Concurrent workshop

Implementations summary and conclusions:

Yotam Manne

Itay Levi

In this workshop we were asked to implement a concurrent priority queue in two different approaches, in addition to a concurrent SSSP algorithm.

Chosen implementations:

- **MultiQueue:** A relaxed priority queue based on multiple sequential priority queues. The data structure is comprised of an array of cp sequential priority queues for some constant c > 1, where p is total number of threads. Each sub-queue is a d-ary heap (in our case a binary heap). Insertions go to random queues. The idea behind this is that each queue should contain a representative sample of the globally present elements. Deleted elements are the minimum of the minima of two randomly chosen queues. By choosing two rather than one queue, fluctuations in the distribution of queue elements are stabilized.

- **Skiplist:** A lock free, linearizable concurrent priority queue, based on the skiplist data structure. The paper implies that in experimentation it was found that the global update operations in the DeleteMin operation are the bottleneck that limits scalability of priority queues. Due to that, in order to increase scalability, the number of such updates should be minimized. Accordingly, the main advantage of the algorithm is that almost all DeleteMin operations are performed using only a single global update to shared memory. This is achieved by not performing any physical deletion of nodes in connection with logical deletion. Instead - perform physical deletion in batches when the number of logically deleted nodes exceeds a given threshold. Each batch deletion is performed by simply moving a few pointers in the sentinel head node of the list. Thus only one CAS per DeleteMin operation is required (for logical deletion).

SSSP Algorithm:

Our concurrent SSSP algorithm, is based on Dijkstra's algorithm with a minor change:
After a vertex's 'distance' field is being updated, the vertex is re-inserted to the priority queue. This method ensures that if two or more threads are concurrently changing same vertex data, it would be re-checked due to being re inserted to the queue. According to that - if shorter path exists, the data would be updated again repeatedly until no improvement (shorter path) is found, and this conserves Dijkstra's correctness.

Implementation challenges:

Global:

- Mostly at the beginning and before termination of each execution, it is very likely that there are more available threads than nodes in the priority queue. In such case we wanted to send unnecessary threads to 'sleep' so they won't waste CPU time. We used the POSIX Semaphore object which stores 'credit' according to the available work – If a thread inserts a new node to the queue, it adds one token to the semaphore (opposite happens when removing a node). If a thread wants to extract a node from the queue but no node is available – the semaphore will block and send him to sleep, and wake him up when a new node is available.
- The algorithm's termination condition was a challenging issue. Logically – we would want to terminate when only one thread is awake and the queue is empty. Practically, due to race conditions, a thread might wake up after the terminating thread already saw that he's sleeping and the execution will be terminated before the algorithm is finished. To solve this, we first **atomically** check if the current thread is the only one awake, and then if the queue is empty. This order guaranties that no false termination will occur.

MultiQueue:

- MultiQueue's extract_min method requires comparison between the minimum elements of two randomly chosen binary heaps. During testing we noticed that it is challenging to safely inspect the head of the two. While the inspection is being done, there is a possibility that another thread will delete the head of one or both of the queues. We solved this issue by trying to lock (with CAS operation) both of the heaps before comparing the minimum. If only one heap was locked successfully – we unlock it and try again.
- According to our implementation of concurrent SSSP Dijkstra's algorithm- it is likely that more than one node pointing to same vertex can appear in different heaps in the Multi_Queue. This fact causes a problem when these nodes are concurrently checked (extracted from the different heaps) and have their fields updated, and so they are re inserted to the priority queue – possibly not with the minimal distance found (this case occurs with high probability due to the fact that a vertex with small distance from all current nodes in the queue is likely to be minimal in different heaps and so being concurrently extracted), and so is likely to cause wrong results. We solved this case by using a CAS operation to lock each vertex, such that one's data can be checked and updated exclusively and atomically.

Skiplist:

- We found the Skiplist paper very detailed and informative, and so we did not encounter implementation challenges beside normal coding bugs during our implementation.

Interesting heuristics:

As taught in class, when data is changed in a certain cache line, within a multithreaded shared memory environment, it causes invalidation of the same line to all other threads using it. When data stored in this line would be accessed by one of the other threads, a cache miss would occur and performance are likely to decrease. In order to try and minimize cache misses, we added to each node data structure (BH_Node, Skiplist_node) 'junk' padding up to full cache line size. The said above optimization had minor improvement, nevertheless, we decided to keep it as described.

Execution and Build instructions:

1. Navigate to the project directory:
   ```
   cd /path/to/concurrent-workshop
   ```
2. Use makefile to compile project:
   ```
   make clean
   ```
   ```
   make
   ```
3. Two executables will be created:
   ```
   main_multiqueue
   main_skiplist
   ```
4. Run the program with command line arguments as shown below (example for multiqueue):
   ```
   ./main_multiqueue [/path/to/edges_file] [THREAD_NUM - optional]
   ```
   Where edges_file is the input file, and THREAD_NUM is the number of threads to use (default is 80 if unspecified).
5. When the program terminates, an output file called ‘sssp_distances’ will be created in the project directory.

Files description:

sssp.h: Public declarations for function implementations of concurrent single source shortest path algorithm.

sssp.cpp: Function implementations of concurrent single source shortest path algorithm.

priority_queue.h: Definition of 'abstract class' "Priority_Queue", including constant definitions for the class.

priority_queue.cpp: "Priority_Queue" abstract functions implementations (only functions implementation which are shared by MultiQueue and Skiplist).

graph.h: Definitions of "Graph" and "Vertex" classes and of "neighbor" struct.

graph.cpp: "Graph" and "Vertex" classes functions implementations.

main_multiqueue.cpp: Main file for running sssp with "Multi_Queue" data structure.

multiqueue/binary_heap.h: Definitions of "Binary_Heap" and "BH_Node" classes.

multiqueue/binary_heap.cpp: "Binary_Heap" and "BH_Node" classes functions implementations.

multiqueue/multi_queue.h: Definition of "Multi_Queue" (inherits from "Priority_Queue") class.

multiqueue/multi_queue.cpp: "Multi_Queue" class functions implementations.

main_skiplist.cpp: Main file for running sssp with "Skiplist" data structure.

skiplist/skiplist.h: Definitions of "Skiplist" class (inherits from "Priority_Queue") and "Skiplist_node" class.

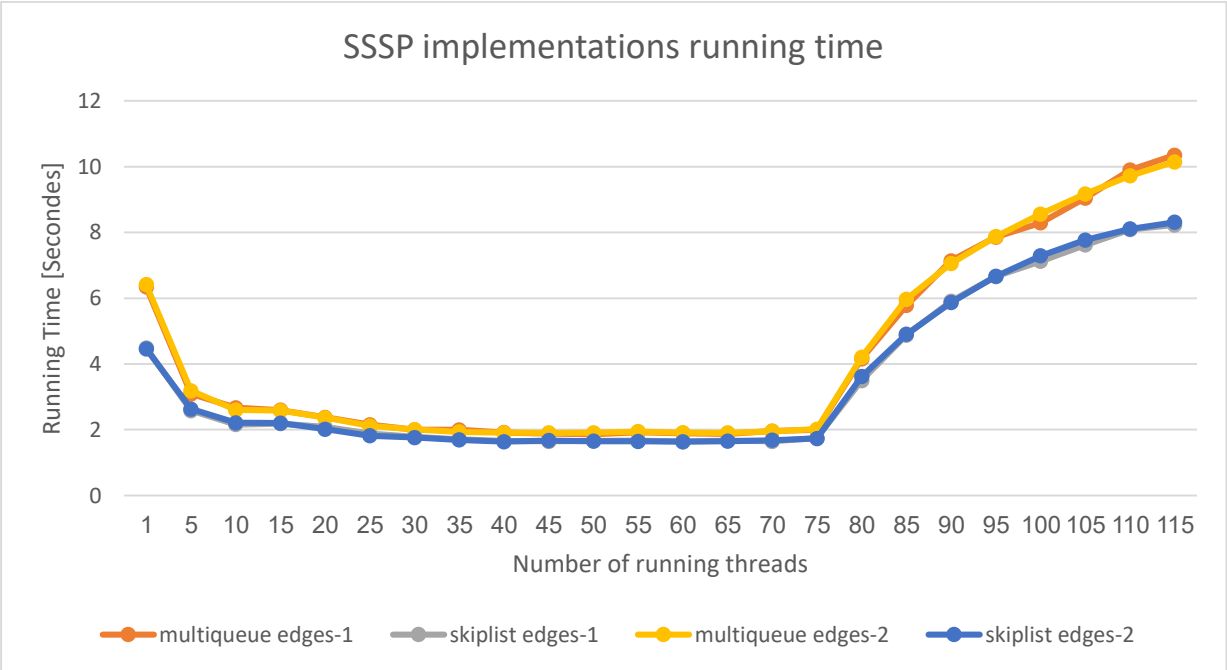
skiplist/skiplist.cpp: "Skiplist" and "Skiplist_node" classes functions implementations.

makefile: Makfile to build and clean the project objects and executables.

recordmgr: "Debra" memory manager library suggested by workshop staff (with small changes in Debug section due to compilation problems caused by the library original implementation – discussed and solved with teaching assistant Orr Fischer).

Results and Conclusions:

Out experiments included Averaging 3 running times for each implementation with different edges input files and with different number of running threads.
In our running experiments on the course server (Running times results figure is presented below) we found that our Skiplist implementation has better running time than our MultiQueue implementation. We assume that different implementation for d-heap – using d≠2 and using optimizations might accelerate running time of MultiQueue. Moreover, we assume different edges input files (for example having different graph diameter) might result different running times for the implementations.



Known bug:

On our running experiments over the course server we noticed that running of the concurrent_skiplist implementation, when number of running threads is set to 2/3/5 (only these numbers causes a fault, and only over the server) the program collapses due to segmentation fault. The backtrace implies that there is a problem when calling the debra library function mgr->leaveQuiescentState(tid). The problem is solved when the line calling this function is commented out (line 253 in skiplist/skiplist.cpp) and correctness of the output is maintained (done for timing expiriments).