PFIND: Parallel File Inspection and Node Discovery

Denis Koshelev
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
koshelev@gatech.edu

Aditya Kaushik
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
akaushik48@@gatech.edu

Ryan Thomas Lynch
College of Computing
Georgia Institute of Technology
Atlanta, Georgia
ryan.lynch@gatech.edu

I. Abstract

The Linux `find`[1] command does not currently support multithreading or any form of parallelism, which can be problematic in larger codebases. We propose a modified version called `pfind` that introduces multithreading using OpenMP. We benchmark it against actual `find` on multiple datasets.

II. Introduction

Currently, the `find` command operates sequentially and doesn't utilize the multiple processors available in modern systems. Our objective is to create a native implementation that supports parallel execution.

Our version, `pfind`, is designed to optimize search efficiency, particularly in systems with extensive data sets or intricate directory structures. By dividing the search task across multiple processors, we aim to significantly reduce the time required to locate files. This is especially crucial in environments where rapid file retrieval is key, such as in large data centers or complex computational scenarios.

We developed multiple versions of `pfind` that implement parallel BFS with different granularity: recursive, queue-based, and a lock-free modification of it. Prior to this work, we did not find any parallel implementation of the `find` command using OpenMP. We found an implementation of multiple Linux commands [1], including `find`, based on MPI. However, the scope of our project is to make it faster on shared memory devices, so we do not concentrate much on distributed memory ones.

III. Background

The `find` command searches a given directory recursively and prints out all the files and directories with a given target name. Note, this refers to one command option of the Unix `find` command that we will specifically be focusing on, which is find by `-name`.

Notably, this command searches the directory tree exhaustively, not stopping until it has processed every file in the subtree starting from the start directory. Thus, we can model this as a graph traversal problem, which in this case is traversing all the nodes and outputting the path of those nodes that have a given name. There are many traversal strategies, of which the two most well known are Breadth-first-search and Depth-first-search.

For this problem, we decided to implement the directory search with a BFS traversal, although it is noted that one of our three approaches does not use this strategy. This would consist of keeping track of a queue of directories which are modeled as nodes in the tree. These would be pushed onto the queue as we discover them in the traversal, and popped off the queue as we process them.

As far as parallelism, we used the OpenMP framework to add parallelism to our implementations. For this problem, we see that there is an oppurtunity for the directory traversal to be parallelized as different paths of the tree can be searched completely independently, as they have no interdependences.

IV. Project contribution

We created three different implementations of parallel `find` that all have the same functionality and correctness. We detail below how each of the implementations work and the parallelization used within each one.

Recursive `pfind`. Our first approach uses a function `process_directory()` that is the core of this implementation. For each entry, if it's a directory and not a special directory, a new OpenMP task is created to recursively process that directory in parallel using this helper function.

The parallel tasks are managed within an OpenMP parallel region, initiated with `#pragma omp parallel`, and the actual task distribution is handled using `#pragma omp single`, ensuring that the directory listing and task spawning are done by a single thread, while the actual processing of directories is distributed among available threads.

This implementation is straightforward and simple, requiring minimal additional lines of code on top of the serial `find` implementation while still offering some parallelization benefits. The listing of directory entries and the creation of tasks

[1] find(1) - Linux manual page

for each subdirectory are serialized, which we plan to address in our future implementations.

Queue-based `pfind`. Our second approach implemented a BFS traversal using a queue that would serve as a sort of task queue, with one thread popping from the queue and creating tasks that could be picked up by any available thread, and having all threads being allowed to push tasks to the queue.

The task queue works by storing the directories that are on the frontier of the BFS traversal. Thus, the algorithm starts by pushing the root directory to the queue, indicating that this is the start node of the traversal. In addition, we initialize an atomic integer `active_tasks` to keep track of the number of tasks that are currently running. The use of this will be explained later.

After this, one thread is responsible for popping directories from the queue and creating tasks for this using the OpenMP directive `#pragma omp task`. This directory queue modification is guarded by locks as this queue structure is modified by many threads. In addition, upon popping the directory from the queue, the variable `active_tasks` in incremented. The OpenMP tasks being created is one which uses the path of the directory which was popped from the queue by the single thread. This task involves iterating through all the directory entries of that particular directory, pushing all the dirents which are type directory to the directory queue, and printing out all the dirents which have the same name as the target name value. Upon finishing this, the directory is closed and `active_tasks` is decremented.

The use of this atomic integer is to keep track if there are any active tasks running. This is important because there could be a case when the directory queue is empty but there are still tasks running. Thus, if we only check for the directory queue being empty when running the single thread, then we could end the program before exhaustively searching the directory tree if we run into the case where there are tasks running but the queue is empty. Things may be later added to the queue, but the thread popping from the queue has terminated so we don't check anymore. Thus, we instead check if either the queue is empty or if there are any active tasks running, that way we don't stop the traversal prematurely. As far as this implementation goes, for concurrency there are locks around each access and modification to the directory queue, and the integer used to store the number of active tasks is atomic, so concurrent modifications are handled accordingly.

One problem with this approach is that while searching the directory entries and finding entries with the target name is parallelized, the process of popping directories from the queue and creating tasks is still serial. Thus, we also wanted to develop an approach that would parallelize both the pushing and popping from the queue. This would require a different approach than was implemented before.

Lock-free queue-based `pfind`. Our third approach sought to parallelize both the pushing and popping from the queue. Instead of having one thread pop from the queue and multiple threads push to the queue and process the directories, we will have all the threads pop from the queue and process the directories. In this implementation, instead of using the standard STL queue, we instead use the Boost lockfree queue. We have each available thread run a function called `searchDirectories()` by wrapping it in the OpenMP directive `#pragma omp parallel`. This function essentially keeps popping from the directory queue and does the same task as before, which is iterating through all the directory entries, printing the entries with the target name, and pushing the subdirectories to the queue. All threads keep doing this until the queue is empty or there are no active tasks running. Once this is over we know that the entire directory tree has been explored, so the program stops. This approach ideally gives a greater degree of parallelism because all the threads are able to be parallelized to run all the tasks included in the traversal. The lockfree queue, while not strictly necessary, made programming the queue accesses and modifications much more simple due to not having to worry about dealing with queue access race conditions. We refer to this version as `b_pfind` in out tests since we use Boost library for a lock-free queue.

V. Evaluation

A. Implementation

As discussed, we used OpenMP and C++ to implement our program. Our lock-free queue uses the Boost library, and we compiled for our tests with Boost version 1.79.0. We used GCC version 10.3.0 as the compiler. Our source code is available on our GitHub repository along with the scripts we used to collect our data.

B. Experiments

1) *Collection:* We used the hyperfine command-line utility to control our runs, measure execution time across runs, and to create our charts. For the Git, synthetic, and "in-the-wild" tests, we used 3 warmup rounds and 50 data collection rounds for each command. For the strong scaling tests, we used 3 warmup rounds and 25 data collection rounds for each command.

2) *Datasets:* We had three different datasets: Git repositories, synthetic directories, and "in-the-wild" tests using the `/usr` repository on the ICE PACE cluster. For the Git repositories, we used eight popular repositories from GitHub which contain about 1GB of data across almost 100K files and directories. For the synthetic directories, we created a script that creates about 28K files and directories across two directories, one with two levels of directories with a fanout of 20 and 20

files per leaf directory and one with five levels of directories with a fanout of 5 and 5 files per leaf directory.

3) *Competitors:* We compare two of our programs, queue-based `pfind` and lock-free queue based `b_pfind`, with two other programs, UNIX `find` version 4.9.0 and `fdfind` version 9.0.0 available from https://github.com/sharkdp/fd. `fdfind` is a parallel upgraded version of the `find` utility written in Rust and extensively tested, so we expect that it will outperform our programs, though we hope to achieve performance near it.

We intended to use an MPI implementation of find created by Argonne National Laboratory [1], but the source code link was broken and the only source code available was outdated and did not include the source for the find program.

To test scaling, we compiled different versions of our `b_pfind` executable (as we believed it would perform best) with fixed thread counts of 1, 2, 4, 8, 16, 32, and 64. For the other executables, we used the default number of threads provided by the OpenMP environment, which is either determined by the environment variable `OMP_NUM_THREADS` or by defaulting to the number of cores.

C. Machine

We tested on the ICE PACE cluster. For each test, we used one node on a 32-core machine, which were usually AMD EPYC 32-core processors.

D. Results

1) *Git Tests:* For our Git tests, the performance was very similar across the board between our implementation, `find`, and `fdfind`. The only notable performance difference is that our `pfind` executable performed slightly worse than the rest of the competitors, approximately 2x slower.

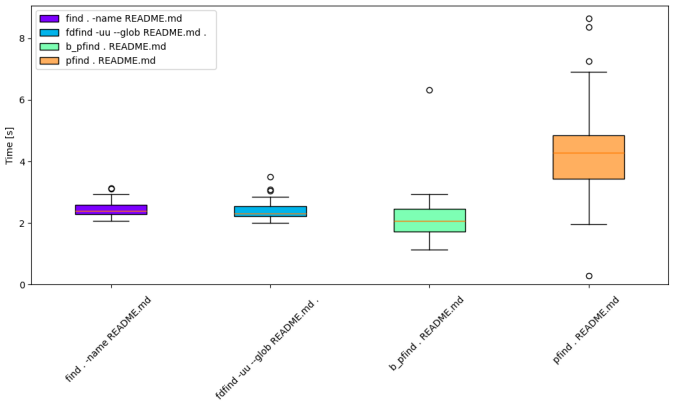

Figure 1: Queries on 1GB of Git repositories on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, linear scale on runtime axis

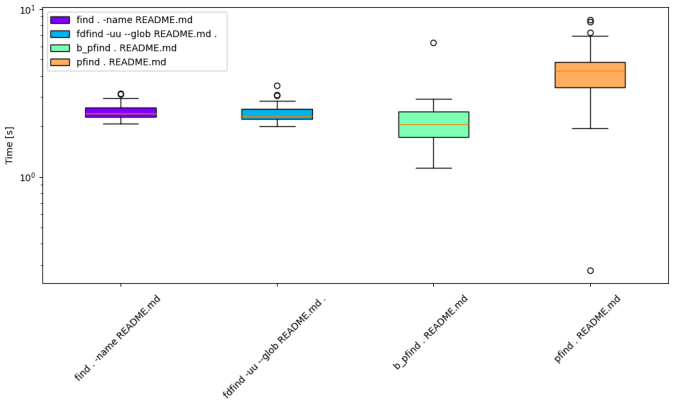

Figure 2: Queries on 1GB of Git repositories on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, log scale for runtime axis

2) *Synthetic Tests:* Our synthetic tests showed different results depending on the fanout of the directories. For the shallow directory with high fanout, which has about 8,400 files and directories to traverse, our implementations performed about as well as the default `find`, and consistently much slower than `fdfind`. However, for the deeper directories with lower fanout, which had about 19,500 files and directories, our implementations performed much closer to the performance of `fdfind`, and outperformed `find` by a factor of 11x. When we tested a search across both directories (not shown), the performance was very similar to a search across just the larger directory, which makes sense as it likely dominated the slowdown for `find`.

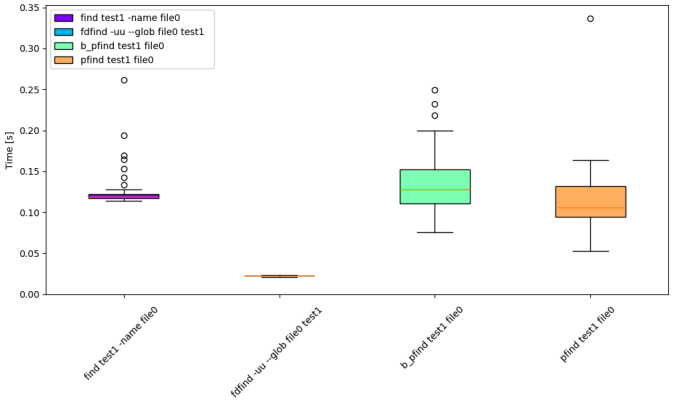

Figure 3: Queries on shallow synthetic directories with fanout of 20 on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, linear scale on runtime axis

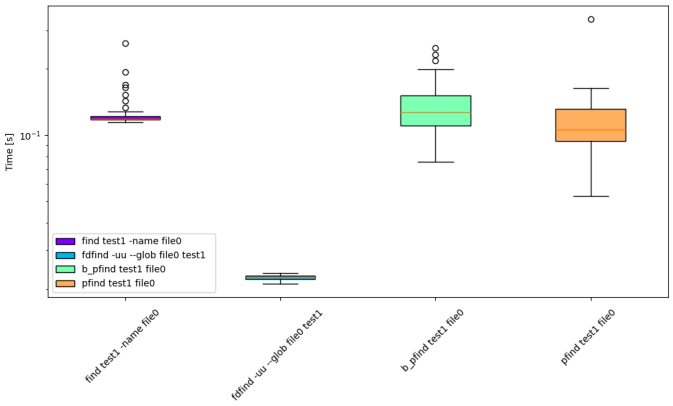

Figure 4: Queries on shallow synthetic directories with fanout of 20 on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, log scale for runtime axis

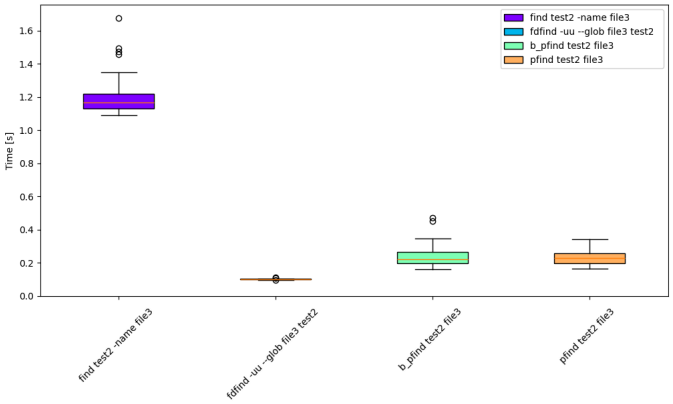

Figure 5: Queries on deep synthetic directories with fanout of 5 on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, linear scale on runtime axis

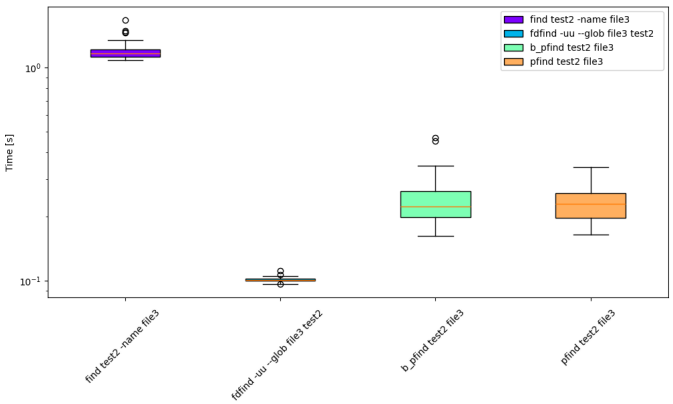

Figure 6: Queries on deep synthetic directories with fanout of 5 on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, log scale for runtime axis

3) *"In-the-Wild" Tests:* We performed tests using the `/usr` directory on ICE PACE which contains about 250K files and directories to traverse. We expected that this would be a good test to show how our program performs when there is higher disk latency and many more files to traverse, but the results tell a different story. There was a much higher variability in the latency of our programs than expected, as you can see that the whisker bars in Figure 7 and Figure 8 for our programs show that sometimes our programs were up to 1.5x faster than `find`, but other times they would be 3x slower. While the nature of accessing the `/usr` directory on ICE PACE should lead to some variance in the latency of disk accesses, we cannot explain this much difference in latency for our overall program without assuming some sort of defect in our programming which amplifies certain latencies.

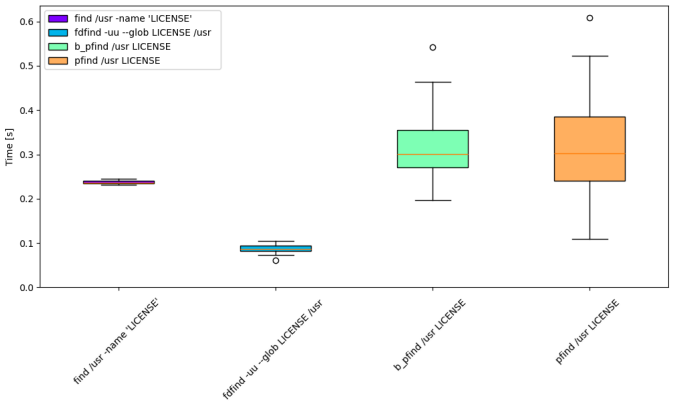

Figure 7: Queries on `/usr` on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, linear scale on runtime axis

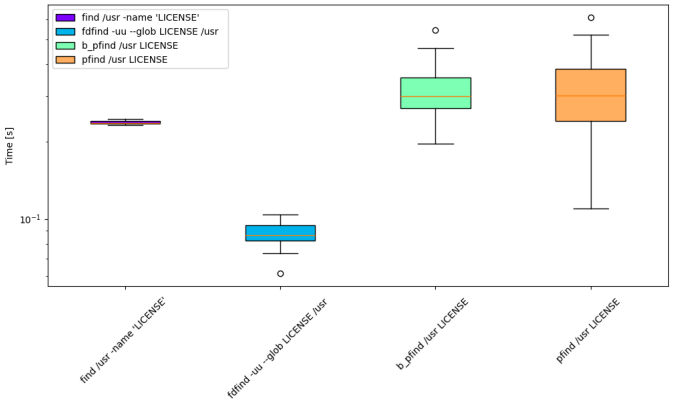

Figure 8: Queries on `/usr` on a PACE 32-core machine with 3 warmup rounds and 50 rounds of data collection for each command, log scale on runtime axis

4) *Strong Scaling Tests:* For the strong scaling tests, we tested the `/usr` directory and the Git repositories, and it showed that our program `b_pfind` does scale with more threads, but the performance again varies greatly. For the Git repositories, 16 threads performed the best, while fewer than 10 threads performed the best for the `/usr` directory.

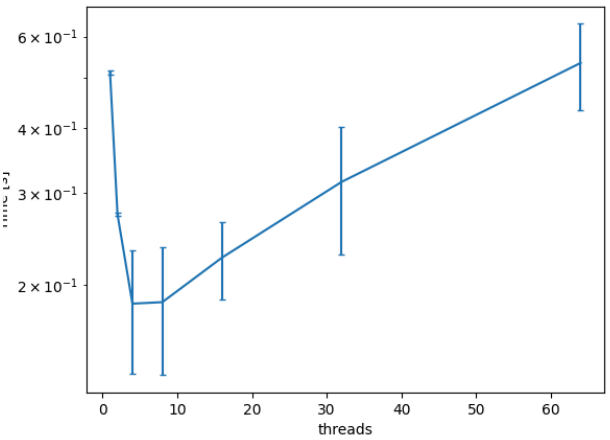

Figure 9: Queries using `b_pfind` with fixed number of threads on `/usr` on a PACE 32-core machine with 3 warmup rounds and 25 rounds of data collection for each command, log scale on runtime axis

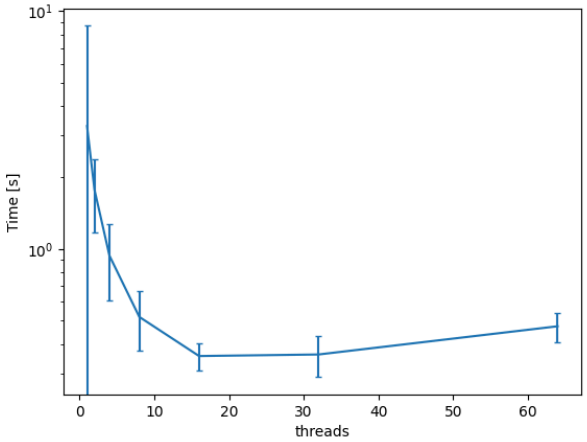

Figure 10: Queries using `b_pfind` with fixed number of threads on 1GB of Git repositories on a PACE 32-core machine with 3 warmup rounds and 25 rounds of data collection for each command, log scale on runtime axis

5) *User and System Time:* For all of our tests, the user time and system time (representing the total time spent across all threads) was always higher for our programs than the real time, meaning that the overall computational cost of running our programs is much higher even when there is a real execution time savings.

VI. Conclusion

We have developed several parallel versions of the Linux `find` command, which we believe could enhance the existing utility by introducing an optional flag for enabling parallel search capabilities. Although our comparative testing revealed that the traditional `find` command outperforms our parallel versions in certain scenarios, there are still instances, particularly with large file systems, where the benefits of parallelism and multi-core utilization are clear. This suggests that our parallel implementations could provide significant performance improvements in specific contexts. Furthermore, this was achieved using straight-forward techniques, such as lock-free queues and parallel BFS, that could be complemented with additional features, such as heuristics for understanding what folder should be prioritized by parallelization. We believe this could further improve the performance of `pfind`.

VII. Breakdown of contributions - 1 point

Team member	Contribution
Denis Koshelev	Implemeneted recursive `pfind` Debugged queue-based `pfind`
Aditya Kaushik	Implemeneted STL queue `pfind` Implemented boost queue `pfind`
Ryan Thomas Lynch	Developed testing metholodogy Gathered and evaluated datasets Wrote testing harnesses Collected testing data Created results graphs

References

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