David Wade

Concurrent Programming

Final Project

Concurrent Containers

For this project, I built and optimized my own concurrent data structures. There were a total of four data structures, a Treiber Stack, Michael and Scott Queue, Single Global Lock Stack, and a Single Global Lock Queue. None of the data structures leak memory which was accomplished through Epoch based Reclamation. The stack data structures were optimized with an elimination algorithm and the lock algorithms were optimized with flat combining.

The Treiebr stack and the M&S queue are both non-blocking algorithms while the Single global lock stack and single global lock queue are blocking. The non-blocking algorithms use atomic operations and more specifically compare and swap operations in order to linearize the algorithms across threads. The blocking algorithms simply emulate the standard stack and queue algorithms, however the algorithms are protected by a lock in order to avoid data races.

When testing the performance of each data structure compared to each other, I tested the structures with 100,000 operations with varying thread counts (50,000 pushes/queues and 50,000 pops/dequeues). I ran each data structure 10 times with 5, 25, and 50 threads.

<u>100,000 Operations w/ 5 threads</u>

	Treiber No elim (s)	Treiber w/ Elim (s)	M&S Queue (s)	SGL Stack Elim (s)	SGL Stack Flat Comb (s)	SGL Stack No Optim (s)	SGL Queue Flat Comb (s)	SGL Queue No Optim (s)
	1.332461233	1.492175971	1.555661466	0.080486385	0.081544654	0.08718275	0.075141286	0.072819348
	1.345985345	1.309788372	1.392108302	0.100223405	0.099276557	0.081171512	0.064181731	0.06413581
	1.468395312	1.312779111	1.413374087	0.079099261	0.08797523	0.08392841	0.067222213	0.067052768
	1.730561442	1.511314692	1.533491967	0.078143924	0.081285517	0.111008305	0.073321257	0.067387731
	1.323112342	1.490915173	1.397160645	0.101336251	0.083785336	0.084621858	0.068564067	0.066528566
	1.659887488	1.318181872	1.44139656	0.082864398	0.086992015	0.079764911	0.067449371	0.008509315
	1.475820697	1.336571864	1.526933825	0.081922121	0.082705094	0.087534976	0.066534652	0.071407626
	1.48584103	1.444714429	1.395606808	0.081385863	0.086192395	0.09037552	0.06597292	0.066150909
	1.310489632	1.460546391	1.598251316	0.08641067	0.083917918	0.076855309	0.060776163	0.066386128
	1.340788117	1.334688089	1.523031973	0.082939892	0.077684076	0.105172051	0.061623175	0.070563928
AVG.	**1.447334264**	**1.401167596**	**1.477701695**	**0.085481217**	**0.085135879**	**0.08876156**	**0.067078684**	**0.062094213**

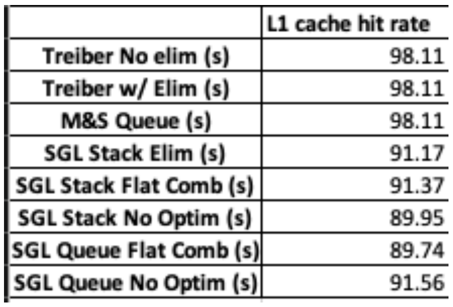
<u>100,000 Operations w/ 25 threads</u>

	Treiber No elim (s)	Treiber w/ Elim (s)	M&S Queue (s)	SGL Stack Elim (s)	SGL Stack Flat Comb (s)	SGL Stack No Optim (s)	SGL Queue Flat Comb (s)	SGL Queue No Optim (s)
	1.892912483	1.898891642	2.49570795	3.676105545	2.612479605	2.896786783	1.32523906	2.575912993
	2.205768666	1.696236056	2.186720995	3.695754681	3.194407327	2.810120221	2.323615041	1.795244729
	1.610328545	1.529490387	2.099124306	2.904753958	3.200368481	3.376122184	3.887948806	1.720952132
	1.82685018	1.885938555	1.802273015	2.516919016	3.597189694	2.705307272	1.574338826	2.976160444
	2.101002834	2.00958086	2.105079637	3.12335231	2.200582313	3.010028135	1.70893325	2.37658982
	2.287891178	2.018668186	2.610301609	2.800053852	2.00444429	3.684454452	1.394580804	2.280155736
	1.895399052	1.410726006	1.497459867	2.420020818	1.803378512	2.913763568	1.188021722	2.010859784
	1.922109184	1.800976361	1.892859871	2.902683592	4.097263343	3.320488793	1.079818316	2.712443352
	1.536411317	1.601752994	1.829266111	2.810510998	4.110519563	3.109680633	1.414033236	3.087721397
	2.106117665	1.002423893	2.287832393	3.610631818	3.903020265	4.782608547	1.912591768	1.992124739
AVG.	**1.93847911**	**1.685468494**	**2.080662575**	**3.046078659**	**3.072365339**	**3.260936059**	**1.780912083**	**2.352816513**

<u>100,000 Operations w/ 50 threads</u>

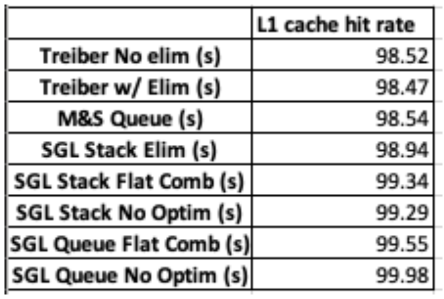
	Treiber No elim (s)	Treiber w/ Elim (s)	M&S Queue (s)	SGL Stack Elim (s)	SGL Stack Flat Comb (s)	SGL Stack No Optim (s)	SGL Queue Flat Comb (s)	SGL Queue No Optim (s)
	1.299641095	0.985096337	1.312811547	9.776898071	6.516879267	7.40464993	4.90867846	6.103729933
	1.309713187	1.189472749	1.008843131	10.11928465	6.672384499	10.08352013	5.293194427	7.300039882
	1.209636494	1.212794868	1.300026043	6.016217706	8.416744535	6.800809079	4.498516914	5.2003501
	1.201120605	1.281876808	1.209745535	9.992474785	7.572411906	6.972153933	5.284731998	9.007546672
	1.183683717	0.78449783	1.31044389	5.920161348	7.096731766	10.09679314	4.200476681	6.904119438
	0.913327782	1.093485859	1.301768695	8.998265672	8.011062504	6.420739502	6.399961293	5.091575992
	1.093446476	1.185791428	1.404298957	9.092662923	10.41271732	8.499891198	5.71712988	7.794026239
	1.121829583	1.176715821	1.204860023	11.59350868	6.777598789	9.798764887	5.924731097	6.495365331
	1.301353582	1.20927702	1.497215461	5.876823026	7.492737413	10.10874659	6.214492279	8.078755479
	1.189561938	0.909783414	1.18115183	9.705948032	7.793291999	12.20277733	5.696817031	10.50454929
AVG.	**1.182331446**	**1.102879213**	**1.273116511**	**8.709224489**	**7.676255999**	**8.838884572**	**5.413873006**	**7.248005835**

5 threads



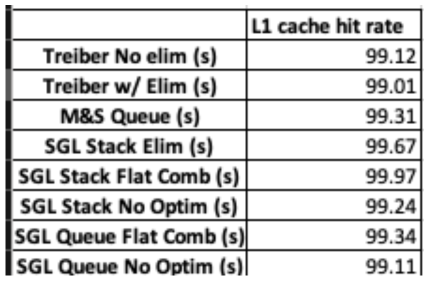
	L1 cache hit rate
Treiber No elim (s)	98.11
Treiber w/ Elim (s)	98.11
M&S Queue (s)	98.11
SGL Stack Elim (s)	91.17
SGL Stack Flat Comb (s)	91.37
SGL Stack No Optim (s)	89.95
SGL Queue Flat Comb (s)	89.74
SGL Queue No Optim (s)	91.56

25 threads



	L1 cache hit rate
Treiber No elim (s)	98.52
Treiber w/ Elim (s)	98.47
M&S Queue (s)	98.54
SGL Stack Elim (s)	98.94
SGL Stack Flat Comb (s)	99.34
SGL Stack No Optim (s)	99.29
SGL Queue Flat Comb (s)	99.55
SGL Queue No Optim (s)	99.98

50 thread



	L1 cache hit rate
Treiber No elim (s)	99.12
Treiber w/ Elim (s)	99.01
M&S Queue (s)	99.31
SGL Stack Elim (s)	99.67
SGL Stack Flat Comb (s)	99.97
SGL Stack No Optim (s)	99.24
SGL Queue Flat Comb (s)	99.34
SGL Queue No Optim (s)	99.11

After collecting the data, it appears that the Treiber stack with the elimination optimization was the best performing data structure overall. The only time the Treiber stack with elimination was out performed was when there was a low thread count, where the single global lock data structures shined. I believe this to be the case due to my garbage collection implementation. With the non-locking algorithms, garbage collection was done via epoch based reclamation, but the locking algorithms collected garbage more efficiently as the deletes were per5formed once the lock was held. However, the garbage collection efficiency became less prevalent as the number of threads increased. Once the threads increased, the locking algorithms' performance drastically declined as more contention on the lock was introduced to the system. In addition, it is clear from the data that each optimization improved the performance of its corresponding non optimized counterpart. Overall, it is clear that the non blocking algorithms performed better than the locking algorithms as the number of threads increased. In addition, my flat combining optimization performed better than my elimination optimization. This makes sense as flat combining had a better cache hit rate.

My code is organized into 8 c++ files, each with a corresponding header file. Each of the files, with the exception of the driver file, is organized in an object oriented fashion. The driver file essentially just grabs command line arguments in order to perform different tests. The arguments are the number of desired threads, the number of iterations each thread will perform, and the desired data structure that I want to be tested. Then there is the data structure tester file where the parallelism occurs. This files handles whatever inputs I grabbed from the command line and tests the data structures accordingly. Each data structure has their own parallel fork function. In addition, this file contains the epoch based reclamation garbage collection algorithm. In addition, I have my barriers and locks file that I created in Lab 1 in order to use the API that I built to implement parallelism. Lastly, each data structure then has their own files in which the corresponding fork methods call to from the tester file. The optimizations are implemented in each of the corresponding data structure files.

In addition to the coding files listed above, there is also a makefile for easy compiling, and an excel spreadsheet where I created the tables shown above that I'm submitting with this project.

In order to compile the project, one simply needs to be in the correct5 directory and type a make command in the command line. Each file will create an object file and an executable called "concurrent_containers" will be created.

In order to execute the code, the syntax is ./concurrent_containers -t [NUM_THREADS] -I [NUM_ITERATIONS] –structure=<treiber,msQ,sglS,sglQ>. Furthermore, in order to turn on and off certain optimizations, one must go to the corresponding header file and comment/uncomment the desired #define that will enable the optimization. The code is designed to only implement a single optimization at a time.

This project does not have any extant bugs, and each data structure/optimization does not leak any memory.