420-326 – Operating Systems
Winter 2019

Worth 4%

Assignment #2:
PowerShell (Parallel Processing)

Notice

This is version #2 of this document. Be aware that revisions may be made to add details and/or requirements. If a revision is made, it will be posted to Lea. No revision will be made later than one week before the due date. You are responsible for meeting the latest requirement.

This document has a ton of detail. Read it from top-to-bottom and then re-read at least twice to make sure you don't miss anything.

Preamble

With the fast Intel CPUs usually supporting at least 8 cores, a program written without support for parallelism (multi-tasking) will use less than 15% of the overall processor capacity at any time. This means no matter how well your write an application, unless you write in support for multi-tasking, your application's performance will be nowhere near meeting its potential.

With CPU-bound applications, sometimes the easiest performance fix is to refactor for multi-threading support (i.e. rewrite a serial program so that it's parallel).

A good parallel re-write of a serial program isn't found by optimizing each of its steps. Instead, the best parallelization often involves a new algorithm.

Requirements

For this assignment, we'll be re-writing the following pseudo-code:

```
rows[]=...
sum = 0;
primes[] = ();
for (i = 0; i < n; i++) {
    if(isPrime($rows[$i])){
        primes[]=$rows[$i];
    }
}

foreach(prime in primes){
   sum+=prime
}
print $sum
```

We are going to re-write the pseudocode to support *symmetric multi-processing*. To parallelize, assume that we have m threads and m is smaller than n. Each core could be assigned a subset of numbers to sum via a *thread* (or *runspace* in Powershell-speak).

You are required to use runspaces to code the assignment. The use of a RunspacePool is optional. Microsoft has a four-part tutorial here: https://blogs.technet.microsoft.com/heyscriptingguy/2015/11/26/beginning-use-of-powershell-runspaces-part-1/

To keep things simple, we'll use private variables for each thread (instead of *shared memory*). This will allow each thread to execute its block of code independently of the other threads.

For example, if there are eight threads, n = 24, and each call to `isPrime()` returns true, then given the list:

3, 5, 3, 2, 11, 13, 5, 3, 3, 7, 7, 3, 11, 13, 17, 3, 2, 5, 17, 3, 2, 5, 13, 3

Then the values stored in each threads subtotal might be:

Core	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Subtotal	8	17	27	8	24	31	35	9

When each subset was done being calculated, the "master" thread can simply sum the subtotals by retrieving the results from each thread:

```
if (I'm the master core) {
      sum = 0;
      for each thread other than myself {
            receive value from thread; sum += value;
      }
}
```

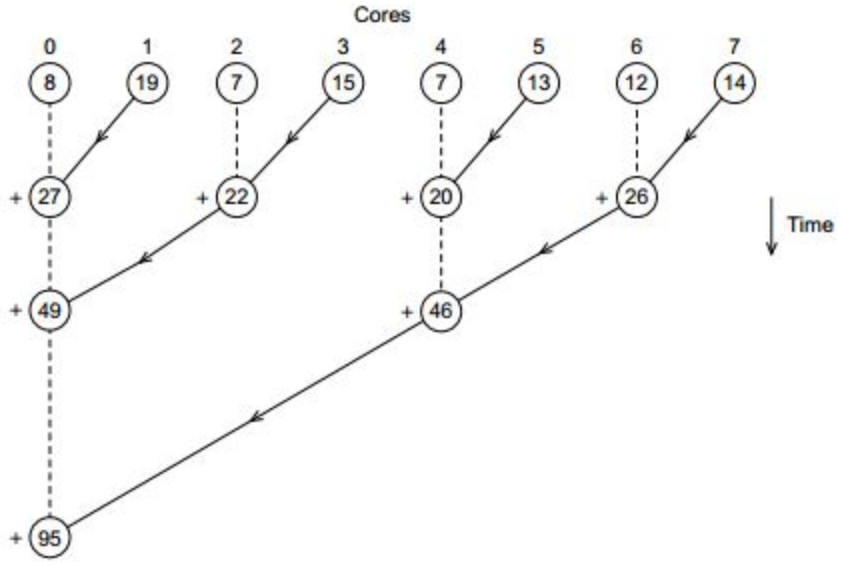
In our example, if the master core is core 0, it would add the values 8 + 19 + 27 + 8 + 24 + 31 + 35 + 9 = 161.

NOTE: You need to write your own `isPrime()` function. Remember that 1 is not a prime number, but 2 is! Wikipedia has decent pseudocode for a simple implementation here.

Bonus (50% of assignment)

(A perfect score on the assignment plus the bonus will cause the bonus to be applied against lost marks on tests, etc.)

There is a better way to do this — especially if the number of threads is large. Instead of making one thread do all the work of computing the final sum, we can pair threads so that thread 0 adds the result of thread 1, thread 2 can add in the result of thread 3, thread 4 can add in the result of thread 5 and so on. Then we can repeat the process with only the even-numbered threads: 0 adds in the result of 2, 4 adds in the result of 6, and so on. Now threads divisible by 4 repeat the process, and so on (see diagram below). The circles contain the current value of each thread's subtotal, and the lines show thread's sending their sums to another thread.



For both solutions (regular assignment vs. bonus) the master thread (0) does more work than the other threads, so the length of time it takes the program to find the final sum should be roughly equal to the length of time it takes for the master thread to complete. However, with eight threads, the master will carry out seven additions using the first method, while with the (bonus) method it will only carry out three. The difference becomes much more significant with a large numbers of threads. With 1000 threads, the first method will require 999 additions, while the second will only require 10, an improvement of almost a factor of 100!

To accomplish the bonus, you will probably need to mix runspaces and PS-Jobs. No further guidance will be provided for the bonus. The spec is being left purposefully vague to encourage creativity and independence.

Other Thoughts about Parallelism

There are a number of ways to solve this problem, but all of them depend on the basic idea of partitioning the work so that it can be done in parallel.

There are two widely used approaches: *task-parallelism* (asymmetric multiprocessing) and *data-parallelism* (symmetric multiprocessing). In task-parallelism, we partition the various tasks carried out in solving the problem among the threads. With data-parallelism, we partition the data used in solving the problem among the threads, and each thread carries out more or less similar operations on its part of the data. **For the purposes of this assignment we've chosen to use data-parallelism.**

The illustration we discussed was our math department using common exams for each course (so that all Cal I students write the same final exam for example). Suppose that a course has one hundred students across all sections with four teachers in total. After the final exam (containing 20 questions) is over, the teachers divide the corrections. They have the following two options: each of them can grade all one hundred responses to five of the questions; say T1 grades questions 1-5, T2 grades questions 6-10, and so on. Alternatively, they can divide the one hundred exams into four piles of 25 exams each, and each of them can grade all the exams in one of the piles; T1 grades the papers in the first pile, T2 grades the papers in the second pile, and so on.

In both approaches the "threads" are the teachers. The first approach is an example of task-parallelism. There are 20 tasks to be carried out: grading the first question, grading the second question, and so on. Presumably, the graders will be looking for different information in question 1 than question 2 and so on. So the teachers will be "executing different instructions" (independent tasks).

The 2nd approach is to use data-parallelism. The "data" are the exams, which are divided among the threads, and each thread performs the same work for each paper.

The assignment divides the data and each threads carries out the same operations on its assigned elements: finding the sum of the prime numbers.

When threads can work independently, writing a parallel program is similar to writing a serial programs. Things get more complex when the threads need to coordinate their work. In the bonus portion of the assignment, although the tree structure in the diagram is very easy to understand, writing the actual code is relatively complex. Unfortunately, it's much more common for the threads to require coordination!

The assignment also involves some degree of *load balancing*: it's clear that we want the threads all to be assigned roughly the same number of values to compute. If, for example, one core has to compute most of the values, then the other threads will finish much sooner than the heavily loaded core, and their computational power will be wasted.

Input

Your program will read the asg2-200000.txt text file included with the assignment. The text file contains 20,000 random numbers between 1 and 2000. You should define a constant in your script pointing to this file (which should be in the same directory) - you *MUST* use the `$PWD` environment variable as part of the filename!

Your program should take two parameters (using the **`Param`** keyword and the two names below):

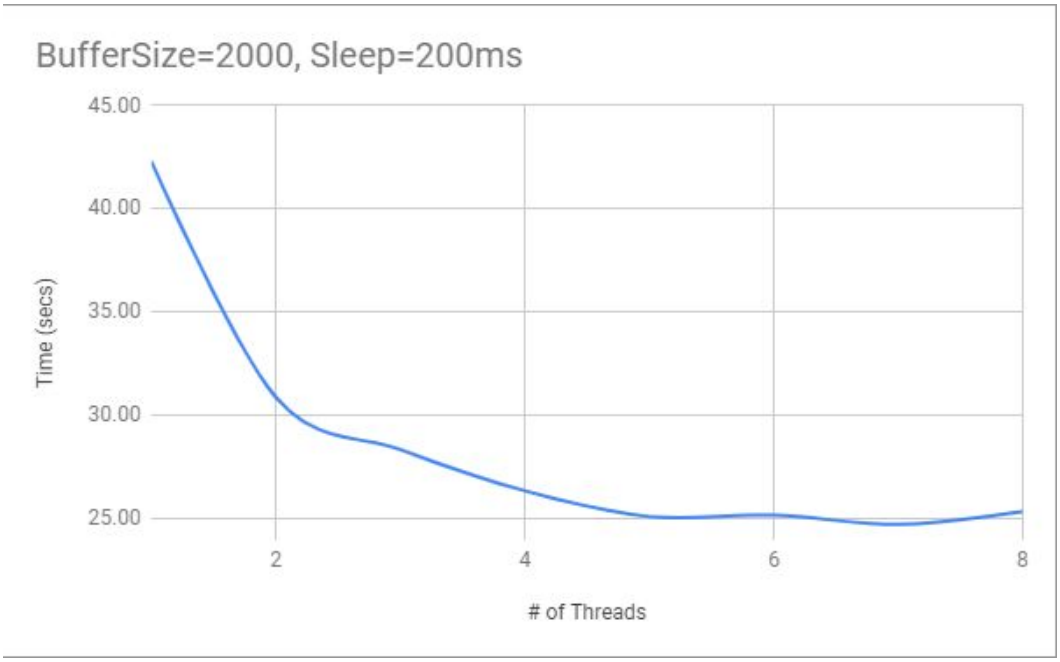
```
PS C:\Users\Jim> .\Desktop\ass2.ps1 -bufferSize 100
-numThreads 8
```

Output

- Your script must output the final sum of the prime numbers in the file (3,578,424,242) on its last line of output. Any other output prior to this will be ignored. It's a fantastic idea to output debugging information throughout your script.

- Your script should run in under 2 minutes using 1 thread and under 30 seconds once you are using several (4+) threads. These are rough estimates to guide you so that you know immediately if you've done something inefficient.

- On an older machine, I coded the assignment (without any serious optimization) and saw results like the following using a 2000 digit buffer:



(Visually you should be able to guess that I wrote the assignment on a computer with 8 cores).

A Note about Coding Using Larger Datasets

You've been given a test file with 200,000 rows and been told that a decent runtime is 30 seconds. If you immediately start coding against the test file, you are going to spend A LOT of time waiting for your program to complete. Be smart, create a separate ~200 line file (using the top 200 lines of my file) and write your code to use that until you are sure your algorithm works! You should consider using Excel or Google Sheets to validate your answer (with the smaller dataset).

Important details

- If your script uses filenames (to load or write files), you must use the `$PWD` environment variable. The penalty for failing to do so will be harsh! (10%)
- You should test your script under PowerShell 5 (run `$PSVersionTable.PSVersion.Major` to check). Our computer labs run PowerShell 5. If you have an older version installed at home, simply update it (PowerShell is a free download: https://www.microsoft.com/en-us/download/details.aspx?id=50395).
    - Using an older version is going to make your life much harder; there are numerous features in PowerShell 5 that aren't in early versions.
- Use Visual Studio Code with the PowerShell extension, you need a debugger!
    - When using the debugger, make sure the debug mode is set to "PowerShell Launch Current File in Temporary Console", otherwise your **`Set-Variable`** constant declarations will fail between runs (you did use constants, right?!)
- Your script must use "**`set-strictmode -version latest`**" right at the top of the script.
- We may build off of this assignment in the future, it is important that you complete it so as to not jeopardize your future grades.
- While programming style is not the primary focus of this class, you are expected to produce quality code. Please use common sense (i.e. functions where appropriate, liberal use of whitespace, consistent indentation, comments where needed, but not excessive, etc.). Grades will be deducted for inconsistency and/or incompetence re: coding style. See handout re: coding style.
- Be aware of bugs caused by having your code sleep for long periods of time!

Submission

A zip file containing:

- Your .ps1 script file
- A screenshot of Process Explorer (https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx) showing your script running with 4 active threads (note that depending how you code your script, 4 threads may be with the parameter `numThreads=3 or 4`).
- You will run (and time) your script using 1-16 threads with a **2000** buffer size (2000 digits passed to each thread). You will do this 5 times. The goal is to capture the average (median) run time for each configuration (eliminate any outlier results). You will create an Excel document with your results and include a graph exactly like the one in the "Output" section of this document.
    - NOTE: If the shape of your graph doesn't look like mine, your code is wrong (up to a point, an i7 CPU will have a different curve than an i3). It's important that you prove better performance by adding threads (up to a point).
    - NOTE #2: If you are using a single core processor at home, you should still see performance improvements using two threads thanks to Hyper-Threading. Please time your script at school though on one of our i7-based PCs.
- In your spreadsheet, calculate an approximation of the serial % of your program using Amdahl's law based on a measurement of the execution times of the serial and parallel portions of your code using one thread (which we'll cover in class). Once you have an approximation, calculate the theoretical maximum speedup for each additional thread and contrast it against what you saw (note that once you exceed the number of cores on your machine, Amdahl's Law is useless)

You can capture your results using the following PowerShell `for` loop:

```
foreach($x in 1..16){
echo $x;
   (Measure-Command { .\Desktop\ass2.ps1 $x 8192
   }).TotalSeconds; (Measure-Command { .\Desktop\ass2.ps1 $x
   8192 }).TotalSeconds; (Measure-Command { .\Desktop\ass2.ps1
   $x 8192 }).TotalSeconds;
}
```

Grading Rubric

- To earn 100% on this assignment your code must support parallel processing up to 16 processes.
    - Performance must increase as threads are added (until # threads == # cores)

A note about strategy:

- Students should focus on the single process approach first. Once this returns the correct result, refactor for multi-process.
- Don't attempt the bonus portion without first completing the non-bonus portion!

% of Grade	Description
30%	Demonstrated mastery of introductory PowerShell (conditionals, loops, data structures). Bug-free code, logical approach
20%	Correct result
20%	Code quality: KISS principle respected, commenting where appropriate, proper and consistent formatting (indenting, constants, etc.), **code re-use**.
30%	Complete submission (including results spreadsheet, screenshots, Amdahl's law)

NOTES:

1. Code that does not work will clearly violate several of the above categories (e.g. broken code is clearly not of good quality, probably does not return the correct result, etc).
2. Code that requires more than three minutes to run on a lab PC will be considered broken.

Late Penalty

10% per day to a maximum of three days late. If you have a legitimate reason, ask for an extension (a few days in advance) rather than losing grades for nothing.

Resources

Runspaces:
https://blogs.technet.microsoft.com/heyscriptingguy/2015/11/26/beginning-use-of-powershell-runspaces-part-1/

Runspaces & pools: https://blog.netnerds.net/2016/12/runspaces-simplified/

Hints

- The amount of time you sleep between checking running threads has a huge performance impact.
- Try to do as much work as possible in parallel mode and as little as possible in serial mode.