O
- Have a clear semantics
- Independent of storage system
- Allows high-level libraries to be layered on top
- Should be useful on local systems
- Maybe a prospective replacement of API in Linux VFS

IO Layers


AP2
POSIX
FUSE
Wrapper
IOVerbs POSIX
NETCDF
ESDM
IOVerbs
lib-IOVerbs
AP3
IML-UserSpace
IOVerbs
IML
BeeGFS-UserSpace
RPC
BeeGFS
AP5
IO500
FSDM

Work Plan


AP7: Management
AP1: IOVERBS
AP2: Implementation
AP3: MCS2
AP4: Campaign Storage
AP5: Anwendungsportierung
AP6: Evaluation


Theory & Design
Use Cases
AP1
Feedback
FDSM
ESDM
IML
IO500
- IO-Verbs API Spec
- Use Cases
- Abbildung IO-Verbs zu IML
- High-Level API Strömungsmechanik
Coding
AP2
AP3
AP4
AP5
Feedback
lib-IOVerbs
Frontends
- POSIX
- ESDM
Backends für
- IML
- BeeGFS
MCS2 Komponenten
- User-Client
- IML Server
BeeGFS User-Space
- Workflow Integration
- Slurm Einbindung
- HCPSerA Anpassung
- IO500 Backend
- FDSM Integration
Evaluation
AP6
Use Cases
CODA
IO500
NetCDF-Bench
- Microbenchmarks
- Refinement
- Optimizations

Outline

1 Introduction

2 Semantics

3 Conclusions

Aspects for Semantics (Excerpt)

- Visibility: when will another proc see changes
- Durability: when is data written back
- Concurrency: what happens with concurrent access
- Sychronous/Asynchronous API: App can proceed during I/O – background
- In-transit: Allow computation during I/O for pre-/post-processing
- Transparency: no hidden optimization
- Optimistic vs. pessimistic approaches

$\Rightarrow$ There are many ways to describe semantics in the API

Motivating Example: Shallow Water Model

Facts about the model

- Stores data column-wise in memory
- Separates compute phase and IO phase[1]

Existing NetCDF code for IO phase limits performance

```
size_t start[] = {0, 0};
size_t count[] = {nY, 1};
for(unsigned int col = 0; col < nX; col++) {
  start[1] = col; //select col (dim "x")
  nc_put_vara_float(dataFile, i_ncVariable, start, count,
    &i_matrix[col+boundarySize[0]][boundarySize[2]]);
}
```

[1] DSLs will help to separate those phases

ESDM Code for the Application (that worked)

```c
int64_t offset[] = {(int64_t) timeStep, offsetY, offsetX};
int64_t size[] = {1, (int64_t) nY, (int64_t) nX};

esdm_wstream_float_t stream;
esdm_wstream_start(&stream, dset, 3, offset, size);
for(int y = 0; y < nY; y++) {
 for(int x = 0; x < nX; x++) {
   esdm_wstream_pack(stream, i_matrix
       [x + boundarySize[0]][boundarySize[2] + y])
   // this may trigger actual IO and postprocessing!
 }
}
esdm_wstream_commit(stream);
```

- Ultimately, using DSLs an IO phase could switch between compute and "stream output" to minimize memory pressure (and trigger initial post processing)

IO500 Example - IOR Easy Write

- Each process operates on an (independent) file

```
fd = open(path, O_CREAT)
offset = 0
for I in SegmentCount:
    write(fd, offset, Blocksize)
    offset += Blocksize
close(fd)
```

An alternative formulation

```
fd = iov_open_async(path) // we can proceed *before* file is opened
offset = 0
iov_write_limit(fd, SegmentCount * Blocksize) // organizes space - note: Stonewall
for I in SegmentCount:
    res = iov_write_excl_data_ready(fd, offset, Blocksize) // caching, no sync
    offset += Blocksize // also user must check for error
iov_wait_completion(fd)
iov_destroy(fd)
```

IO500 Example - IOR Hard Write

- Processes operate on a shared file - write strided

An alternative formulation

```
fd = iov_open_async_coll(path) // collective
// includes amount of data that this proc and all procs write
iov_write_coll_limit(fd, SegmentCount*Blocksize, nproc*SegmentCount*Blocksize)
for I in SegmentCount:
    res = iov_write_excl_data_ready(fd, offset, Blocksize) // exclusive: non-overlapping
iov_wait_completion(fd)
iov_destroy(fd)
```

IO500 Beispiel - MDTest Hard Write

- Each process creates files ... (suboptimal anyway)

```
for f in file: // the recursive truth is too complex ;-)
    fd = open(path/f, O_CREAT)
    write(fd, 0, Blocksize)
    close(fd)
```

An alternative formulation

```
iov_limit_objects(no of files locally) //system could create container/return limits
for f in file:
    fd = iov_open_async(path) // we can proceed *before* file is opened
    iov_write_limit(fd, Blocksize)
    iov_write_excl_data_ready(fd, 0, Blocksize)
    iov_destroy(fd)
    err = iov_check_state() // check async if there was an error so far, alternatively
        ↪ use iov_register_err_handler() before
    if err != IOV_NO_ERR invoke error()
iov_wait_completion_all()
```

Outline

1 Introduction

2 Semantics

3 Conclusions

Conclusions

- MCSE will systematically explore parallel I/O semantics
- MCSE will develop IO-verbs
  - Addressing metadata and data
- MCSE develops an integrated solution for campaigns

PMDK Read (Simple case, no transactions)

- pmemobj API version 2.3
- Example Code/Github

```
struct my_root {
    size_t len;
    char buf[MAX_BUF_LEN];
};
void readIn(){
    PMEMobjpool *pop;
    pop = pmemobj_open(path, LAYOUT);
    // Get the PMEMObj root
    PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
    // Pointer for structure at the root
    struct my_root *rootp = pmemobj_direct(root);
    printf("%s\n", rootp->buf);
    pmemobj_close(pop);    // Close PMEM object pool
}
```

PMDK Write

```
void writeOut(){
    PMEMobjpool *pop;
    pop = pmemobj_create(path, LAYOUT, PMEMOBJ_MIN_POOL, 0666);
    // Get the PMEMObj root
    PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
    // Pointer for structure at the root
    struct my_root *rootp = pmemobj_direct(root);
    // code for write:
    pmemobj_memcpy_persist(pop, rootp->buf, buf, strlen(buf));
    // Assign the string length and persist it
    rootp->len = strlen(buf);
    pmemobj_persist(pop, &rootp->len, sizeof (rootp->len));
    pmemobj_close(pop);    // Close PMEM object pool
}
```