EECE 5640 Homework 2

Emily Costa (costa.em@northeastern.edu)

October 11, 2021

Question 1.

In this section, I implement a solution to the Dining Philosopher's problem given by Edsger W. Dijkstra. Dijkstra was a physicist based in the Netherlands who made great contribution to the computer science field. He was an early pioneer of distributed computing and developed several well-known algorithms such as the shortest path-algorithm, or Dijkstra's Algorithm [2]. Dijkstra's Algorithm determines the shortest path between two nodes, u and w, by taking advantage of the fact that the shortest paths to an intermediary node, v, from u and w gives the shortest path between the two original nodes. During the algorithm, each visited node is kept track of and potentially modifies a value in the array tracking visited node and shortest distances [1]. This concept is similar to the Dining Philosopher problem in which each fork and philosopher needs to be tracked. This concept has several real world application, most interestingly in determining the quickest route when using a GPS.

To run one of my implementations, simply run the provided bash files (e.g. 'source run_q1.sh'). After my original implementation, I explored three other implementations.

a. **Does the number of philosophers impact your solution in any way?** Not really because my original solution allowed for the user to input the number of philosophers. If I had not done that originally, then a level of abstraction would be necessary to run the program. However, it would be a combination of simple changes. When the number of philosophers is increased, there seems to be more randomness at play as the forks are exchanged more often.

   How about if an even number of forks are used? To do this, I needed to modify my original implementation. This new implementation is shown by running 'source run_q1-a.sh'. I simply added an additional fork at the end and had the last philosopher use that last spoon. When this runs, I noticed that the odd numbered philosophers tend to be able to eat more often. This is likely because the first and last (both odd-numbered) have access to one of their own forks. This effect cascades to the other odd-numbered philosophers.

b. **What happens to your solution if we give one philosopher higher priority over the rest?** To do this, I would need to set the priority attribute of a thread before creating it. This can be done using functions from the 'sched.h' header file. Once an attribute is created, it can be fed into the 'pthread_create' function to give that particular thread a higher priority. In this case, the higher priority philosopher will likely always be eating as he is immediately taking back the forks when he puts them down and will not permit those around him to eat. That effect will cascade to other philosophers as they try to grab forks.

c. **What happens to your solution if the time to eat for every philosopher is the same?** To modify my implementation, I simply put the thread to sleep for a set number of seconds then had it put down its forks instead of using a random number of seconds to sleep. This implemenation can be run using 'source run_q1-c.sh'. This cause the switches to happen at about the same time and therefore the print statements were coming out all at once within the set time I allowed the philosophers to eat. The behavior seemed to be similar except maybe that more philosophers were able to eat at once more frequently. I suspect this is due to the clean slate every couple seconds allowing for more philosophers to participate in the feast.

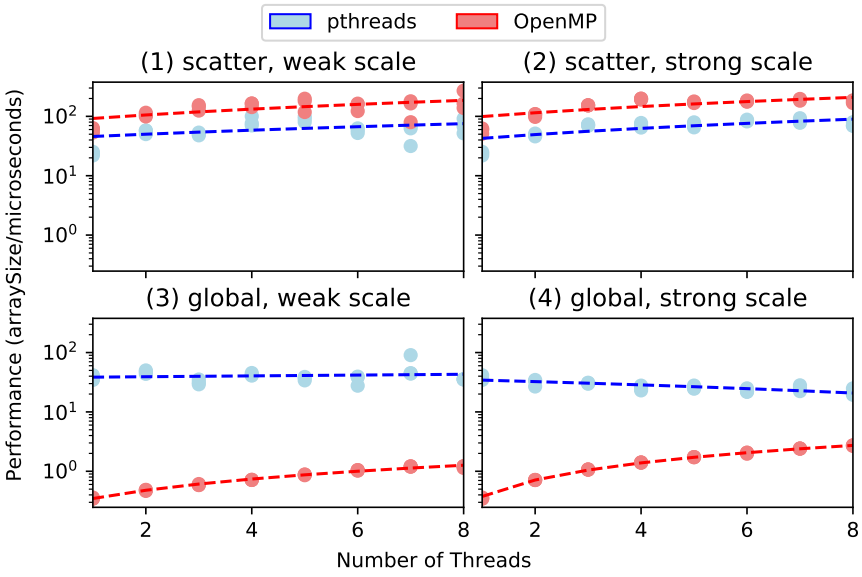

Figure 1: Benchmarks of parallelized versions of Sieve of Eratosthenes.

Question 2.

In the section, I implement an algorithm, Sieve of Eratosthenes, to find the prime numbers in an array of size N. The programs were compiled and run using the Ubuntu 20.04 OS on my personal computer which has a quad-core Intel® Core™ i7-8550U CPU @ 1.80GHz and 16 GB RAM. I had two ideas for implementing parallel versions of the program so I decided to do both and compare them using weak/strong scaling and pthreads/OpenMP. My two implementations of the parallelized algorithm were as follows:

1. **Global variable ("global").** In this initial method, I held the array showing primes at the global level such that multiple threads can access that array to change a value to false. In the algorithm itself, the parallelized part is that each thread grabs a "prime number" to process through the loop and identify non-primes. The major downside is that a mutex lock is required to prevent race conditions when modifying the array containing primes. The mutex lock is a potential congestion point that significantly reduces the benefit of using threads as it is critical.

2. **Scattered data ("scatter").** To avoid using mutex locks and creating congestion, I simply redid my algorithm to process the array of size N in chunks equal to the number of threads. Each chunk processed by a thread is size N/numberOfThreads. This reduces the workload of a thread, though the downside is that the workload spread among threads is not even as the chunks on the higher end of the prime array require more work be done to determine which numbers are primes.

To make a full comparison, I ran each unique combination of parallel framework (pthreads/OpenMP), scaling (weak/strong), and parallization technique a total of 3 times. In Fig. 1, I demonstrate the performance of my programs by the number of threads used. Performance is measured by number of elements, N, processed per microsecond such that all primes in [1...N] (or [1...N*numberOfThreads] for strong scaling) are identified. So, the higher the value on the plot's y-axis , the better that run performance is as it took less time to complete the function on a larger array. We make the following observations:

1. **OpenMP outperforms pthreads in the scatter implementation.** This is because a benefit

of managing threads at a lower level, such as with pthreads, is that you can control how and when the threads access different memory which is not needed in this implementation as each thread uses data independent of other threads. OpenMP just automates and controls the scattering of data to each thread better than pthreads which in turn helps the threads complete their given tasks faster. However, both methods performed well in this implementation.

2. **Pthreads outperforms OpenMP in the global implementation.** Though OpenMP slightly outperforms pthreads in the scatter implementation, pthreads by far performs better than OpenMP in the global implementation. This is because pthreads does a better job of managing the flow of threads accessing and modifying the global array.

3. **During strong scaling on global implementation, pthreads worsens in performance.** This is because pthreads has a higher overhead from the bottleneck when more threads trying to access the global array. As only one thread can modify the array at a time, the efficiency is not improved when the threads are already waiting for the mutex to be unlocked.

4. **OpenMP scales better than pthreads in the global implementation.** Though pthreads far outperforms OpenMP, OpenMP scales better than pthreads, especially during strong scaling. As previously mentioned, performance gains are hindered as the global array is locked by a thread and the other threads are required to wait. However, OpenMP is better at automatically managing and making use of additional threads.

5. **Scattering the data gave better performance than allowing for global access.** Both parallel frameworks performed better and scaled when the scatter implementation was used. This is because the threads did not need to manage access to memory as each worker completed its own task. This avoided racing conditions and resource congestion when memory was temporarily blocked to some threads.

Question 3.

In this section, I discuss the paper by Castelló et al.that considers lightweight threads versus sticking with operating system threads.

a. The main tradeoff of using OpenMP alone versus adding the capabilities of light-weight threading libraries efficiency in massive parallel environments. OpenMP's expensive context swicthing and synchronization mechanisms make it difficult to leverage massive parallelism. Lightweight threads, such as user-level threads, offer more efficient context switching and synchronization operations.

b. I selected one of the lightweight threading libraries, Go, discussed and provide an example of how I would use it to parallelize a simple vector addition kernel (adding two vectors of single-precision floating point numbers together). To use goroutines in threading, the function needs to have time allocated to it or else it will immediately be returned. The following would be my approach where I use goroutines to create the LWTs and Go syntax:

```
package main

import{
    "fmt"
    "time
}

var N int = 100
var vector0 [N]float32
```

```
var vector1 [N]float32
var output [N]float32

func add(n int){
    time.Sleep(time.Second)
    output[i] = vector0[i] + vector1[i]
}

func main(){
    for i:=0; i<N; i++{
        go add(i)
    }
}
```

c. This paper evaluates the performance of BLAS-1 functions. The BLAS-1 subprograms include scalar and vector operations such as the Sscal function. To achieve task-level parallelism in these subprograms, the elements in the vectors are divided amongst threads.

d. This paper considers the parallelization of nested parallel structures and nested tasks. These parallel patterns, such as the nested for loop, are common in high performance applications because the master thread can easily leverage the structure to divide the iteration space among several threads. This will give spawned threads clear work with a function pointer to be executed. When a specifically nested structures are used, the threads spawned on an outer iteration can be used to divide the inner loop iterations.

4. References

[1] algodaily. *Dijkstra's algorithm.* URL: https://algodaily.com/lessons/an-illustrated-guide-to-dijkstras-algorithm.

[2] IEEE. *Edsger Dijkstra.* URL: https://www.computer.org/profiles/edsger-dijkstra.