Implementation of the Quadratic Sieve Algorithm

October 20, 2024

Abstract

The Quadratic Sieve (QS) is one of the most efficient algorithms for factoring large composite numbers, particularly effective for numbers with up to 100 digits. This report details the implementation of the Quadratic Sieve algorithm in C++, leveraging the GNU Multiple Precision Arithmetic Library (GMP) for handling large integers and the Message Passing Interface (MPI) for parallelization. The report covers the theoretical foundations of QS, the design and structure of the implementation, challenges encountered, optimizations applied, and the results obtained.

Contents

1 Introduction

The Quadratic Sieve (QS) is a highly effective integer factorization algorithm, second only to the General Number Field Sieve for very large integers. Its primary application is in cryptographic systems where the security relies on the difficulty of factoring large composite numbers. This report presents the implementation of the QS algorithm in C++, utilizing MPI for parallel processing to enhance performance and GMP for precise large integer arithmetic.

2 Background

2.1 Integer Factorization

Integer factorization involves decomposing a composite number N into a product of smaller integers, typically prime numbers. The difficulty of this task underpins the security of many cryptographic protocols, such as RSA.

2.2 Quadratic Sieve Algorithm

The Quadratic Sieve algorithm operates by finding congruent squares modulo N. Once such squares are found, their difference yields a non-trivial factor of N. The algorithm comprises several key steps:

1. Selection of a polynomial.
2. Generation of a factor base.
3. Sieving to find smooth numbers.
4. Solving a system of linear equations to find dependencies.
5. Extraction of factors using the dependencies.

3 Implementation

3.1 Tools and Libraries

- **C++**: The primary programming language used for implementation.
- **GMP (GNU Multiple Precision Arithmetic Library)**: Facilitates operations on large integers required by the QS algorithm.
- **MPI (Message Passing Interface)**: Enables parallel computation across multiple processes, significantly speeding up the sieving step.

3.2 Code Structure

The implementation is organized into several key functions, each corresponding to a step in the QS algorithm:

- `generatePrimes(int limit)`: Generates all prime numbers up to a specified limit using the Sieve of Eratosthenes.
- `legendreSymbol(const mpz_class &a, int p)`: Computes the Legendre symbol to determine if n is a quadratic residue modulo p.
- `compute_Qx(int x, const mpz_class &m, const mpz_class &n)`: Computes $Q(x) = (x+m)^2 - n$.
- `factorize_Qx(mpz_class Qx, const std::vector<int` &factor_base): Attempts to factorize $Q(x)$ over the factor base.
- `printMatrix(const std::vector<std::vector<int` &matrix, const std::vector¡int &factor_base): Prints the exponent matrix modulo 2.
- `findDependencies(std::vector<std::vector<int` matrix_mod2): Performs Gaussian elimination over GF(2) to find dependencies.

3.3 Prime Generation

The `generatePrimes` function employs the Sieve of Eratosthenes to efficiently generate all prime numbers up to a given limit. This factor base is crucial for identifying smooth numbers during the sieving process.

```cpp
std::vector<int> generatePrimes(int limit) {
    std::vector<bool> is_prime(limit + 1, true);
    is_prime[0] = is_prime[1] = false;
    int sqrt_limit = static_cast<int>(std::sqrt(limit));
    for (int p = 2; p <= sqrt_limit; ++p) {
        if (is_prime[p]) {
            for (int multiple = p * p; multiple <= limit; multiple += p) {
                is_prime[multiple] = false;
            }
        }
    }
    std::vector<int> primes;
    for (int p = 2; p <= limit; ++p) {
        if (is_prime[p]) {
            primes.push_back(p);
        }
    }
    return primes;
}
```

Listing 1: Prime Generation Function

3.4 Legendre Symbol Calculation

The Legendre symbol $\left(\frac{a}{p}\right)$ determines whether a is a quadratic residue modulo p. This calculation is essential for building the factor base by selecting primes for which n is a quadratic residue.

```cpp
int legendreSymbol(const mpz_class &a, int p) {
    mpz_class a_mod_p = a % p;
    if (a_mod_p == 0) {
        return 0;
    }
    int exponent = (p - 1) / 2;
    mpz_class result;
    mpz_powm_ui(result.get_mpz_t(), a_mod_p.get_mpz_t(), exponent,
        mpz_class(p).get_mpz_t());
    if (result == 1) {
        return 1;
    } else if (result == p - 1) {
        return -1;
    } else {
        return 0;
    }
}
```

Listing 2: Legendre Symbol Calculation

3.5 Computing $Q(x)$

The function `compute_Qx` calculates the quadratic polynomial $Q(x) = (x+m)^2 - n$, which is central to the QS algorithm. Identifying smooth values of $Q(x)$ over the factor base is critical for finding dependencies.

```cpp
mpz_class compute_Qx(int x, const mpz_class &m, const mpz_class &n) {
    mpz_class x_plus_m = x + m;
    mpz_class Qx = x_plus_m * x_plus_m - n;
    return Qx;
}
```

Listing 3: Compute Q(x) Function

3.6 Factorization Over Factor Base

The `factorize_Qx` function attempts to factorize $Q(x)$ using the primes in the factor base. If $Q(x)$ can be fully factorized, it is considered smooth, and the exponents of the primes are recorded.

```cpp
std::vector<int> factorize_Qx(mpz_class Qx, const std::vector<int>
    &factor_base) {
    std::vector<int> exponents(factor_base.size(), 0);
    if (Qx == 0) {
        return {}; // Cannot factor zero
    }
    if (Qx < 0) {
        exponents[0] = 1; // Exponent of -1 is 1
        Qx = -Qx;
    }
    for (size_t i = 1; i < factor_base.size(); ++i) {
        int p = factor_base[i];
        while (mpz_divisible_ui_p(Qx.get_mpz_t(), p)) {
            exponents[i]++;
            Qx /= p;
```

```
        }
    }
    if (Qx == 1) { // Successfully factorized over Factor Base
        return exponents;
    } else {
        return {}; // Not smooth
    }
}
```

Listing 4: Factorization Function

3.7 Exponent Matrix Printing

The `printMatrix` function outputs the exponent matrix modulo 2, which is used in the Gaussian elimination step to find dependencies.

```
void printMatrix(const std::vector<std::vector<int>> &matrix, const
    std::vector<int> &factor_base) {
    std::cout << "Exponent Matrix (mod 2):\n";
    // Header
    std::cout << "Row\t";
    for (const auto &p : factor_base) {
        std::cout << p << "\t";
    }
    std::cout << "\n";
    // Rows
    for (size_t i = 0; i < matrix.size(); ++i) {
        std::cout << i + 1 << "\t";
        for (const auto &val : matrix[i]) {
            std::cout << val % 2 << "\t";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}
```

Listing 5: Matrix Printing Function

3.8 Gaussian Elimination for Dependency Finding

The `findDependencies` function performs Gaussian elimination over the binary field GF(2) to identify dependencies in the exponent matrix. These dependencies are essential for constructing the congruent squares needed to factor N.

```
std::vector<std::vector<int>>
    findDependencies(std::vector<std::vector<int>> matrix_mod2) {
    int num_rows = matrix_mod2.size();
    if (num_rows == 0)
        return {};
    int num_cols = matrix_mod2[0].size();
    std::vector<int> pivot_col(num_cols, -1);
    // Initialize an identity matrix to track dependencies
    std::vector<std::vector<int>> identity(num_rows,
        std::vector<int>(num_rows, 0));
    for (int i = 0; i < num_rows; ++i) {
        identity[i][i] = 1;
```

```cpp
    }
    // Perform Gaussian elimination
    int row = 0;
    for (int col = 0; col < num_cols && row < num_rows; ++col) {
        // Find a pivot row
        int pivot_row = -1;
        for (int r = row; r < num_rows; ++r) {
            if (matrix_mod2[r][col] == 1) {
                pivot_row = r;
                break;
            }
        }
        if (pivot_row == -1) {
            continue; // No pivot in this column
        }
        // Swap current row with pivot_row if necessary
        if (pivot_row != row) {
            std::swap(matrix_mod2[row], matrix_mod2[pivot_row]);
            std::swap(identity[row], identity[pivot_row]);
        }
        pivot_col[col] = row;
        // Eliminate all other 1's in this column
        for (int r = 0; r < num_rows; ++r) {
            if (r != row && matrix_mod2[r][col] == 1) {
                for (int c = 0; c < num_cols; ++c) {
                    matrix_mod2[r][c] ^= matrix_mod2[row][c];
                }
                for (int c = 0; c < num_rows; ++c) {
                    identity[r][c] ^= identity[row][c];
                }
            }
        }
        row++;
    }
    // Identify dependencies (nullspace vectors)
    std::vector<std::vector<int>> dependencies;
    // Rows without a pivot correspond to dependencies
    for (int r = 0; r < num_rows; ++r) {
        bool is_zero = true;
        for (int c = 0; c < num_cols; ++c) {
            if (matrix_mod2[r][c] != 0) {
                is_zero = false;
                break;
            }
        }
        if (is_zero) {
            // The corresponding row in the identity matrix represents the dependency
            dependencies.push_back(identity[r]);
        }
    }
    return dependencies;
}
```

Listing 6: Gaussian Elimination Function

3.9 Main Function

The `main` function orchestrates the entire Quadratic Sieve process, utilizing MPI for parallel processing. Below is an overview of its workflow:

1. **MPI Initialization**: Sets up the MPI environment and determines the number of processes and their ranks.
2. **Input Handling**: The root process initializes the target number n and broadcasts it to all processes.
3. **Factor Base Construction**: Generates primes, filters out those dividing n, computes Legendre symbols, and constructs the factor base including -1.
4. **Sieving Process**: Distributes the range of x values among processes, computes $Q(x)$, and identifies smooth numbers.
5. **Gathering Smooth Relations**: Collects smooth relations from all processes to the root process.
6. **Gaussian Elimination and Dependency Analysis**: The root process constructs the exponent matrix, performs Gaussian elimination, and identifies dependencies.
7. **Factor Extraction**: Utilizes dependencies to compute potential factors of n and verifies them using GCD.
8. **Finalization**: Cleans up the MPI environment and terminates the program.

```
int main(int argc, char *argv[]) {
    // Initialize MPI environment
    MPI_Init(&argc, &argv);
    int world_size; // Number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    int world_rank; // Rank of the current process
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Define the target number 'n' as a string (input)
    std::string n_str;
    if (world_rank == 0) {
        // Example large number (replace with desired 35-40 digit number)
        n_str = "1001";
        std::cout << "Quadratic Sieve (QS) Implementation\n";
        std::cout << "===================================\n";
        std::cout << "Target number (n): " << n_str << std::endl;
    }

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Broadcasting n_str and m to all processes
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    // Broadcast the number string length to all processes
    int n_str_length = n_str.size();
    MPI_Bcast(&n_str_length, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```

    // Broadcast the number string to all processes
    char *n_str_cstr = new char[n_str_length + 1];
    if (world_rank == 0) {
        std::copy(n_str.begin(), n_str.end(), n_str_cstr);
        n_str_cstr[n_str_length] = '\0';
    }
    MPI_Bcast(n_str_cstr, n_str_length + 1, MPI_CHAR, 0,
        MPI_COMM_WORLD);

    // Convert the received string to mpz_class
    mpz_class n(n_str_cstr);

    delete[] n_str_cstr; // Clean up

    // Compute 'm' = floor(sqrt(n))
    mpz_class m;
    mpz_sqrt(m.get_mpz_t(), n.get_mpz_t());

    if (world_rank == 0) {
        std::cout << "Computed m (floor(sqrt(n))): " << m << "\n" <<
            std::endl;
    }

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Broadcasting m to all processes
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    // Broadcast 'n' and 'm' to all processes
    // Since mpz_class cannot be directly broadcasted, we can
        serialize it
    std::string n_serialized = n.get_str();
    std::string m_serialized = m.get_str();

    // Broadcast the lengths first
    int n_len = n_serialized.size();
    int m_len = m_serialized.size();
    MPI_Bcast(&n_len, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&m_len, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Broadcast the strings
    char *n_cstr = new char[n_len + 1];
    char *m_cstr = new char[m_len + 1];
    if (world_rank == 0) {
        std::copy(n_serialized.begin(), n_serialized.end(), n_cstr);
        n_cstr[n_len] = '\0';
        std::copy(m_serialized.begin(), m_serialized.end(), m_cstr);
        m_cstr[m_len] = '\0';
    }
    MPI_Bcast(n_cstr, n_len + 1, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Bcast(m_cstr, m_len + 1, MPI_CHAR, 0, MPI_COMM_WORLD);

    // Convert back to mpz_class
    n = mpz_class(n_cstr);
    m = mpz_class(m_cstr);

    delete[] n_cstr;
    delete[] m_cstr;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Factor Base Construction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

std::vector<int> factor_base_primes; // Primes where (n/p) = 1
std::vector<int> factor_base;        // Including -1

// Step 1: Generate primes up to a certain limit
int prime_limit = 200; // Adjust as needed for larger numbers
std::vector<int> primes = generatePrimes(prime_limit);

// Step 2: Exclude primes that divide 'n'
std::vector<int> primes_filtered;
for (const auto &p : primes) {
    if (mpz_divisible_ui_p(n.get_mpz_t(), p) == 0) { // Exclude if
        p divides n
        primes_filtered.push_back(p);
    }
}

// Compute Legendre symbols and build Factor Base
for (const auto &p : primes_filtered) {
    int ls = legendreSymbol(n, p);
    if (ls == 1) { // Include in Factor Base if n is a quadratic
        residue modulo p
        factor_base_primes.push_back(p);
    }
}

// Step 4: Construct the Factor Base by adding -1
factor_base.push_back(-1); // Always include -1
factor_base.insert(factor_base.end(), factor_base_primes.begin(),
    factor_base_primes.end());

if (world_rank == 0) {
    // Display the Factor Base
    std::cout << "Final Factor Base (including -1):\n";
    std::cout << "Size of final factor base including -1 : " <<
        factor_base.size() << "\n";
    for (const auto &p : factor_base) {
        std::cout << p << " ";
    }
    std::cout << "\n\n";
}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Broadcasting Factor Base
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

// Broadcast the size of the Factor Base to all processes
int fb_size = factor_base.size();
MPI_Bcast(&fb_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Broadcast the Factor Base to all processes
if (world_rank != 0) {
    factor_base.resize(fb_size);
}
```

```
    MPI_Bcast(factor_base.data(), fb_size, MPI_INT, 0, MPI_COMM_WORLD);

    // Ensure all processes have received the Factor Base
    MPI_Barrier(MPI_COMM_WORLD);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Sieving Process
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    // Step 5: Sieving Process - Compute Q(x) for x in a range
    const int x_min = 0;
    const int x_max = 99; // Increase the range to collect more relations
    const int total_x = x_max - x_min + 1;

    // Determine the number of x's per process
    int x_per_process = total_x / world_size;
    int remainder = total_x % world_size;

    // Determine the start and end x for each process
    int local_x_start, local_x_end;
    if (world_rank < remainder) {
        // Processes with rank < remainder get (x_per_process + 1) x's
        local_x_start = x_min + world_rank * (x_per_process + 1);
        local_x_end = local_x_start + x_per_process;
    } else {
        // Processes with rank >= remainder get x_per_process x's
        local_x_start = x_min + world_rank * x_per_process + remainder;
        local_x_end = local_x_start + x_per_process - 1;
    }
    // Handle edge cases where x_end might exceed x_max
    if (local_x_end > x_max) {
        local_x_end = x_max;
    }

    // Each process computes Q(x) for its assigned x's and factorizes them
    std::vector<std::vector<int>> local_smooth_relations; // Exponent vectors
    std::vector<int> local_smooth_x; // Corresponding x values

    std::cout << "\nThis sieve is done by Process :" << world_rank << "\n";
    for (int x = local_x_start; x <= local_x_end; ++x) {
        mpz_class Qx = compute_Qx(x, m, n);
        std::vector<int> exponents = factorize_Qx(Qx, factor_base);
        std::cout << "Q(" << x << ") = " << Qx << " at x :" << x
                  << " by process " << world_rank << " ----> ";
        for(int i = 0; i < exponents.size(); i++) {
            std::cout << exponents[i] << " ";
        }
        std::cout << "\n";
        if (!exponents.empty()) { // Q(x) is smooth
            local_smooth_relations.emplace_back(exponents);
            local_smooth_x.push_back(x);
        }
    }

```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Gathering Smooth Relations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

// Gather the counts of smooth relations from each process
int local_count = local_smooth_relations.size();
std::vector<int> recv_counts(world_size, 0);
MPI_Gather(&local_count, 1, MPI_INT, recv_counts.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);

// Prepare for Gatherv
std::vector<int> displs(world_size, 0);
int total_recv = 0;
if (world_rank == 0) {
    total_recv = recv_counts[0];
    for (int i = 1; i < world_size; ++i) {
        displs[i] = displs[i - 1] + recv_counts[i - 1];
        total_recv += recv_counts[i];
    }
}

// Gather smooth x values
std::vector<int> all_smooth_x(total_recv);
MPI_Gatherv(local_smooth_x.data(), local_count, MPI_INT, all_smooth_x.data(),
            recv_counts.data(), displs.data(), MPI_INT, 0, MPI_COMM_WORLD);

// Flatten exponents for MPI communication
std::vector<int> local_exponents_flat;
for (const auto &exponents : local_smooth_relations) {
    local_exponents_flat.insert(local_exponents_flat.end(), exponents.begin(), exponents.end());
}
int exponents_per_relation = factor_base.size();
std::vector<int> recv_counts_exponents(world_size, 0);
int local_exponents_count = local_exponents_flat.size();
MPI_Gather(&local_exponents_count, 1, MPI_INT, recv_counts_exponents.data(),
           1, MPI_INT, 0, MPI_COMM_WORLD);
std::vector<int> displs_exponents(world_size, 0);
int total_exponents_recv = 0;
if (world_rank == 0) {
    total_exponents_recv = recv_counts_exponents[0];
    for (int i = 1; i < world_size; ++i) {
        displs_exponents[i] = displs_exponents[i - 1] + recv_counts_exponents[i - 1];
        total_exponents_recv += recv_counts_exponents[i];
    }
}
std::vector<int> all_exponents_flat(total_exponents_recv);
MPI_Gatherv(local_exponents_flat.data(), local_exponents_count, MPI_INT,
            all_exponents_flat.data(), recv_counts_exponents.data(),
            displs_exponents.data(), MPI_INT, 0, MPI_COMM_WORLD);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```cpp
    %%% Root Process: Assembling Smooth Relations and Dependency Analysis
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    if (world_rank == 0) {
        // Reconstruct exponent vectors
        int num_relations = total_exponents_recv / exponents_per_relation;
        std::vector<std::vector<int>> smooth_relations(num_relations, std::vector<int>(exponents_per_relation));
        for (int i = 0; i < num_relations; ++i) {
            for (int j = 0; j < exponents_per_relation; ++j) {
                smooth_relations[i][j] = all_exponents_flat[i * exponents_per_relation + j];
            }
        }
        std::cout << "Total smooth relations found: " << num_relations << "\n" << std::endl;
        if (smooth_relations.empty()) {
            std::cout << "No smooth relations found. Increase the sieving range or adjust the Factor Base."
                      << std::endl;
            MPI_Finalize();
            return 0;
        }
        // Step 7: Construct the Exponent Matrix
        std::cout << "Constructing the Exponent Matrix...\n" << std::endl;
        printMatrix(smooth_relations, factor_base);
        // Step 8: Perform Gaussian Elimination to Find Dependencies
        std::cout << "Performing Gaussian Elimination over GF(2) to find dependencies...\n" << std::endl;
        // Create a copy of the matrix with exponents modulo 2
        std::vector<std::vector<int>> matrix_mod2 = smooth_relations;
        for (auto &row : matrix_mod2) {
            for (auto &val : row) {
                val = val % 2;
            }
        }
        std::vector<std::vector<int>> dependencies = findDependencies(matrix_mod2);
        // Display Dependencies
        if (dependencies.empty()) {
            std::cout << "No dependencies found.\n" << std::endl;
        } else {
            std::cout << "Dependencies Found:\n";
            for (size_t i = 0; i < dependencies.size(); ++i) {
                std::cout << "Dependency " << i + 1 << ": ";
                for (size_t j = 0; j < dependencies[i].size(); ++j) {
                    if (dependencies[i][j] == 1) {
                        std::cout << "Relation " << j + 1 << " ";
                    }
                }
                std::cout << "\n";
            }
            std::cout << "\n";
            // Step 9: Use dependencies to compute 'a' and 'b', and find factors
```

```
std::cout << "Attempting to find factors using dependencies...\n" << std::endl;
for (size_t i = 0; i < dependencies.size(); ++i) {
    // Initialize 'a' and 'b'
    mpz_class a = 1;
    mpz_class b = 1;
    // Exponent vector for 'b'
    std::vector<int> total_exponents(factor_base.size(), 0);
    // Multiply corresponding x + m for 'a' and collect exponents for 'b'
    for (size_t j = 0; j < dependencies[i].size(); ++j) {
        if (dependencies[i][j] == 1) {
            int x = all_smooth_x[j];
            mpz_class x_plus_m = x + m;
            a = (a * x_plus_m) % n;
            // Sum exponents
            for (size_t k = 0; k < factor_base.size(); ++k) {
                total_exponents[k] += smooth_relations[j][k];
            }
        }
    }
    // Divide exponents by 2 for 'b' (since exponents are even)
    for (size_t k = 0; k < total_exponents.size(); ++k) {
        total_exponents[k] /= 2;
    }
    // Compute 'b' as the product of primes raised to the total_exponents
    for (size_t k = 0; k < factor_base.size(); ++k) {
        if (total_exponents[k] > 0) {
            mpz_class temp = factor_base[k];
            mpz_class temp_pow;
            mpz_pow_ui(temp_pow.get_mpz_t(), temp.get_mpz_t(), total_exponents[k]);
            b = (b * temp_pow) % n;
        }
    }
    // Compute gcd(a - b, n)
    mpz_class diff = a - b;
    if (diff < 0)
        diff += n;
    mpz_class gcd_value;
    mpz_gcd(gcd_value.get_mpz_t(), diff.get_mpz_t(), n.get_mpz_t());
    // Check if gcd is a non-trivial factor
    if (gcd_value > 1 && gcd_value < n) {
        mpz_class complementary_factor = n / gcd_value;
        std::cout << "Non-trivial factor found: " << gcd_value << std::endl;
        std::cout << "Complementary factor: " << complementary_factor << "\n" << std::endl;
        MPI_Finalize();
        return 0;
    }
}
```

```
            std::cout << "No non-trivial factors found with the
                current dependencies.\n" << std::endl;
        }
    }

    // Finalize MPI environment
    MPI_Finalize();
    return 0;
}
```

Listing 7: Main Function (Partial)

4 Challenges and Optimizations

4.1 Parallelization with MPI

Implementing parallel processing using MPI introduced several challenges:

- **Work Distribution**: Ensuring an even distribution of x values among processes to avoid idle processors.
- **Communication Overhead**: Managing data broadcasts and gathers efficiently to minimize communication time.
- **Synchronization**: Ensuring all processes have consistent data, especially the factor base, before proceeding with computations.

4.2 Handling Large Integers with GMP

Utilizing GMP allowed for precise arithmetic with large integers, but it required careful management:

- **Memory Management**: Ensuring that dynamically allocated memory (e.g., character arrays for broadcasting) was properly deallocated to prevent memory leaks.
- **Performance**: While GMP is efficient, operations on very large numbers can still be time-consuming. Optimizing the usage of GMP functions was necessary.

4.3 Gaussian Elimination Over GF(2)

Performing Gaussian elimination in the binary field introduced complexities:

- **Matrix Size**: As the number of smooth relations increases, the exponent matrix grows, making elimination more resource-intensive.
- **Dependency Detection**: Accurately identifying dependencies required a robust implementation to avoid missing critical relations.

4.4 Optimizations Applied

To enhance performance and scalability, several optimizations were implemented:

- **Efficient Sieving**: By distributing the sieving workload across multiple processes, the time to identify smooth numbers was significantly reduced.
- **Early Termination**: The program exits early upon finding non-trivial factors, saving computational resources.
- **Selective Output**: Limiting debug output to essential information helps in reducing I/O overhead during parallel execution.

5 For Compile

5.1 In linux :

mpic++ code.cpp -o code -lgmp -lgmpxx

5.2 In Mac :

mpic++ -std=c++17 code.cpp -o code -I/opt/homebrew/Cellar/gmp/6.3.0/include -L/opt/homebrew/Cellar/gmp/6.3.0/lib -lgmp -lgmpxx

6 For Run

6.1 In linux :

mpirun -np 4 ./submit

6.2 In Mac :

mpirun -np 5 ./submit

6.3 Performance Analysis

- **Execution Time**: The implementation efficiently factors small to medium-sized numbers. Execution time decreases with an increasing number of MPI processes.
- **Scalability**: The parallel approach scales well for larger numbers, provided sufficient smooth relations are found within the sieving range.
- **Limitations**: For very large numbers, the sieving range and factor base size may need to be increased, leading to higher memory and computational requirements.

7 Conclusion

The implementation of the Quadratic Sieve algorithm in C++ successfully factors composite numbers by leveraging MPI for parallel processing and GMP for handling large integers. While effective for numbers up to a certain size, further optimizations and enhancements are necessary to scale the algorithm for extremely large integers. Future work may include implementing advanced sieving techniques, optimizing Gaussian elimination, and exploring distributed storage solutions to handle larger datasets.