MONASH University

Lab Session Week 3

Student ID	Student Name	Student Email
32844700	Teh Jia Xuan	Jteh0015@student.monash.edu

Task 1 - Code

Task1.c > main()

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>

bool isprime(int number);
int main ()
{
    //declaration and initialisation
    int *arr = (int *)malloc(1000000 * sizeof(int));
    struct timespec start, end, startComp, endComp;
    double time_taken;
    int n;
    int counter = 0;

    // Get current clock time.
    printf("Compute:\n");
    clock_gettime(CLOCK_MONOTONIC, &start);

    //loop through the numbers to check prime
    for (int i = 0; i < 1000000 ; i++){
        if (isprime(i)){
            //update the array if there is prime
            counter++;
            arr[counter] = i;
        }
    }
    //print out the numbers in array
    for(int i = 0; i <= counter; i++){
        printf("%d\n", arr[i]);
    }
    //get time to calculate how long the algo runs
    clock_gettime(CLOCK_MONOTONIC, &end);
    //minus 2 times as second and nano second stored differently
    time_taken = (end.tv_sec - start.tv_sec) * 1e9;
    time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
    printf("Overall time: %f sec \n", time_taken);
}
```

```
bool isprime(int number){

    if (number <= 1){
        return false;
    }

    if (number == 2){
        return true;
    }

    if (number % 2 == 0){
        return false;
    }

    //calculate prime number return false if it can be divided by a number before it
    for (int i = 3; i <= sqrt(number); i++ ){
        if (number % i == 0){
            return false;
        }
    }
    return true;

}
```

```
Overall time: 0.533003 sec

real    0m0.536s
user    0m0.212s
sys     0m0.323s
```

Serial code took 0.533 seconds to run

MONASH University

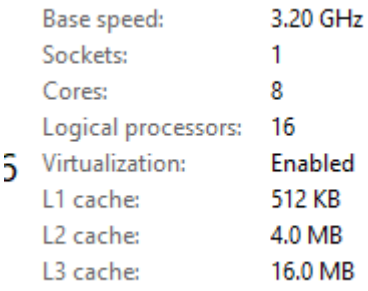
Task 2 - Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#define THREADS 8

bool isprime(int number);

//define global variable
size_t num_elem = 1000000;
int *arr;
int counter = 0;
pthread_mutex_t mutex;

//call prime number function
void *calprime (void *arg){

    //indicate start and end
    int start = *((int*)arg);
    int end = *((int*)arg + 1);

    //looping through the numbers to check prime
    for (int i = start; i < end ; i++){
        if (isprime(i)){

            //if is prime then lock the threads so that one thread can update each time
            //to prevent race condition
            pthread_mutex_lock(&mutex);
            arr[counter] = i;
            counter++;
            pthread_mutex_unlock(&mutex);
            //unlock after done updating
        }
    }
}
int main ()
{
    //declaration and initialisation
    pthread_mutex_init(&mutex, NULL);
    struct timespec start, end, startComp, endComp;
    double time_taken;
    pthread_t threads[THREADS];
    int thread_args[THREADS][2];
    pthread_mutex_t mutex;
    arr = (int *)malloc(num_elem * sizeof(int));

    //get step for each threads
    int step = 1000000 / THREADS;

     // Get current clock time.
    printf("Compute:\n");
    clock_gettime(CLOCK_MONOTONIC, &start);

    for(int i = 0; i < THREADS; i++){
        //set each thread for its start and end in a 2d array
        thread_args[i][0] = i * step;
        if (i == THREADS - 1){
            thread_args[i][1] = 1000000;
        }
        else{
            thread_args[i][1] = (i + 1) * step;
        }
        //create the threads using i
        pthread_create(&threads[i], NULL, calprime, &thread_args[i]);
    }

    //join the threads together
    for (int i = 0; i < THREADS; i++){
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    //print out the updated array-> all the prime number
    for(int i = 0; i < counter; i++){
        printf("%d\n",arr[i]);
    }
```

```
76      pthread_mutex_destroy(&mutex);
77
78      //print out the updated array-> all the prime number
79      for(int i = 0; i < counter; i++){
80          printf("%d\n",arr[i]);
81      }
82
83      clock_gettime(CLOCK_MONOTONIC, &end);
84      //get the clock time and get the time that the application run
85      time_taken = (end.tv_sec - start.tv_sec) * 1e9;
86      time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
87      printf("Overall time: %f sec \n", time_taken);
88
89  }
90
91  //determine prime number
92  bool isprime(int number){
93
94      if (number <= 1){
95          return false;
96      }
97
98      if (number == 2){
99          return true;
100     }
101
102     if (number % 2 == 0){
103         return false;
104     }
105
106     //loop until sqrt of n as p*q = n so loop until one of them is sufficient
107     for (int i = 3; i <= sqrt(number); i++ ){
108         if (number % i == 0){
109             return false;
110         }
111     }
112     return true;
113
114 }
```

```
Overall time: 0.442950 sec

real    0m0.446s
user    0m0.234s
sys     0m0.391s
```

Task 2 – Explanation



Base speed:	3.20 GHz
Sockets:	1
Cores:	8
Logical processors:	16
Virtualization:	Enabled
L1 cache:	512 KB
L2 cache:	4.0 MB
L3 cache:	16.0 MB

My serial code run 0.533 seconds and my parallel code run 0.442950 seconds. Which is speed up by 16.88%, 0.09 seconds. For my parallel code I created 8 threads to speed up as I have 8 cores in my CPU. In my expectation, the code is supposed to speed up 8 times faster as I am using 8 threads compared to using 1 thread in my serial code.

1. I/O operations

The reason it is not meeting the expectations is the code performs a lot of I/O operations like printing to the console which can be a bottleneck as printing to console is slow and it not easily be parallelised. Even with multiple threads the console only handles the output operations once at a time.

2. Synchronization overhead

Another reason is each time a thread needs to update the array when they found a prime number. So each time when a thread needs to access one shared memory, it must lock the shared memory to prevent other threads update the data and cause data corruption or confusion. After finish updating it will unlock the shared memory for other threads to access. Thus, the cost of locking and unlocking will significantly reduce the expected speed up as other threads need to wait for the shared memory to be unlocked.

3. Workload imbalanced

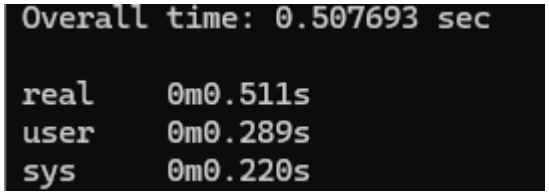
Besides that, the workload is not equally divided among the threads some threads might finish their tasks earlier and remain idle while others are still working. In my implementation, I have divided tasks to the threads for example, thread 1 calculating prime number 0 – 250000, thread 2 calculating 250001 – 500000. So thread 1 will be faster than thread 2 as its data can be calculated faster. So thread 1 will remain idle.

MONASH University

Task 3 – Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
#include <omp.h>

bool isprime(int number);

int main ()
{
    int *arr = (int *)malloc(1000000 * sizeof(int));
    struct timespec start, end, startComp, endComp;
    double time_taken;
    int n;
    int counter = 0;

    // Get current clock time.
    printf("Compute:\n");
    clock_gettime(CLOCK_MONOTONIC, &start);

    //parallel this for loop
    #pragma omp parallel
    {
        //define local array so each thread will create these
        int *localarr = (int *)malloc(1000000 * sizeof(int));
        int localcounter = 0;
```

```
30          #pragma omp for schedule(guided)
31          {
32              //each thread will update their prime number in their own local array
33              for (int i = 2; i < 1000000 ; i++){
34                  if (isprime(i)){
35                      localarr[localcounter] = i;
36                      localcounter++;
37                  }
38              }
39          }
40
41          //use to lock the shared memory to prevent race condition and data corruption
42          #pragma omp critical
43          {
44              //after they calculate they own value in their local array they update this global array a
45              //only one thread able to update everytime
46              for(int i = 0; i < localcounter; i++){
47                  arr[counter] = localarr[i];
48                  counter++;
49              }
50          }
51      }
52
53      //print all the output
54      for(int i = 0; i < counter; i++){
55          printf("%d\n", arr[i]);
56      }
58       //get the time
59       clock_gettime(CLOCK_MONOTONIC, &end);
60       time_taken = (end.tv_sec - start.tv_sec) * 1e9;
61       time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
62       printf("Overall time: %f sec \n", time_taken);
63
64  }
65
66
67
68  bool isprime(int number){
69
70      if (number <= 1){
71          return false;
72      }
73
74      if (number == 2){
75          return true;
76      }
77
78      if (number % 2 == 0){
79          return false;
80      }
81
82      //loop until sqrt of n as p*q = n so loop until one of them is sufficient
83      for (int i = 3; i <= sqrt(number); i++ ){
84          if (number % i == 0){
85              return false;
86          }
87      }
88      return true;
89
90  }
```



Task 3 - Explanation

Task 3 I have used openMP implementations to parallelized the code and it speeds up by 0.26 seconds. The threads is set to 8 to make the program run as fast as possible as there is 8 cores in my machine.

The expected speed up is supposed to be 8 times faster than the serial one. But due to several reasons it does not meet with the actual time.

1. Synchronization Overhead
   When the thread is updating the array we need to lock the shared memory to prevent race condition. In other words, this is to prevent conflicts between two threads trying to write data to the same location. Thus, we lock the shared memory, but locking the shared memory will cause threads to wait for it to be unlock. Hence, reduces the speed. Although in my implementation, I have declared each local array for each thread, so they can update in their own array without locking and unlocking. But after the calculation they need to write to the global array so it still introduce synchronization overhead.

2. Memory bandwidth limitation
   When multiple threads are accessing data from RAM the bandwidth is not sufficient to support the CPU's processing speed. So CPU may spent more time to wait for the data and cause it to be slow.