# CS380L Final Project: Async IO and cp –r

Taijing Chen

May 2022

## 1 Introduction

### 1.1 Background

#### 1.1.1 AIO

As we know, when applications issue most I/O requests, they won't get back control til the requested I/O is completed, which can undermine performance. In contrast, Linux asynchronous I/O interface allows users to issue non-blocking I/O requests so that they can better utilize their time slice for other works. This library provides 5 system call interfaces: `io_setup`, `io_destroy`, `io_submit`, `io_cancel` and `io_getevent`. Users can submit I/O requests through `io_submit`. When the request is completed, kernel will create an `io_event`. Users can obtain these responded events using the `io_getevent` call and choose to handle the events of interest with self-defined functions.

#### 1.1.2 cp -r

Through `strace`, we can see some optimization techniques Linux uses to implement `cp -r`: After some initial checks (e.g. permissions, etc), `cp` uses `stat/lstat` to check directory status and open them using `openat` with `O_NONBLOCK` set. When it opens the source file, `cp` uses `fadvise64` with the `POSIX_FADV_SEQUENTIAL` flag to notify kernel that it'll access file sequentially. It then calls `mmap` and `ioctl` with `FS_IOC_FIEMAP` flag for better locality. Finally, `cp` performs read and closes fils.

### 1.2 Goal

The goal of this project is to optimize performance of recursive copy on Linux machines via Linux's asynchronous IO interfaces (aio) and multi-threading.

## 2 Design

My implementation exploit 3 main techniques to optimize recursive copy on Linux machines:

1. Linux Asynchronous I/O interface, which allows my program to continue execution without waiting for
2. multi-threading, which allows my program to do copies in parallel
3. fadvise, which allows optimization for the specific access pattern in recursive copy

My design has two main components: 1) `Copier` that operates on a file-granularity for single file copying, and 2) `RecursiveCopier` that operates on a directory-granularity to manage metadata multiple worker threads. I'll briefly explain their jobs and high-level designs in this section.

### 2.1 Copier

`Copier`'s job is to copy a single file. It's implemented with Linux Asynchronous I/O. The I/O requests are submitted and handled in batches, with a maximum batch size that users can specify. Before and after accessing file data, `Copier` uses `fadvise` to inform kernel of the file access pattern it expects. See section 3 for details

### 2.2 RecursiveCopier

`RecursiveCopier` handles metadata (e.g. create directory, check permission). It also manages multiple threads and assign jobs to different workers threads. With such design, multi-threading only happens across files; There's no additional thread management within a single file copy.

## 3 Implementation

### 3.1 Overview

I implemented this project in C++17. I used the Filesystem library in C++'s std to iterate through directory recursively. Apart from this, all other file/directory operations are done through C libraries. The main APIs are shown in Table 1.

| **APIs** | **Description** |
|---|---|
| `mycp::init(const unsigned nEvents)` | initialization |
| `mycp::shutdown` | clean up and shutdown |
| `mycp::Copy::copy()` | copy a single file using Linux AIO |
| `mycp::RecursiveCopier::copy()` | recursive copy files/directories |

Table 1: mycp APIs

## 3.2 AIO

In my implementation, there're two parameters user can tune: 1) `nMaxRCopierEvents`, which is the total queue size users want to register with `io_context_t`, and 2) `nMaxCopierEvents`, which is the max number of I/O requests in one `io_submit`.

To use the recuresive copier, users first call `init` to register the max `io_context_t` size they want. Then, with the source directory path and destination directory path parsed in, `RecursiveCopier` will start to examine directory/file status. For directories, it will create directories at the destination path if they don't exist. If there exist a file in the destination directory whose name conflicts with a directory name in the source directory, or a directory name in the destination directory conflicts with a filename in the source directory, the program gives error and abort.

`RecursiveCopier` assigns files to `Copier` to perform AIO copy. `Copier` opens source file and destination file, and records their file descriptors. After opening the source file, the constructor uses `posix_fadvise` on the source file descriptor with `POSIX_FADV_NOREUSE` set, as we expect to access all blocks in the source file once. In the constructor, `Copier` also creates `nMaxCopierEvents` numbers of I/O callback pointers (i.e., `struct iocb*`) and reserve a block size of buffer for them.

To start single file copy, `Copier` prepares files for I/O requests using `io_prep_pread`: Each I/O request contains read on one filesystem block; If the file size or the remainig unread file size is smaller than filesystem block size, `Copier` checks it and makes sure the requests don't contain unwanted data. When there're `nMaxCopierEvents` of events prepared or there's no remaining data to prepare for read, `Copier` submits all batched I/O requests through `io_submit`. If `io_submit` fails to submit any of the request, the program reports errors and then abort.

Once a while `RecursiveCopier` calls `RecursiveCopier::handleCallback`. `RecursiveCopier::handleCallback` obtains completed I/O requests through `io_getevents` and handles events based on self-defined I/O callbacks. When a read request is completed, it triggers the registered `RecursiveCopier::readCallback`. This callback checks if the returned read sizes match with the requested sizes. If so, it will prepare write requests and submit them. When a write request is completed, `RecursiveCopier::writeCallback` check if there's unread data in the source file. If there is, it continues to prepare new read requests.

When the lifetime of a single `Copier` object comes to an end, its deconstructor performs clean-ups such as freeing memory and close source and destination files. For correctness, deconstructor waits till all I/O callbacks associated with itself to complete and then performs clean-ups.

## 3.3 Multi-threading

To implement Asynchronous I/O, I/O callbacks need to know the status of their associated files. For example, in the write callback, we need to check if there's remaining un-copied data; The Copier deconstructor also needs to know if all data have been read/written before it closes the file descriptors. To do so, my program keeps track of the in-use I/O callbacks and statuses of their target file in a global structure.

This implementation is trivial for single-thread AIO. However, for multi-thread AIO, it becomes a bit tricker. Unlike in Java, I'm not aware of any thread-safe containers in C++'s standard library. For correctness, an easy solution is to lock all operations to this global tracking structure, but for obvious reason such locks can lead to terrible performance bottleneck. In my current multi-threaded AIO implementation, I resorted to a lock-free solution: I reserved a large static array (65536 pointers) and used a self-defined hash function to map callback pointers to array entries. It's worth noting that there's no additional checking when program tries to modify values stored in this hashtable, assuming conflicts are rare. I'm aware that it's not a safe assumption to make. I'll discuss more about it in section 4.

### 3.4 Code delivery

GitHub Repo (https://github.com/TieJean/mycp/tree/aio-mt) branches:

- master: contains executable to generate synthetic data for evaluation (It doesn't contain my final multi-threaded aio implementation) (`./bin/eval_main`)
- aio: AIO without multi-threading
- mt: multi-threading without AIO
- aio-mt: multi-threaded AIO

To run the implementation, do:
`./bin/main --src=<abs-path-to-src-dir> --dst=<abs-path-to-dst-dir>`.

## 4 Limitations

In this section, I'll discuss about two main limitations of my current implementation.

### 4.1 Multi-threading Correctness

As mentioned in section 3.3, my implementation stores a global structure to keep track of all submitted but unfinished I/O callbacks so that `Copier` doesn't close file descriptors or free spaces that are still in-use. To do so, I created a static array and mapped all `iocb` pointers to their target file descriptors using a self-defined hash function. When my implementation accesses or makes changes to this structure, with the hope that collisions are rare, it doesn't do any additional checking. However, when the workload is large, this assumption is not safe. When using my codes on the `aio-mt` branch to copy a large directory, users sometimes will see an error message and then the program aborts. There're multiple scenarios where this can happend. For example, my structure currently stores a mapping from the `iocb` pointers to their target file status (e.g. opened file descriptors, offset, etc). When the `Copier` gets deconstructed, the values stored in the mapped indices are set to `NULL` (i.e., 0). When `iocb` pointers from two different `Copier`'s get mapped to the same entry, it is possible that one `Copier` thought it finishes all jobs and calls the deconstructor while the other one is still needing its file descriptor. Even if the file descriptor is not closed, the files are corrupted as the `Copier`'s are reading from/writing to wrong files. A correct implementation should checks for those conflicts or implement more fine-grain locking. For the purpose of this project, I didn't implement one myself as I'm sure there're existing third party C++ libraries that offer thread-safe hash tables; But using them would make it difficult to hypothesize and understand the performance. Thus, they're out of scope of this project.

### 4.2 Workload balance

Ideally, `RecursiveCopier` should use some policy to balance workloads assigned to different worker threads based on file sizes. `RecursiveCopier` can achieve this by 1) using some hash function to map files to workers randomly or 2) (better) recursively scanning through the source directory first and assign files to workers based on their sizes. Neither strategy is implemented in my vanilla multi-threaded aio (mainly to make debugging easier). Currently, `RecursiveCopier` creates one thread for each directory under the source root directory, and handles rest files in source root directory by itself. Thus, file sizes that are not evenly spread out at different directories will result in poor performance.

## 5 Evaluation

### 5.1 Experimental Setups

The experiemnts were conducted on a 8-core Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz with Ubuntu 20.04. It has a 64 KB L1 data Cache and 2M/4M pages L1 TLB. All experiments used synthetic datasets created by a script I wrote. In all experiments, caches and TLBs both are cold before every run (all files were deleted and recreated in every trial) to minimize impacts from caches and TLBs.

Moreover, since `RecursiveCopier` in my current implementation doesn't assign jobs to workers smartly (section 4.2), there's no point to evaluate my implementations on imbalanced workload. This means, in all my experiments, if possible, the directories under the source root directory have roughly the same total file sizes and file numbers. In the following experiments, if not mentioned otherwise, the multi-threaded version launched 10 threads.

### 5.2 Tested Prototypes

I evaluated 3 versions of implementation. Summaries shown in table 2

| **name** | **fadvise before access** | **fadvise after access** | **multi-threaded** | **copy** | **data structure** |
|---|---|---|---|---|---|
| `aio` | NOREUSE | DONTNEED | no | AIO | C++ STL |
| `mt` | NOREUSE + SEQUENTIAL | DONTNEED | yes | sendfile | C++ STL |
| `aio-mt` | NOREUSE | DONTNEED | yes | AIO | static arrays |

Table 2: Tested Prototypes

#### 5.2.1 aio

This version implemented my recursive copy with single-thread Linux Asynchronous I/O. It reserves a 65536 (system-wide maximum number) `io_context_t` entry queue. Each `Copier` batches a maximum of 64 blocks of I/O. When it accesses a block, it suggests kernel with `POSIX_FADV_NOREUSE` (`POSIX_FADV_SEQUENTIAL` is not set as I'm not sure about AIO's response to batched read request); When it finishes writing a block, it sets `POSIX_FADV_DONTNEED`.

#### 5.2.2 mt

This version implemented my recursive copy multiple threads. Since this project only focuses on Linux machines, copies are done through `sendfile`. According to its man page, `sendfile` handles copy in kernel so it's more efficient than combinations of `read` and `write`. When it accesses a block, it suggests kernel with `POSIX_FADV_NOREUSE` and `POSIX_FADV_SEQUENTIAL`; When it finishes writing a block, it sets `POSIX_FADV_DONTNEED`.

#### 5.2.3 aio-mt

This version implemented my recursive copy with multi-threaded Linux Asynchronous I/O. Due to limitation discussed in section 4.2, the number of worker threads is determined by the number of directories under the source root directory. Copies through AIO is accomplished in a similar fashion to the one in the aio version with exceptions of the data structures used: All C++ container structures are replaced by statically-allocated arrays, with hash functions that map values to entries. Due to the limitation discussed in section 4.1, this version is not evaluated with large datasets.

## 5.3 Experiment1: Copy single large file

In this experiment, programs were asked to copy a single large file (512 MB, 1GB, and 2GB) from the source directory to the destination directory. Since we only focus on recursive copy, this experiment mainly served as a baseline to help us better understand the performance. As there's only one files, `RecursiveCopier` only issue one thread for this job, mt in fact measures the

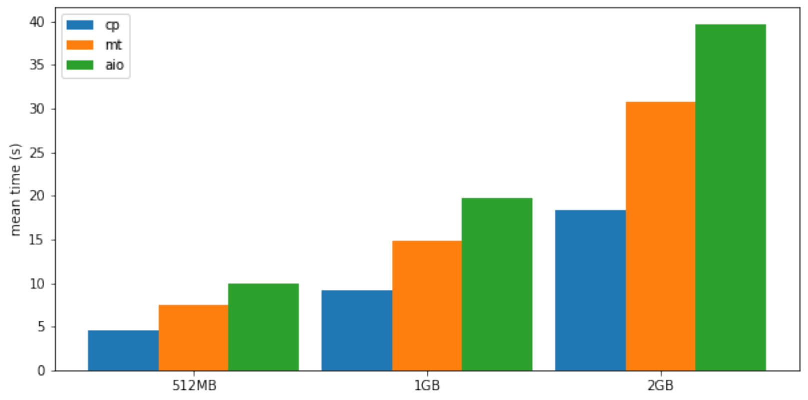

Figure 1: copy a single large file

time `sendfile` takes for a single file copy, together with other overhead in my implementation. For the same reason, aio-mt does the same operations as aio, so it's not separately included in Figure 1. As we can see, both mt and aio takes a lot more time compared to `cp`, with aio taking almost twice more time than the ones for `cp`. The result suggests that my implementations may have high overheads.

## 5.4 Experiment2: Copy directories with various file sizes

### 5.4.1 Small file tests

In this experiment, programs were asked to recursively copy  500 files from the source directory to the destination directory. All tests have the same number of directories as well. In the small file test (shown in Figure 2, all files are only a few (4-8) KBs. This means programs have a higher metadata-to-data ratio relative to other test cases. Both mt and aio-mt takes significantly less time to finish this tasks, suggesting multi-threading plays a key role here. This is expected, as all threads are performing non-conflicted metadata checkings and creations simultaneously. However, I was surprised to observe that my aio performances a lot worse than `cp`, as I hypothesized that aio can perform other metadata operations while waiting for I/Os to complete. Again, I think this result suggests my aio implementation has a high overhead.

## 5.5 Medium and Large file tests

These two test cases have the same setup as the small file test in section 5.4.1 apart from the file sizes. The medium file test is consisted of files whose sizes are within a few (1-4) MB. In the large file test, there're ten 1GB files, while the

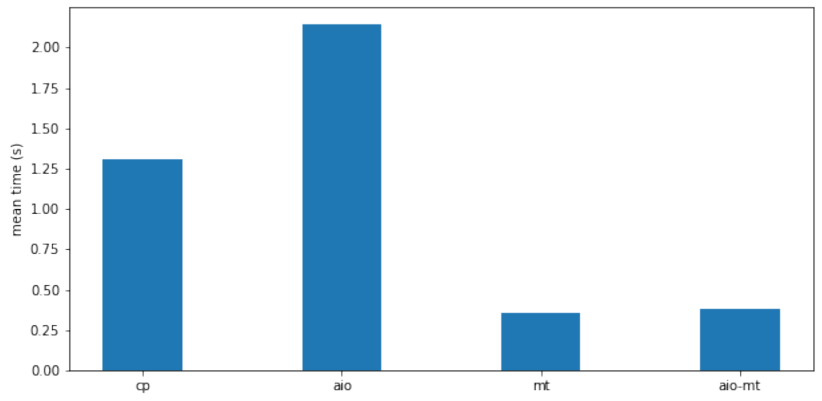

Figure 2: Small file test

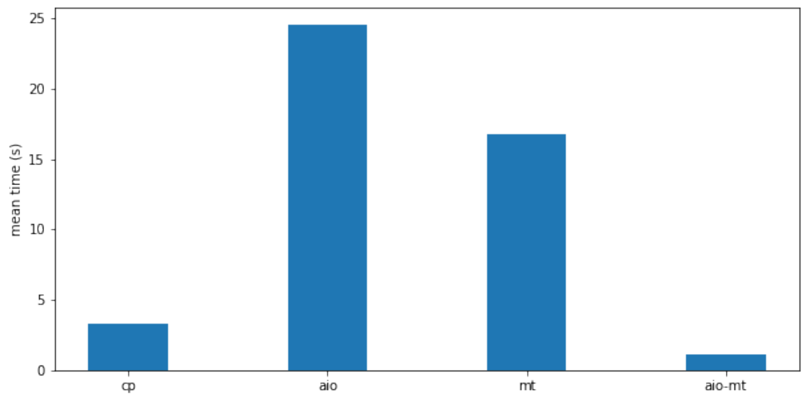

Figure 3: Medium file test

rest files are of size 10 MB (so that the resulting total file size is not enormous). Due to the multi-threading limitation discussed in section 4.1, aio-mt cannot consistently succeed for the large file test. Thus, I didn't include its result here. In both tests, we can see aio and mt along performs poorly compared to `cp`: mt takes 4x more time in the medium file test that `cp`, while aio takes 6x more time in the this task. Given results from previous section, this is expected: These two tasks have much lower metadata-to-data ratio, so the speed for data copying determines the performance. In the medium file test, aio-mt outperforms `cp`, which shows the combined power of combining Linux Asynchronous I/O and

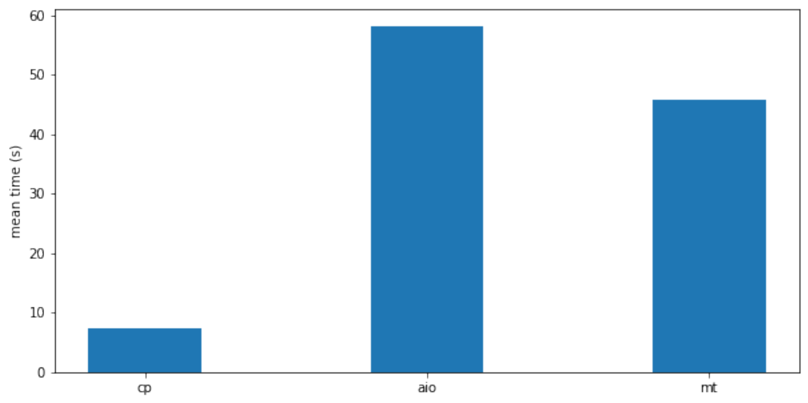

Figure 4: Large file test

multi-threading.

### 5.6 Experiment3: Copy directories with various total numbers of files

In this experiment, all files are of size 8 KB, but the total numbers of files under the source root directory differ. Figure 5 shows the average time `cp` and `aio-mt` took to complete different tasks. As shown in the graph, aio-mt significantly outperforms `cp`. Given the results from experiment 1 and 2, this is expected, as aio-mt can better exploit parallelism on the metadata.

### 5.7 Experiment4: Number of threads

Since experiment1-3 shows the great benefits that aio plus multi-threading can bring, this experiment investigates the effect of the number of threads on aio-mt's performance.

This experiment has basically the same work load as the medium file test in experiment 2. The only difference is that in one dataset, there are only two directories under the source root directory, so my aio-mt only launches two thread to copy files in this dataset. Figure 6 demonstrates a significant improvement when more threads were used, which matches results shown in experiment 1-3 and my hypothesis.

## 6 Conclusion

To conclude, at least for my implementation, Linux Asynchronous I/O or multi-threading along failed to achieve similar performance as `cp -r`. For efficient

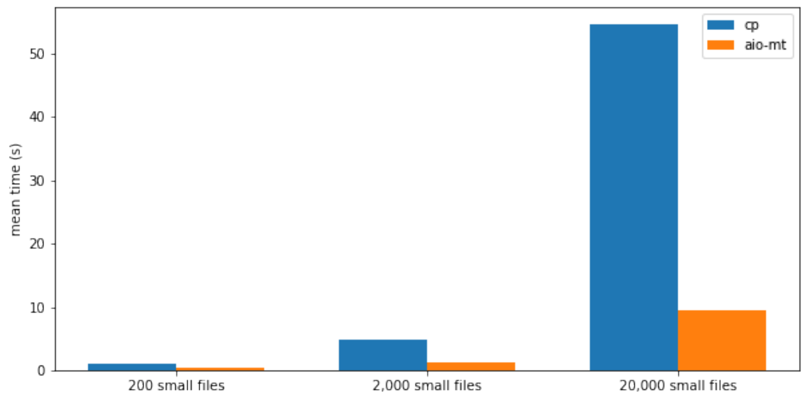

Figure 5: Effects of different total file numbers

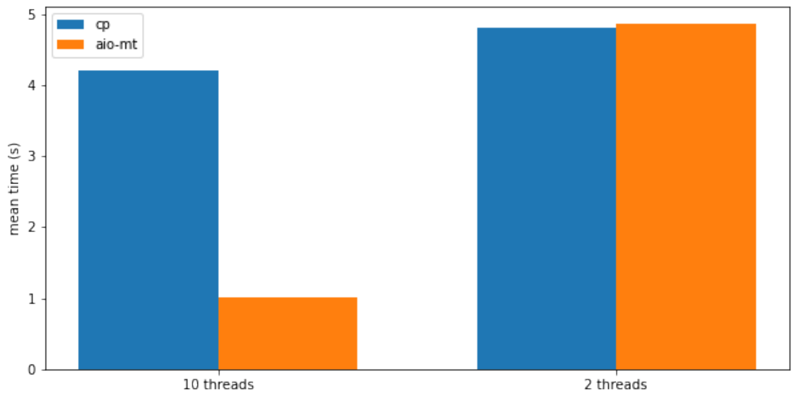

Figure 6: Effects of number of threads

recursive copy, we need to exploit the power of both non-blocking I/O and multi-threading.

I'd also like to note that the purpose of this project is to optimize recursive copy. Therefore, in my evaluation, I focused more on evaluating my implementation rather than understanding the mechanical details behind Linux Asynchronous I/O. As a result, I didn't perform micro benchmarks on every operations I used, nor did I measure each function independently. However, if I have more time, I'm very interested in doing this benchmarks so I can have a deeper understanding on why my single-thread aio has such poor performance and why multi-threading helps the performance so much.

## 7 Time spent

I spent a week reading about the interface and drew out my skeleton design. It took me about 3 days to implement a vanilla prototype with aio (no multi-threading). I spent most of the rest weeks trying to get/debug on a multi-threaded aio version. It took me about 3 days for the report and presentation.

## 8 Reference

Linux Asynchronous I/O. https://oxnz.github.io/2016/10/13/linux-aio/io-models
posix_fadvise man page. https://linux.die.net/man/2/posix_fadvise
Asynchronous I/O and event notification on linux. http://davmac.org/davpage/linux/async-io.html