PFIND: Parallel File Inspection and Node Discovery

Denis Koshelev, Aditya Kaushik, Ryan Lynch

Georgia Institute of Technology

April 23, 2024

Agenda

1. Problem statement
    - Limitations of the current `find` command

Agenda

1. Problem statement
   - Limitations of the current `find` command
2. Our parallel implementation
   - How `pfind` works
   - Approach 1: Recursive version
   - Approach 2: Iterative BFS

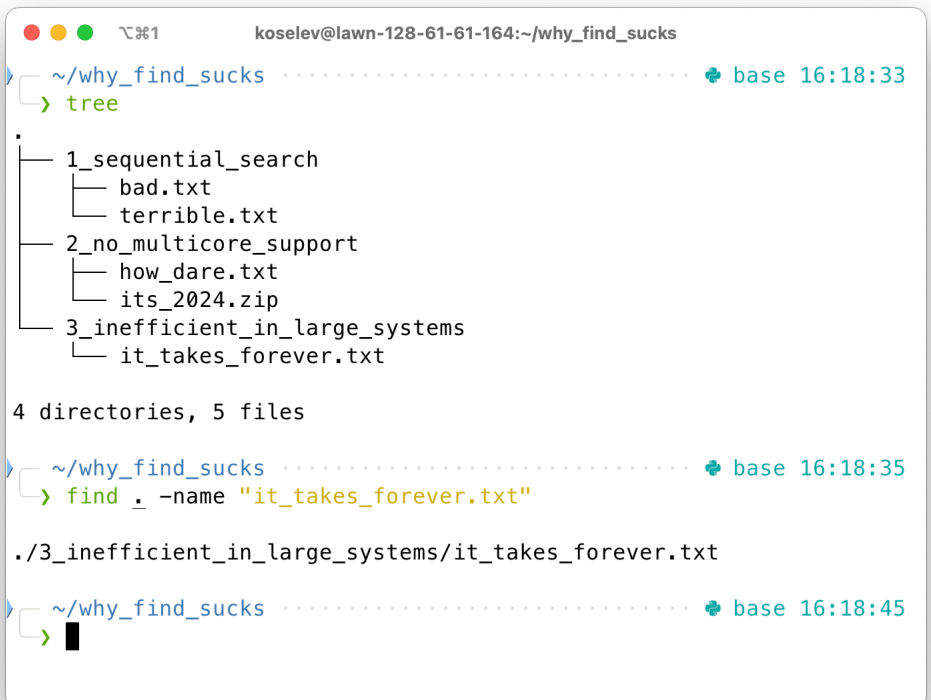
Agenda

1. Problem statement
   - Limitations of the current `find` command
2. Our parallel implementation
   - How `pfind` works
   - Approach 1: Recursive version
   - Approach 2: Iterative BFS
3. Current progress

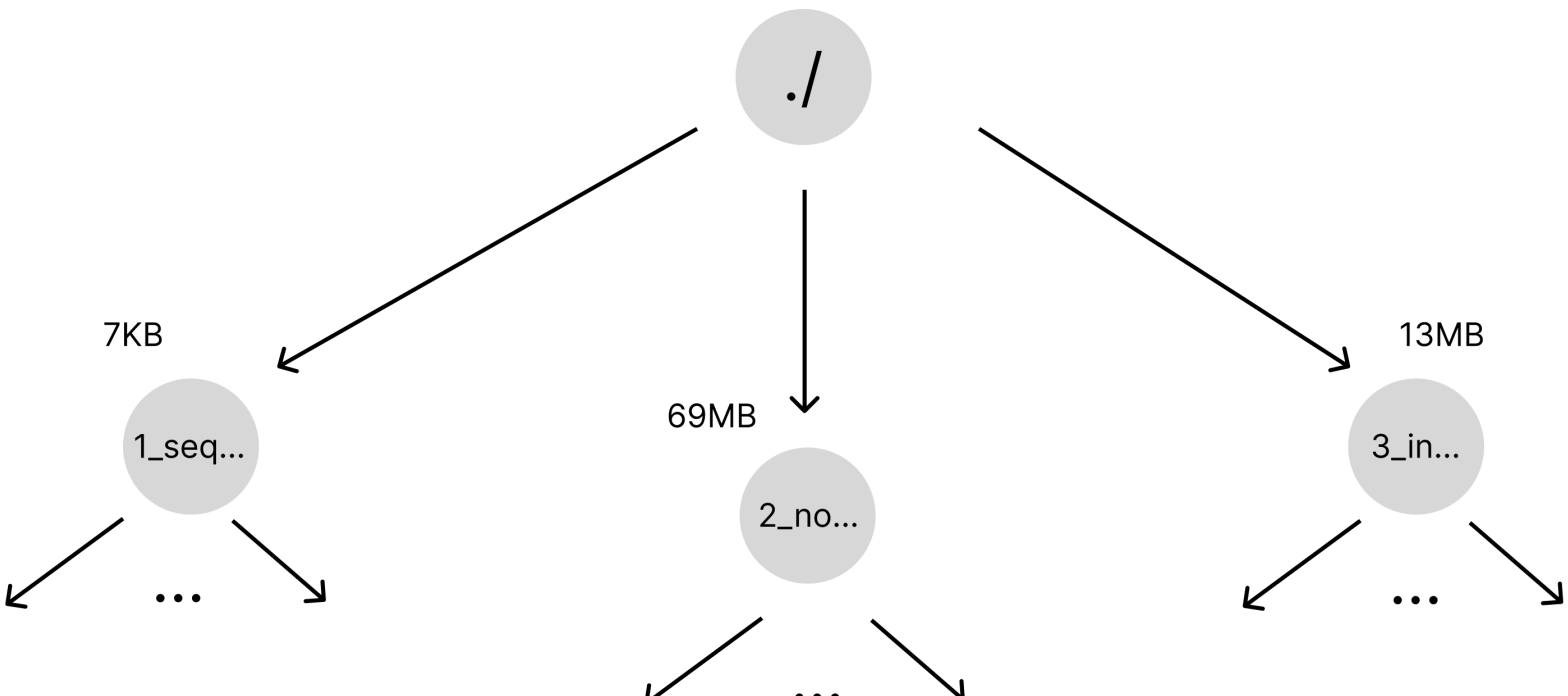
Agenda

1. Problem statement
   - Limitations of the current `find` command
2. Our parallel implementation
   - How `pfind` works
   - Approach 1: Recursive version
   - Approach 2: Iterative BFS
3. Current progress
4. Evaluation
   - Data collection and challenges
   - Benchmarking with competitors

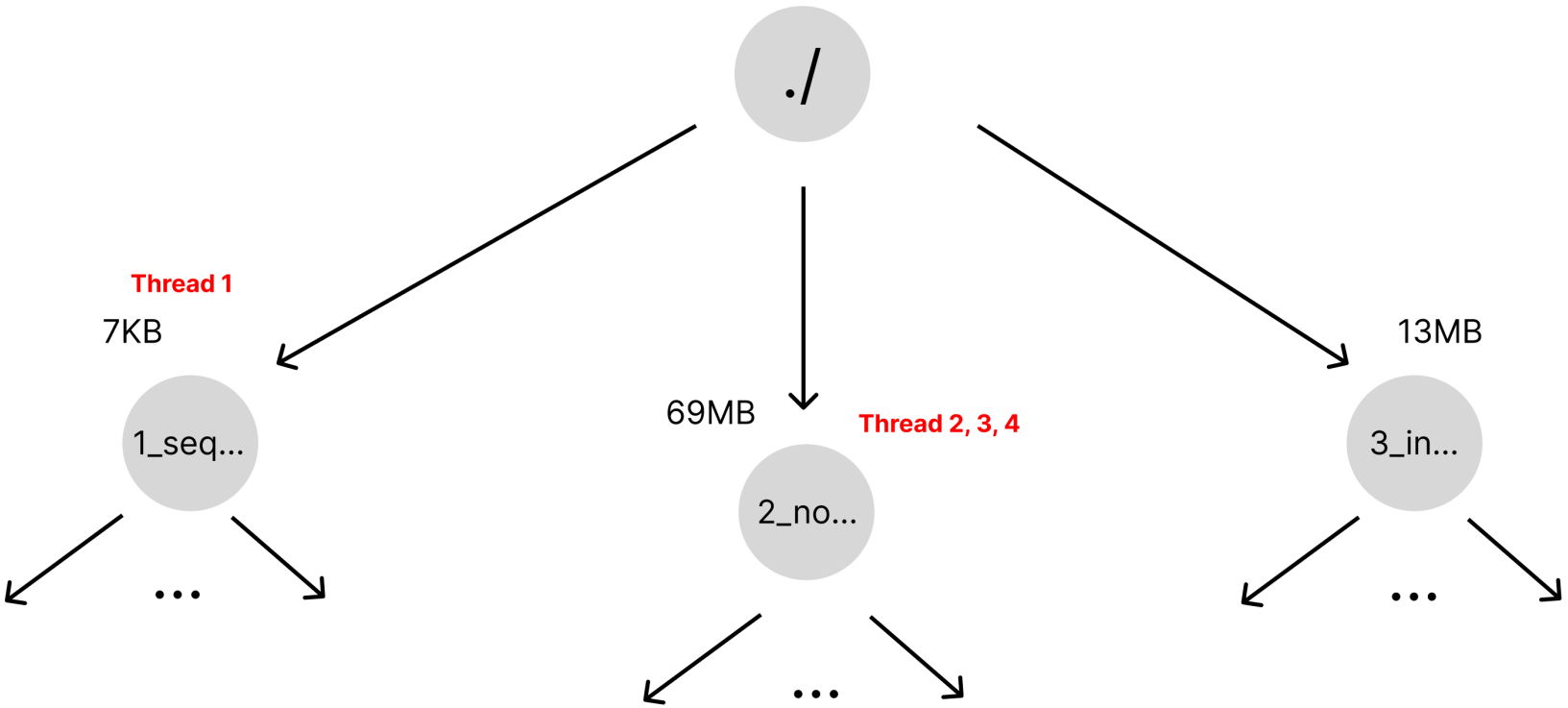
Limitations of the current `find` command



How `pfind` works



How `pfind` works



How `pfind` works

Our main ideas for parallel implementation are:

How `pfind` works

Our main ideas for parallel implementation are:

- Using **directories** as parallelizable units of work

How `pfind` works

Our main ideas for parallel implementation are:

- Using **directories** as parallelizable units of work
- Using **heuristics** such as directory size and/or number of files to split up work amongst threads

Approach 1: Recursive approach

An outer function calls a recursive function `processDirectory()` that takes start folder and target to find as parameters:

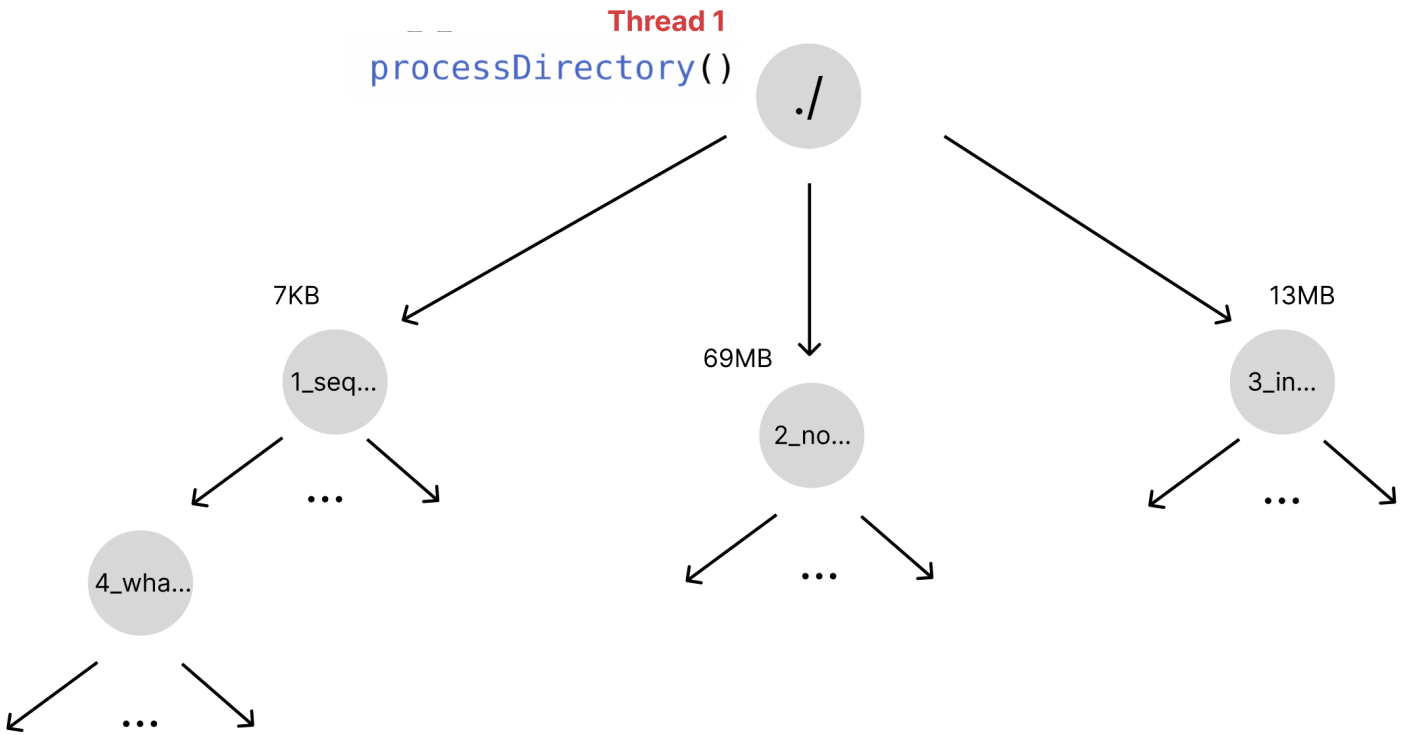
```
function parallelFind(root, target) {
    #pragma omp parallel single
    processDirectory(root, target)
}
```

Approach 1: Recursive approach

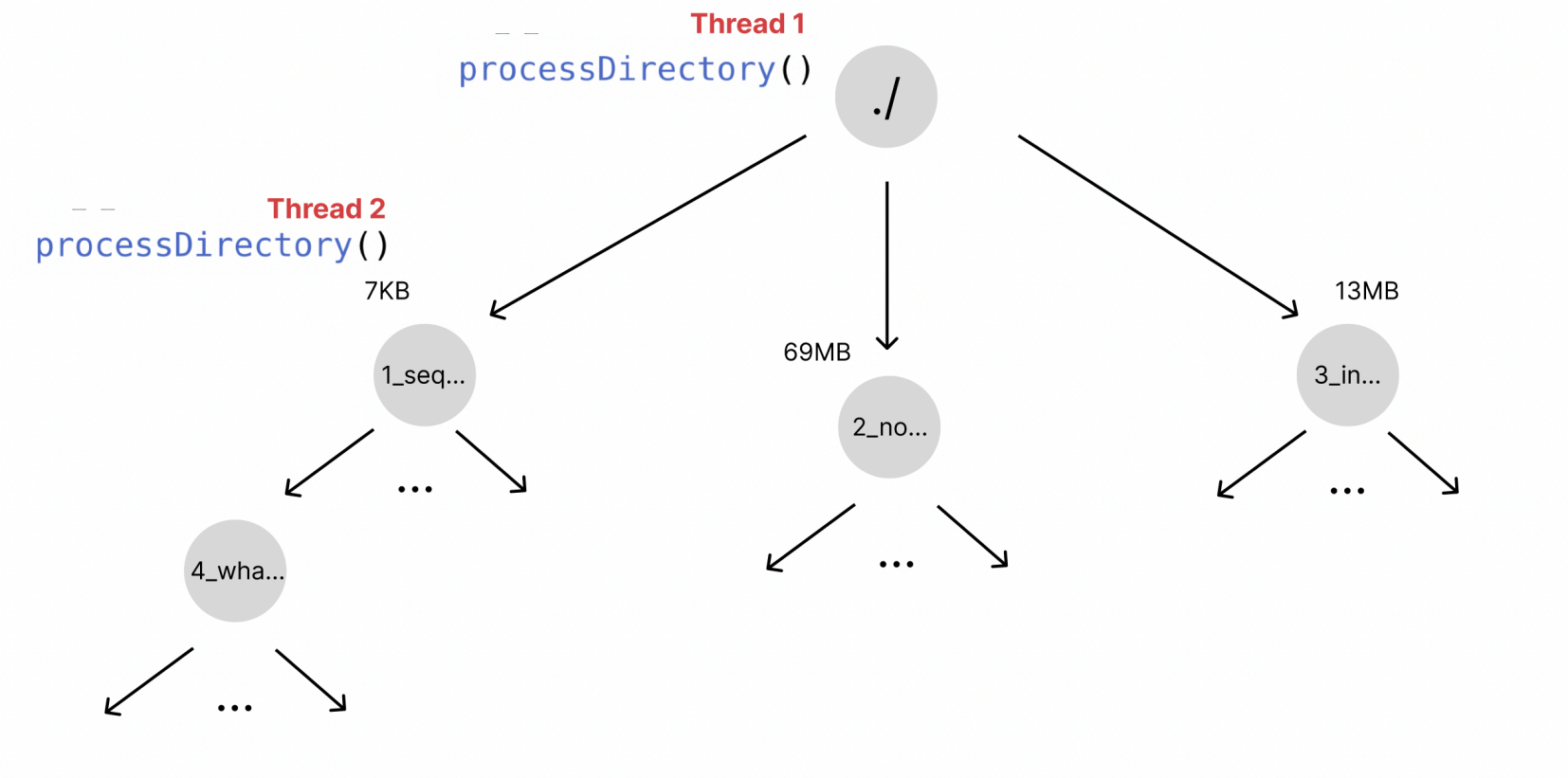
`processDirectory()` function recursively makes parallel calls to itself whenever new directory is encountered or prints path if file found:

```
function processDirectory(directory, target) {
    for entry in directory:
        if entry is directory:
            #pragma omp task
            processDirectory(entry, target)
        else if entry matches target:
            print(entry)
}
```

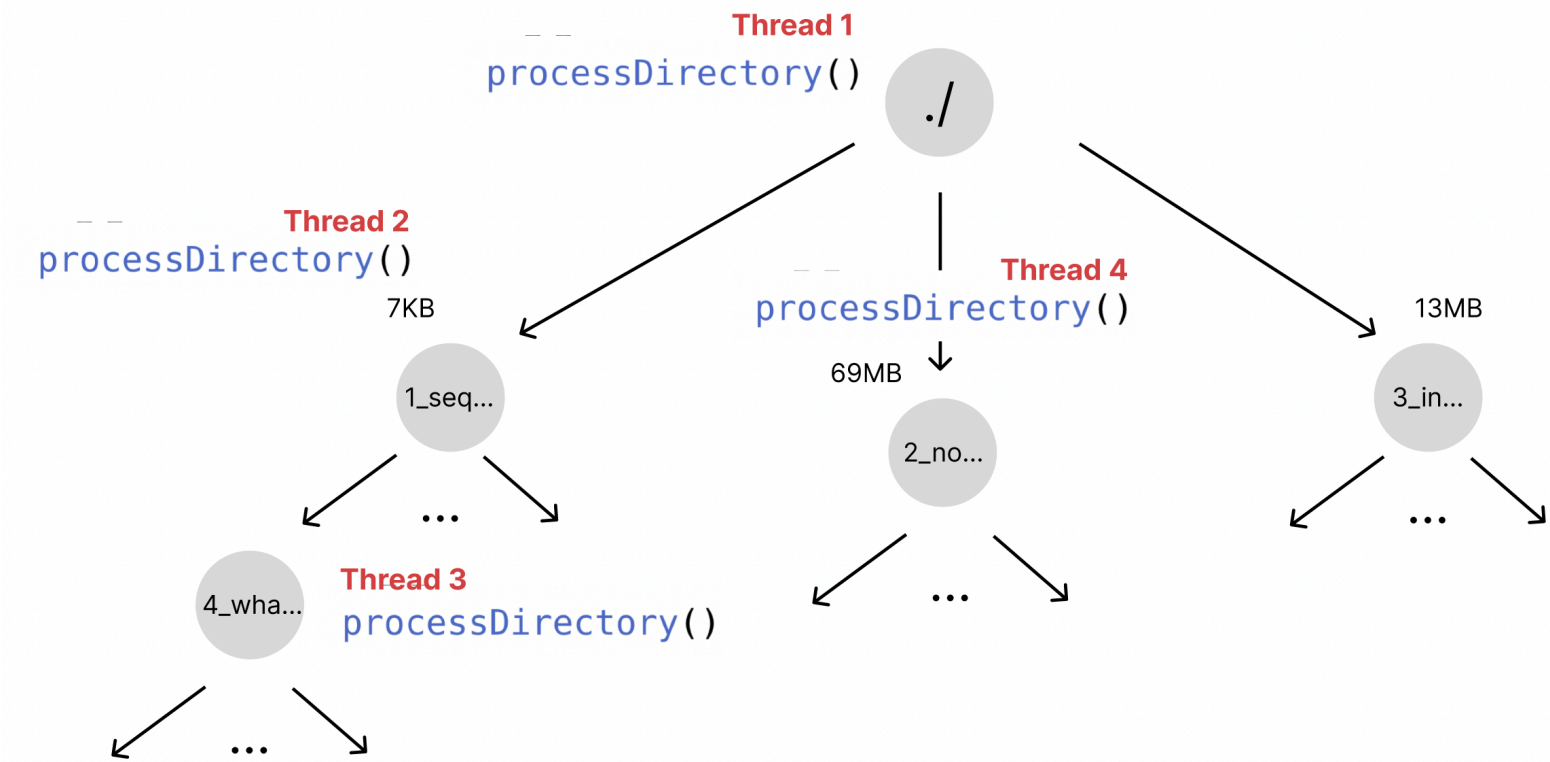
Approach 1: Recursive approach



Approach 1: Recursive approach



Approach 1: Recursive approach



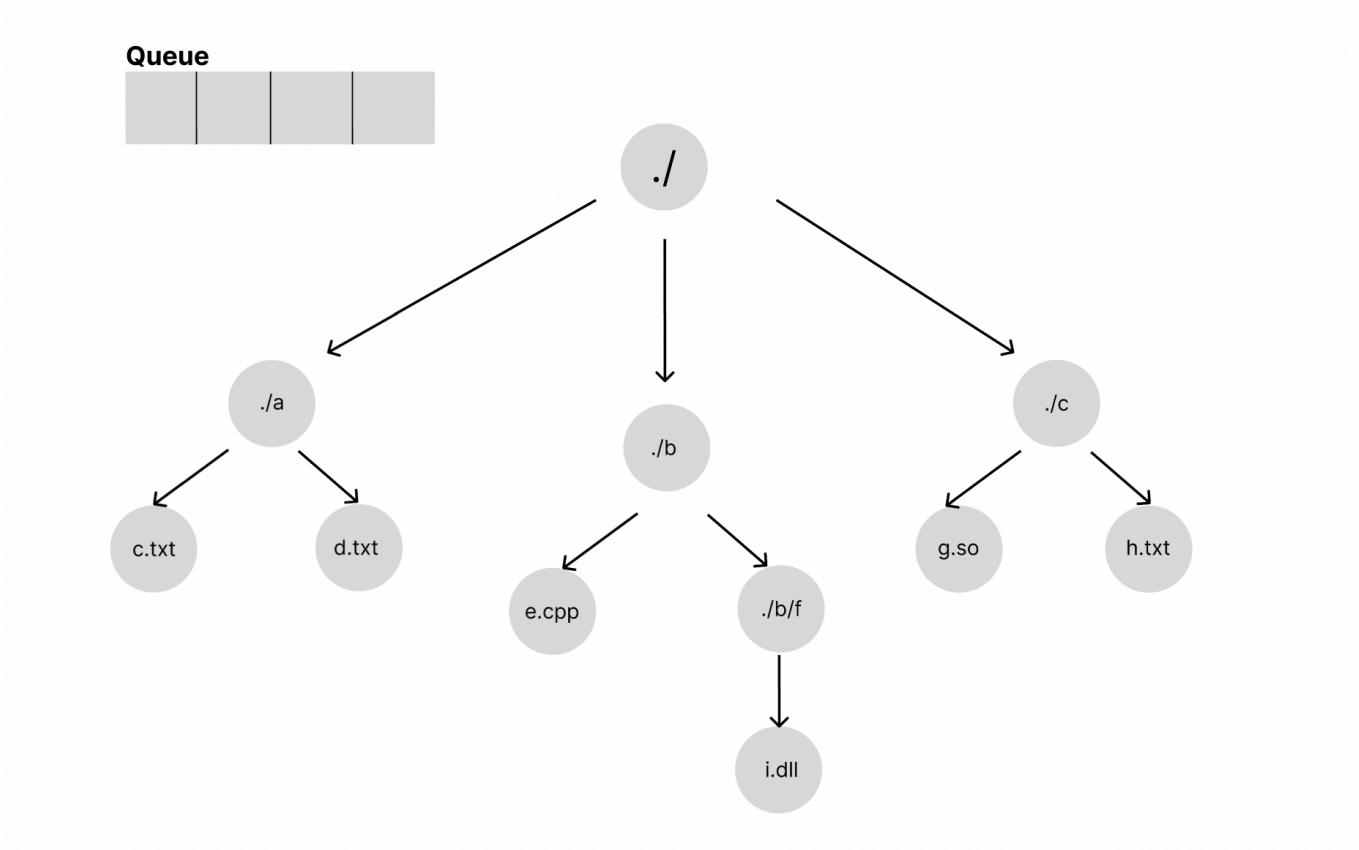
Approach 2: BFS

```
omp_set_lock(&queue_lock);
if (!directory_queue.empty()) {
    current_directory = directory_queue.front();
    directory_queue.pop();
    active_tasks++;
}
omp_unset_lock(&queue_lock);
```

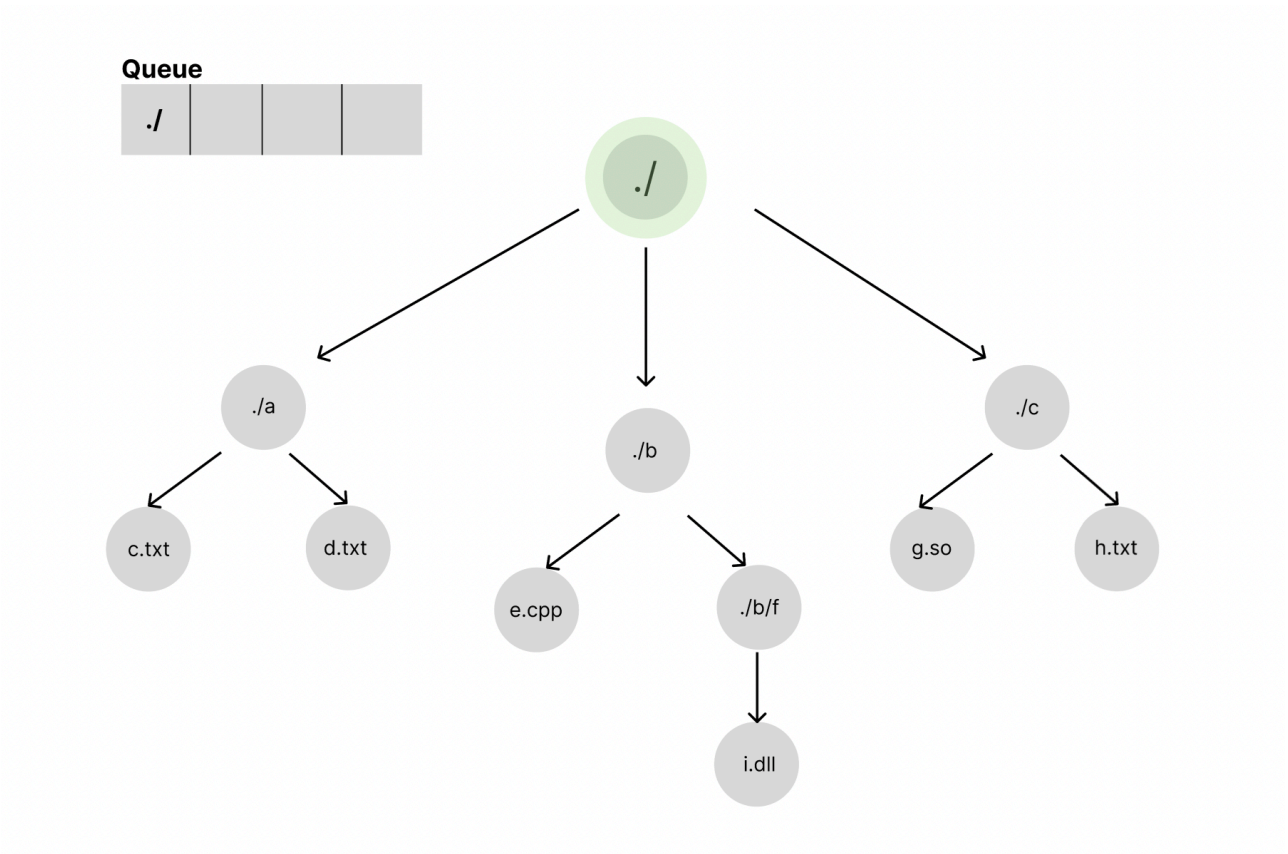
Approach 2: BFS

```
omp_set_lock(&queue_lock);
directory_queue.push(full_path);
omp_unset_lock(&queue_lock);`)
```

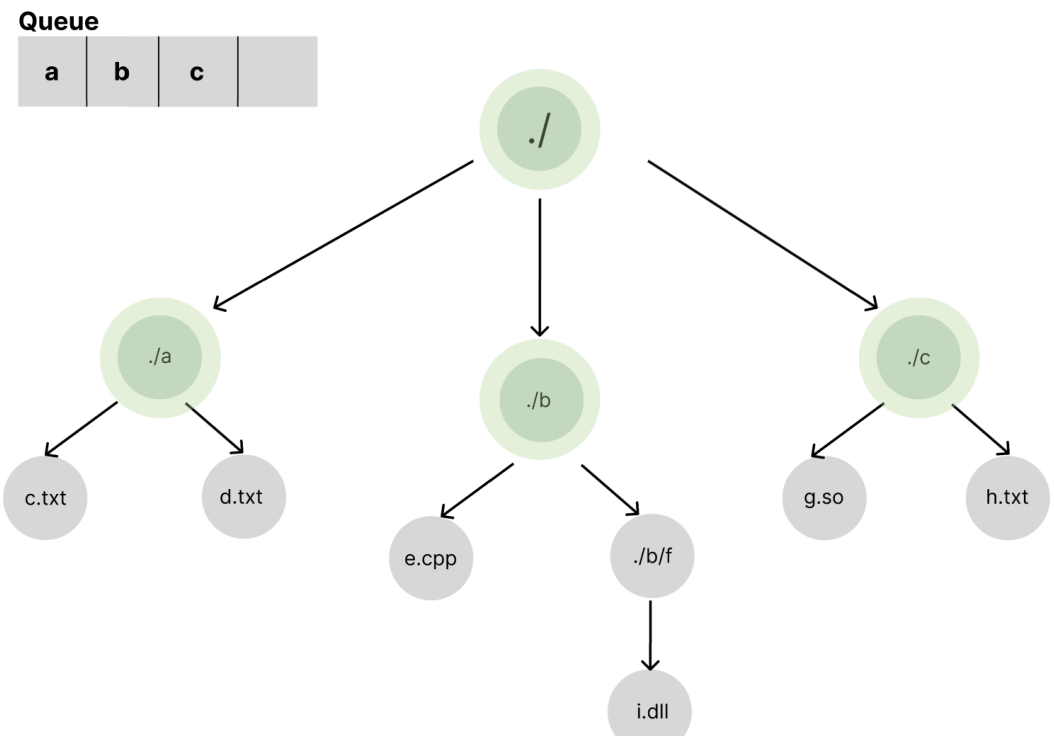
Approach 2: BFS



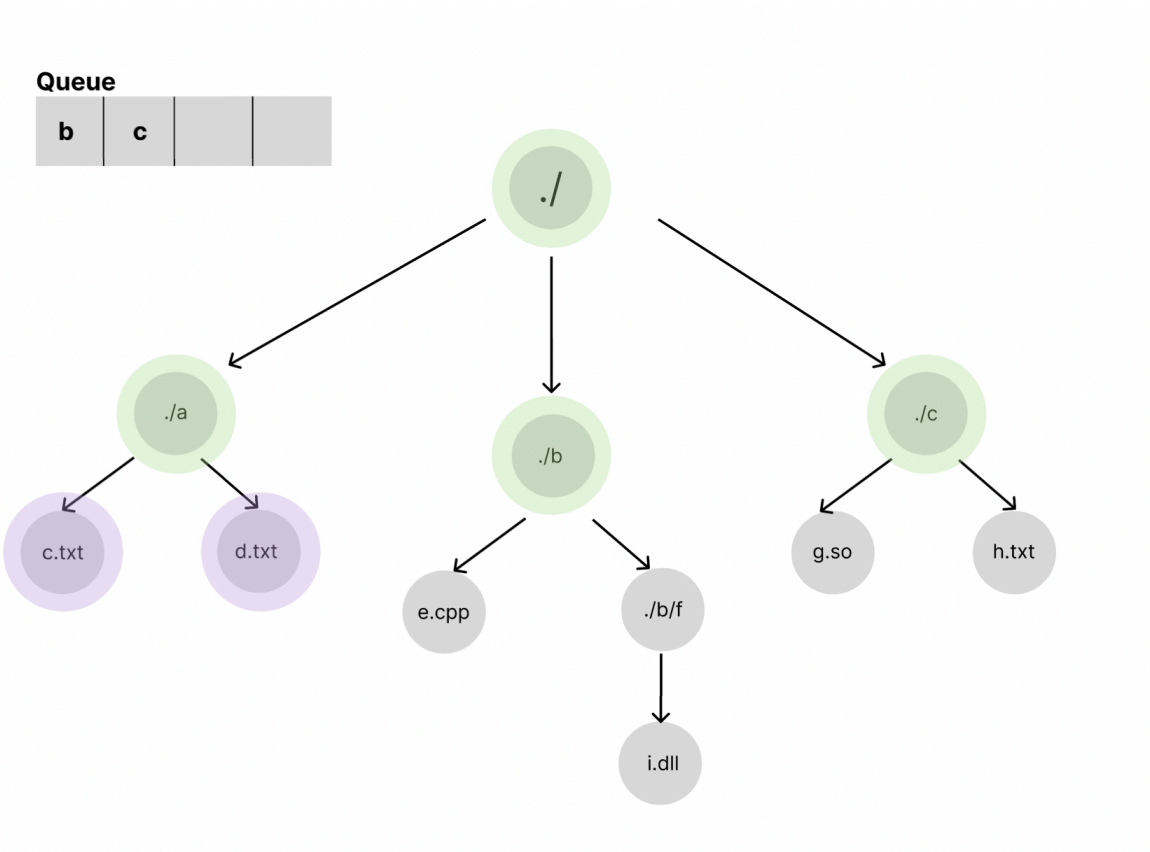
Approach 2: BFS



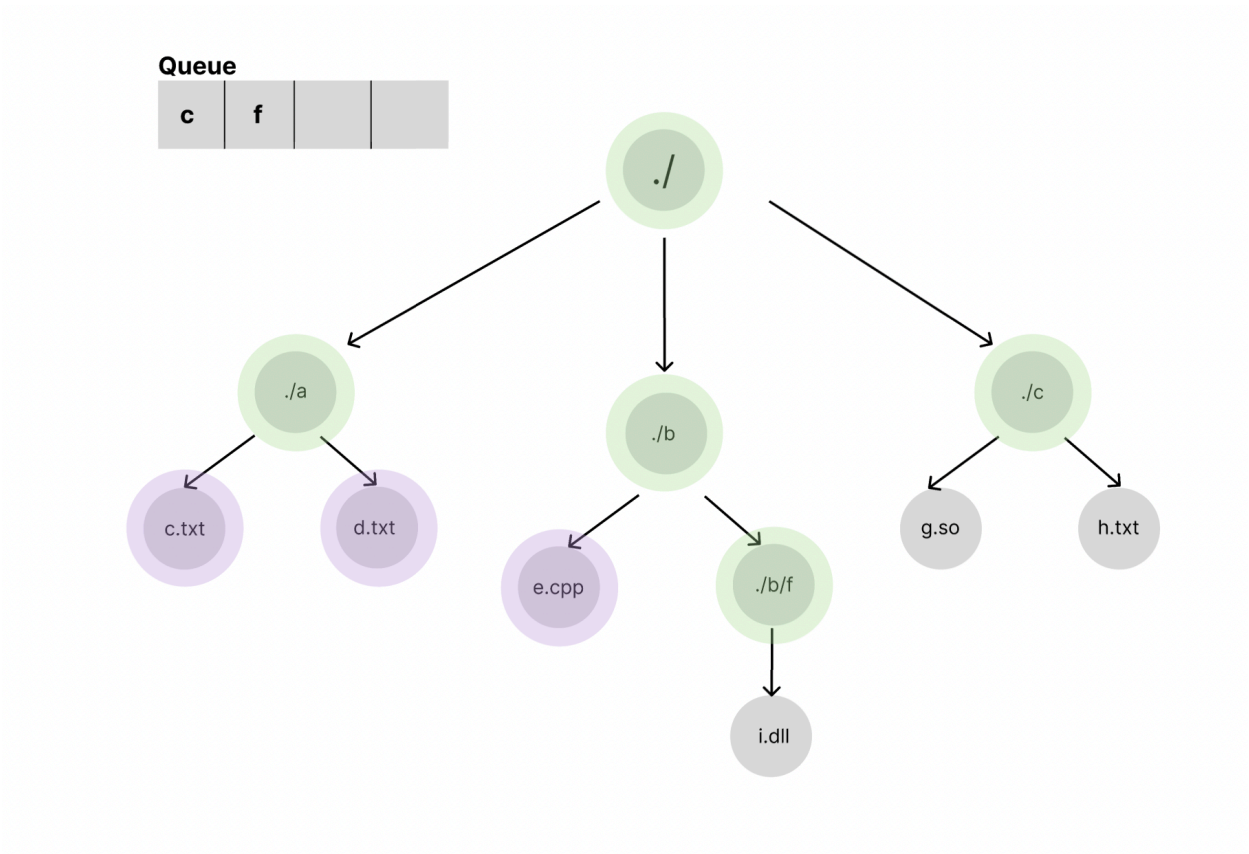
Approach 2: BFS



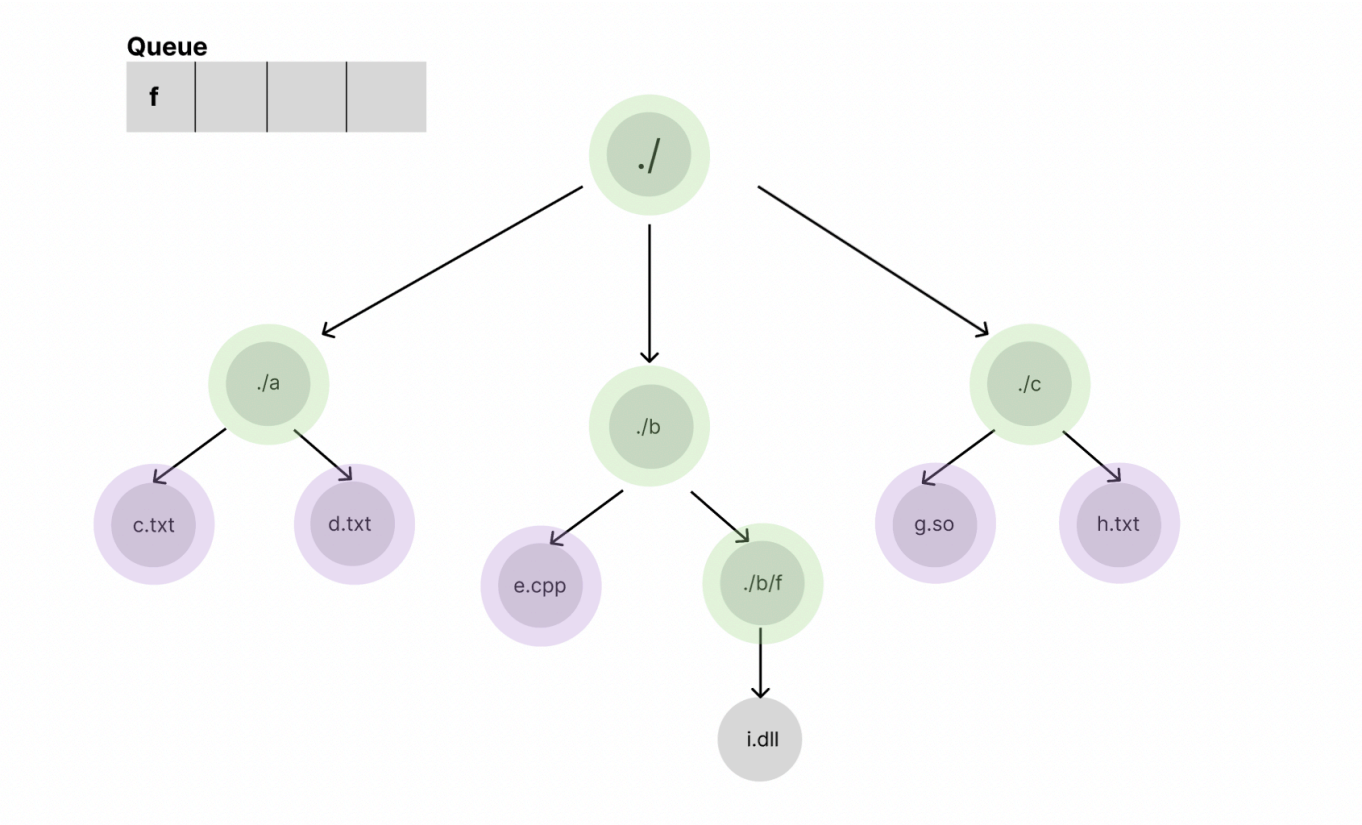
Approach 2: BFS



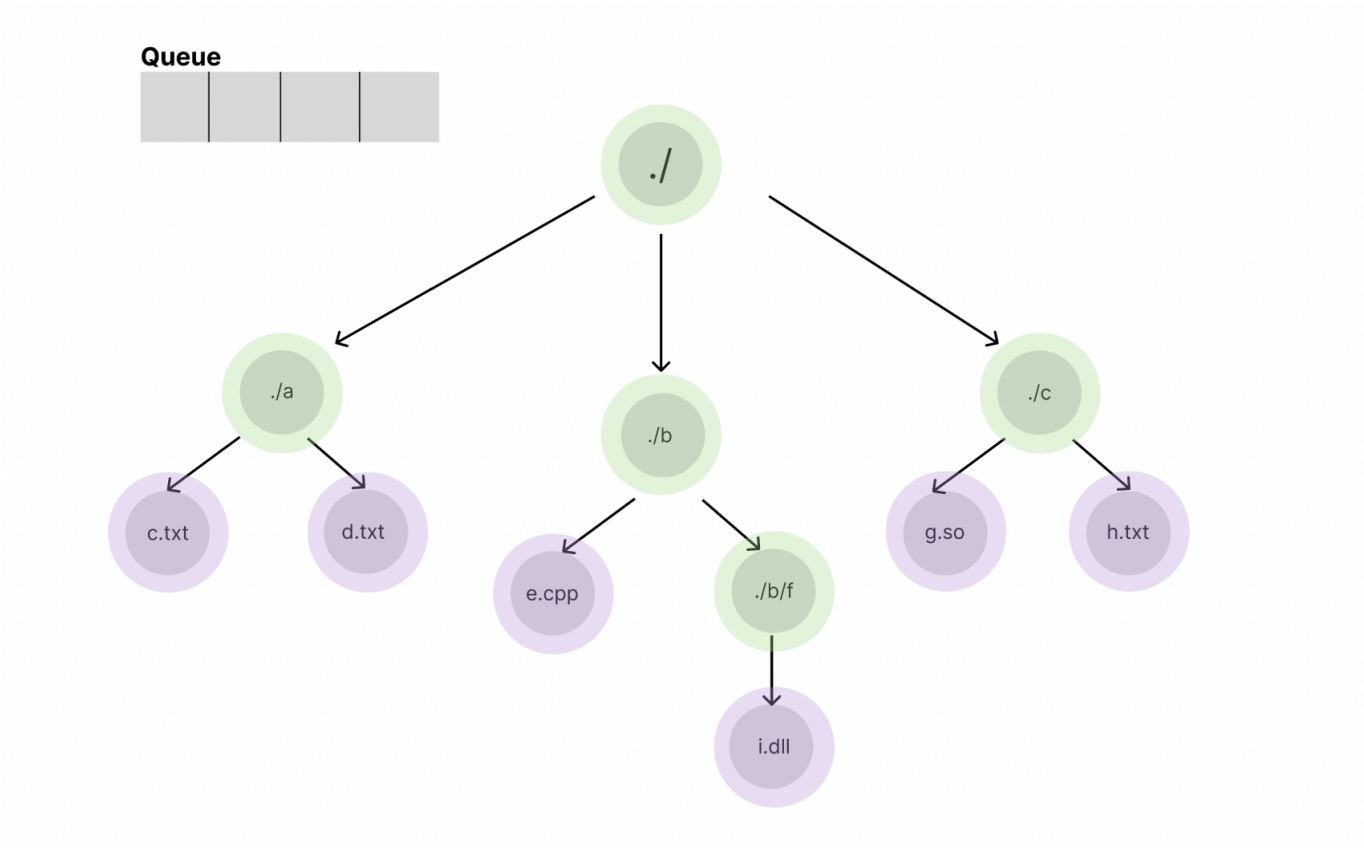
Approach 2: BFS



Approach 2: BFS



Approach 2: BFS



Current progress

1. Implemented a serial implementation of `pfind` using BFS
2. Implemented recursive and iterative `pfind` versions
3. Prepared a dataset suite for benchmarking
4. Performed basic performance analysis of iterative `pfind` version

Current progress

1. Implemented a serial implementation of `pfind` using BFS
2. Implemented recursive and iterative `pfind` versions
3. Prepared a dataset suite for benchmarking
4. Performed basic performance analysis of iterative `pfind` version

Work in progress

1. Implementing heuristics to allow more granular parallelism
   - Switching to priority queue to prioritize larger directories
2. Performing more thorough benchmarking with all competitors

Evaluation plan

Competitors for `pfind`

- Standard `find` that is used as baseline serial implementation
- MPI version of `find` from [1]

Scoring

1. Runtime
2. Resource usage

Evaluation plan: datasets

Datasets

- 1GB of Git repositories
  - “Awesome” list
  - Electron app framework
  - freeCodeCamp’s curriculum
  - JavaScript algorithm implementations
  - AMD MLIR AI engine toolchain
  - NVIDIA Linux GPU kernel module
  - Paper Minecraft server

Evaluation plan: datasets

- **Synthetic directory structures**
  - from high fan-out trees with few files to shallow trees with many files
- **Directories "in the wild"**
  - `/home/` on PACE and our own Unix home directories

Evaluation plan: challenges

1. **Varying Directory Sizes**
   - Large directories take too long to process
   - Small directories don't show parallelization benefits

Evaluation plan: challenges

1. **Varying Directory Sizes**
   - Large directories take too long to process
   - Small directories don't show parallelization benefits
2. **Ideal Directory Structure Hard to Find**
   - Structures showcasing parallel benefits are rare
   - Need balance in depth, file count, and size

Evaluation plan: challenges

1. **Varying Directory Sizes**
   - Large directories take too long to process
   - Small directories don't show parallelization benefits
2. **Ideal Directory Structure Hard to Find**
   - Structures showcasing parallel benefits are rare
   - Need balance in depth, file count, and size
3. **Modern vs. Older Hardware**
   - Default `find` is efficient on new hardware
   - Older systems with slow I/O benefit more from parallelization

Benchmarking results

Git:

- `pfind` does better on real time, but uses more compute time.
- 1.8x to 4x speedup on real time, but 2x to 4x as much compute time

Benchmarking results

Git:

- `pfind` does better on real time, but uses more compute time.
- 1.8x to 4x speedup on real time, but 2x to 4x as much compute time

Synthetic:

- Similar performance between `pfind` and `find` (but more compute time)

Benchmarking results

Git:

- `pfind` does better on real time, but uses more compute time.
- 1.8x to 4x speedup on real time, but 2x to 4x as much compute time

Synthetic:

- Similar performance between `pfind` and `find` (but more compute time)

In the wild:

- `pfind` is usually about 2x faster, sometimes 2x slower

Conclusions

- The main bottleneck is I/O, so it's hard to get lots of speedup
- Faster real time/wall time comes at higher computation time cost
- Computing heuristics takes time and resources, which incurs overhead

Bibliography

[1] E. Ong, E. Lusk, and W. Gropp, “Scalable Unix Commands for Parallel Processors: A High-Performance Implementation,” pp. 410–418, 2001.