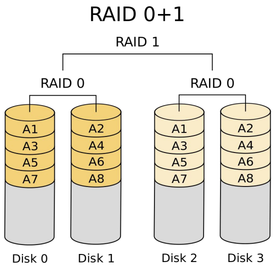
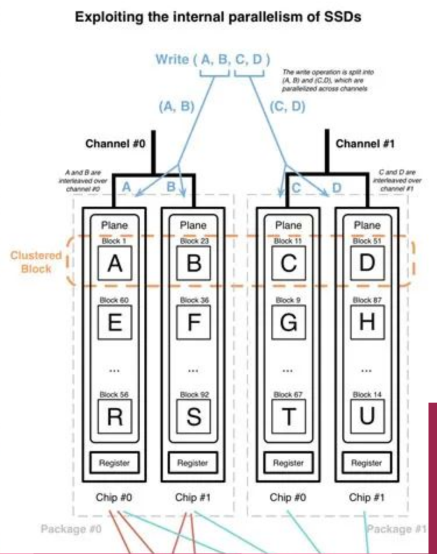
Concurrency-Aware Tree/Graph Traversal Algorithms

Randy Collado, Taishan Chen, Yizheng Xie

Background

- Parallelism in data storage
  - *Multiple* drive: Distributed, RAID
  - *Single* drive: **Solid-State Disk (SSD)**
- SSD has **internal concurrency**
  - Can handle multiple requests simultaneously
  - Serial operations cannot achieve best performance
- Parallel algorithm for data
  - Tree/Graph structure widely used in data and file system
  - Traversal & Searching Algorithms



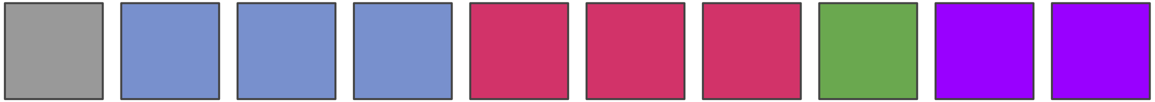


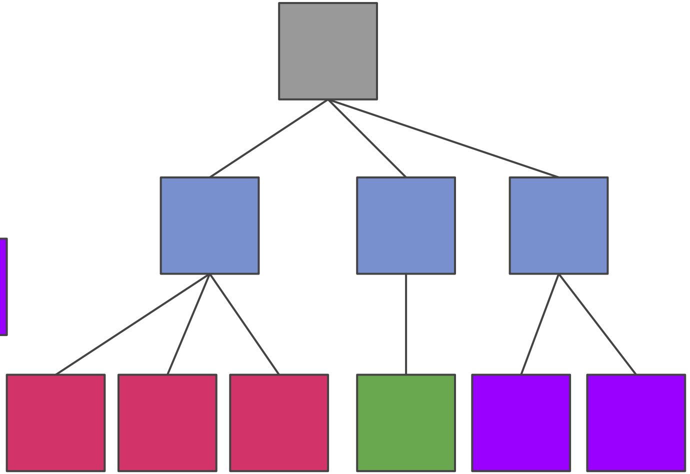
Coding for SSDs – Part 5: Access Patterns and System Optimizations, Emmanuel Goossaert

Algorithm - Serial BFS

Breadth-first Search

Layer by layer





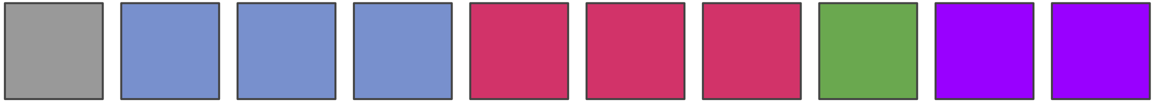
Implement by a **queue** (FI**F**O)

Pop_**front**() and push_**back**() its all descendants

Parallel BFS

Nodes in **one layer** can be processed **in parallel**

Straightforward

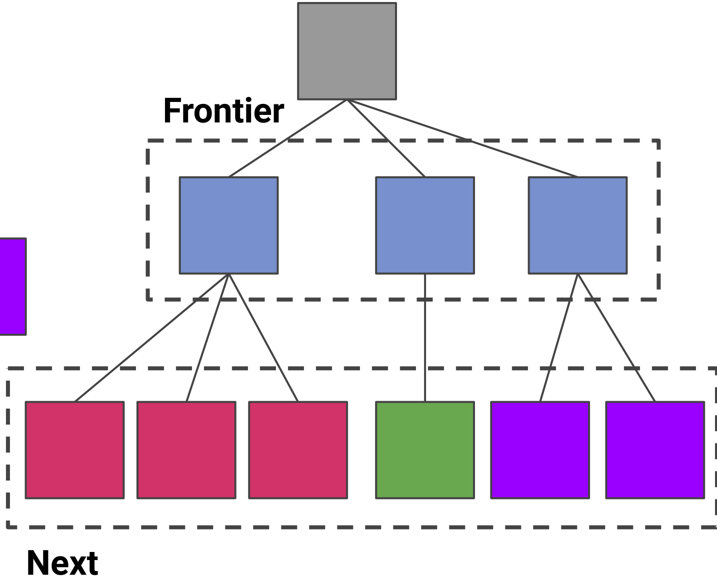


Need to distinguish **current** layer and **next** layer

Frontier and **Next** Queue

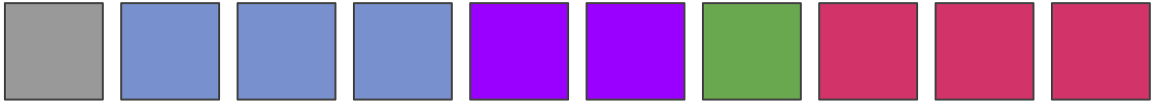
Loop *frontier* in parallel, push descendants in *next*

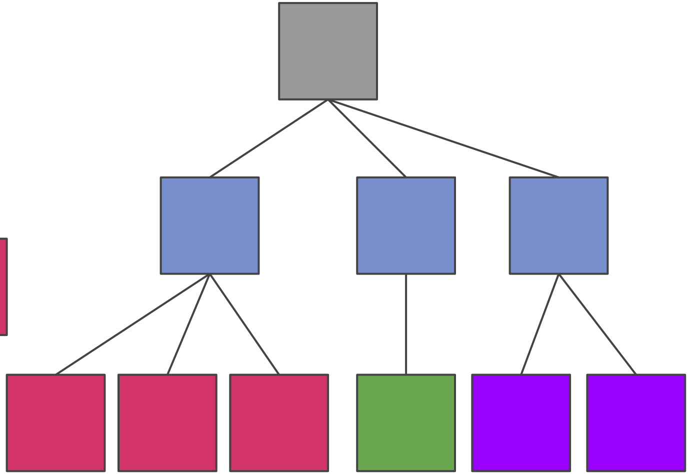
Frontier = Next



Serial DFS

Depth-first Search





Use **Stack** (FI**L**O)

Pop_**back**() and push_**back**()

More common in searching

Parallel DFS

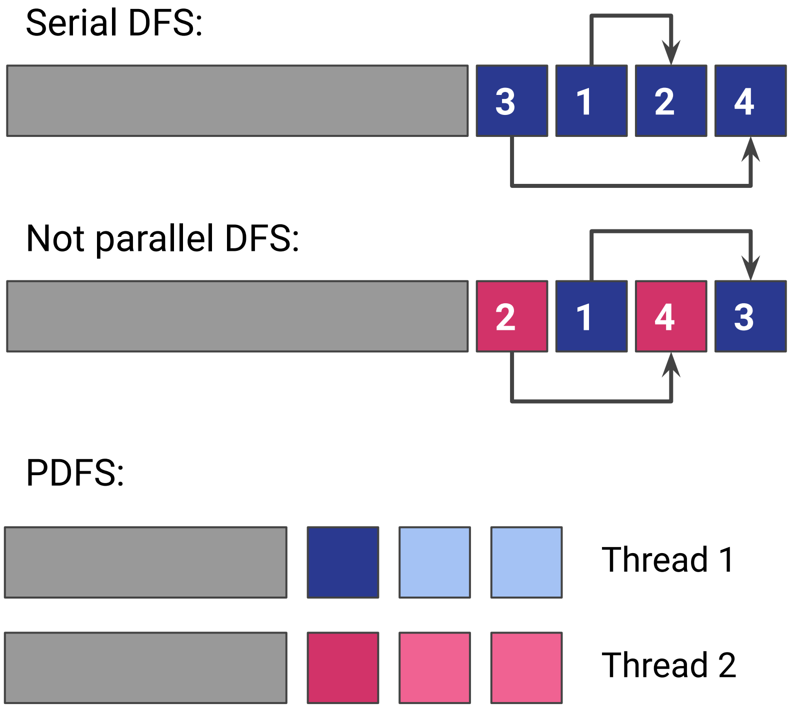
Each processing relies on last result

Both operation are on stack top, have to wait

A *strict order* DFS **cannot** be paralleled efficiently

Unordered/Pseudo Parallel DFS: DFS for **each thread**

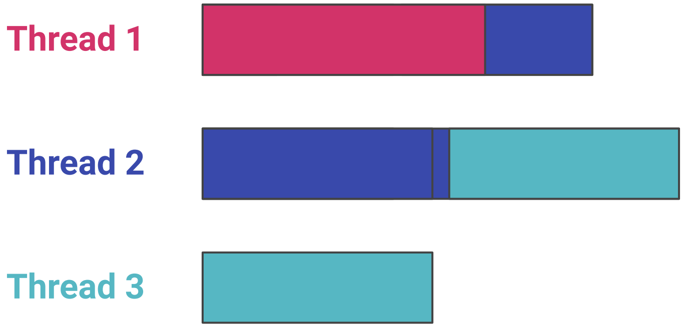
Not *globally* depth-first, but still prefer depth



Parallel DFS - cont.

How to distribute nodes to threads?
Keep all threads busy for most parallelism.

- Set a **threshold** size **(fSize)** for stack
- Each thread works on own stack
- When one thread's stack is larger than threshold
  - **Split** into two part
  - Given one part to a **new** thread



We let the OpenMP automatically handle the scheduling.
For better performance, control scheduling and balancing.

Acar, Umut A., Arthur Charguéraud, and Mike Rainey. **"A work-efficient algorithm for parallel unordered depth-first search."**, 2015.

Code

OpenMP

OpenMP: Compiler-level directives, no/minimum changes on codes.

Serial and Parallel **BFS**:

```
std::vector<int> frontier, next;
bool isFound = false;
while (!isFound && frontier.size() > 0):
  for offset in frontier:
    node = read_node(offset);
    if (node.key == key) isFound = true;
    if (isFound || node's children count == 0)
      continue;
    next.insert(node.children);
  frontier = next;
  next.clear();
```

```
std::vector<int> frontier, next;
bool isFound = false;
#pragma omp parallel
while (!isFound && frontier.size() > 0):
#pragma omp for nowait
  for offset in frontier:
    node = read_node(offset);
    if (node.key == key) isFound = true;
    if (isFound || node's children count == 0)
      continue;
#pragma omp critical
    next.insert(node.children);
  frontier = next;
  next.clear();
```

Code

Serial and Parallel **DFS**:

```
std::vector<int> frontier;
bool isFound = false;
frontier.push_back(0);
while (!frontier.empty()):
  node = read_node(frontier.back());
  if (node.key == key) isFound = true;
  frontier.pop_back();
  frontier.insert(node.children);
return isFound;
```

```
std::vector<int> frontier;
bool isFound = false;
frontier.push_back(0);
#pragma omp taskgroup
while (!frontier.empty()):
  node = read_node(frontier.back());
  if (node.key == key) isFound = true;
  frontier.pop_back();
  frontier.insert(node.children);
  while (frontier.size() > fSize):
    frontier, frontier_new = frontier.split();
#pragma omp task shared(isFound)
    if (run(frontier_new)){
      isFound = true;
#pragma omp cancel taskgroup
    }
return isFound;
```

We also implement IDDFS and some hybrid approach, see code if interested.

Tree Structure

- Our testing was conducted on a randomized tree-like graph, with control over its branching factor, the branching factor of the individual nodes, and the number of values allowed per key
- This offers significant flexibility is studying configurations that are beneficial for concurrent search performance
- This also allows us to test worst and best-case scenarios for our search algorithms

File Structure

- Memory-mapped structs allow efficient byte-level access of all the data within each node of tree
- Tree is serialized in BFS order, converting each node encountered into an S_Node, a node format that allows for efficient concurrent access of a node and it's children
- S_Node structs store the offsets of the children nodes in the file, making the file compatible with both DFS and BFS-based search schemes[1]
- Low-level syscalls allow us to avoid OS intervention and get more accurate results vs. Standard Library

1. Sussenguth, E. H. (1963). Use of tree structures for processing files. Communications of the ACM, 6(5), 272–279. https://doi.org/10.1145/366552.366600

Serializer and Operators

- File is opened via the open(2) syscall (or CreateFileA on Windows) with the appropriate flags (see Direct I/O)
  - The file descriptor is stored in the serializer object for reading from and writing to the file.
- Operators:
  - **S_Node* readNode()**: reads node at current fd position
  - **S_Node* readNodefromOffset(size_t offset)**: reads node at position offset bytes forward relative to start of file
  - **void writeNodeWithOffset(size_t offset)**: writes node at position offset bytes forward relative to start of file
  - **void write_offset_metadata()**: unused
  - **void read_offset_metadata():** unused

```
struct S_Node {
    int key;
    int numChildren;
    int payload[8];
    int children[8];
}
__attribute((aligned(512)));
```

```
struct Node {
    size_t numChildren;
    size_t maxChildren;
    size_t numValues;
    size_t maxValues;
    int values[8];
    Node* children[8];
}
```

File I/O

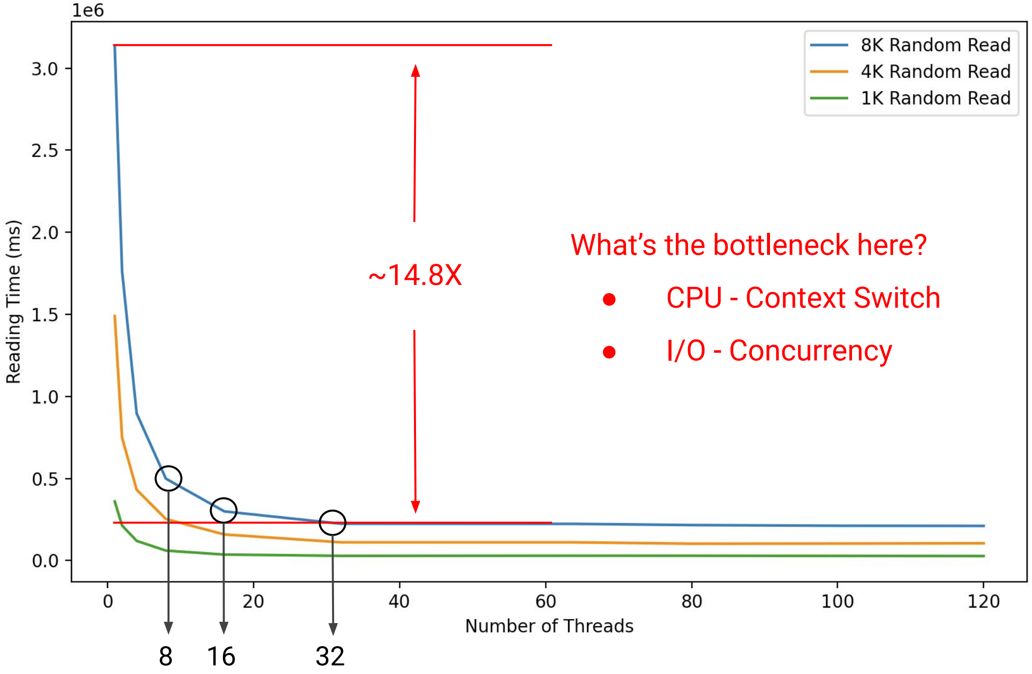
No caching: Test concurrency of **storage**

Concurrent: I/O operations should ***not*** be **serialized** by OS somehow

No universal libs for now, have to use system calls.

- SSD device which supports concurrent I/O
- O_DIRECT, pread/pwrite (Linux)
  - Windows: FILE_FLAG_NO_BUFFERING, FILE_FLAG_OVERLAPPED, GetOverlappedResult()
- Read/write with offsets

Experiments - Direct I/O (Random Read)



Experimental Setting: 8-CPU

`omp_set_num_threads`

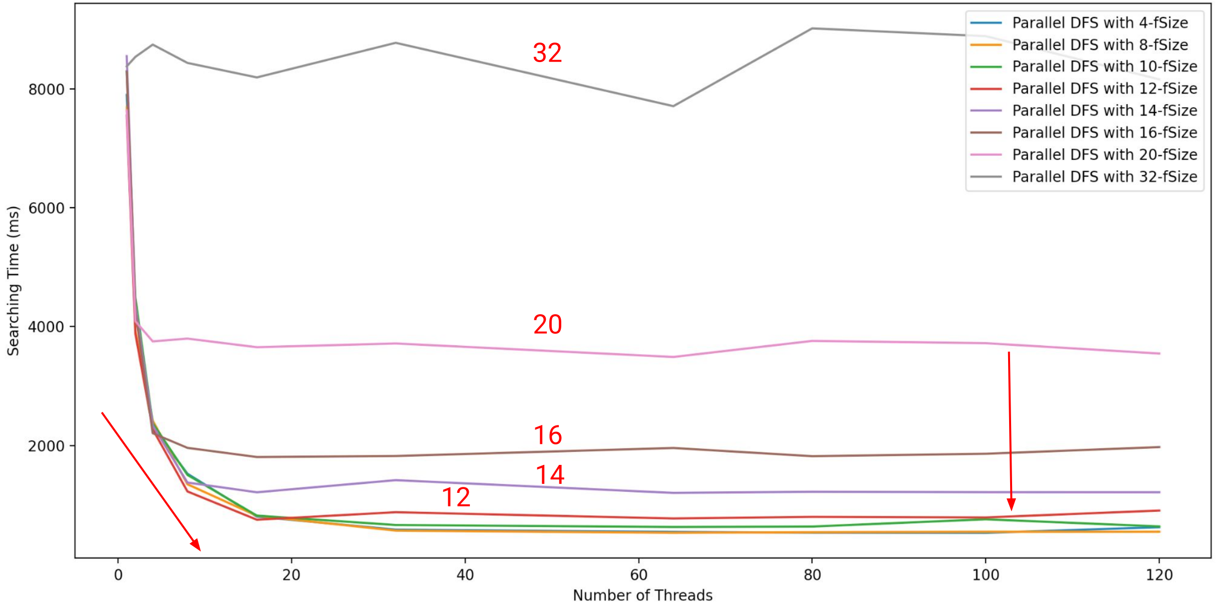
Summary The `omp_set_num_threads` routine affects the number of threads to be used for subsequent parallel regions that do not specify a `num_threads` clause, by setting the value of the first element of the *nthreads-var* ICV of the current task.

Format

C/C++

```
void omp_set_num_threads(int num_threads);
```

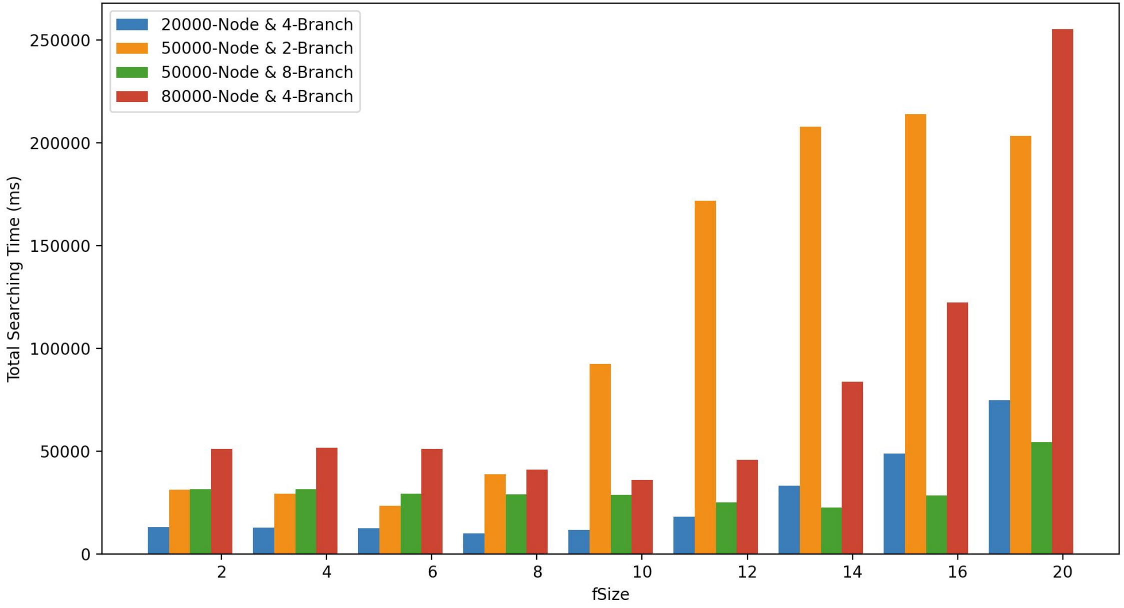
Experiments - Tuning fSize (Target Key Not Exist)



Experimental Setting:

- Num of Nodes: 20000
- Branch Size: 8
- Target Key Not Exist
- 8 CPU

Experiments - Tuning fSize (Random Target Key)



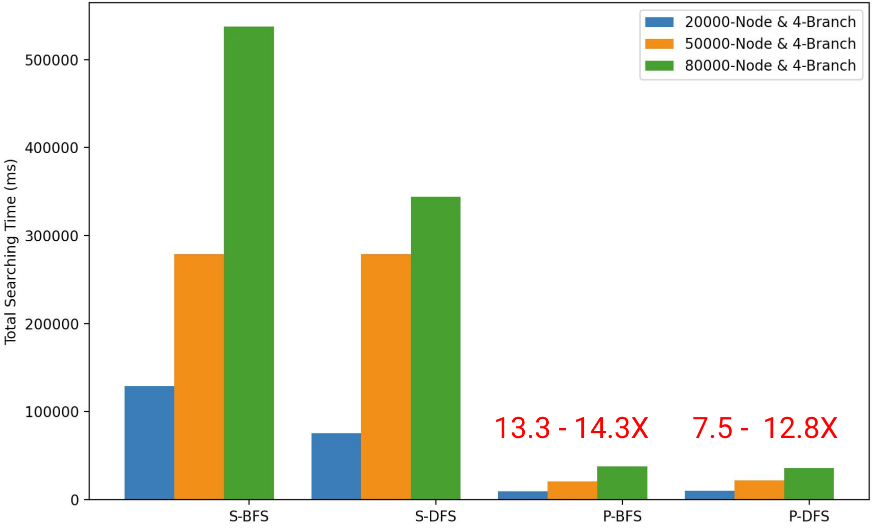
Experimental Setting:

- Num of Nodes: 20000/50000/80000
- Branch Size: 2/4/8
- 20 Random Target Key + 1 Not Exist
- 8 CPU & 32 Threads

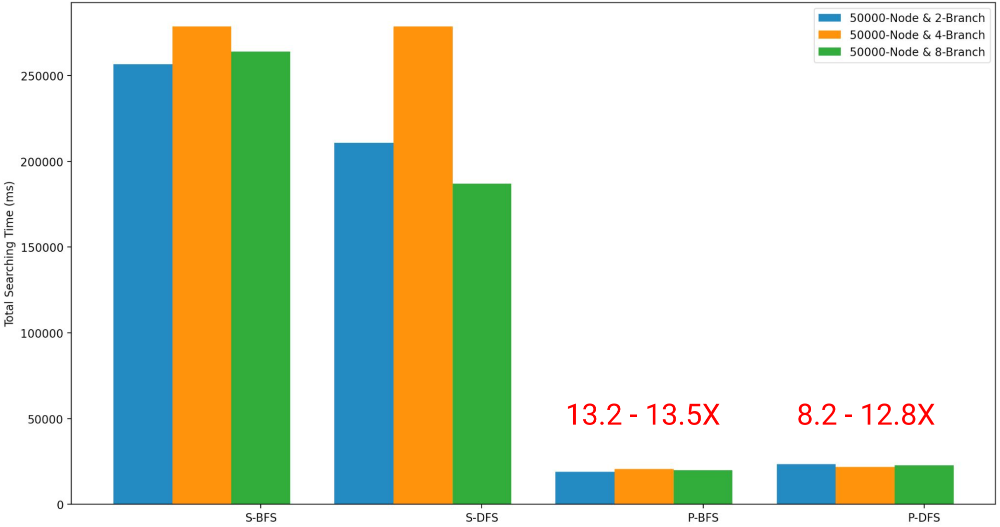
Experiments - Algorithm Efficiency

Experimental Setting:

- 20 Random Target Key + 1 Not Exist
- 8 CPU & 32 Threads



Change Num of Nodes



Change Branch Size

Future Works

- Algorithm
  - Better controls over scheduling and task stealing
  - Hybrid search, adjust parameters automatically
  - Extend algorithm to general graph structures
- Testing
  - More different workload
  - I/O Matrices like IOPS

Some Lessons!

- Try and choose way of implementation wisely.
  - Go: chan, operator(<-, ->)
  - C/C++: Std::thread, OpenMP
- Test in correct way.
  - All tests in memory before midterm, no performance gain, puzzled.
- Leave time for debugging!
  - especially for concurrent programming…

Thanks!

Reference

Acar, U. A., Charguéraud, A., & Rainey, M. (2015, November). A work-efficient algorithm for parallel unordered depth-first search. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-12).

Korf, R. E. (1986). Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 28(1), 123. https://doi.org/10.1016/0004-3702(86)90035-4

Sussenguth, E. H. (1963). Use of tree structures for processing files. Communications of the ACM, 6(5), 272–279. https://doi.org/10.1145/366552.366600