Assignment 2

CS 531
Due January 27, 2019

Jason Graalum

January 26, 20190

Part A: Running and timing the example pthreads "sum" code from the lecture I am reporting data on four implementations of the "sum" function. These four are the result of a series of solutions I developed. I choose these as they show several different effects:

- Thread count
- Cache effects
- Global vs. Thread local variables
- Passing data into and out of a thread

The first version of the pthread code is a naive implementation which does not take into account the overhead of synchronizing the cache between threads. From the data, we see that this solution actually runs slower that the single threaded version. The reason for this is the additional overhead added to manage the threads. But, because each thread must lock the global sum variable while it iterates, the summation is effectively serial.

The second version tries to take into account the fact that the variables passed to each thread may be local - within a single cache line. If this is the case, then each thread must continually synchronize with each other thread. But, due to the locking of the global sum variable, this solution is also slower than the single threaded case. One modification I made was to move the lock outside of the full loop in the summation routine as shown below:

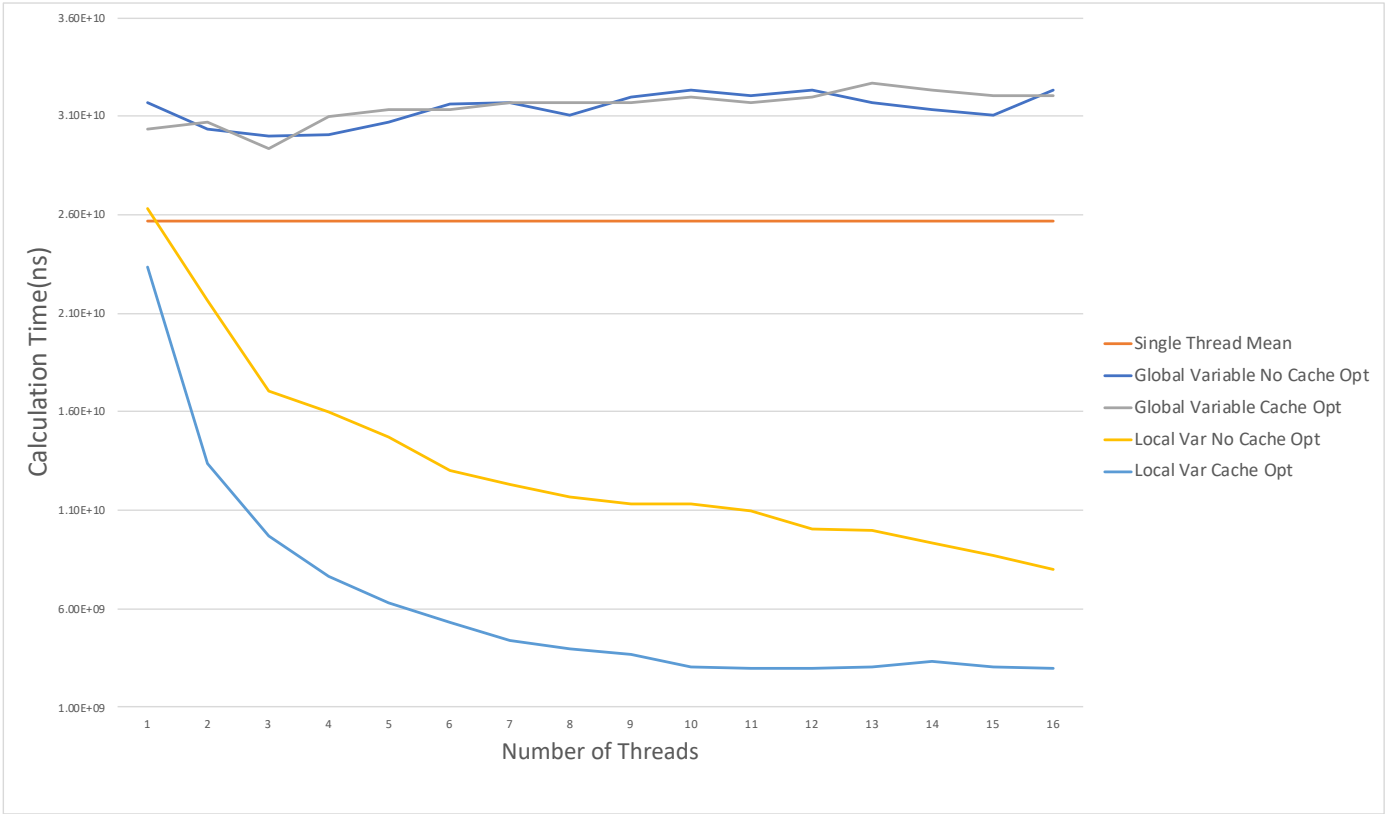
```
void *thread_loop_fine_mutex(void *vargp)
{
    int myid = *((int *)vargp);
    pthread_mutex_lock(&(nelems_per_thread.lock));
    size_t start = myid * nelems_per_thread.val;
    size_t end = start + nelems_per_thread.val;
    pthread_mutex_unlock(&(nelems_per_thread.lock));
    size_t i;
    data_t local_sum = 0;
    // Critical Section
    //printf("Locked(%d)\n",myid);

    for (i = start; i < end; i++) {
        local_sum += i;
    }
    pthread_mutex_lock(&(global_sum.lock));
    global_sum.val += local_sum;
    //printf("Unlocked(%d)\n",myid);
    pthread_mutex_unlock(&(global_sum.lock));

    return NULL;
}
```

In the last two version, to get around the problems with these global variables, I instead passed in a data structure to each thread call. This data structure, which is shown below, provides a self-contained and independent set of data needed by each thread. This partially removes the cache locality issues.

```
typedef struct thread_data_t {
    data_t start_num;
    data_t end_num;
    data_t sum;
} thread_data_t;
```

To complete accounting for cache locality, each structure must be stored a cache line apart. This is included in the final version of the sum code.

A summary of the data is shown in the graph below. Also, the code and raw data will be included in the assignment submission.


3.60E+10
3.10E+10
2.60E+10
2.10E+10
1.60E+10
1.10E+10
6.00E+09
1.00E+09
Calculation Time(ns)
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Number of Threads
Single Thread Mean
Global Variable No Cache Opt
Global Variable Cache Opt
Local Var No Cache Opt
Local Var Cache Opt

Part B: Amdahl's Law and Speedup The following is the gprof data from the single threaded version of the summation program.

```
     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.04% of 27.41 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     99.8   27.34    0.00                 thread_loop [1]
-----------------------------------------------
                0.07    0.00       1/1           main [3]
[2]      0.2    0.07    0.00       1         multi_thread_global_var [2]
-----------------------------------------------
                                                 <spontaneous>
[3]      0.2    0.00    0.07                 main [3]
                0.07    0.00       1/1           multi_thread_global_var [2]
-----------------------------------------------
```

From this output, we can see that 99.8

Using Amdahl's law, $time_{new} = time_{org} * ((1 - f) + f/s)$ where f =fraction of time enhanced and s = speed up factor, $f = 99.8\%$ and s is based on the number of threads used.

s	New Time/Original Time
1	1
2	0.501
3	0.335
4	0.252
5	0.202
6	0.168
7	0.145
8	0.127
9	0.113
10	0.1018
11	0.093
12	0.085
13	0.078
14	0.073
15	0.069
16	0.064

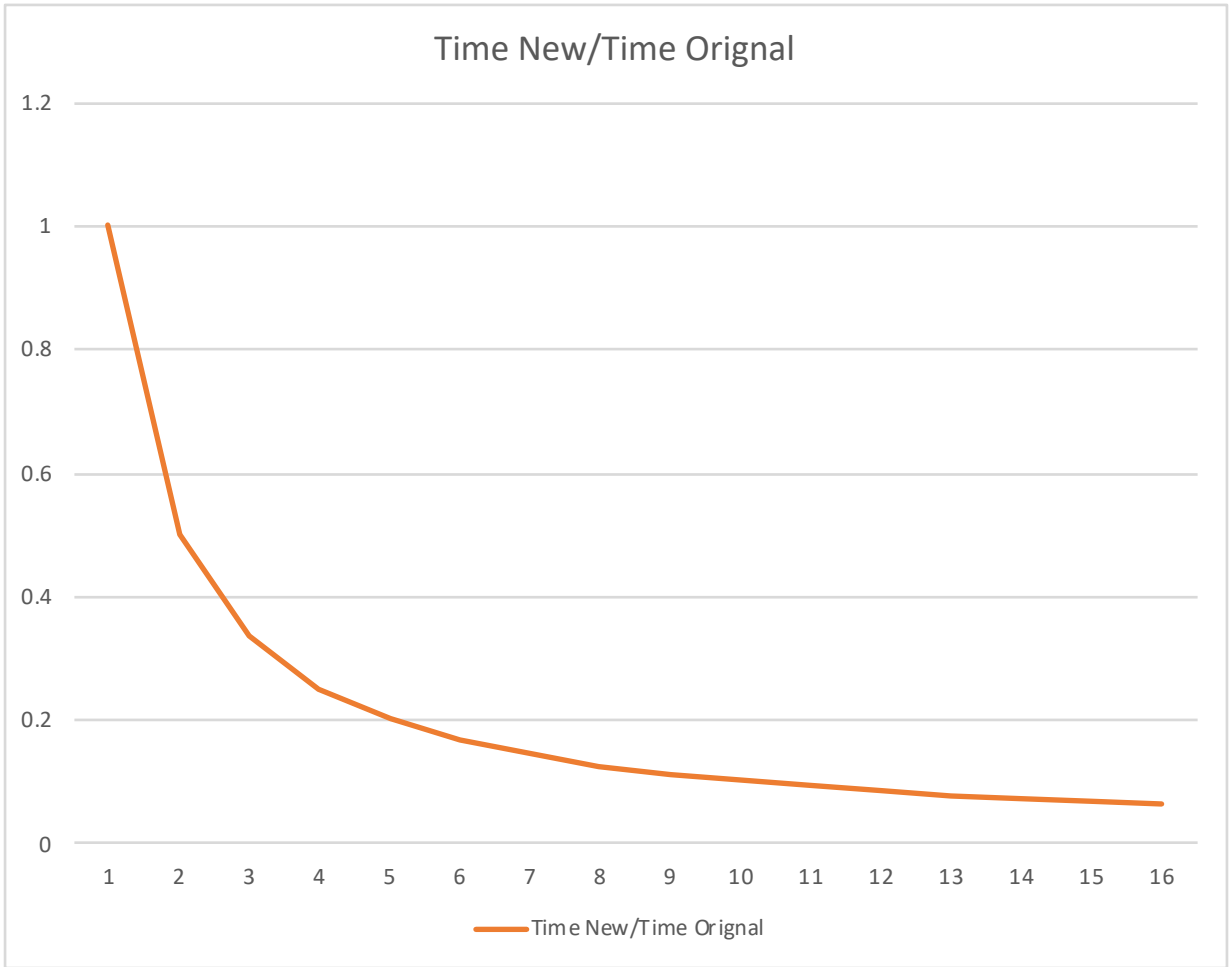
This is the speed up for the version using the local variables and adjusted cache locality. The versions using the global variable are expect to have a negative speed up since there is no portion of the code that can truly be paralellized.

The results for #3 and #4 using the 16 thread cases for each version of the code.

Below are the raw mean data for each version with data point for 1 to 16 threads.

Next is the mean difference data, the standard deviation, and the 95

My understand of ANOVA is not deep enough to generate a useful conclusion. From what I understand, I need to find the p-value, but I need to study up on statistics a bit more.


Time New/Time Orignal
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Time New/Time Orignal

f	0.998

s	Speed Up	New Time	Global 1	Global 2	Local 1	Local 2
1	1	25652225743	31682481841	30338127145	26318384686	23315495457
2	0.501	12851765097	30326496608	30675151952	21664983078	13354968599
3	0.334666667	8584944882	29985214638	29339626471	17022191139	9670669622
4	0.2515	6451534774	30028230954	30980577481	15980746573	7659726699
5	0.2016	5171488710	30669205658	31328471453	14677725489	6325152492
6	0.168333333	4318124667	31641253497	31340917597	13003166013	5325348438
7	0.144571429	3708578922	31663892041	31658393149	12337804403	4354289163
8	0.12675	3251419613	31026140351	31677973846	11665402556	3981325453
9	0.112888889	2895851262	31998375846	31652305913	11332024869	3662263102
10	0.1018	2611396581	32306607739	31998361124	11332306742	3020024278
11	0.092727273	2378660932	32017219121	31692945271	10978066026	3003677136
12	0.085166667	2184714559	32316649485	31993599904	10030836190	2995065896
13	0.078769231	2020606089	31672593699	32664490496	9983638657	3006778408
14	0.073285714	1879941687	31333530098	32320702527	9322712767	3309182469
15	0.068533333	1758032538	31019384833	32004962985	8664847696	3011420084
16	0.064375	1651362032	32317725257	32008686506	8018721258	3001348017

s	Diff Means Global 1	Diff Means Global 2	Diff Means Local 1	Diff Means Local 2
1	-6030256099	-4685901403	-666158943.7	2336730286
2	-17474731511	-17823386855	-8813217981	-503203502.3
3	-21400269756	-20754681589	-8437246258	-1085724740
4	-23576696179	-24529042707	-9529211799	-1208191924
5	-25497716949	-26156982744	-9506236780	-1153663782
6	-27323128830	-27022792930	-8685041346	-1007223771
7	-27955313120	-27949814227	-8629225481	-645710241.2
8	-27774720738	-28426554233	-8413982944	-729905840.4
9	-29102524584	-28756454652	-8436173607	-766411840.9
10	-29695211158	-29386964543	-8720910161	-408627697.3
11	-29638558189	-29314284338	-8599405093	-625016203.1
12	-30131934925	-29808885345	-7846121631	-810351337.1
13	-29651987610	-30643884407	-7963032567	-986172318.7
14	-29453588412	-30440760840	-7442771080	-1429240783
15	-29261352295	-30246930447	-6906815158	-1253387547
16	-30666363225	-30357324473	-6367359225	-1349985985
StdDev	6421982874	6754998864	2081192406	872150943.5
Conf Interval	100675463.5	105896053.4	32626217.51	13672443.89

Part C: Calculating the Mean