CS342 Operating Systems Spring 2019

Project 2 Report

Student 1:

Name: Ünsal Öztürk

ID: 21601565

Section: 1

Student 2:

Name: Faruk Oruç

ID: 21601844

Section: 3

Experiment Setup, Methodology, and Statistics

The experiment is located in test.c. It is very similar to the integer-count program. It takes five parameters from the command line. The following is an example invocation of the test program:

$$test\ 100\ 10000\ 50\ 100\ results.txt$$

The first parameter is the number of threads operating on the hash table. The second parameter is the number of the set of operations performed in total. The third parameter is the number of regions in the hash table, the fourth parameter is the number of buckets in the hash table, and the last parameter is the name of the file to which the relative statistics and data about the execution time of the program.

The test program creates T number of threads, and each thread inserts K randomly generated keys between 1 and 10000 to the hash table, with the value 1. If a randomly generated key is present in the table, its value is incremented by one. In one iteration, $hash\_get(3)$ is called once, $hash\_insert(3)$ is called once with probability p, and $hash\_update(3)$ is called with probability $1-p$, where p is the probability that a key is present in the table. For each thread, the time of execution of the operations on the hash table is measured and recorded. The program then calculates the minimum, maximum, average, variance, and the standard deviation of execution times, and then dumps them in two files. In this invocation, the file name is $results.txt$. It also creates another file called $matlab-results.txt$ which has the thread execution times in MATLAB array notation.

We ran the program with varying T, W, K, N parameters and recorded our findings in the tables below. These parameters correspond to:

$T - number\ of\ threads\ operating\ on\ the\ hash\ table$

$W - number\ of\ set\ of\ operations(insert, get, delete, update)\ performed\ in\ total$

$K - number\ of\ regions\ in\ the\ hash\ table$

$N - number\ of\ buckets\ in\ the\ hash\ table$

The time taken to execute the test program is measured in seconds. The rest of the columns denote:

$Min.thread: The\ minimum\ time\ it\ took\ for\ a\ thread\ to\ finish\ execution: \max_i T_i$

$Max.thread: The\ maximum\ time\ it\ took\ for\ a\ thread\ to\ finish\ execution\ \min_i T_i$

$$Average: The\ average\ time\ it\ took\ for\ a\ thread\ to\ finish\ execution\ \mu_T = \frac{1}{N}\sum_{i=0}^{N-1} T_i$$

$$Variance: The\ variance\ of\ thread\ execution\ time\ (sample\ variance): S^2 = \frac{1}{N-1}\sum_{i=0}^{N-1}(T_i - \mu_T)^2$$

$$Standard\ Deviation: The\ standard\ devation\ of\ thread\ execution\ time: \sqrt{S^2}$$

Experiment Results

T	W	K	N	Min(s)	Max(s)	Avg(s)	Variance(s)	StdDev(s)	Total(s)
1	100000000	100	1000	14.475459	14.475459	14.475459	0	0	14.475541
10	100000000	100	1000	15.013644	15.135221	15.091562	0.001623	0.040281	15.136962
25	100000000	100	1000	13.764948	14.276855	14.126725	0.023323	0.152718	14.282874
50	100000000	100	1000	10.667157	13.594504	12.895390	0.374170	0.611696	13.601243
100	100000000	100	1000	11.237664	13.073812	12.736677	0.114490	0.338365	13.092533
250	100000000	100	1000	10.667157	13.594504	12.895390	0.374170	0.611696	13.601243
500	100000000	100	1000	8.452625	14.106449	12.808840	1.381859	1.175526	14.125302
1000	100000000	100	1000	4.773749	14.721947	12.703047	3.818240	1.954040	14.748121

Table 1 Variable T (other variables kept constant)

T	W	K	N	Min(s)	Max(s)	Avg(s)	Variance(s)	StdDev(s)	Total(s)
200	**100000000**	100	1000	11.290289	14.079068	13.262844	0.470154	0.685679	14.086886
200	**200000000**	100	1000	26.881542	31.269815	30.100840	1.103717	1.050580	31.283463
200	**300000000**	100	1000	36.596394	41.081924	39.962902	0.980184	0.990047	41.094440
200	**400000000**	100	1000	49.587456	56.194920	54.580238	2.292671	1.514161	56.204311
200	**500000000**	100	1000	68.956825	78.033737	76.048897	3.827308	1.956359	78.043976
200	**600000000**	100	1000	84.122498	91.958900	90.164772	2.817961	1.678683	91.968239
200	**700000000**	100	1000	93.040947	100.725121	98.900017	3.008949	1.734634	100.729767
200	**800000000**	100	1000	99.725266	108.062561	106.072159	3.600757	1.897567	108.070671

Table 2 Variable W (other variables kept constant)

T	W	K	N	Min(s)	Max(s)	Avg(s)	Variance(s)	StdDev(s)	Total(s)
200	100000000	**1**	1000	8.789445	12.437372	11.420913	0.694256	0.833223	12.453827
200	100000000	**10**	1000	10.569728	13.432539	12.848156	0.280022	0.529171	13.444483
200	100000000	**25**	1000	11.184059	13.465724	12.985914	0.204557	0.452279	13.470882
200	100000000	**50**	1000	11.539142	14.074930	13.549130	0.235105	0.484879	14.079588
200	100000000	**100**	1000	10.859344	13.422381	12.813807	0.261897	0.511759	13.431146
200	100000000	**250**	1000	10.394932	13.580857	12.896502	0.419451	0.647652	13.591878
200	100000000	**500**	1000	10.889360	13.531170	12.882162	0.348900	0.590678	13.539494
200	100000000	**1000**	1000	10.872176	13.413576	12.775683	0.337268	0.580748	13.424737

Table 3 Variable K (other variables kept constant)

T	W	K	N	Min(s)	Max(s)	Avg(s)	Variance(s)	StdDev(s)	Total(s)
200	100000000	100	**1000**	12.162053	15.284687	14.497527	0.457098	0.676091	15.293446
200	100000000	100	**2000**	8.983260	12.083741	11.407519	0.396719	0.629859	12.088714
200	100000000	100	**3000**	7.915183	10.832044	10.252878	0.266262	0.516006	10.836843
200	100000000	100	**4000**	7.745450	10.013089	9.436481	0.279070	0.528271	10.025676
200	100000000	100	**5000**	6.342074	9.579698	8.909039	0.384154	0.619804	9.588587
200	100000000	100	**6000**	7.410914	9.522144	8.866539	0.256684	0.506641	9.526928
200	100000000	100	**7000**	7.369834	9.064311	8.627587	0.242522	0.492465	9.071327
200	100000000	100	**8000**	6.586128	8.813473	8.320871	0.216281	0.465060	8.819048

Table 4 Variable N (other variables kept constant)

Theoretical Discussion, Experimental Results and Observations

Theoretically, assuming that the rest of the parameters are kept constant, increasing the number of locks (K) should decrease the number of execution due to lower contention among threads when a thread wants to access a portion of the table. With increasing N, the execution time should decrease due to the depth of the buckets being shallower on average, which leads to less memory accesses for duplicate element checks. With increasing T, the execution time should decrease, since the workload is equally divided between threads, and the work is done concurrently. With increasing W, the execution time should scale linearly given that everything else stays the same.

The results of this experiment, however, do not entirely conform to this theoretical discussion. This is caused by the fact that we ran the experiment on a single core machine. This means that even though the threads are being executed concurrently, only one thread at a time may be executed, therefore the process execution time of this test program is the same as the execution time of a process which does the same operation in a sequential manner. This is reflected in Table 1: The number of threads does not have a big impact on the total execution time. The average time of execution is around 12 seconds for most setups. What the number of threads changes; however, is the variance of the execution times: the more threads there are, the higher the variance for the execution time of the threads. The increase in variance is due to the scheduling of the threads: more threads for the same job means more context switches, and more context switches means a higher probability of a context switch during the execution of a critical section, which means higher contention. The overhead caused by this contention, however, is not a massive one, since the execution times are nearly identical. The variance of the execution times is also caused by the indeterministic nature of the execution of the driver program and the threads: the driver may launch a thread and before creating another thread in the loop, there might be a context switch, causing some threads to be launched way later than intended (initialization of all threads is not atomic). If the number of threads executing at a given time is few, then there will be less contention. This type of delayed executions therefore reduces the amount of concurrency and the more this happens, the more the variance will be. Increasing the number of maximum threads creates such an effect.

For different values of W, the results conform to the theoretical discussion. The execution times scale nearly linearly with linearly increasing values of W. In terms of the variance of the execution times, the variance increases as W increases. An argument similar to the argument for the effect of the choice of T on execution time applies here as well: the more work a thread has to complete, the higher the probability that all the threads are executing at the same time: the main driver may have enough time to launch all the threads eventually if W is large enough due to the sheer time it takes for a thread to finish a task. With higher W, it is easier to context switch from a worker thread back to the main program before the thread finishes execution, which allows more threads to be scheduled for running. This in turn results in more threads running concurrently, some nearing the end of their execution and some freshly launched. And the more threads there are running concurrently, the more the context switches. This results in slower execution times for threads launched when there are lots of threads executing, and a faster execution time when there are fewer number of threads running. This results in an increase in variance.

The number of locks does not seem to affect the execution times severely in our experiment. This means that the number of context switches while a thread is executing a critical section is low, therefore decreasing the contention, which means that the execution time is largely unaffected. This might partly be due to the low probability of context switching while the critical section runs, and might also be due to

the number of processes running in the background while the experiment was performed and the scheduling algorithm provided by the operating system. Another reason why such an invariance is present in the data is the random number generator. We used $rand()$ for the experiment, which is known to sometimes produce non-uniformly distributed values. By the central limit theorem, given that there are different seeds for different threads, the probability of contention for T threads follows a Gaussian distribution for all the threads, which means that in such a case (scheduled thread execution on a single core), contention will be normalized for different lock values. For the variance of the execution times, the variance should decrease with increasing number of locks. This is because of contention: if a thread busy loops before entering the critical section, and if there is a context switch back to the driver program, which launches more threads, which also busy loop for a while before actually starting execution. This causes the number of concurrently executing threads at a given time to be more stable, which decreases execution time variance.

Another reason why the K does not change execution times that much is the load factor of the hash table. The keys used in the experiment when $hash\_insert(3)$ is called do not, empirically, do not result in very long linked-lists for a given bucket. The average length of a bucket was measured to be 10 nodes, which means that the critical sections do not take too long to execute. This results in low contention overall.

The effect of parameter N on execution time should be quite obvious: more buckets mean less memory accesses (due to low load factors), which means less time to complete a hash table operation, which means faster execution overall. Table 4 clearly conforms to this result. The average execution time is lower for higher number of buckets. Unsurprisingly, the variance of the execution time for T threads also decreases as the number of buckets increase. This is because contention is decreased when the linked-lists associated with the buckets are shorter, which means that the critical section is executed faster, which decreases the time window in which a contention causing context-switch may take place, which results in more stable execution times.

Possible improvements and method drawbacks: We did not have a machine with a multicore CPU which has a Linux installation, therefore concurrency was not achieved in the sense that we originally intended it to be. Another possible improvement would be to scale the number of unique keys more than linearly so that the buckets are lengthier so that the critical sections are longer. In this way, one would observe real contention, which would allow one to observe the contention-decreasing effect of having more locks. In our experiment, we were not quite able to observe the effect in its amplified form.

test.c contents

```
#include <pthread.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include "pthread.h"
#include "hash.h"


HashTable *ht1; // space allocated inside library

//Note: ./test 1000 100000 10 1000 results.txt -> good params

// Fast inverse square root, taken directly from Quake III Arena, including the original comments.
// One letter was changed from the original comments to avoid profanity.
// This is here because makefile shouldn't be changed, and to compile test, one had to include the
// math library in the makefile for the sqrt(1) funciton. Instead, we used computed 1/sqrt(x) using
// the following method.
float Q_rsqrt( float number ) {
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;                       // evil floating point bit level hacking
        i  = 0x5f3759df - ( i >> 1 );               // what the duck?
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
        y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

        return y;
}

// Test: Generate a random key between 0 and 10000
// then insert the key with value 1 into the table.
// if the key exists, update the value by incrementing it
// by 1.
void* thread_driver(void* param) {
        struct timeval tv1, tv2;
        int iter_no = (int)param;
        int i;
        gettimeofday(&tv1, NULL);
        for(i = 0; i < iter_no; i++) {
                int key = rand() % 10000;
                void* vp;
                if(hash_get(ht1, key, &vp) == 0) {
                        hash_update(ht1, key, vp + 1);
                } else {
                        hash_insert(ht1, key, (void*)1);
                }
        }
        gettimeofday(&tv2, NULL);
        return (void*)(((tv2.tv_usec - tv1.tv_usec) + 1000000 * (tv2.tv_sec - tv1.tv_sec)));
}

int main(int argc, char **argv) {
        // Expected arguments in order:
        // T W K N
        // T:Threads
        // W: number of operations
        // K: Number of locks
        // N: Size of table
        // last arg: output file name
```

```
        int T = atoi(argv[1]);
        int W = atoi(argv[2]) / T;
        int K = atoi(argv[3]);
        int N = atoi(argv[4]);
        pthread_t threads[T];
        int times[T];

        ht1 = hash_init(N,K);
        int i;
        // Measure time of execution
        struct timeval tv1, tv2;
        gettimeofday(&tv1, NULL);
        for(i = 0; i < T; i++) {
                pthread_create(&threads[i], NULL, thread_driver, (void*)W);
        }
        for(i = 0; i < T; i++) {
                void* time;
                pthread_join(threads[i], &time);
                times[i] = (int)time;
        }
        gettimeofday(&tv2, NULL);
        float exec_time = ((tv2.tv_usec - tv1.tv_usec) + 1000000 * (tv2.tv_sec - tv1.tv_sec)) / 1000000.0;

        // Compute statistics: Compute min, max, mean, variance, and standard deviation.
        // Then, write results to a file.
        float min = times[0] / 1000000.0;
        float max = times[0] / 1000000.0;
        float avg = 0;
        FILE* fp = fopen(argv[5], "w");
        char matlab_filename[strlen(argv[5])+7];
        strcpy(matlab_filename, "matlab-");
        strcat(matlab_filename, argv[5]);
        FILE* fp_matlab = fopen(matlab_filename, "w");
        fprintf(fp_matlab, "[ ");
        for(i = 0; i < T; i++) {
                float inc = times[i] / 1000000.0;
                if(inc < min) {
                        min = inc;
                }
                if(inc > max) {
                        max = inc;
                }
                avg += inc;
                fprintf(fp, "Execution time for thread %d: %f sec\n", i, inc);
                fprintf(fp_matlab, "%f ", inc);
        }
        fprintf(fp_matlab, "]");
        avg /= T;
        float var = 0;
        for(i = 0; i < T; i++) {
                float inc = times[i] / 1000000.0;
                var += (inc - avg) * (inc - avg);
        }
        var /= T - 1;

        // Log results
        fprintf(fp, "Minimum execution time for a thread: %f sec\n", min);
        fprintf(fp, "Maximum execution time for a thread: %f sec\n", max);
        fprintf(fp, "Sample mean of execution time for a thread: %f sec\n", avg);
        fprintf(fp, "Sample variance of execution time for a thread: %f sec\n", var);
        fprintf(fp, "Sample std. dev. of execution time for a thread: %f sec\n", 1.0 / Q_rsqrt(var));
        fprintf(fp, "Total execution time: %f sec\n", exec_time);
        fclose(fp);
        fclose(fp_matlab);
        hash_destroy(ht1);
}
```