Final Project - Skippy the Skip List

Derek Prince

Overview

For this project I chose to implement a concurrent skip list from primitives such as `std::atomic_flag` s and pthreads to hand the creating and joining of threads. Fine-grained synchronization was achieved through hand-over-hand locking to support multiple thread's accesses. The main deviation came from my implementation keeping forwarding pointers in an array so whenever a new node was added that needed to be inserted in the upper levels, it would have to go back and reacquire the lock for the other forwarding nodes to ensure function.

Assumptions, Decisions, and Trade-offs

For this project I simply had nodes keyed in with an integer value that was pulled from a randomly generated text file. In a non-academic setting there would likely be a data field that gets hashed into the key but as this was about contention and parallel performance, the existence of a data blob attached to each node adds no real value to the analysis.
Additionally, I did not add a remove function as it was not requested in the Final Project requirements. I did implement sequential and parallel `insert` , `get` and `range` as requested, with `range(low, high)` returning the set of items including both low and high. As this code is not running off into the wild I did not attempt to cleanse inputs and check things like upper/lower bounds to ensure they were passed in the right order so that case is undefined and likely will just return an empty vector at best.
For the trade-offs, in choosing to only forward on nodes that exist in upper levels it forced me to re-acquire locks further away that just current and precious. One alternative would be to have all nodes prior forward to the next node in an upper level so that any entry point could move as if all in between nodes are transparent, leaving the lock acquisition closer to a linked-list hand-over-hand locking implementation. The risk here is the number of disconnected nodes at upper levels that would need to have their pointers forwarded on insertion. That would also create a very

large restructuring overhead on many inserts so I moved forward with paralleling my sequential implementation.

Code Structure and Organization

The skip list is implemented in a c++ class file pair named skippy.cpp and skippy.h while the threads are managed in `main()` as this is just a benchmark. There are forwarding functions called from `pthread_create()` to redirect the function call to the class member function. The only reason this was necessary was to maintain the implicit `this->` pointer for member access while doing node traversals. Otherwise it would just have been a plain C implementation with C++ i/o. Memory order was acquire/release whenever possible.

Benchmarking

Benchmarking was done with a randomly generated input file of 1000 integers ranging from 0 to 10000 over 100 threads. If another file is desired, one can pipe the output of ./tools/gen_rand into a file and direct the program executable to that for input (see: running section).
The program inserts all the numbers, queries a range on them, and fetches a `<random>` number from the skiplist, timing each section and printing them at the end. It should be noted that the `<random>` library seems to be broken by taking the timings so I manually checked for numbers in the end of the range to get a standard case.
After that, I manually programmed queries into main (which was ugly but effective) to spin up 100 threads and attempt to acquire the same node at the end of the list to measure maximum contention. Querying the end of the list also showed how the threads were navigating down the list rather than just vying for the lock on the last nodes. The random access was done by generating numbers from `std::uniform_int_distribution<int>` and accessing athe completed list with 100 threads.
Analysis was done using system clock and the gnu perf. The gnu perf command used was `perf stat -e L1-dcache-store,L1-dcache-load,L1-dcache-prefetches,branch-instructions,branch-misses,branch-loads,branch-loadmisses ./skiplist /input/input.txt -t 100`
Though the high contention cases are worse than random, the difference is not as much as most concurrent data structures I would imagine as all queries have to enter through `head` and acquire a lock on the way. So it is not quite as advantageous as one might think. Here is an example output from my tests earilier:

```
Skip list inserted in approximately 2073437 ns.
Skip list fetched range in approximately 18852 ns.
Skip list returned {8284} in approximately 36292 ns.
Skip list contention: 1296882 ns.
Skip list random access: 876923 ns.
```

Here we can see that random access took almost 70% of the time that it took for the high contention case. Though as we see it took 36292 ns to reach (nearly) the end of the list, having 100 threads try to race to random spots in light of that seems correct.

Build Instructions

Unzip file and cd into top level with the Makefile. The file structure should look as such:

```
├── main.cpp
├── Makefile
├── Readme.md
├── Writeup.pdf
├── include
│   ├── cpu_relax.hpp
│   ├── cxxopts.hpp
│   ├── skippy.cpp
│   └── skippy.h
├── input
│   └── input.txt
└── tools
    └── gen_rand
```

To build simply run `make` while in the FinalProject directory.
The makefile supports `remove` `uninstall` and `clean` though they do no error checking to see if the file-to-be-removed exists.

Execution Instructions

The program requires and input file directly after the program and then a number of threads specifier with `-t|--threads`. `skiplist --name` will also return author name like the labs.
`--help` output:

```
$ ./skiplist --help
Concurrent Skip List final project implemented with fine-grained hand-over-hand loc
```

```
Usage:
  ./skiplist [OPTION...] positional parameters

  -n, --name                 My name (for grading purposes)
  -t, --threads NUM_THREADS  Number of threads to be used
  -h, --help                 Display help options
```

Example used during benchmarking: `./skiplist ./input/input.txt -t 100`

improvements

For this project I first wrote a sequential version of a skip list to make sure I understood what was going on before converting it to a concurrent version. If I were to do this again I would have completely started from scratch with a new concurrent version of the skip list as its own class with better outside handlers written as well. When I started the conversion it went very nicely and functioned seemingly without error. Then i started using big data sets and edge cases to see how it performed and realized that there were some rare edge cases (talking 1 in 20 on a tailored data set) that would segfault the list, leading me to discover the need to go back and lock the previous nodes required for insertion forwarding on a new level. This resulted in a very ugly implementation that is hard to adjust. So I would redesign it completely with a locking scheme as the first and foremost design decision. It is because of this that I did not end up implementing the multiple readers extra credit past using `std::memory_order_acquire / std::memory_order_release` to let readers in faster.

Expected Input

A text file with one number per line up to the size of the range of an int.

Ex:

```
45
1
93
11
345
0
6816
9
```

Letters as far as has been tested are ignored and removed from the input. See extant bugs section for more.

Terminal Output

By default the terminal prints the entire list before getting to the timings so with large inputs it can look hairy.
Here is an example list print from a small data set:

```
Skip list contents
Level: 0 : 1 2 4 5 6 8 9 14 29 67 72 87 92
Level: 1 : 4 5 8 9 14 67
Level: 2 : 5
In range {2,12}: 2 4 5 6 8 9
In range {6,28}: 6 8 9 14
```

File Descriptions

File Descriptions

- **Readme.md**
  - The markdown version of this writeup (I use these in jekyll with prettier rendering is why .md)
- **DerekPrince_writeup.pdf**
  - You are here.
- **Makefile**
  - makefile to automate build with gcc.
- **main.cpp**
  - Entry point of program.
- **include/skippy.h**

    - Class definitions of all the skip list

- **include/skippy.cpp**

    - Class member definitions of skip list class

- **include/cxxopts.hpp**

    - CLI argument parser library

- **include/cpu_relax.hpp**

    - Contains support function cpu_relax() that fit nowhere else.

Credits

- cxxopts for command line parsing because nobody but C++ needs to solve this issue again.
- ifknot For their asm memory relax instruction to ease off the spin locks.
- stackedit for a cleaner render.

Extant Bugs and Undefined Behaviour

As this code has no life out in the wild, I did not take any extra steps to go cleanse inputs to guarantee that they are always in the form expected and so undefined inputs should be treated as errors even if they return. Most likely the results will be empty but I did not test these cases as that is not what this project was about.
There is some rare case of operation that shows up on *some* datasets where the list will still segfault. This is very much not an intended result so I spent close to 8 hours one day trying to find the damn thing but it only shows up outside of debugging (as in if it is run from CLion in debug mode it never shows up. Ever.) and valgrind cannot determine my intents with atomic flags so helgrind could not find the problem, nor could memtest. So it lives another day until I can track it down. I believe it is a result of all the locks release and acquiring so many times, leading to a data race that cannot be solved by a simple atomic flag and somewhat fair locks (due to `cpu_relax()`).