4F03 Assignment 2

Q1 : Pg 1-7
Q2 : Attached

QUESTION 1

Part 1
Given a number n, the program will find its two prime factors, p and q, so that $p \times q = n$. Firstly, we know that one of the two numbers will be in the range [0, $\sqrt{n}$]. This means that it's safe to search in this range for one of the factors. In order to split up this range into k parts so that all processes can search it, I find the number of digits in $\sqrt{n}$ using *mpz_sizeinbase*, and multiply this by the number of processes being used at the time. This helps so that when we have a large number n, we can split it up into many pieces and processes can work on one small piece at a time.
I then create an array with limits, call it *searchsections*. Each entry in the array is a limit of the range that each process will search (see pseudocode below). The processes will all run the same exact loop with different starting positions and each iteration of the *for loop* will be incremented by the number of processes, to ensure that multiple processes don't work on the same numbers. In this loop, p will start at *searchsections[i]* and will be increased until it has covered all the primes between *searchsections[i]* and *searchsections[i+1]*. When it finishes those, the next iteration of the loop will have *i=i+num_of_processes*. Each iteration will check if n is divisible by p. If not, then it moves to the next prime number. If it is divisible, it will check the other factor q for primeness. If it is prime, then we've found our solution. Otherwise, we know that our p is not right, so we need to keep searching for a prime.

In the loop, we also have a flag that shows us if we have found the prime numbers we need. Every certain amount of iterations of the loop, we communicate this flag to all other processes to make sure that when one process finishes, they all know that they can stop looking.

```
n = user input
procs = number of processes

k = mpz_sizeinbase(n) * procs

for j in 0:k
     searchsubsectons[i] = (sqrt(n)/k) * i


for (i = myrank; i<k; i=i+procs)
     while (!done)&&(p <= searchsections[i+1])
          MPI_Allreduce the done flag with MPI_MAX
          if n is not divisible by p
               p = nextprime(p)
               continue
```

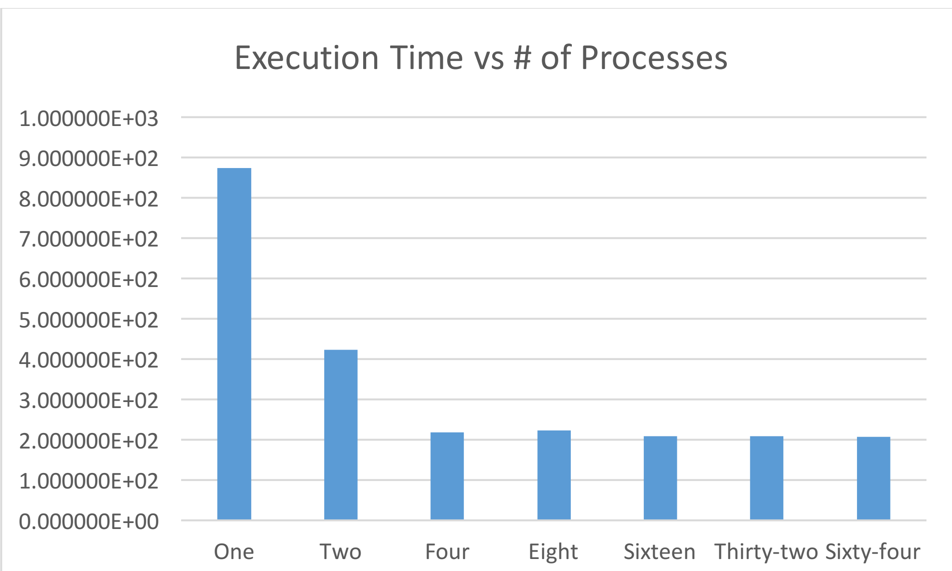
```
            else
                  q = n/p
                  if q is prime
                        done = true
                  else
                        p = nextprime(p)

      if (done) break
      end while
end for
```

Part 2

I found that using the number 1,367,973,661,846,368,403 achieves 15 minutes on one process. Running it with multiple processes gives the following table and graph:

One	Two	Four	Eight	Sixteen	Thirty-two	Sixty-four
8.740535E+02	4.233139E+02	2.194260E+02	2.229587E+02	2.095813E+02	2.086969E+02	2.074842E+02



Here we can see that there is a considerable amount of speedup when using multiple processes.

$$E = \frac{S}{p} = \frac{T_s}{T_p} \times \frac{1}{p} = \frac{8.740535e02}{2.086969e02} \times \frac{1}{32} = 0.1308$$

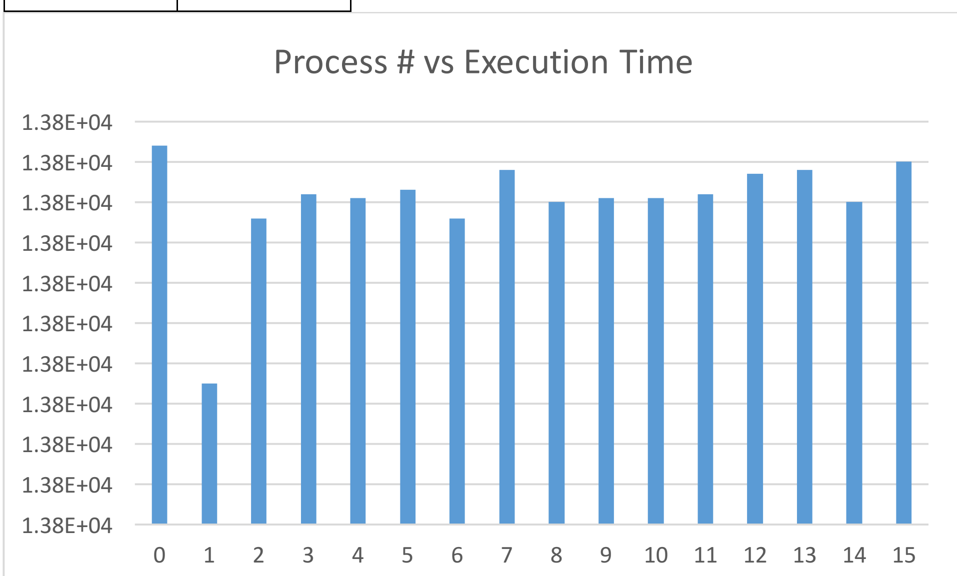
Therefore, our Speedup is 4.188 and Efficiency with 32 processes is 0.1308.

Here, I obtain a decent speedup and efficiency. We can see that the efficiency, however, tapers off very quickly around the time that we use four processors. If we were to compare Ts to Tp using 4 processors, our efficiency would greatly increase.

Part 3

In this section, I found a number which took roughly 3.8 hours. This number is 2,764,206,616,926,988,390,607.

Process#	Exec Time
0	1.38E+04
1	1.38E+04
2	1.38E+04
3	1.38E+04
4	1.38E+04
5	1.38E+04
6	1.38E+04
7	1.38E+04
8	1.38E+04
9	1.38E+04
10	1.38E+04
11	1.38E+04
12	1.38E+04
13	1.38E+04
14	1.38E+04
15	1.38E+04



CODE FOR QUESTION 1

```
#include <gmp.h>
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <math.h>


void writeTime(char *n, double *timearray, int j);

int main(int argc, char** argv)
{
    int         my_rank,
                procs;

    MPI_Status  status;

    mpz_t       q,
                p,
                pq,
                n,
                gap,
                sqn;

    mpz_t             *k;

    double            *timearray;
    double      begin, end;
    double            time_spent;

    int         done = 0,
                found = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &procs);


    begin = MPI_Wtime();

    mpz_init(n);
    mpz_init(q);
    mpz_init(p);
    mpz_init(pq);
    mpz_init(sqn);
    mpz_init(gap);

    mpz_set_str(n,argv[1],10);
    mpz_sqrt(sqn,n);
    mpz_set(gap, sqn);
```

```
    mpz_set_ui(p,1);
    mpz_nextprime(q, p);


    size_t mag = mpz_sizeinbase (gap, 10);
    mag=mag*procs;

    k = (mpz_t*)malloc(sizeof(mpz_t)*(mag+1));

    mpz_t temp;
    mpz_init(temp);
    mpz_tdiv_q_ui(temp,gap,mag);

    for (int i=0;i<=mag;i++)
    {
        mpz_init(k[i]);
        mpz_mul_ui(k[i],temp,i);
    }
    mpz_set(k[mag],sqn);





    int counter=0;


    for (int i=my_rank; i<mag; i=i+procs)
    {
        mpz_set(q,k[i]);

        mpz_sub_ui(q,q,1);
        mpz_nextprime(q,q);


        while (( mpz_cmp(q,k[i+1]) <=
0 )&&(!done)&&(!found)&&(mpz_cmp(q,sqn)<=0))
        {//finding the prime numbers
            counter++;
            if (counter%5000==0)MPI_Allreduce(&found, &done, 1,
MPI_INT, MPI_MAX, MPI_COMM_WORLD);

            if (mpz_divisible_p(n,q)==0)
            {//if n is not divisible by q
                mpz_nextprime(q,q);
                continue;
            }

            //since it is divisible, try n/q and see if result is
```

```
prime
            mpz_divexact(p,n,q);
            int reps;
            if (mpz_probab_prime_p(p,reps)!=0)
            {
                found = 1;
                done = 1;
            }
            else
            {
                mpz_nextprime(q,q);
            }
        }//done finding primes


        if (found || done) break;

    }

    end = MPI_Wtime();



    if (found)
    {
        MPI_Allreduce(&found, &done, 1, MPI_INT, MPI_MAX,
MPI_COMM_WORLD);
        gmp_printf("*************************\nP%d:
Finished\np*q=n\np=%Zd q=%Zd n=%Zd\n*************************\n",
my_rank,p,q,n);
    }
    else if (!done)
    {
        while (!done&&!found)
            MPI_Allreduce(&found, &done, 1, MPI_INT, MPI_MAX,
MPI_COMM_WORLD);
    }



    time_spent = (double)(end - begin);





    if (my_rank!=0)
    {
        MPI_Send(&time_spent,
sizeof(double),MPI_CHAR,0,0,MPI_COMM_WORLD);
```

```
    }
    else
    {
        timearray = (double*)malloc(sizeof(double)*procs);


        timearray[0] = time_spent;

        int j;
        for (j = 1; j<procs;j++)
        {
            double *ptr = timearray+j;
            MPI_Recv(ptr,
sizeof(double),MPI_CHAR,j,0,MPI_COMM_WORLD,&status);
        }


        writeTime(argv[1],timearray,procs);
        free(timearray);

    }



    free(k);

    MPI_Finalize();
    return 0;
}



void writeTime(char *n, double *timearray, int j)
{
    int i;

    // open file for writing
    char filename[100];
    sprintf(filename, "time_%s",n);

    FILE *fd;
    fd = fopen(filename, "w");

    // write the image
    for(i = 0; i < j; i++)
    {
        fprintf(fd, "%d\t%le\n", i,timearray[i]);
    }
    fclose(fd);
}
```