Effect of Data Races on Accuracy and Performance of Aggregative Loops

Aysylu Biktimirova

May, 15th 2012

1 Introduction

Currently, research in concurrent and parallel programming is on the rise, in order to develop techniques and algorithms to allow programmers to take advantage of the many cores available in modern machines. However, many programs designed for multi-core systems do not reach their potential performance due to the limits imposed by synchronization. Synchronization ensures the program gives the correct answer, but it prevents so-called critical sections of the program from running in parallel. If we were to remove synchronization, the program may return inaccurate results but may also scale better. Although many applications require absolute correctness of the results, some applications, like Monte Carlo simulations and video encoder/decoders, can have inaccurate output that's useful as long as it falls within known error bounds.

Most programs spend most processing time in loops, so optimizing and parallelizing loops is a goal for many compiler researchers. I will focus on the type of programs that contain a for-loop, which performs aggregation of values, e.g., summing the elements of an array. Aggregative loops are present in a wide variety of applications, such as matrix multiplication, numerical simulations and others. I will parallelize the for-loop without synchronization in order to explore the extent to which the results are inaccurate. I will propose a model for estimating the expected result of the program, taking into account the introduced inaccuracy, and for predicting the total execution time of the program. The model will take into account the number of cores the program will run on, the number of instructions in one iteration of the loop, and the number of iterations of the loop in order to calculate the expected result of the program and its execution time. Furthermore, I will compare the theoretical model with experimental results from running the program on an eight core machine.

2 Methodology

I analyzed the following type of programs. A global variable *sum* is initialized to some value *init*. With each execution of the loop, the global *sum* is incremented by a local variable x, the value of which depends on the induction variable i.

```
sum = init;
for (int i = 0; i < N; i++) {
 x = f(i);
 sum += x;
}
```

The actual value of x could be set in a variety of ways, such as

```
z = foo(i);
y = z * 2;
x = y + 4;
sum += x;
```

but we abstract away the notion of how exactly the value of the variable x is set by representing all of those instructions with generalized $f(i)$.

We will call the load and the store of the *sum* variable global instructions. All the other instructions in the loop body are local instructions.

We compare the results of the theoretical model and the experimental data. We focus on two aspects of the performance of the program:

1. expected value of the sum, and

2. expected execution time,

both as a function of the number of threads and as a function of the ratio of local instructions to the total number of instructions.

2.1 Potential Data Race and Its Effects

If we parallelize the for-loop without synchronization, there is a non-zero probability that a data race will occur on the global variable *sum*. For example, let us assume that we have two threads, T1 and T2, that execute the loop computation. The thread T1 reads the value of the *sum*. The other thread T2 also reads the same value of the variable *sum*, increments it by x and stores the result to the memory. Note that the thread T1 operates on the old version of the *sum*, so when it sotres the result of the incrementing the old value of the *sum*, this result will overwrite the value that the thread T2 computed. In effect, the contribution of the thread T2 will be discarded.

The theoretical model will enable us to quantitatively reason about the effects of potential data races, as well as the mathematical expectation of such data races.

2.2 Theoretical Model Assumptions

I designed the theoretical model to answer the following question: "Given the number of threads on which the program will run, the number of iterations of the loop, and the number of instructions in the loop body, what is the expected value of the sum and what is expected execution time?"

The following assumptions were made during the construction of the theoretical model:

(1) the total number of instructions is counted from the label of the loop to the jump instruction back to the loop, as produced by x86 Assembly;

(2) all local instructions have the same execution time, 1 unit of time;

(3) every write to the global *sum* is committed to memory as soon as the store instruction is executed;

(4) if the program reads one or more times from another global variable, but never writes to it, treat those load instructions as local for the simplicity of the analysis, because we expect that the values will be cached;

(5) we will focus our analysis on programs that perform one write to the global variable *sum*, programs that perform writes to multiple different global variables are out of scope of the problem definition;

(6) the increment variable x is a random variable from a uniform distribution between a and b;

(7) *sum* is initialized to *init*; without loss of generality, we will consider the case when *sum* is initialized to zero;

(8) the function $f(i)$ has no side effects;

(9) the loop is parallelized assuming a static scheduler, which assigns the workload to m threads before executing them by dividing the work into m equal parts;

(10) all the dead code has been eliminated, so only instructions that compute the incremental variable x affect the analysis.

2.3 Experimental Program

I used the following program in my experiments:

```
initialize array a;
SIZE = 1000000;
sum = 0;
for (int i = 0; i < SIZE; i++) {
  x = a[i];
  sum += x;
}
```

I wrote a Python script that generates random numbers from a uniform distribution between 0 and 1 from a particular seed. The generated numbers are written into a file, which serves as input to initialize the array a. Thus, in this program I cover one of the assumptions I made for the theoretical model that reads from a global variable, in this case array a, are functionally equivalent to local instructions due to the modern processors performing caching and pre-fetching of values, if the memory access pattern can be predicted.

I ran the experiments on the CAGnode machine, which is an 8-core Intel Xeon X5460 @ 3.16GHz machine. I collected the results of 200 runs for the accuracy test, and 100 runs for the timing test, and computed the mean and the standard deviation to compare with the results of the theoretical model.

For the timing test of the experiment, I added an inner loop that adds 1 and then substracts 1 from the variable x. This results in addition of

15 instructions with each iteration of the inner loop, according to the x86 Assembly output of the program. Thus, I was able to control exactly how many instructions were added during the execution.

For the accuracy test, 1933 additional instructions were added in order to obtain more realistic results of how accuracy depends on the number of threads the program runs on. This corresponds to 128 executions of the inner loop, as described in the previous paragraph.

The program was always compiled with gcc with all optimizations disabled with the following command:

```
gcc -O0 experiment.c -o experiment -fopenmp
```

3 Results

3.1 Accuracy

Lower/Upper Bound. Since there are N total executions of the loop, m number of threads performing aggregation, and the inputs come from the uniform distribution between a and b, the upper bound on the expected value of the *sum* for 2 and more threads is

$$\mathrm{B}_{\mathrm{max}} = N \cdot \frac{a+b}{2} \tag{1}$$

and the lower bound is

$$\mathrm{B}_{\mathrm{min}} = N \cdot \frac{a+b}{2m} \tag{2}$$

Then, the expected value of the sum is

$$\mathrm{B}_{\mathrm{min}} \leq \mathrm{E}[sum] \leq \mathrm{B}_{\mathrm{max}} \tag{3}$$

For 1 thread, the expected normalized value of the *sum* is always 1.

Combinatorial approach. We can also compute the expected value of the *sum* in a combinatorial way. Combinatorial approach calculates the number of all possible interleavings of threads, given a fixed number of threads the program runs on and the number of instructions in the loop. From all the possible interleavings, we can calculate which of the interleavings and how many of them would produce data races, and how these data races would affect the resulting value. For simplicity of the calculation, but without loss of generality, we assume that the variable x is always set to 1. In section "Effect of Distribution of Inputs" in the Appendix we show that for any input distribution, the expected value of the distribution has no effect on the probability of data races. Therefore, the results of our calculations will not be affected by the distribution from which value of the variable x comes, and having it always be 1 will simplify intermediate calculations.

"Combinatorial Calculation" section provides a detailed combinatorial derivation of the expected value of the sum. It follows from the combinatorial computations that the expected value of summation for 2 threads is 55% of the correct value, for 4 threads 25.46%, and for 8 threads 12.55%. Thus, the global variable *sum* is incremented by 1 in each iteration of the loop.

If we treat the loop to have only the three instructions, load, addition, and store of the global variable *sum*, and run our program on m threads, then the total number of instructions is $\frac{(3m)!}{6^m}$, out of which $m!$ interleavings result in no data races, and all other interleavings produce one or more data races.

Figure 1 shows the relation of the upper and the lower bound estimates, as well as the combinatorial expected value of the sum. The number of threads is represented by the x-axis, and the y-axis represents the fraction of the value of the *sum*, where 1 is the sum calculated when the program runs serially. We observe that with the increasing number of threads, it becomes much more likely to encounter data races, and the most likely data races are the ones that cancel out contributions of all but 1 thread.

Figure 2 shows how the accuracy changes based on the number of threads the program runs on. We observe that the experimental model produces accuracy very close to that of the theoretical combinatorial model with 2 threads, but the accuracy of the experimental program executions on 3 to 8 threads is higher than the accuracy predicted by the combinatorial model. We also observe that the experimental and theoretical curves become almost linear for 3 to 8 threads. These lines have similar slope (-0.032 for theoretical and -0.026 for experimental), but different intercept. We have calculated the differences between the two intercepts to be 0.17.

In addition, I computed the value of the *sum* in terms of local number of instructions added to the loop body for the 2-threaded version of the program. This approximate computation takes as a base step the results of our previous combinatorial analysis of the case when the only loop instructions are load,

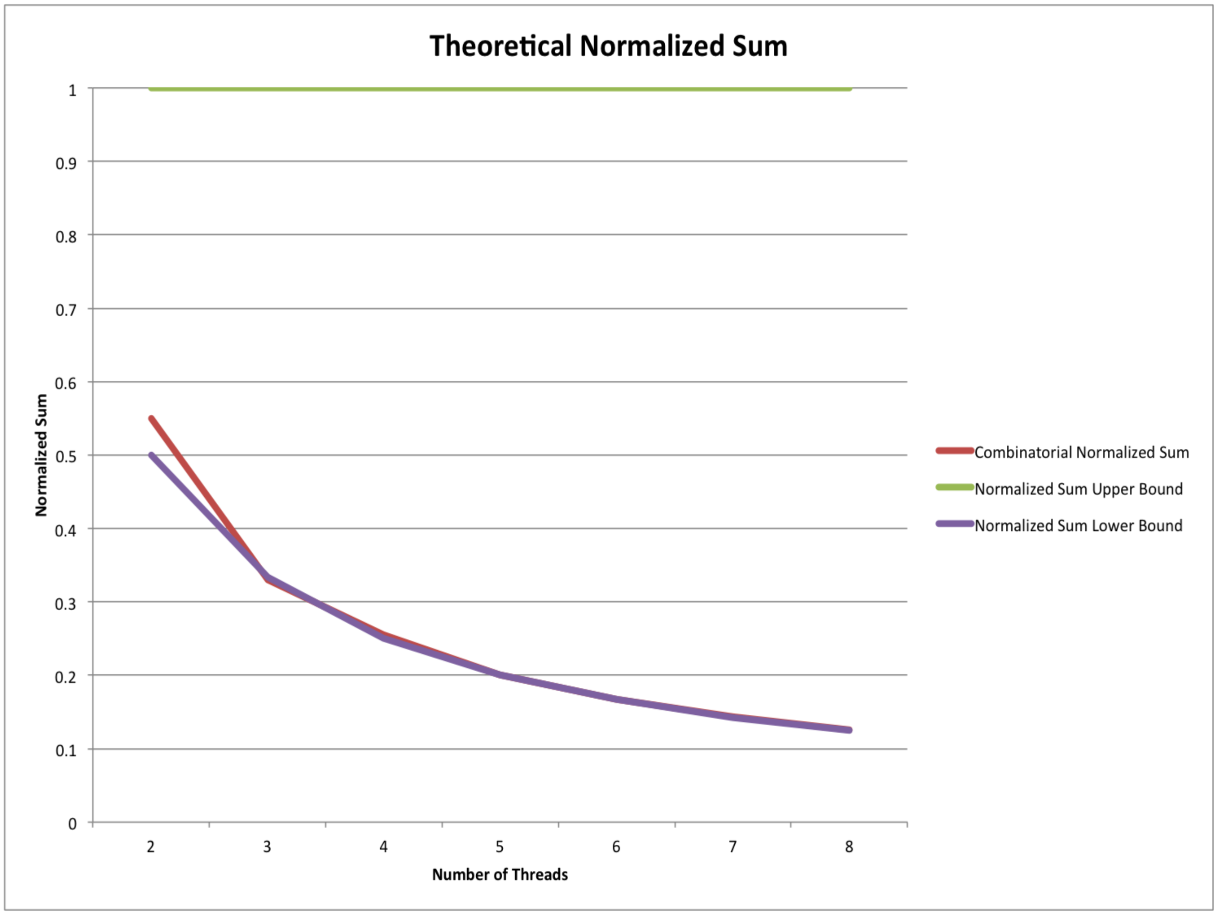


Figure 1: Theoretical Normalized Expected Value of Sum

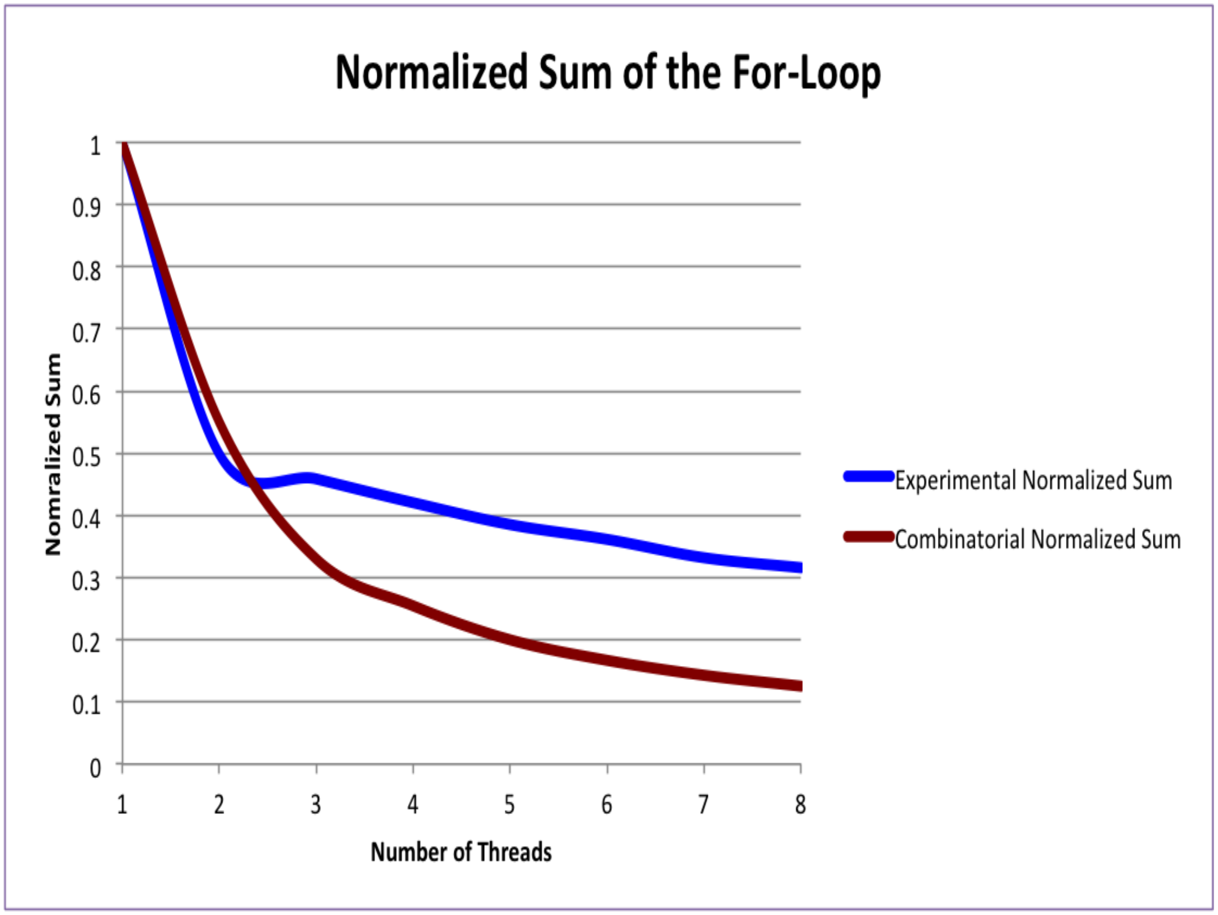


Figure 2: Normalized Sum on Various Number of Threads

add, and store (see “Combinatorial Calculation” in the Appendix). As a guide for this approximation, we assume that most of the data races that will occur in this case are those that we have identified in the base step; the additional instructions, on the other hand, do not cause additional data races.

In the base case, there were 20 possible interleavings of the instructions that operate on the global *sum*. Out of these 18 interleavings produce data races. Now, in the case when we have other instructions in the loop, we approximate the total number of interleavings, *num_interleavings*, as *num_local_instructions* $+ 20$ (where the *num_local_instructions* approximates the contributions of the added instructions, and 20 is the number of interleavings in the base case). Then, the estimated contribution of these 2 interleaved threads is, therefore,

$$\mathrm{Z} = 2 \cdot \frac{num\_interleavings - 18}{num\_interleavings} + 1 \cdot \frac{18}{num\_interleavings} \tag{4}$$

The total estimated sum is equal to $\frac{N}{2} \cdot Z$. Since the sum of the sequential program is N, the normalized estimated sum is $\frac{\mathrm{Z}}{2}$. We want to normalize the result so that our model predicts the expected value of the sum regardless of the number of threads the program runs on.

Figure 3 shows how the increasing number of local instructions affects the accuracy of the program, when run on 2 threads. The expected value of the *sum* increases dramatically with the increasing number of local instructions. Figure 3 also shows the results of the experimental runs of the program on CAGnode machines. Note that since we count all of the instructions in the loop body starting from the first instructions of the loop right below the label and ending at the conditional jump back to the label of the loop, the experimental program starts out with 12 local instructions. Therefore, I plotted the experimental results for 12 and more additional instructions.

3.2 Timing analysis

The theoretical model states that for the single-threaded version, the execution time is just the time it takes to execute local instructions, which is simply the number of local instructions l multiplied by the time it takes to execute each local instruction s. For the m-threaded version, however, it takes $\frac{l \cdot s + t}{m}$, where t is the time it takes to write the value of the global *sum* to memory.

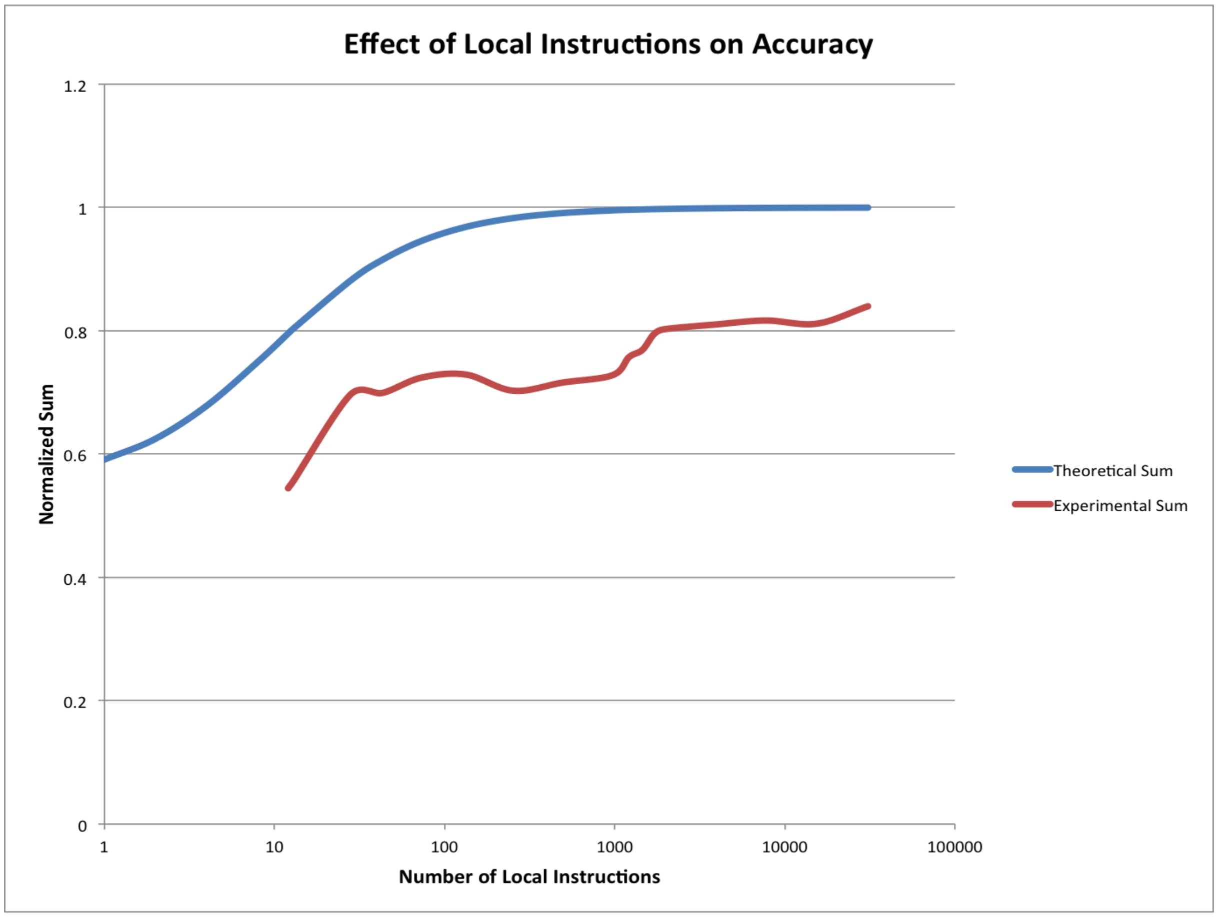


Figure 3: Accuracy with Local Instructions

Figure 4 shows how the number of threads that the program runs on affects the execution time of the program. In both the theoretical model and experimental program we used 1213 local instructions. In addition, time to execute global instructions, t, was set to 100, s was set to 1. We observe that running the experiment on more than 4 cores reveals that the program slows down: it is never as slow as the 2-threaded version, but is slower than 4-threaded version.

Figure 5 shows how the ratio of local instructions to the total number of instructions affects the execution time of the program. The x-axis represents the ratio in percentage, and the y-axis is a logarithmic scale that represents normalized execution time. Normalization of the experimental execution time is necessary to compare the experimental data that was initially in seconds and the theoretical data in the number of local instructions. In Figure 5 (a) we notice when the theoretical model exhibits the largest discrepancies with the experimental results when the ratio of local to total number of instructions is over 90%. For the same number of instructions we observe similar deviation from the theoretical predictions in Figures 5 (b), (c), and (d) for 2, 4, and 8 threads.

4 Discussion

4.1 Accuracy

In Figure 2 we observed that with the increasing number of threads and the static scheduler, the accuracy of the program decreases. Through the combinatorial theoretical model we gain insight into the reasons for this behavior. With the increasing number of threads, the probability of getting an interleaving that results in a data race increases dramatically. We observe that the experimental results perform by about 17% better than the predictions of the theoretical model. Again, we attribute this to aspects of sophisticated architectures of modern machines that perform synchronization of values on the hardware level, as in the example of cache coherency protocols. Also, one of the reasons for the difference in the results of the theoretical model and experimental results is that the theoretical model assumed no local instructions, whereas the experimental model contains 12 additional instructions in the body loop besides the load, add and store to the *sum*. Thus, the theoretical model should be updated with the modern

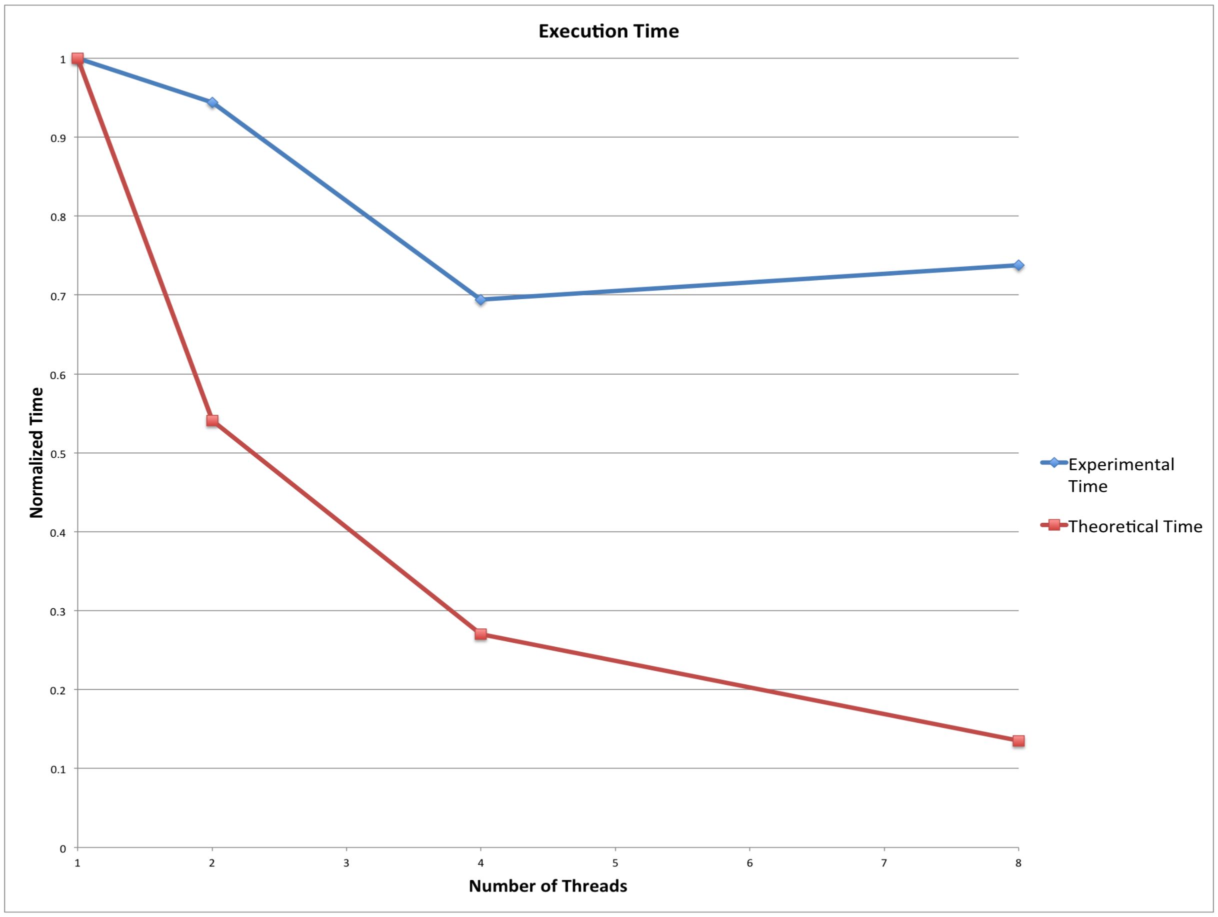
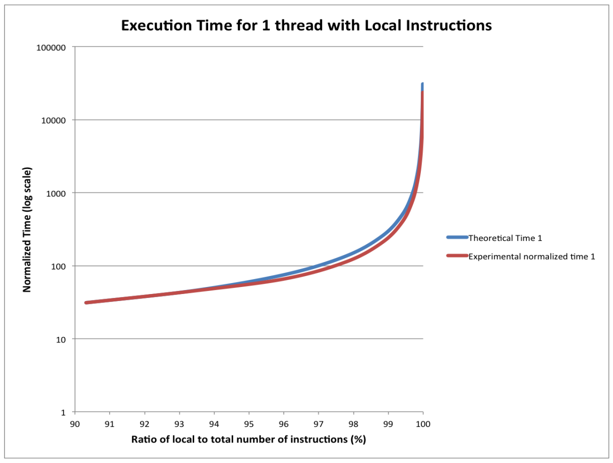
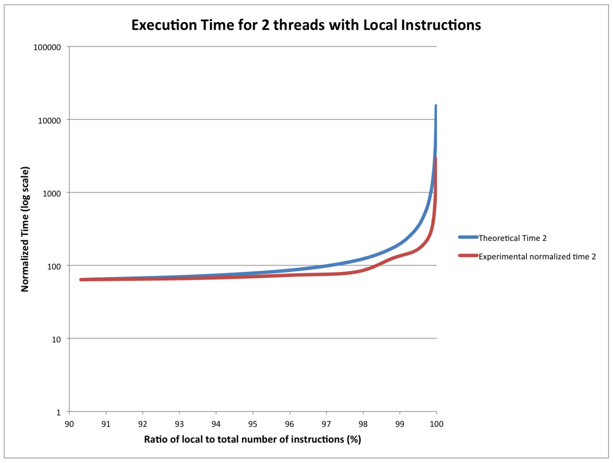


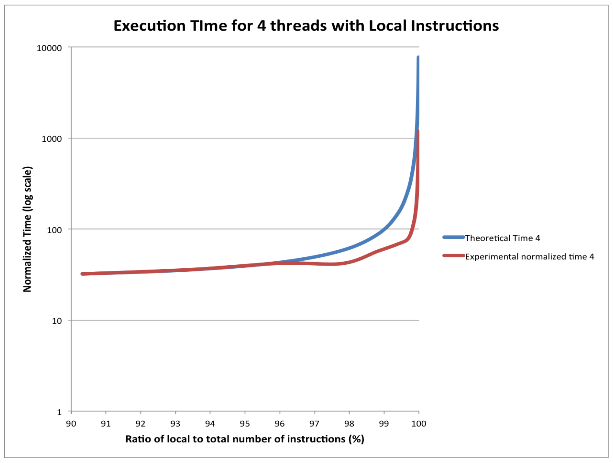
Figure 4: Execution Time on Various Number of Threads



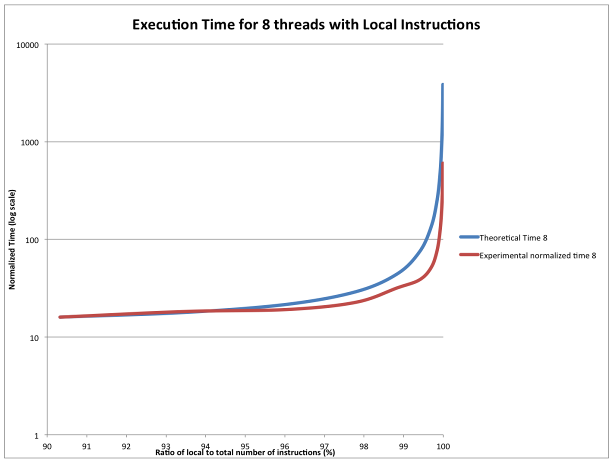
(a) Serial program



(b) 2-threaded program



(c) 4-threaded program



(d) 8-threaded program

Figure 5: Normalized execution time in terms of local instructions

features in order to make better estimates for the expected value of the sum. Constructing such updated model is out of scope of this project, and should be constructed in the future. In the meantime, a better estimate can be given for programs that run on 4 and more threads by adding 0.17 to the expected value of the sum.

In Figure 3 we observe that the accuracy of the experimental results increases rapidly with the number of local instructions added, which corresponds to the predictions of the theoretical model. The behavior of our experimental results deviate from the theoretical model, with accuracy lower than predicted. Thus, a better theoretical model could be constructed in order to provide better estimate. Possible explanation for such discrepancies is the sophisticated features of the modern architectures that the theoretical model didn't take into account.

4.2 Timing

One of the reasons for the slowdown we observe in Figure 4 with 8 threads is the cache coherency protocols: as soon as the computations of one of the threads are committed to memory, the copies of the global variable *sum* in caches of other processors are invalidated, and the processors are forced to wait until the most recently updated value becomes available to them, which causes the additional overhead which affects the execution time. Thus, although we do not explicitly perform any synchronization of threads, the threads implicitly get synchronized through the cache coherency protocols in the hardware. We attribute the differences in theoretical execution times and the experimental results shown in Figure 5 to the aspects of the modern computer architectures that are not reflected in the theoretical model.

5 Related Work

Compiler researchers have done a lot of work to develop compilers that can automatically exploit parallelism available in programs that manipulate dense matrices using affine access fucntions. Several mature compiler systems demonstrated success at exploiting this kind of parallelism [2, 4].

Parallelizing programs while preserving correctness has been one of the major concerns. Researchers developed tools that detect accesses to shared variables in critical sections on-the-fly [3], and data races in threaded programs [1].

More recently, however, engineers found that many applications permit relaxation of the correctness requirements, usually for higher performance and lower resource consumption [8, 5]. Researchers have explored automatic parallelization with potential remaining data races [6]. Concurrent data structures, such as k-FIFO queue, allow a high degree scalability and some correctness guarantees [7]. Such data structures may be distributed, and accesses to them do not have to be synchronized. In addition, synchronization-free algorithms, such as space-division tree construction algorithm [9], produce results that are consistent enough for the purposes despite data races. At last, research is done to develop techniques for recovering from faults in complex hardware and software systems [10].

6 Conclusions

Parallelizing programs without synchronization leads to inaccurate results due to data races. I have developed a theoretical model for the aggregative loops that computes the expected value of the global variable sum, and the execution time of the program based on the number of instructions in the loop body, the number of iterations of the loop, and the number of threads the program will be running on.

I compared the theoretical model to the experimental results. While there is a correlation between the theoretical results and the experimental data, a more updated theoretical model will be developed in the future to improve the accuracy and the execution time predictions.

7 Acknowledgements

I would like to acknowledge and extend my heartfelt gratitute to the following persons who have made the completion of this thesis possible:

My thesis advisor, Prof. Martin Rinard, for the help with the project and inspiration he extended.

Graduate student Sasa Misailovic for the help with the theoretical model and obtaining experimental results, as well as the brainstorming conversations and encouragement.

David Greenberg for the many intellectually stimulating conversations we had that provided me with a better insight into possible explanations of the phenomena I observed with the experimental results, and his vital encouragement and support.

My mother, Gamira Mauzitova, for the encouragement and support she provided throughout the years of my undergraduate studies at MIT.

References

[1] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 69–78, New York, NY, USA, 2006. ACM.

[2] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Effective automatic parallelization with polaris. *International Journal of Parallel Programming*, 1995.

[3] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26(12):85–96, December 1991.

[4] Mary W. Hall, Jennifer M. Anderson, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, December 1996.

[5] Christoph M. Kirsch and Hannes Payer. Incorrect systems: It's not the problem, it's the solution.

[6] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, 2010.

[7] Hannes Payer, Harald Roeck, Christoph M. Kirsch, and Ana Sokolova. Scalability versus semantics of concurrent fifo queues. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of*

distributed computing, PODC '11, pages 331–332, New York, NY, USA, 2011. ACM.

[8] M. Rinard. Acceptability-oriented computing. *ACM SIGPLAN Notices*, 38(12):57–75, 2003.

[9] Martin Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm.

[10] Martin Rinard. What to do when things go wrong: recovery in complex (computer) systems. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 1–2, New York, NY, USA, 2012. ACM.

A Effect of Distribution of Inputs

A note about the summation of 1 or any number from a range [a, b]. If the probability of data race is p, i.e. p is the probability that result will get discarded as a consequence of a data race, then we can write the expected sum as:

$$\mathrm{E}_{\mathrm{Y_i,X_i}}[sum] = \mathrm{E}_{\mathrm{Y_i,X_i}}[Y_1 + Y_2 + \ldots + Y_n] = \sum_{i=1}^{n} \mathrm{E}_{\mathrm{Y_i,X_i}}[Y_i] \tag{5}$$

where Y_i is equal to X_i with probability $1 - p$ and 0 with probability p. So, the expected value is

$$\mathrm{E}_{\mathrm{Y_i}}[Y_i | X_i] = (1 - p) \cdot X_i \tag{6}$$

Since p is independent from parameters of the input distirbution a and b, we can represent the expectation of each element as

$$\begin{aligned} \mathrm{E}_{\mathrm{Y_i,X_i}}[Y_i] = & \\ &= \mathrm{E}_{\mathrm{X_i}}[\mathrm{E}_{\mathrm{Y_i}}[Y_i|X_i]] \\ &= \mathrm{E}_{\mathrm{X_i}}[(1-p) \cdot X_i] \\ &= (1-p)\,\mathrm{E}[X_i] \\ &= (1-p)\frac{b+a}{2} \end{aligned} \tag{7}$$

Therefore, distribution from which the inputs come doesn't matter – whether it is always 1, or a uniform distribution between a and b, or a gaussian, or other – as long as it does not depend on the probability p. This justifies why in our combinatorial calculations of the expected sum we looked only at all increments of the sum equal to 1.

B Combinatorial Calculation

I performed program analysis on a very simple kind of aggregative loop. With each iteration of the loop the program increments the global variable *sum* by 1:

```
sum = 0;
for (int i = 0; i < N; i++) {
 sum += 1;
}
```

We will simplify the model by considering only the three instructions in the loop body:

```
load $SUM, %sum
add  %sum, 1
store %sum, $SUM
```

where \$SUM is the memory location for the global variable *sum*.

Suppose we want to run our program on 2 threads ([1] and [2] refer to the threads 1 and 2 respectively).

```
Thread 1                          Thread 2

[1] load $SUM, %sum               [2] load $SUM, %sum
[1] add  %sum, 1                  [2] add  %sum, 1
[1] store %sum, $SUM              [2] store %sum, $SUM
```

There are 20 different ways the two threads could be interleaved. There are 2 interleavings that would result in no data races. One interleavings is as follows:

```
[1] load $SUM, %sum
[1] add  %sum, 1
[1] store %sum, $SUM
[2] load $SUM, %sum
[2] add  %sum, 1
[2] store %sum, $SUM
```

In the other interleaving the order of the two threads is swapped, i.e. Thread 2 executes all of its instructions before Thread 1 does. No data race will occur because we load *sum* in one of the threads only after the other threads completed storing its updated value of the *sum*.

Other interleavings will result in a data race, like in the following example:

```
[1] load $SUM, %sum
[1] add  %sum, 1
[2] load $SUM, %sum
[2] add  %sum, 1
[2] store %sum, $SUM
[1] store %sum, $SUM
```

Note here that load [2] happened before store[1], so the result of incrementing the *sum* by the Thread 2 is canceled, because store[2] is followed by subsequent store[1]. Analogously, there can be other interleavings that would result in loosing either the contirbuiton of the Thread 1 or Thread 2.

The goal of the analysis is to calculate how many interleavings, relative to the total number of interleavings, will result in data races. For this particular example we have 2 "good" interleavings and 18 "bad" interleavings out of 20 possible instruction interleavings.

Now, we can calculate expected number of data races, as well expected value that will be produced by our parallelized program. In our calculation of expected number of data races we will say there are 0 data races in "good" interleavings and 1 data race in "bad" interleavings. Therefore, the expected number of data races in this example is equal to the probability of dataraces occurring. In the equation below variable D is a Bernoulli distributed variable, i.e. $D = 0$ if no data races, $D = 1$ if there is a data race.

$$\mathrm{E}[D = 1] = \frac{2}{20} \cdot 0 + \frac{18}{20} \cdot 1 = \frac{9}{10} \tag{8}$$

The expected value of the summation in this case is then:

$$\mathrm{E}[Z] = \frac{2}{20} \cdot 2 + \frac{18}{20} \cdot 1 = \frac{11}{10} \tag{9}$$

Thus, the expected value of summation in this case is 55% of correct. So, if we have 2 threads and 100 total iterations of the loop, the expected value of the result of the *sum* is 55.

For 4 threads the expected number of data races is 99.99%, and the expected sum is 25.46% of correct. For 8 threads, the expected sum is 12.55%, approximately $\frac{1}{8}$ of the correct value.

In general, for n threads, each of which executes only the 3 instructions – load, add, and store – the total number of instructions that need to be executed is $3n$. There are $\binom{3n}{3}$ ways to pick which 3 slots in the sequence of instructions are occupied by the Thread 1. After that, there are $\binom{3n-3}{3}$ ways to pick from remaining slots space for instructions of Thread 2. Afterwards, Thread 3 has $\binom{3n-6}{3}$ ways to pick slots for its instructions. For thread k, $3(k-1)$ slots have already been picked, so there are $\binom{3n-3(k-1)}{3}$ ways to pick slots for thread k. Thus, the total number of interleavings is:

$$\binom{3n}{3} \cdot \binom{3n-3}{3} \cdot \binom{3n-6}{3} \cdot \ldots \cdot \binom{6}{3} \cdot \binom{3}{3} = \frac{(3n)!}{(3!)^n} = \frac{(3n)!}{6^n} \tag{10}$$

Out of the total number of possible interleavings, there are $n!$ interleavings that produce no data races. Hence, there are $\frac{(3n)!}{6^n} - n!$ interleavings that result in one or more data races. In order to figure out the expected value of the sum for 3 and more threaded programs, additional calculations need to be performed in order to compute what fraction of interleavings will produce 1 and more data races. In general, with the increasing number of threads, the expected value of the sum approaches $\frac{1}{m}$, where m is the number of threads.