**Project:- Distributed Fault Tolerant Prime Finding Algorithm**

GitHub Link for the code:- [PrabushettySaiKrishna/Distributed-Systems (github.com)](https://github.com/PrabushettySaiKrishna/Distributed-Systems)

Step by step instructions of the code

# Distributed-Systems

A distributed prime number finder. The algorithm is implemented using MPI Programming.

**Development**

* Please set editor to use the Google C++ format.

For VSCode, go to preferences and in the workspace tab, check format on save. Then, set clang-format's fallback style to "Google".

* Useful make targets:

make primetime

make debug

make clean

* Command to run:

mpirun -np <num-procs> -use-hwthread-cpus -oversubscribe ./primetime <base> <offset>

here, <num-procs> is the number of threads are need to be used. <base> is the first element. <offset> is the last element of the loop.

* Output It is the time taken by the algoritm to find the prime numbers.

**Code Details**

**Include Folder**

It contains header files. All the files contains library headers, funtions definition and definition of funtions.

**Makefile**

Make file is a program that's used to build other programs or documents. Make is suitable for any process that has intermediate steps that may or may not be complete. Make doesn't rebuild things that are already up to date.

**main.cpp**

The file first check if the input is valid or not using the funtions:

bool is\_valid\_num(int64\_t num) {

if (num < 0) return false;

return true;

}

bool is\_valid\_nth(int64\_t nth) {

if (nth <= 0) return false;

return true;

}

Initialise the main funtion of MPI, MPI\_Init. Then, Synchronize all processes for time keeping using.

MPI\_Barrier(MPI\_COMM\_WORLD);

After checking the errors with the manager and the input. Start with the basic search.

1. If it is the manager, then begin the loop for search. Broadcast the iteration number till the element is not found.

MPI\_Bcast(&iteration, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

After receiving the result from each process, check for the prime number.

MPI\_Recv(&recv\_arr, batch\_size, MPI\_INT64\_T, i, 0, MPI\_COMM\_WORLD, MPI\_SUCCESS);

if (current\_prime\_count == nth)

{

// Return the answer

cout << x << "," << (iteration-1) \* (num\_procs-1)\* batch\_size + i \* batch\_size << "," << endl;

found = true;

}

1. If follower, then run the loop forever and send the primes between base and offset.

for (auto i = min\_val; i < max\_val; i++) {

if (is\_prime(i)) primes[primecount++] = i;

}

MPI\_Send(&primes, batch\_size, MPI\_INT64\_T, 0, 0, MPI\_COMM\_WORLD);

After that, synchronize all processes for time keeping and return the max time of all processes and shutdown the MPI.

MPI\_Finalize();

**primality\_test.cpp**

First, the basic code of modular binary exponentiation is explained. The power is generally evaluated under the modulo of a large number. Below is the fundamental modular property that is used for efficiently computing power under modular arithmetic.

base %= mod;

while (exponent) {

if (exponent & 1) result = (\_\_int128\_t)result \* base % mod;

base = (\_\_int128\_t)base \* base % mod;

exponent >>= 1;

}

Then, the check\_composite function is checking the Miller-Robin primality condition for a particular witness.

for (int r = 1; r < s; ++r) {

x = (\_\_int128\_t)x \* x % n;

if (x == n - 1) {

return false;

}

}

The above code checks if the number x has any of the prime factors other than 1 or x, then it is not prime and return false, else true.

The deterministic\_miller\_rabin\_64 function is Miller-Rabin primality test. Checks only as many prime witnesses as needed for 64 bit integers. It returns true if n is prime, false otherwise.

for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) {

if (n == a) {

return true;

}

}

It is given the some prime numbers and directly checks if it belongs to them or not. Return true if belongs to them, else

check\_composite(n, a, d, r)

and return true or false based on that.

**Other files**

Other files, utils.cpp and worker.cpp provide basic struture for master and follower nodes.(the codes of the mentioned files are below as follows).

**CODE FILES:-**

**Main.cpp**

|  |
| --- |
| #include "includes/main.h" |
|  |  |
|  | #include <mpi.h> |
|  | #include <stdlib.h> |
|  | #include <sys/types.h> |
|  | #include <unistd.h> |
|  |  |
|  | #include <cassert> |
|  | #include <cmath> |
|  | #include <cstddef> |
|  | #include <cstring> |
|  | #include <fstream> |
|  | #include <iomanip> |
|  | #include <iostream> |
|  | #include <limits> |
|  |  |
|  | #include "includes/primality\_test.h" |
|  | #include "includes/utils.h" |
|  | #include "includes/worker.h" |
|  |  |
|  | using namespace std; |
|  |  |
|  | bool is\_valid\_num(int64\_t num) { |
|  | if (num < 0) return false; |
|  |  |
|  | return true; |
|  | } |
|  |  |
|  | bool is\_valid\_nth(int64\_t nth) { |
|  | if (nth <= 0) return false; |
|  |  |
|  | return true; |
|  | } |
|  |  |
|  | int main(int argc, char \*\*argv) { |
|  | // Initialize. |
|  | MPI\_Init(&argc, &argv); |
|  |  |
|  | int root = 0; |
|  | int rank, num\_procs; |
|  | MPI\_Comm\_size(MPI\_COMM\_WORLD, &num\_procs); |
|  | MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); |
|  |  |
|  | if ((argc < 3) and (rank == 0)) { |
|  | cerr << "Invalid input." << endl; |
|  | MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); |
|  | } |
|  |  |
|  | // Synchronize all processes for time keeping. |
|  | MPI\_Barrier(MPI\_COMM\_WORLD); |
|  | double start\_time = MPI\_Wtime(); |
|  |  |
|  | // Do stuff. |
|  |  |
|  | int64\_t num = (int64\_t)stoll(argv[1]); |
|  | int64\_t nth = (int64\_t)stoll(argv[2]); |
|  | int batch\_size = 1000; // Have it possibly calculated or input by the user |
|  |  |
|  | bool found = false; |
|  |  |
|  | // Error checking with the manager |
|  | if (rank == 0) { |
|  | if (!is\_valid\_num(num)) { |
|  | cerr << "Specify n greater than 0" << endl; |
|  | MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); |
|  | } |
|  |  |
|  | if (!is\_valid\_nth(nth)) { |
|  | cerr << "Specify nth greater than 0" << endl; |
|  | MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE); |
|  | } |
|  | } |
|  |  |
|  | // If manager |
|  | if (rank == 0) { |
|  | // Begin the loop for search |
|  | int64\_t current\_prime\_count = 0; |
|  | int64\_t iteration = 1; |
|  |  |
|  | while (!found) { |
|  | // Broadcast the iteration number to each process |
|  | MPI\_Bcast(&iteration, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); |
|  |  |
|  | // Receive the result from each of the processes |
|  | for (int i = 1; i < num\_procs; i++) { |
|  | int64\_t recv\_arr[batch\_size]; |
|  |  |
|  | MPI\_Recv(&recv\_arr, batch\_size, MPI\_INT64\_T, i, 0, MPI\_COMM\_WORLD, |
|  | MPI\_SUCCESS); |
|  |  |
|  | // Check for nth prime |
|  | if (!found) { |
|  | for (auto x : recv\_arr) { |
|  | if (x != 0) { |
|  | current\_prime\_count += 1; |
|  |  |
|  | if (current\_prime\_count == nth) { |
|  | // Return the answer |
|  | cout << x << "," << (iteration-1) \* (num\_procs-1) \* batch\_size + i \* batch\_size << "," << endl; |
|  | found = true; |
|  | } |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | // Increment iteration |
|  | ++iteration; |
|  | } |
|  |  |
|  | // Ask other nodes to break |
|  | iteration = -1; |
|  | MPI\_Bcast(&iteration, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); |
|  | } |
|  | // For the followers |
|  | else { |
|  | // Run forever |
|  | while (true) { |
|  | int iteration; |
|  |  |
|  | MPI\_Bcast(&iteration, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); |
|  |  |
|  | if (iteration == -1) break; |
|  |  |
|  | int64\_t min\_val = num + (iteration - 1) \* batch\_size \* (num\_procs - 1) + |
|  | batch\_size \* (rank - 1); |
|  | int64\_t max\_val = min\_val + batch\_size; |
|  |  |
|  | int64\_t primes[batch\_size] = {0}; |
|  | int primecount = 0; |
|  | for (auto i = min\_val; i < max\_val; i++) { |
|  | if (is\_prime(i)) primes[primecount++] = i; |
|  | } |
|  |  |
|  | MPI\_Send(&primes, batch\_size, MPI\_INT64\_T, 0, 0, MPI\_COMM\_WORLD); |
|  | } |
|  | } |
|  |  |
|  | // Synchronize all processes for time keeping. |
|  | MPI\_Barrier(MPI\_COMM\_WORLD); |
|  | double end\_time = MPI\_Wtime(); |
|  | double elapsed\_time = end\_time - start\_time; |
|  |  |
|  | // Get max time across all processes. |
|  | double max\_time; |
|  | MPI\_Reduce(&elapsed\_time, &max\_time, |
|  | /\*count=\*/1, MPI\_DOUBLE, MPI\_MAX, root, MPI\_COMM\_WORLD); |
|  |  |
|  | if (rank == root) { |
|  | cout << max\_time << endl; |
|  | } |
|  |  |
|  | // Shutdown. |
|  | MPI\_Finalize(); |
|  | return 0; |
|  | } |

**Primarily\_test.cpp**

|  |
| --- |
| #include "includes/primality\_test.h" |
|  |  |
|  | #include <mpi.h> |
|  | #include <stdlib.h> |
|  | #include <sys/types.h> |
|  | #include <unistd.h> |
|  |  |
|  | #include <cassert> |
|  | #include <cmath> |
|  | #include <cstddef> |
|  | #include <cstring> |
|  | #include <fstream> |
|  | #include <iomanip> |
|  | #include <iostream> |
|  | #include <limits> |
|  |  |
|  | #include "includes/utils.h" |
|  |  |
|  | using namespace std; |
|  |  |
|  | // Deterministic Miller Rabin guaranteed to be correct for 64 bit integers. |
|  |  |
|  | // Computes base^exponent modulo mod using binary exponentiation. |
|  | int64\_t binary\_power(int64\_t base, int64\_t exponent, int64\_t mod) { |
|  | int64\_t result = 1; |
|  | base %= mod; |
|  | while (exponent) { |
|  | if (exponent & 1) result = (\_\_int128\_t)result \* base % mod; |
|  | base = (\_\_int128\_t)base \* base % mod; |
|  | exponent >>= 1; |
|  | } |
|  | return result; |
|  | } |
|  |  |
|  | // Checks the Miller-Rabin primality condition for a particular witness. |
|  | bool check\_composite(int64\_t n, int64\_t a, int64\_t d, int s) { |
|  | int64\_t x = binary\_power(a, d, n); |
|  |  |
|  | if (x == 1 || x == n - 1) { |
|  | return false; |
|  | } |
|  |  |
|  | for (int r = 1; r < s; ++r) { |
|  | x = (\_\_int128\_t)x \* x % n; |
|  | if (x == n - 1) { |
|  | return false; |
|  | } |
|  | } |
|  |  |
|  | return true; |
|  | } |
|  |  |
|  | // Miller-Rabin primality test. Checks only as many prime witnesses as needed |
|  | // for 64 bit integers. |
|  | // Returns true if n is prime, false otherwise. |
|  | bool deterministic\_miller\_rabin\_64(int64\_t n) { |
|  | if (n < 2) { |
|  | return false; |
|  | } |
|  |  |
|  | int r = 0; |
|  | int64\_t d = n - 1; |
|  | while ((d & 1) == 0) { |
|  | d >>= 1; |
|  | r++; |
|  | } |
|  |  |
|  | for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}) { |
|  | if (n == a) { |
|  | return true; |
|  | } |
|  |  |
|  | if (check\_composite(n, a, d, r)) return false; |
|  | } |
|  |  |
|  | return true; |
|  | } |
|  |  |
|  | // Exported function for checking primality. |
|  | bool is\_prime(\_\_int64\_t n) { return deterministic\_miller\_rabin\_64(n); } |

**Utils.cpp**

|  |
| --- |
| #include "includes/utils.h" |
|  |  |
|  | #include <mpi.h> |
|  | #include <stdlib.h> |
|  | #include <sys/types.h> |
|  | #include <unistd.h> |
|  |  |
|  | #include <cassert> |
|  | #include <cmath> |
|  | #include <cstddef> |
|  | #include <cstring> |
|  | #include <fstream> |
|  | #include <iomanip> |
|  | #include <iostream> |
|  | #include <limits> |
|  |  |
|  | using namespace std; |

**Worker.cpp**

|  |
| --- |
| #include "includes/worker.h" |
|  |  |
|  | #include <mpi.h> |
|  | #include <stdlib.h> |
|  | #include <sys/types.h> |
|  | #include <unistd.h> |
|  |  |
|  | #include <cassert> |
|  | #include <cmath> |
|  | #include <cstddef> |
|  | #include <cstring> |
|  | #include <fstream> |
|  | #include <iomanip> |
|  | #include <iostream> |
|  | #include <limits> |
|  |  |
|  | #include "includes/primality\_test.h" |
|  | #include "includes/utils.h" |
|  |  |
|  | using namespace std; |