Ulysses Palomar

Wednesday, October 12, 2022

Parallel Programming Comparison of Two Multithreaded Programs

The purpose of this assignment is to compare the efficiency of threads by comparing two different programs that accomplish the same task. The Sqrt program will find the square root of a number using the java Sqrt() method. The Factor program will also find the square root of a number but will do so by factoring.

To accomplish this, I will be using the provided c++ files on Canvas and translating them to java. This can be accomplished by just breaking the program down into small easily translatable sections and translating. Once translated, I will run the program calculating the square root starting from 1 to a given number while also incrementing the number of threads created and used. I used my personal which has 4 cores.

In theory, the more threads I use, the faster the program should run. So, if I take the actual time taken using 1 thread and divide it in half, I should get the time that 2 threads will take. And so forth for 3 and 4 threads. My results were as follows for the Sqrt program:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sqrt = 200,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 1335 | 618.6 | 412 | 310.2 | 246.2 | 205.6 | 179.8 | 165.4 | 198.6 | 202.6 | 208 |
| Expected | 1335 | 667.5 | 445 | 333.75 | 267 | 222.5 | 190.714 | 166.875 | 148.333 | 133.5 | 13.35 |
| SpeedUp | 1335 | 2.1581 | 3.24029 | 4.30368 | 5.42242 | 6.49319 | 7.42492 | 8.07134 | 6.72205 | 6.589339 | 6.418269 |
| Sqrt = 20,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 154.4 | 67 | 47 | 32 | 25.6 | 21 | 18.8 | 19.2 | 28.2 | 25.6 | 22.8 |
| Expected | 154.4 | 77.2 | 51.4667 | 38.6 | 30.88 | 25.7333 | 22.0571 | 19.3 | 17.1556 | 15.44 | 1.544 |
| SpeedUp | 154.4 | 2.30448 | 3.28511 | 4.825 | 6.03125 | 7.35238 | 8.21277 | 8.04167 | 5.47518 | 6.03125 | 6.77193 |
| Sqrt = 2,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 15.4 | 7 | 4.8 | 4 | 3.4 | 2.6 | 3.4 | 5 | 4.2 | 4 | 12.6 |
| Expected | 15.4 | 7.7 | 5.13333 | 3.85 | 3.08 | 2.56667 | 2.2 | 1.925 | 1.71111 | 1.54 | 0.154 |
| SpeedUp | 15.4 | 2.2 | 3.20833 | 3.85 | 4.52941 | 5.92308 | 4.52941 | 3.08 | 3.66667 | 3.85 | 1.222222 |
| Sqrt = 200,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 3 | 1 | 1 | 1 | 1 | 0.8 | 1 | 1.4 | 1.2 | 1.4 | 10.2 |
| Expected | 3 | 1.5 | 1 | 0.75 | 0.6 | 0.5 | 0.42857 | 0.375 | 0.33333 | 0.3 | 0.03 |
| SpeedUp | 3 | 3 | 3 | 3 | 3 | 3.75 | 3 | 2.14286 | 2.5 | 2.142857 | 0.294118 |
| Sqrt = 20,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 1 | 0.4 | 1 | 0.8 | 1 | 0.8 | 1 | 1.4 | 1.2 | 1.8 | 11 |
| Expected | 1 | 0.5 | 0.33333 | 0.25 | 0.2 | 0.16667 | 0.14286 | 0.125 | 0.11111 | 0.1 | 0.01 |
| SpeedUp | 1 | 2.5 | 1 | 1.25 | 1 | 1.25 | 1 | 0.71429 | 0.83333 | 0.555556 | 0.090909 |

As you can see, the larger the Sqrt range, the more the actual results reflected the expected results. Of course, making 1 thread calculate the square root of 1 to 200 million will take time. So, it makes sense to see that adding more threads to calculate the square roots up to 200 million significantly reduces the time required. My data reflects that assumption for large ranges (200 million and 20 million). But when calculating the square roots for values up to 2 million, the data showed that it would take longer when using more threads. This shows that calculating square roots of such large numbers is such a heavy workload that the program benefits from creating multiple threads to complete the task. But when calculating smaller ranges of square roots, the task is so simple and fast, that it takes longer for the program to create the threads and run them, than it does to just calculate the square roots. So, if you look at the graphs for 200 and 20 million, the actual data lines up almost identically to the expected data. Since it is such a heavy workload, the increase in threads help lower the total time. But if you look at 20 thousand for example, it’s such a light workload that the computer takes more time creating and running multiple threads than it does to just calculate the square roots.

And these were my results for the Factor program:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Factor = 2,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 144773 | 106135 | 79950 | 63616 | 56916 | 54366 | 50974 | 47745 | 45279 | 42715 | 36368 |
| Expected | 144773 | 72386.5 | 48257.7 | 36193.3 | 28954.6 | 24128.8 | 20681.9 | 18096.6 | 16085.9 | 14477.3 | 1447.73 |
| SpeedUp | 144773 | 1.36405 | 1.81079 | 2.27573 | 2.54363 | 2.66293 | 2.84013 | 3.03221 | 3.19735 | 3.38928 | 3.98078 |
| Factor = 200,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 1884 | 1453 | 1061 | 1028 | 840 | 829 | 666 | 729 | 576 | 529 | 470 |
| Expected | 1884 | 942 | 628 | 471 | 376.8 | 314 | 269.143 | 235.5 | 209.333 | 188.4 | 18.84 |
| SpeedUp | 1884 | 1.29663 | 1.77568 | 1.83268 | 2.24286 | 2.27262 | 2.82883 | 2.58436 | 3.27083 | 3.56144 | 4.00851 |
| Factor = 20,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 31 | 24 | 21 | 20 | 18 | 19 | 17 | 17 | 15 | 18 | 19 |
| Expected | 31 | 15.5 | 10.3333 | 7.75 | 6.2 | 5.16667 | 4.42857 | 3.875 | 3.44444 | 3.1 | 0.31 |
| SpeedUp | 31 | 1.29167 | 1.47619 | 1.55 | 1.72222 | 1.63158 | 1.82353 | 1.82353 | 2.06667 | 1.72222 | 1.63158 |

Because Factor calculates the square root in a much more extensive way compared to Sqrt, even small ranges take large amounts of time compared to Sqrt. But nonetheless, there was a decrease in total time when increasing the number of threads, but only to a certain number of threads. As mentioned before, my laptop has 4 cores. And you can see from the graphs that once the number of threads exceeded 4, there was no longer a significant decrease in time. The data shows that after 4 threads, the time data plateaus and no longer reflects the expected data. It doesn’t take longer as it did with the Sqrt program. But, like I said, I believe it is because the Factor program Is already a longer version of Sqrt so even with the small range of 20 thousand, the data matches the similarly to the 2 million range of the Sqrt program. Sqrt has a significant decrease in time as more threads get added when calculating a large enough range. And an increase in time as more threads get added when calculating smaller ranges. Factor, however, exhibits a plateau in time as threads increase past available cores on the laptop, regardless of range.

These tables show my actual vs expected speed up:

These speedup graphs show that in Sqrt, the actual speedup positively correlates to the expected speedup. At least until the thread count reaches 100. This is simply due to the fact that the program exceeds the number of physical cores on my laptop. When the number of threads exceeds the number of cores on my laptop, there is no more efficiency that the program can squeeze out of my laptop. The speedups on the Factor program reflect the same principle. Once the number of threads increases to 100 threads, the actual speedup can no longer keep up with the estimated speedup.

While in the process of getting my Sqrt program to run properly, I was having doubts that my data was accurate. Someone mentioned to me that it may be because my laptop is not in the best of conditions. So, I tried running my program on the lab computers via SSH. The lab computers also have 4 cores, so the results should have been similar to the results from my laptop. I ran the Sqrt on the terminal because I couldn’t get IntelliJ to work on the lab computers, the JDK wasn’t setup, and it wasn’t letting me set it up. So, maybe if I ran it on the lab computer’s IntelliJ instead of the terminal, I would have gotten different results. Nonetheless, here are my results for the Sqrt program from running them on the lap computers’ terminal while

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sqrt = 200,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 1412 | 687.2 | 460.8 | 348.4 | 407.6 | 364.2 | 373.8 | 358 | 363.4 | 375.6 | 381.6 |
| Expected | 1412 | 706 | 470.667 | 353 | 282.4 | 235.333 | 201.714 | 176.5 | 156.889 | 141.2 | 14.12 |
| ASpeedUp | 1412 | 2.05471 | 3.06424 | 4.05281 | 3.46418 | 3.87699 | 3.77742 | 3.94413 | 3.88553 | 3.75932 | 3.70021 |
| ESpeedUp | 1412 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Sqrt = 20,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 145.6 | 85 | 58 | 67 | 62 | 58 | 62 | 51 | 58 | 54 | 65 |
| Expected | 145.6 | 72.8 | 48.5333 | 36.4 | 29.12 | 24.2667 | 20.8 | 18.2 | 16.1778 | 14.56 | 1.456 |
| ASpeedUp | 145.6 | 1.71294 | 2.51034 | 2.17313 | 2.34839 | 2.51034 | 2.34839 | 2.8549 | 2.51034 | 2.6963 | 2.24 |
| ESpeedUp | 145.6 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Sqrt = 2,000,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 16 | 12 | 14 | 13 | 12 | 14 | 13 | 13 | 16 | 12 | 17 |
| Expected | 16 | 8 | 5.33333 | 4 | 3.2 | 2.66667 | 2.28571 | 2 | 1.77778 | 1.6 | 0.16 |
| ASpeedUp | 16 | 1.33333 | 1.14286 | 1.23077 | 1.33333 | 1.14286 | 1.23077 | 1.23077 | 1 | 1.33333 | 0.94118 |
| ESpeedUp | 16 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Sqrt = 200,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 |
| Expected | 3 | 1.5 | 1 | 0.75 | 0.6 | 0.5 | 0.42857 | 0.375 | 0.33333 | 0.3 | 0.03 |
| ASpeedUp | 3 | 1 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 0.6 |
| ESpeedUp | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Sqrt = 20,000 | | | | | | | | | | | |
| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |
| Actual | 4 | 4 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 2 | 39 |
| Expected | 4 | 2 | 1.33333 | 1 | 0.8 | 0.66667 | 0.57143 | 0.5 | 0.44444 | 0.4 | 0.04 |
| ASpeedUp | 4 | 1 | 4 | 4 | 4 | 4 | 2 | 4 | 4 | 2 | 0.10256 |
| ESpeedUp | 4 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 100 |

being SSH’ed in from my laptop:

Compared to the times from my laptop, the lab computer showed a significant increase in time. Where my laptop was almost identical to the lab computers for large square root numbers, the lab computers had times greater than expected in all cases. While I’m not sure the root cause for this, I do have quite a few theories as to why. Since I was SSHing into the lab computers from home over the internet, there is a delay due to the internet. I could have had periods in which my internet was slow which could’ve caused the increase in time. Also, when SSHing into the lab, it could’ve connected me to a computer that another student was already using. That would then be 2 students using the same computer; me having my program utilize the 4 cores the computer has while the student is also utilizing an unknown number of cores. So, the power of the cores is being split between the other student and I, causing the increase in times. The one thing that is similar is the fact the when I do 100 threads at low numbers, the time drastically increases. As mentioned before, this is because the workload is so low that it is taking more to create the threads, meant to speed up the process, than it is to run the program.

I ran several tests to compare the time taken when printing each square root, saving each square root to a file, and only printing the sum. For some reason, my code couldn’t save the square roots to a file when testing large numbers of square roots. But nonetheless, here is my data:

This data shows that when printing each square root to the display causes the program to take the same amount of time regardless of the number threads. This is because although the threads split the workload, only one thread can print to the display at one time. So, they still take turns printing to the display. Therefore, the time stays consistent as the number of threads increases. However, when saving the output to a file instead, the time does decrease as the number of threads increases. But only when increased to 2 threads. After 2 threads, the time plateaus. It is because of the same principle as printing the output. The threads take turns saving the output to the file. And finally, when only printing the sum total, the program follows the expected times almost identical. The only thing of note worthy though is that when the number of threads reaches higher numbers, it actually takes more because my program is printing the sum of each thread. So, for example, when printing the sum of 20,000 square roots, it took significantly longer to print the sum of each thread when using 100 threads.

In conclusion, there are many different factors that can affect a program’s runtime. My data shows just a few examples of variables that can either increase or decrease a program’s runtime. This is important to remember because there is no correct way to write a program. As long as it runs and runs correctly, that is all that matters. But if you wish to make it run more efficiently, it is important to be aware of the factors that can make it run more efficiently. And one way to do that is to know what system you are using. Some machines are more powerful than others so they may produce extremely different results when running the same program. Nonetheless, creating a more efficient program isn’t always about writing better functions, but also about knowing your operating system and system information.

1. Sqrt Function

//  
// SquareRoot Java Program  
// This program spawns children, who will help to determine square roots.  
// To compile: javac SquareRoot.java  
// To run: java SquareRoot  
// To monitor system resources: (Linux): Applications->System Tools->System Monitor->Resources  
// Use Spreadsheet Applications->Office->LibreOffice Calc to show efficiency of multiple threads.  
//  
  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.lang.Math;  
import java.lang.Thread;  
import java.util.ArrayList;  
import java.util.List;  
  
  
  
class SquareRoot {  
 public static void main(String[] args) {  
// String filename= "Time.txt";  
// File timeFile = new File(filename);  
// if (timeFile.exists()){  
// timeFile.delete();  
// }  
 int numChild = 1;  
 for (int k = 1; k <= 11; k++) {  
  
// for (int j = 1; j <= 5; j++) {  
 if (k == 11) {  
 numChild = 100;  
 }  
 //int numChild = 1;  
 final int total = 20000;  
 double range = total / numChild;  
 double begin = 0.0;  
 List<Thread> threads = new ArrayList<Thread>();  
  
  
 System.*out*.println("Run SquareRoot " + total + ":" + numChild);  
 long start\_s = System.*currentTimeMillis*();  
 // Spawn children processes  
 for (int i = 0; i < numChild; i++) {  
 Thread t = new Thread(new Child(begin, begin + range));  
 t.start();  
 threads.add(t);  
 begin += range + 1;  
 }  
  
 // Wait for children to finish  
 for (Thread t : threads) {  
 try {  
 t.join();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 long stop\_s = System.*currentTimeMillis*();  
 System.*out*.println("time: " + (stop\_s - start\_s));  
// try  
// {  
//  
// FileWriter fw = new FileWriter(filename,true); //the true will append the new data  
// fw.write((stop\_s - start\_s) + "\n");//appends the string to the file  
// fw.close();  
// }  
// catch(IOException ioe)  
// {  
// System.err.println("IOException: " + ioe.getMessage());  
// }  
 System.*out*.println("All Children Done: " + numChild);  
 //}  
 numChild++;  
 }  
 }  
}  
  
class Child implements Runnable {  
 private double begin;  
 private double end;  
 double memAttr;  
 double global;  
  
 public Child(double begin, double end) {  
 this.begin = begin;  
 this.end = end;  
 }  
  
 @Override  
 public void run() {  
 // Print the current CPU number  
 //System.out.println("CPU=" + Thread.currentThread().getName());  
  
 // Get the current time  
  
 String filename= "Time.txt";  
 File timeFile = new File(filename);  
 if (timeFile.exists()){  
 timeFile.delete();  
 }  
  
 double totalSum = 0.0;  
 for (int local = (int)begin; local < end; local++) {  
 double root = Math.*sqrt*(local);  
// try  
// {  
//  
// FileWriter fw = new FileWriter(filename,true); //the true will append the new data  
// fw.write((local + ":" + root) + "\n");//appends the string to the file  
// fw.close();  
// }  
// catch(IOException ioe)  
// {  
// System.err.println("IOException: " + ioe.getMessage());  
// }  
 // revised lines to do prints:  
 //System.out.println(local + ":" + root);  
 //if (local%5==0) System.out.println();  
 totalSum += root;  
 }  
  
 // Are class attributes and globals shared?  
 System.*out*.println(" totalSum=" + totalSum + " global=" + ++global + " memAttr=" + ++memAttr);  
 // Calculate execution time in ms and print  
  
  
 }  
}

1. Factor Program

import java.io.\*;  
import java.lang.Process;  
import java.lang.ProcessBuilder;  
import java.lang.ProcessBuilder.Redirect;  
import java.util.\*;  
  
public class Factor  
{  
 public static void main(String[] args) throws IOException, InterruptedException  
 {  
 int numChild = 1;  
 int total = 600000;  
 // Spawn children processes  
 for (int i = 0; i < numChild; i++)  
 {  
 ProcessBuilder pb = new ProcessBuilder("java", "FactorChild", String.valueOf(i));  
 pb.redirectOutput(Redirect.INHERIT);  
 pb.redirectError(Redirect.INHERIT);  
 Process process = pb.start();  
 }  
 // Wait for children to finish  
 for (int i = 0; i < numChild; i++)  
 {  
 Process process = Runtime.getRuntime().exec("wait");  
 process.waitFor();  
 }  
 System.out.println("All Children Done: " + numChild);  
 }  
}  
  
class importFactorChild  
{  
 public static void main(String[] args)  
 {  
 int begin = 0;  
 int end = 0;  
 int id = Integer.parseInt(args[0]);  
 int range = 600000 / 1;  
 begin = id \* range;  
 end = begin + range;  
 int start\_s = (int) System.currentTimeMillis();  
 int val, i;  
 for (val = begin; val < end; val++)  
 {  
 for (i = 2; i <= val / 2; i++)  
 {  
 if (val % i == 0) break;  
 }  
 if (i > val / 2)  
 {  
 System.out.println("F:" + val);  
 }  
 }  
 int stop\_s = (int) System.currentTimeMillis();  
 System.err.println("time: " + (stop\_s - start\_s));  
 System.exit(0);  
 }  
}