|  |  |
| --- | --- |
| **[80 pts] GDPARCM Problem Set: Prime Number Verifier**  To be accomplished in pairs. | Instructor: Neil Del Gallego  neil.delgallego@dlsu.edu.ph |

**GENERAL PROGRAMMING INSTRUCTIONS**

You are to create a multithreaded program that asks for a very large number and checks whether it is a prime number or not. The number, **2,147,483,647,** is the highest prime number that fits on a 32-bit integer data type. You may use this number as test case for running the experiment.

A written technical report is to be submitted to the instructor on the specified deadline.

**Restrictions**

* Only C++ and its threading support library should be used in this problem set. No other libraries or game engines should be used.

**Specific Instructions**

* Your experiment should at least check all the possible divisors applicable for the test case, **2,147,483,647.** If results are not yet observable, consider increasing the value to **2,305,843,009,213,693,951**, which is the next prime number (but requires a long data type).
* Number of Threads **-** It is up for the students to specify explicitly in the report, the number of threads they used for the experiment. Students may choose to increase the threads for comparison in the following manner: **1,2,4,8,16,32,64,128 (threads increase by 2i).**
* Consider running your experiment in multiple trials. Then get the mean/standard deviation of processing time.
* The goal of this experiment is for you to discover an observable performance increase of a multi-threaded application, over a single-threaded implementation.

**Some Tips on Creating Threads**

* If your physical environment for testing is too fast, consider adding **TIME DELAYS** (i.e. adding a sleep() function periodically to pause your thread execution). Make sure you specify this in your report.

**TECHNICAL REPORTING INSTRUCTIONS**

Written reports are to be printed using letter-size paper with font style, Arial 10. You may choose to write your report in your own format, provided that the following guide questions and important points are answered in your report:

* What physical environment did you use? Processor and speed, memory, windows/mobile, etc.
* How did you partition the checking of divisors?
* Did you use any time delays?
* What is your experiment setup? What are the number of threads you used for comparison? Did you consider multiple trials? How many?
* Discuss your program implementation. **DO NOT INCLUDE YOUR WHOLE CODE** here in this discussion. Present implementation and discussion for: your prime number tests (did you use the INT-32 prime number or the LONG-64 prime number?), partitioning the divisors and assigning it to threads. Present pseudocode for your threading implementation.
* Present your results in visualized form and provide observations. You may use time in seconds as your measure for performance.
* Discuss your conclusion. What did you observe from increasing/decreasing the number of threads? In your experiment, what’s the recommended number of threads applicable for the said program? Support this answer by your results.
* Refer to the appendix for a sample report.

**Grading Scheme**

This activity is worth 80 points with the rubric seen below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation Correctness** | | | |
| 0 – 9 points | 10 – 15 points | 16 – 20 points |  |
| The report hints an incorrect implementation. Performance boost is not observed on 50% or more on the test cases. | The report partially shows correctness in implementation. The performance boost is only observed on 30 – 40% of the test cases. | The report shows evidence that the multi-threaded implementation is correct and offers an increase in performance. |  |
| **Evidence Presentation** | | | |
| 0 – 9 points | 10 – 15 points | 16 – 20 points |  |
| The report do not present evidences through figures/tables, or the figures/tables are not easy to understand. | The report presents figures/tables that somewhat shows evidences of the results. The figures/tables can be more detailed or presented better. | The report presents figures/tables that shows clear evidences of the results obtained from the experiments. |  |
| **Correctness of Experiment and Testing** | | | |
| 0 – 9 points | 10 – 15 points | 16 – 20 points |  |
| The report fails to have information related to experiment and testing. | The report does not have enough information regarding the experiment and testing scheme performed. It is difficult to assess how the results were obtained. | The report discusses the experiment and testing scheme clearly. |  |
| **Thoroughness of Testing** | | | |
| 0 – 9 points | 10 – 15 points | 16 – 20 points |  |
| The report exhibits evidence that testing was not performed properly or was rushed. | The report presents evidence of basic testing with limited columns/bars seen in figures/tables. The testing does not completely prove that a performance boost was achieved. | The report presents evidence that a rigorous testing was performed, such as varying number of threads, varying test devices, different implementations and comparing them, etc. |  |

**Submission Instructions**

* Submit the technical report in PDF.
* A 1 - 2 min video demo showing the sample program in MP4.
* Submit the contribution list document.

**APPENDIX A: Sample Report**

Neil Patrick Del Gallego

XXXXXX – id number

**OPERSYS Multithreading Report #1: A Sample**

**Introduction**

This report contains the observations from creating a simple multi-threaded application running on Android devices. The application involves searching for a number in a large shuffled list. The application also asks the user the number of threads required to perform the task [[1]](#footnote-1).

**Device Specifications**

All data in this report are gathered from THL T100, an Android device with the following specifications which can affect the performance of the application:

* Operating System: Android 4.2.2 (JellyBean)
* CPU : 1.7 Ghz Octa-Core MediaTek Processor Cortex-A7
* GPU : Mali450 – MP4 700 Mhz
* RAM : 2GB
* Display : 5.0" Full HD IPS OGS 441ppi

**Multithreaded Number Search**

Multithreaded number search is an application that searches for a specified integer in a large shuffled collection. The application has an option to generate a shuffled collection with a data size of *1,999,999* indexes. Given *MAX\_ARRAY\_SIZE* as *1,999,999*, the application counts from 0 up to *MAX\_ARRAY\_SIZE,* stores them into the list,and shuffles them using Java's *shuffle*() method found in Collections package.

The application requires the user to enter the number to search and the number of threads required to perform the task. By default, the number to search can be zero and the number of threads is set to 1. Once done, results are shown to the user and the total running time it took for the application to finish traversing the large collection is displayed.

**Time Delays**

Initial findings on the multithreaded number search application are not significant and the time it took the threads to finish were fast and not meaningful to provide comparisons. To provide more meaningful and observable data, the application was coded with **console logging enabled**. This provided significant delay but still bearable for the application to finish at a reasonable amount of time.

**Implementation**

Given the specified amount of threads preferred by the user, threads receive an even amount of partition to search such that no thread checks the same index of the large collection. Two classes are used to divide the tasks, the **SearchThreadInitiator** and the **SearchThread** class.

Class SearchThreadInitiator

public SearchThreadInitiator(numberToSearch, numberOfThreads) {

this.numberToSearch = numberToSearch;

this.numberOfThreads = numberOfThreads;

}

public void beginThreads()

{

//pseudocode here

}

End Class

The **SearchThreadInitiator** class manages and handles the creation and execution of threads. When *beginThreads()* method is called, the start time is logged which denotes the beginning time that the search task is started. Based from the number of threads specified by the user, the next code block partitions the indexes evenly among the search threads. The search threads are immediately started upon creation. The *onNotify()*method is fired once a number was found by any of the search threads. The result is reported back to the UI. If all threads are finished running and the number is not found in the list, *onNotify()* method is also fired.

Class SearchThread

public SearchThread(threadId, numberToSearch, minPartition, maxPartition) {

this.threadId = threadId;

this.numberToSearch = numberToSearch;

this.minPartition = minPartition;

this.maxPartition = maxPartition;

}

public void run() {

//logic for number searching here

}

End Class

**SearchThread** class implements a simple way to check if the number exists in the given partition. If the number in the current index being checked is equal to the number, then it means that the number is found.

**Results**

Results show two test cases, one is to search for the number 1 and another is to search for a number out of range. The number of threads are increased exponentially, starting from 1, then 2, then 4, then up to 128 threads running all at the same time.

Searching for number 1, the results is shown in the graph below.

Figure 1: Total time it took to find a number based on number of threads

As seen in Figure 1, 8 threads seems to be the most optimal solution as it only took a total amount of time of about *34.937 seconds* to finish the amount of work. 16 threads and onwards resulted in higher time. However, the list is shuffled so there are cases where the number of threads may not matter due to the manner in which the partitions are made. If for example, number 1 is positioned nearer towards a lower partition of thread #5, then thread #5 will immediately report the result back to the UI, resulting in a much faster turnaround time. Though the philosophy of this problem is that the more threads running, then there are more work being done. Starting from 16 threads onwards, there is already a noticeable latency in terms of performance and observed that the CPU utilization of the device is around 70% or higher. Running the application continuously resulted in the device heating up.

Second test was to intentionally search for a number outside of the range of the collection. This is to force all threads to traverse everything in their assigned boundaries. Ideally, having a smaller partition will yield a faster turnaround time for each thread.

Figure 2: Total time it took to search for a number outside of range

As seen from Figure 2, the time is significantly reduced starting from 2 threads. However, there is no longer a noticeable difference starting from 8 threads up to 128 threads. Therefore, it can be concluded that exceeding to more than 8 threads resulted into a degraded performance and unoptimized CPU utilization. This result may vary on other devices especially on how many physical processors the device has.

**Conclusion**

Based from the problems and solutions presented, multithreading significantly improves the performance of the application and results in better CPU utilization. However, increasing threads too much may no longer yield better results and may cause the application to actually run slower than intended.

Based from the two results and considering the device specifications, the optimal number of threads to run on the device is around 8 to 11 since the scale factor and the amount of physical processors (which is 8) should also be considered. CPU utilization was around 70% to 80% during testing which is better compared to running the task on one thread only which resulted in around 40% to 50% CPU utilization.

|  |  |
| --- | --- |
| **GDPARCM Contribution List** | Instructor: Neil Patrick Del Gallego  neil.delgallego@dlsu.edu.ph |

**General Instructions:** This form serves as a formal indication of the member’s contribution. Unless the student is excused, a non-contributing student may incur deduction penalties from their score.

**Format and Submission Instructions**

Please follow this format with the same font style of this document. Write down contributions as specific as possible.

**GDPARCM <Section> - Peer Evaluation Form**

|  |  |
| --- | --- |
| Did both of you contribute almost equal amount of work? Check the box that applies. | |
| **YES\*** | **NO** |

***\*****If YES, you may leave the contribution list unanswered.*

|  |  |
| --- | --- |
| **Student Name** | **Contributions** |
| <Name 1> |  |
| <Name 2> |  |

We hereby certify that the information provided above is true and correct, and in good faith.

|  |  |
| --- | --- |
| <Signature over Printed Name> | <Signature over Printed Name> |

Submit this as PDF.

1. Multithreaded Application Download Link: <https://dl.testfairy.com/download/6GW32D9H5MR2TG5NYFYG29CZ1G6PEYSB9F1XRJ43J822W/SampleThreading_0601-testfairy.apk>

   Or email [neil.delgallego@aninogames.com](mailto:neil.delgallego@aninogames.com) for a direct copy of the APK file. Android device minimum specs should be 1GHz dual-core processor, 512MB RAM and Android 2.3 or higher. [↑](#footnote-ref-1)