# Parallel Programming
# Assignment 2

Mahmood Naseer
Computer Science Department

October 25, 2023

## Prime Number Code Generation

This report compares different methods for optimizing prime number generation in the fastest way possible by converting the code into a parallel cyclic approach.

- $P_0$ starts from 3 and checks for prime numbers, then increases the step size by the number of total processes.
- $P_1$ starts from the next odd number after 3, which is 5, and continues in a similar manner.
- Each process follows this cyclic pattern for generating prime numbers efficiently.

Also compares how fast parallel processing distributes numbers in a cyclic manner, either by dividing them among processes.

**Q1 : Why the algorithms checks only up to sqrt(n) not up to N?**

$n$ is the input number. The algorithm checks if $n$ is prime by starting from 2 up to $\sqrt{n}$.
This is because if there is a factor divisible by $n$, it must be less than or equal to $\sqrt{n}$.
This method reduces the number of iterations for the loop, making the algorithm more efficient.

**Q3 : pseudocode for the mpiprime.cpp code**

Terminal 1: pseudocode

```
function main()
    if process_id == 0 then
        begin = 3
        stop = input_from_user() // Get the stop value from user input
        int block_test=stop/number_of_processes;
        for i: integer = 1 to number_of_processes - 1
            send (i*block_test) to process i
            send (i*block_test+block_test) to process i
        end for
    else
        receive begin from process 0
        receive stop from process 0
        for num: integer = begin to stop
            // test for prime number and print
            begin=begin+1
        end for
    end if
end function
```

This method divides all numbers from 'begin' to 'stop' among all processes
Let the number of processes be 4 and the stop value be 12. When we divide 12 by 4, each process will test 3 numbers after removing the even numbers:

- Process 0 will test 3, 4, and 5.
- Process 1 will test 6, 7, and 8.
- Process 2 will test 9, 10, and 11.
- Process 3 will test 12.

**Q4 : pseudocode for the mpiprimecyclic.cpp code**

Terminal 2: pseudocode

```
function main()
    if process_id == 0 then
        begin = 3
        stop = input_from_user() // Get the stop value from user input
        rank = # process
        for i: integer = 1 to number_of_processes - 1
            if rank is even then
                begin = begin + 2
            else
                begin = begin + 3
            send begin to process i
            send stop to process i
        end for
        end if
    else
        receive begin from process 0
        receive stop from process 0
    for num: integer = begin to stop
        // test for prime number and print
        //step size is number_of_processes*2 to move in cyclic way
        begin=begin+number_of_processes*2
    end for

end function
```

1. Obtain the input value from the user using command-line arguments.

2. Check if the process ID is odd.

3. If the process ID is odd, add 3 to begin,else add 2 . This will be the starting point for the process.

4. Send begin and stop for each process

5. Each process moves in steps of size equal to the number of processes multiplied by 2. Multiplying by 2 eliminates even numbers from consideration.

# Result

**Definition 1: Time effect**

If a process takes more time than other processes, it will slow down the entire program. Therefore, it is important for most processes to have similar execution times.

| n | Process 0 | Process 1 | Process 2 | Process 3 |
|---|---|---|---|---|
| 4 | 2.0188e-05 | 9.514e-06 | 1.4312e-05 | 1.447e-05 |
| 27 | 1.9036e-05 | 1.4312e-05 | 1.4072e-05 | 1.5406e-05 |
| 256 | 2.0566e-05 | 1.7013e-05 | 1.3513e-05 | 1.3792e-05 |
| 3125 | 4.6505e-05 | 4.4858e-05 | 5.5102e-05 | 5.9473e-05 |
| 46656 | 0.000902282 | 0.00131633 | 0.00128097 | 0.00165753 |
| 823543 | 0.033588 | 0.0612693 | 0.0649758 | 0.0749103 |
| 16777216 | 2.19437 | 3.51942 | 4.38887 | 4.93196 |

Table 1: Test with Block mpiprime.cpp

| n | Process 0 | Process 1 | Process 2 | Process 3 |
|---|---|---|---|---|
| 4 | 2.1778e-05 | 1.674e-05 | 1.5346e-05 | 1.5547e-05 |
| 27 | 1.9529e-05 | 1.2089e-05 | 1.5456e-05 | 1.0563e-05 |
| 256 | 2.3556e-05 | 2.0128e-05 | 1.853e-05 | 1.6039e-05 |
| 3125 | 5.7887e-05 | 3.9829e-05 | 3.7447e-05 | 5.1583e-05 |
| 46656 | 0.0012891 | 0.000718887 | 0.00125227 | 0.000739666 |
| 823543 | 0.0564921 | 0.056242 | 0.0532188 | 0.0530984 |
| 16777216 | 4.01768 | 4.03032 | 3.96029 | 4.02423 |

Table 2: Test with cyclic mpiprimecycilc.cpp

**Note 1: 16777216**

Note that when n is 16777216, the execution time of the mpiprime code differed, while in the mpiprimecycilc code, we saved 1 second

cyclic was faster, and all the processes worked equally. On the other hand, without cyclic by block, the time between processes was different. Considering my high-speed CPU and the heavy programs I work with, these factors might have affected the results. However, it is clear that cyclic was more stable.

**Note 2: Windows Subsystem for Linux**

I work on WSL (Windows Subsystem for Linux) on my PC; perhaps this could affect the results.

## How to Run the Code

Terminal 3: Bash : mpiprimecyclic.cpp

```
$ mpic++ ./mpiprimecyclic.cpp
# Comilpe the code
$ mpirun -np 8 --allow-run-as-root a.out 256
# mpirun -np 8 --allow-run-as-root a.out [Argument ]
process 3: 41 73 89 137 233  time = 2.9024e-05
process 4: 11 43 59 107 139 251  time = 2.0499e-05
process 6: 31 47 79 127 191 223 239  time = 1.7274e-05
process 0: 3 19 67 83 131 163 179 211 227  time = 1.5743e-05
process 1: 5 37 53 101 149 181 197 229  time = 1.6261e-05
process 2: 7 23 71 103 151 167 199  time = 1.5841e-05
process 7: 17 97 113 193 241  time = 1.6951e-05
process 5: 13 29 61 109 157 173  time = 1.4858e-05
```

Terminal 4: Bash : mpiprime.cpp

```
$ mpic++ ./mpiprime.cpp
# Comilpe the code
$ mpirun -np 8 --allow-run-as-root a.out 256
# mpirun -np 8 --allow-run-as-root a.out [Argument ]
process 4: 131 137 139 149 151 157  time = 2.4752e-05
process 5: 163 167 173 179 181 191  time = 1.575e-05
process 6: 193 197 199 211 223  time = 1.7078e-05
process 7: 227 229 233 239 241 251  time = 1.591e-05
process 0: 3 5 7 11 13 17 19 23 29 31  time = 1.4383e-05
process 1: 37 41 43 47 53 59 61  time = 1.8054e-05
process 2: 67 71 73 79 83 89  time = 2.7491e-05
process 3: 97 101 103 107 109 113 127  time = 2.7615e-05
```