# Description

I chose to write algorithms for concurrent containers (stacks and queues). I wrote a stack and queue implementation that uses SGL as a synchronization tool. The SGL was done using a test and test and set lock with a single global flag. I wrote a treiber stack implementation that uses compare and swap to spin. Next I wrote a Michael & Scott queue that uses compare and swap to enqueue and dequeue. I also wrote a flat combining stack/queue (FC Stack and FC Queue). This was done using helper functions from Moran Tzafrir (Copyright 2009). This was implemented using an abstract class provided in the library named ITest. Using the same libraries, I also implemented a baskets queue using ITest. Enqueue and dequeue were written with an included compare and swap function.

# Experimental Results

In this section we will examine the execution time of loading 100 values into my different concurrent data structures. This will hopefully allow us to run performance checks on how easily or easily items are added and removed from a data structure. It can be assumed that the longer the execution time for each data structure, the more contention or spinning that the locks must overcome in order to access the data structure. Regardless of execution time however, each data structure can have pros and cons, and execution time is not necessarily the golden benchmark for performance. Even though one may have a very fast execution time, it may experience starvation or livelock more than another.

Below are 7 graphs appropriately titled of execution time from 3 threads to 20. I chose to evaluate from 3 to 20 because outside of that range, the execution time would have very unpredictable behavior. In addition, anything more than 100 threads would actually slow the execution time down in a linear fashion. This is likely due to thread management and limitations by the hardware threads.

# Analysis of Results

I find that over all containers, the Flat Combining Stack has the most consistent and fastest execution times. Why the other containers have a high standard deviation of execution time I am not quite sure. However, out of all data structures, the FC Stack seems to have the best performance in terms of range of execution time, as well as average execution time. This would be my goto choice of concurrent data structure if a LIFO implementation was required.

The algorithms that this data structure implements attempts to reduce the amount of contention on accesses. Instead of having every thread sending synchronization messages to the master, each individual thread can do its own separate combining. Once this is done, the flat\_combining() function that I call combines all of these combined thread copies that decreases the total amount of locking and synchronization variables needed.

# Description of Organization

My zip folder is quite a jumble (just a forewarning). All of my dependent classes and the main header file is included in “stackqueue.h”. This contains all of the classes, structures, and defines that my cpp files require.

You will find 5 different cpp files that are separated by data structure type. fcstackqueue.cpp, basketqueue.cpp, msqueue.cpp, sglstackqueue.cpp, and tstack.cpp are the main files that my data structures are implemented in. These files contain the flat combining stack/queue, basket queue, MS queue, single global lock stack/queue, and treiber stack respectively. Each of these files contain definitions for add and remove functions that are declared inside of modular classes inside of “stackqueue.h”.

Each of these main cpp files contain a main method that have test cases that I have written to concurrently add and remove integers from these respective data structures.

# Description of Files

I have implemented many library files that I pulled from a publication written by Hendler, Incze, Shavit, and Tzafrir. Folders “framework” and “data\_structures” contain dependency files that I include in most of my own files. I call functions from these files that help me add, remove, and print from these data structures. I will not describe every single one of these files because there are a lot.

Basketqueue.cpp – implementation of my basket queue

Fcstackqueue.cpp – implementation of flat combining stack and queue

Msqueue.cpp – implementation of Michael & Scott queue

Sglstackqueue.cpp – implement single global lock stack and queue

Sourcefile.txt – file with 20 numbers in it used for testing

Sourcegeneration.cpp – script I wrote to generate any number of numbers from 1 – n (testing)

Stackqueue.h – main header file that contains all necessary classes and other dependencies

Tstack.cpp – implementation of treiber stack functions

Makefile – my makefile that generates 5 different executables

# Compilation Instructions

All files are run using command “make”

# Execution Instructions

All files are ran using:

<./data\_structure\_executable> <sourcefile.txt> -t <threadcount>

One can print the execution instructions with the -h flag

Name is printed with <executable> --name

# Extant Bugs

There is a bug with my FC stack where after adding 20 elements, only 4 are inside of the queue. And these 4 numbers can be any one of the 20 that I popped or pushed.