Wren Paris-Moe
9/19/2019
CS 317
Professor Jeffery Miller

# Lab Project 4 – Parallel Processing

Multi-threading is a programming technique that divides tasks into smaller, more manageable workloads that can be completed concurrently. The method allows multiple threads within a single-processor system to execute independently while sharing the same resources. Almost all modern processors have multiple cores. Within the cores are threads which are utilized to make running multiple processes at once possible. Without using threads, a processor performs tasks consecutively one after the other. This causes tasks to be run in queue fashion where a single time-consuming task forces the remaining ones to wait for it to finish. When implemented properly, taking control of thread use within a processor can greatly increase runtimes of a program.

The effectiveness of multi-threading was tested by computing the sum of the first 100,000,000 positive integers in several ways. That is by computing $\sum_{k=1}^{N} k$ where N = 100,000,000. The sum of the first N positive integers equals a total of 50,000,000,500,000,000. First, the sum was computed in a single for loop without the use of threading. The first test of threading was implemented by computing the entire sum in a single for loop within a single thread. Next N was split into R equal sized integer partitions to be computed in parallel on separate threads. Afterwards, the partial sums were added to get the total sum. Many variations were tested with R = 2, 3, 4, 5, 6, 7, and 8. Then R was set as 16, 64, 256, and 1024. Note that these values exceed the physical number of threads existing within the CPU of the computer used to compute the sum. First the programs were implemented in Java and then in Python.

## Java:

Computing the sum in a single for loop without multi-threading was simple. The sum was timed over 10 trials to record an accurate average runtime. The number of nanoseconds since the start of the day was recorded directly before and after the program was run. The difference between the two values was calculated to get the runtime of the process.

```
int N = 100000000;
long sum = 0;
long timeSum = 0;
for(int i = 0; i < 10; i++) {
   long start = System.nanoTime();
   for (int k = 1; k <= N; k++) {
      sum += k;
   }
   long elapsed = System.nanoTime() - start;
   timeSum += elapsed;
}
long avgElapsed = timeSum / 10;
System.out.println(N + ", " + sum + ", " + avgElapsed);
```

Computing the total sum with multi-threading was more complex and required the use of an object class that extended the Java Thread class. A tester class was used to facilitate the process by creating and initializing threads for the given R values. The threadSum class and tester class are shown below to illustrate how the technique was implemented with an ArrayList of threads.

```
public class threadSum extends Thread {
   public int start, end;
   public long sum;
   public threadSum(int s, int e) {
      this.start = s;
      this.end = e;
      this.sum = 0;
   }
   @Override
   public void run(){
      for (int k = start; k <= end; k++) {
         sum += k;
      }
   }
   public long getSum(){
      return sum;
   }
}
```

```
public class threadSumTester {
   static final int R = 1;
   public static void main(String[] args){
      int N = 100000000;
      long sum = 0;
      long timeSum = 0;
      for(int j = 0; j < 10; j++) {
         ArrayList<threadSum> tArrayList = new ArrayList<threadSum>();
         int r = N / R;
         int e = r;
         int s = 1;
         long start = System.nanoTime();
         for (int i = 0; i < R; i++) {
            if (i == (R - 1)) {
               e = 100000000;
            }
            threadSum t = new threadSum(s, e);
            t.setName("Thread" + (i + 1));
            t.start();
            tArrayList.add(t);
            s += r;
            e += r;
         }
```

```
        for (int i = 0; i < R; i++) {
            threadSum twr = tArrayList.get(i);
            while (twr.getState() != Thread.State.TERMINATED);
            sum += twr.getSum();
        }
        long elapsed = System.nanoTime() - start;
        timeSum += elapsed;
    }
    long avgTime = timeSum / 10;
    System.out.println(N + "   " + sum + "   " + avgTime);
  }
}
```

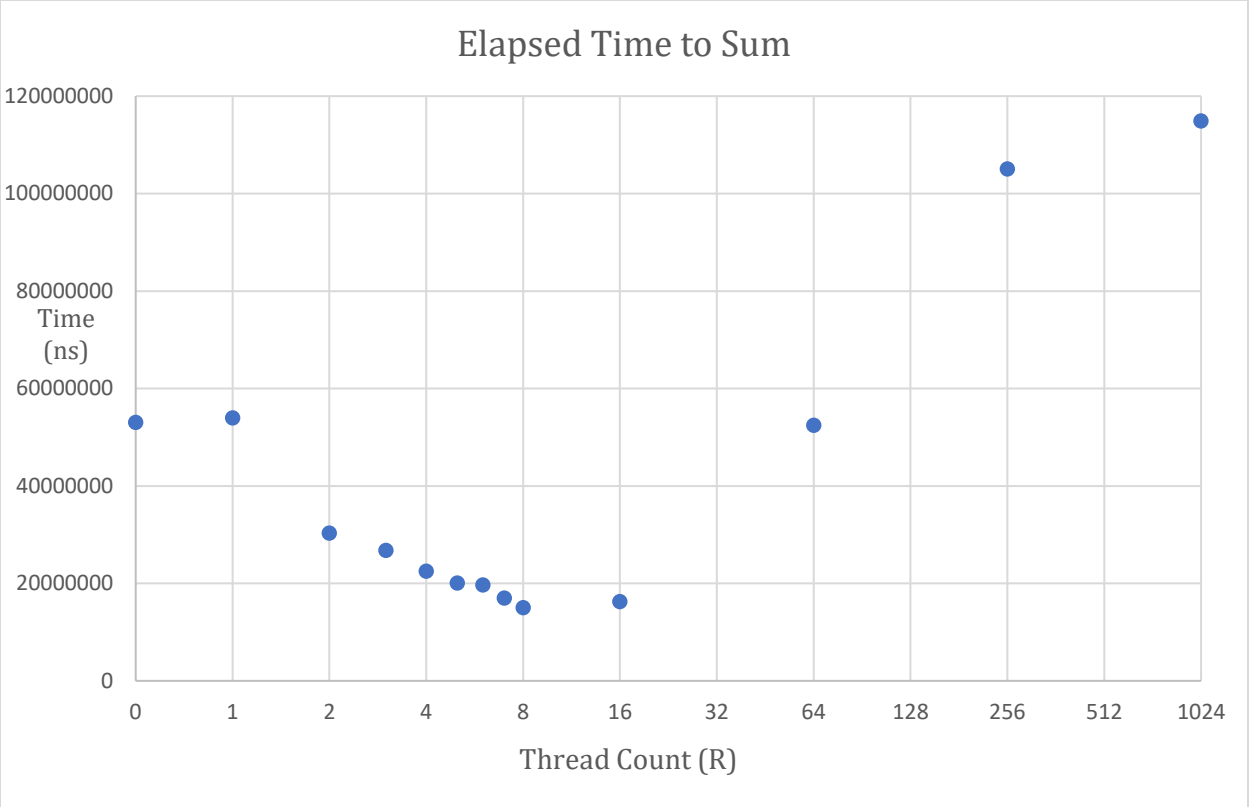
Data:

On the right is the elapsed time taken to compute the total sum for each R value in Java. The elapsed time for R = 0 represents when the sum was calculated in a single for loop with no use of threading. The first takeaway from the data was that multi-threading decreased runtimes from the original simple sum test. Using a single thread was slower than running the regular sum method. Though, by incrementing the R value the runtimes quickly became faster. The elapsed time to sum decreased for each value as R increased from 1 to 8 where the fastest runtime was reached. Dividing the sum among 8 threads made the runtime 3.5 times faster than without multi-threading. After the use of 8 threads, the trend reversed where runtimes slowed as R increased towards 1024. The elapsed time for R = 64 increased by over 3 times that of R = 16 while the use of 256 threads took 2 times longer to compute than it did when using 64.

| R | Elapsed Time (ns) |
|---|---|
| 0 | 53017820 |
| 1 | 53962490 |
| 2 | 30311110 |
| 3 | 26756320 |
| 4 | 22473970 |
| 5 | 20048650 |
| 6 | 19668890 |
| 7 | 17006150 |
| 8 | 14995760 |
| 16 | 16249460 |
| 64 | 52424740 |
| 256 | 105090000 |
| 1024 | 114936950 |

| R | Elapsed Time (ns) | Approximation ($5E+07R^{-0.567}$) |
|---|---|---|
| 1 | 53962490 | 50000000 |
| 2 | 30311110 | 33750950 |
| 3 | 26756320 | 26818980 |
| 4 | 22473970 | 22782532 |
| 5 | 20048650 | 20074929 |
| 6 | 19668890 | 18103321 |
| 7 | 17006150 | 16588206 |
| 8 | 14995760 | 15378642 |

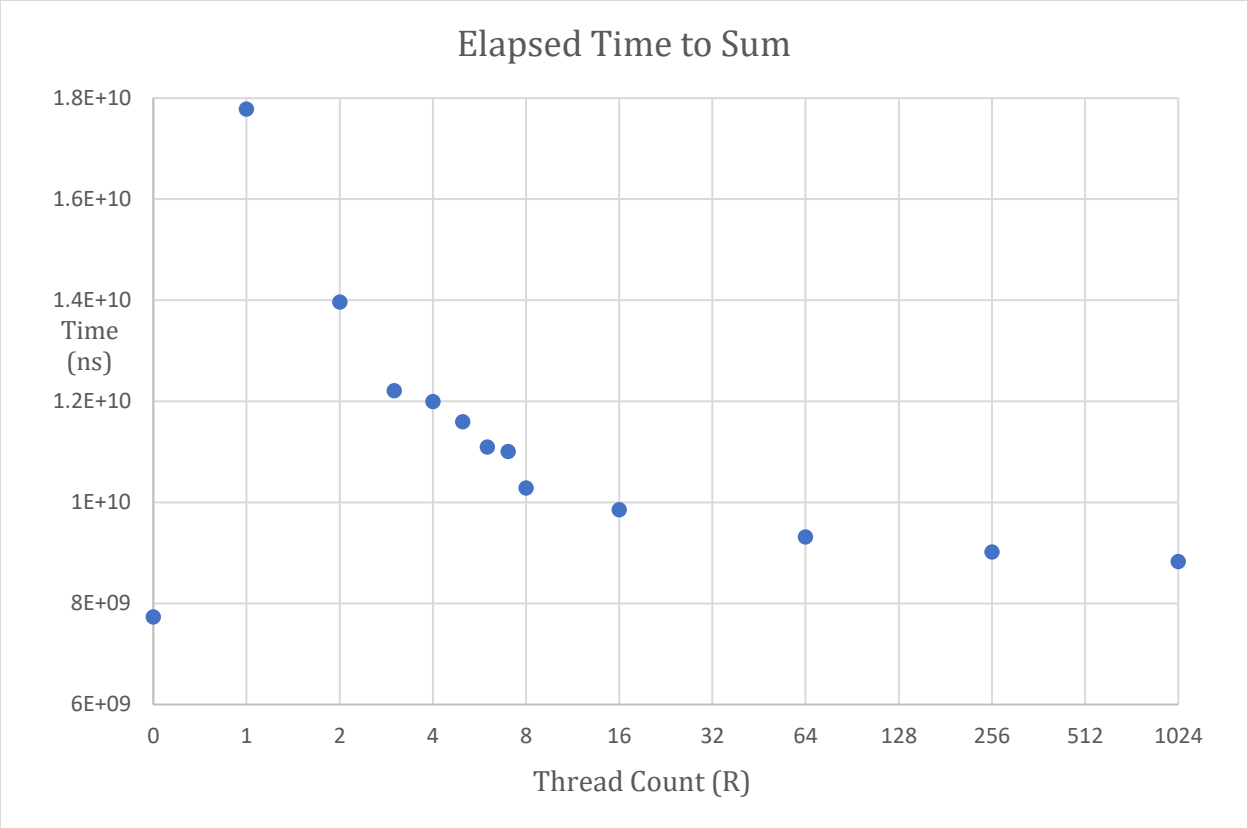
A closer look at the data for R = 1 through R = 8 showed a noteworthy trend for these thread ranges. An approximation formula was found that fit the data almost exactly. The power formula of $y = 5E+07R^{-0.567}$ had an $R^2$ value of 0.9775. This means as R approached 8, the runtimes decreased at a rate similar to $\frac{1}{\sqrt{R}}$. The rate of decrease started out fast and slowed as the data progressed. The jump in runtime from R = 7 to R = 8 was minimal compared to the differences between the lower values. Interestingly, the data for R = 8 to R = 1024 increased at a logarithmic rate with an approximation function of 2E+07ln(R) - 5E+07 and an $R^2$ value of 0.9529 (not shown). The entire dataset is plotted on the next page to show the progression of runtimes in Java.




## Python:

Python was found to take much longer to compute the total sum than Java. When ran individually, the runtimes for a given R value seemed to vary immensely. For any R, the program could be run once to find one elapsed time and ran a minute later to find a greatly different runtime. A solution was found to fix this issue which ended up showing a trend in the data. The R values were placed in a list which were iterated through by a for each loop computing the runtimes consecutively. The simple sum (R=0) was computed immediately before the for each loop was initialized. Python had a trend distinct from its equivalent implementation in Java. Unlike Java, the sum was computed the fastest without the use of multi-threading. Using a single thread for the entire sum caused runtimes to be the slowest. The runtime of R = 1 was 2.3 times slower than when R = 0. For each addition of the thread count, runtimes became faster. Java reached its best runtime at R = 8 while Python continued to decrease runtimes as R approached 1024. This illustrates a distinct difference between implementations of multi-threading in Java and Python. A scatter plot with the data is given on the next page to illustrate the trend in runtimes as R got larger.

| **R** | **Elapsed Time (ns)** |
|---|---|
| 0 | 7731385600 |
| 1 | 17783419200 |
| 2 | 13963388700 |
| 3 | 12212295100 |
| 4 | 11995349800 |
| 5 | 11594223600 |
| 6 | 11096585900 |
| 7 | 11003953800 |
| 8 | 10286177500 |
| 16 | 9856963900 |
| 64 | 9315780100 |
| 256 | 9016336400 |
| 1024 | 8826056300 |




The graph makes it obvious how much faster the algorithm was in Python when run without the use of multi-threading. From R = 1 onward, each addition of threads caused the total runtime to decrease with the margin between points decreasing as R got larger. By the time R = 64, 256, and 1024, the difference between elapsed times was minimal. It's almost as if there was a horizontal asymptote directly above the point for R = 0. As R increased the runtimes came closer to that of the original sum time as the jumps in runtimes became smaller. No approximation method was found to be accurate when treating R as its exact values. However, when looking at R = 1 through R = 1024, once R was treated as a linear progression from R = 1 to 12 a distinct power formula was found to fit the data. The approximation method of $2E+10R^{-0.26}$ had an $R^2 = 0.9727$ meaning it was extremely accurate. This rate of increase was similar to $\frac{1}{\sqrt[4]{R}}$.

## Discussion/Analysis:

Java was shown to perform better than Python when computing the sum of the first 100,000,000 positive integers. Finding the total sum in Java was almost instantaneous while the same program in Python forced the user to wait a significant amount of time for it to complete. There was a distinct difference between the two programming languages. Java reached its fastest runtime when a thread count of R = 8 was used while Python was its most efficient without the use of threading. For Python, the attempt to divvy up the workload through multi-threading ended up making the algorithm slower, the exact opposite of the goal of the technique. The reason why the two languages experienced such a difference in trends was due to Python's use of the Global Interpreter Lock (GIL). The GIL makes true concurrent multi-threading impossible. The python compiler uses the GIL to jump between threads in order to complete a task in the most efficient

manner. The GIL will move between threads but no two will truly be active at the same time. This causes multi-threading to be an inefficient technique in Python. Having R = 1 restricted the GIL from processing a program on multiple threads. Essentially, instead of letting the compiler find the best way to complete a task by moving the GIL around, it is restricting it to only one thread. Using two threads restricted the GIL to only jumping between two so it is faster than using a single thread but slower than letting the compiler complete the task on its own. As the thread count increased, more partial sums are computed at once with the GIL.

Java on the other hand was able to implement true multi-threading without the existence of a GIL equivalent. The CPU used in the computer to implement the programs was an Intel Core i7-8850H. This CPU is quad core with hyperthreading giving it up to eight processable threads. This explains why Java reached its fastest runtime at R = 8. All 8 threads were run in parallel with a partial sum assigned to each available thread. This is the most efficient way to utilize true multi-threading on a single-processor system. Initializing more than 8 threads caused additional threads to wait until a thread finished when it could take its place. After using 8 threads, every increment of R resulted in worse runtimes than the last. When utilizing multi-threading it is important to take note of the specific computer hardware that is being used. This allows the technique to be implemented effectively.