НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені Ігоря СІКОРСЬКОГО»
Навчально-науковий фізико-технічний інститут
Кафедра математичних методів захисту інформації

Звіт до
лабораторної №1 за темою
Algorithms for Arithmetic Operations on Large Numbers Over Finite Fields and Groups: GNU GMP Library in Parallel Computing Environments

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

October 1, 2025
м. Київ

1 Extended Description of Research Topic

1.1 Introduction

Multi-precision arithmetic operations on large numbers are fundamental to numerous computational domains including cryptography, number theory, computer algebra systems, and distributed computing applications. The GNU Multiple Precision Arithmetic Library (GMP) provides highly optimized implementations of arithmetic operations that exceed the native integer precision of standard computer architectures.

1.2 Research Context

This research investigates the efficiency of GMP arithmetic operations in parallel computing environments characterized by:

- Multiple processors (possibly multi-core)
- 64-bit architecture
- Memory capacity up to 128 GB RAM
- Application scenarios: transaction processing servers and cloud service provider equipment

1.3 Significance of Multi-Precision Arithmetic

Modern cryptographic protocols, such as RSA, elliptic curve cryptography (ECC), and post-quantum cryptographic algorithms, require arithmetic operations on integers with thousands of bits. Standard 64-bit integer types are insufficient for these applications, necessitating specialized libraries like GMP that can handle arbitrary-precision integers.

1.4 GNU GMP Library Overview

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers. Key features include:

- No practical limit on precision (constrained only by available memory)
- Highly optimized assembly implementations for many processor architectures
- Rich set of functions for arithmetic, number-theoretic, and bit manipulation operations
- Well-established performance as one of the fastest multi-precision libraries available

1.5 Parallel Computing Considerations

While GMP itself is not inherently parallelized at the function level, parallel computing environments can exploit GMP through:

- Task-level parallelism: distributing independent arithmetic operations across multiple cores
- Data parallelism: performing the same operations on different data sets simultaneously
- Pipeline parallelism: overlapping different stages of complex computations

1.6 Research Objectives

The primary objectives of this research are to:

1. Analyze the algorithmic foundations of GMP’s core arithmetic functions
2. Evaluate computational complexity in terms of time and space
3. Assess performance characteristics on 64-bit multi-core architectures
4. Identify optimization opportunities for parallel computing scenarios
5. Provide practical guidance for implementing GMP in high-performance computing environments

2 Multi-Precision Arithmetic Functions

2.1 Integer Initialization and Assignment Functions

2.1.1 mpz_init

Description: Initializes an integer variable and sets its value to 0.
Algorithm: Allocates memory for the limb array (typically one limb initially) and initializes metadata.

```c
void mpz_init(mpz_t x);
```

Complexity: $O(1)$ time, $O(1)$ space
Input: Uninitialized mpz_t variable (passed by reference)
Output: Initialized mpz_t variable with value 0
Return Code: void (no return value)

2.1.2 mpz_set

Description: Assigns the value of one integer to another.
Algorithm: Copies limbs from source to destination, reallocating if necessary.

```c
void mpz_set(mpz_t rop, const mpz_t op);
```

Complexity: $O(n)$ time where n is the number of limbs in op, $O(n)$ space
Input:

- rop: destination integer
- op: source integer

Output: rop contains copy of op
Return Code: void

2.2 Basic Arithmetic Operations

2.2.1 mpz_add

Description: Computes the sum of two integers: $rop = op1 + op2$
Algorithm: School-book addition with carry propagation. For each limb position i:

Algorithm 1 Multi-precision Addition

```
1: carry ← 0
2: for $i = 0$ to max$(n_1, n_2)$ do
3:     sum ← op1[i] + op2[i] + carry
4:     rop[i] ← sum mod $2^{64}$
5:     carry ← sum div $2^{64}$
6: end for
7: if carry $\neq 0$ then
8:     rop[max$(n_1, n_2)$] ← carry
9: end if
```

```c
void mpz_add(mpz_t rop, const mpz_t op1, const mpz_t op2);
```

Complexity: $O(n)$ time where $n = \max(\text{size}(op1), \text{size}(op2))$, $O(n)$ space
Input:

- rop: result integer
- op1: first operand
- op2: second operand

Output: rop contains op1 + op2
Return Code: void

2.2.2 mpz_sub

Description: Computes the difference: $rop = op1 - op2$
Algorithm: Similar to addition but with borrow propagation. Handles sign changes appropriately.

```c
void mpz_sub(mpz_t rop, const mpz_t op1, const mpz_t op2);
```

Complexity: $O(n)$ time, $O(n)$ space
Input: rop, op1, op2 (same as mpz_add)
Output: rop contains op1 - op2
Return Code: void

2.2.3 mpz_mul

Description: Computes the product: $rop = op1 \times op2$
Algorithm: GMP uses multiple algorithms depending on operand size:

- **Basecase multiplication** (small operands): School-book $O(n^2)$ algorithm
- **Karatsuba algorithm** (medium operands): Divide-and-conquer approach
- **Toom-Cook algorithm** (large operands): Generalized Karatsuba
- **FFT-based multiplication** (very large operands): Using Fast Fourier Transform

```c
void mpz_mul(mpz_t rop, const mpz_t op1, const mpz_t op2);
```

Complexity:

- Basecase: $O(n^2)$
- Karatsuba: $O(n^{\log_2 3}) \approx O(n^{1.585})$
- Toom-3: $O(n^{\log_3 5}) \approx O(n^{1.465})$
- FFT: $O(n \log n \log \log n)$

Space Complexity: $O(n)$ where n is sum of operand sizes
Input: rop, op1, op2
Output: rop contains op1 $\times$ op2
Return Code: void

Algorithm 2 Karatsuba Multiplication

```
1: function KARATSUBA($x, y, n$)
2:     if $n \leq$ threshold then
3:         return Basecase-Multiply($x, y$)
4:     end if
5:     $m \leftarrow \lceil n/2 \rceil$
6:     Split $x = x_1 \cdot 2^m + x_0$, $y = y_1 \cdot 2^m + y_0$
7:     $z_0 \leftarrow$ Karatsuba($x_0, y_0, m$)
8:     $z_2 \leftarrow$ Karatsuba($x_1, y_1, n - m$)
9:     $z_1 \leftarrow$ Karatsuba($x_0 + x_1, y_0 + y_1, m + 1$) - $z_0$ - $z_2$
10:    return $z_2 \cdot 2^{2m} + z_1 \cdot 2^m + z_0$
11: end function
```

2.2.4 mpz_div (mpz_tdiv_q)

Description: Computes quotient: $q = \lfloor n/d \rfloor$
Algorithm: GMP uses several division algorithms:

- **Schoolbook division**: For small divisors
- **Divide-and-conquer division**: Based on recursive techniques
- **Barrett reduction**: For modular arithmetic
- **Exact division**: Optimized when remainder is known to be zero

```c
void mpz_tdiv_q(mpz_t q, const mpz_t n, const mpz_t d);
void mpz_tdiv_r(mpz_t r, const mpz_t n, const mpz_t d);
void mpz_tdiv_qr(mpz_t q, mpz_t r, const mpz_t n, const mpz_t d);
```

Complexity: $O(n \cdot m)$ for schoolbook where n is dividend size and m is divisor size; improved algorithms achieve $O(M(n))$ where $M(n)$ is multiplication time
Input:

- q: quotient result
- r: remainder result (for qr variant)
- n: dividend
- d: divisor

Output: q contains quotient, r contains remainder
Return Code: void

2.3 Modular Arithmetic Operations

2.3.1 mpz_mod

Description: Computes $r = n \mod d$ with non-negative result

```c
void mpz_mod(mpz_t r, const mpz_t n, const mpz_t d);
```

Algorithm: Performs division and returns remainder, adjusting sign if necessary
Complexity: $O(n \cdot m)$ similar to division
Input: r (result), n (dividend), d (divisor)
Output: r contains n mod d
Return Code: void

2.3.2 mpz_powm

Description: Modular exponentiation: $rop = base^{exp} \mod mod$
Algorithm: Binary exponentiation (square-and-multiply) with modular reduction

Algorithm 3 Modular Exponentiation

```
1: function ModPow(base, exp, mod)
2:     result ← 1
3:     base ← base mod mod
4:     while exp > 0 do
5:         if exp is odd then
6:             result ← (result × base) mod mod
7:         end if
8:         exp ← exp ≫ 1
9:         base ← (base × base) mod mod
10:    end while
11:    return result
12: end function
```

```c
void mpz_powm(mpz_t rop, const mpz_t base, const mpz_t exp,
              const mpz_t mod);
```

Complexity: $O(k \cdot M(n))$ where k is bit-length of exponent and $M(n)$ is multiplication complexity
Input: rop (result), base, exp (exponent), mod (modulus)
Output: rop contains base^{exp} mod mod
Return Code: void

2.3.3 mpz_invert

Description: Computes modular multiplicative inverse: $rop \cdot op \equiv 1 \pmod{mod}$
Algorithm: Extended Euclidean Algorithm

```c
int mpz_invert(mpz_t rop, const mpz_t op, const mpz_t mod);
```

Complexity: $O(n^2)$ where n is size of modulus
Input: rop (result), op (value to invert), mod (modulus)
Output: rop contains multiplicative inverse if it exists
Return Code: Non-zero if inverse exists, 0 if inverse does not exist (when gcd(op, mod) $\neq$ 1)

Algorithm 4 Extended Euclidean Algorithm

```
1: function ExtGCD(a, b)
2:     if $b = 0$ then
3:         return (a, 1, 0)
4:     end if
5:     (g, x', y') ← ExtGCD(b, a mod b)
6:     x ← y'
7:     y ← x' - $\lfloor a/b \rfloor$ · y'
8:     return (g, x, y)
9: end function
```

Algorithm 5 Euclidean Algorithm

```
1: function GCD(a, b)
2:     while $b \neq 0$ do
3:         temp ← b
4:         b ← a mod b
5:         a ← temp
6:     end while
7:     return a
8: end function
```

2.4 Number-Theoretic Functions

2.4.1 mpz_gcd

Description: Computes greatest common divisor: $rop = \gcd(op1, op2)$
Algorithm: Binary GCD (Stein's algorithm) or Euclidean algorithm

```c
void mpz_gcd(mpz_t rop, const mpz_t op1, const mpz_t op2);
```

Complexity: $O(n^2)$ for Euclidean algorithm where n is bit length
Input: rop (result), op1, op2
Output: rop contains gcd(op1, op2)
Return Code: void

2.4.2 mpz_probab_prime_p

Description: Probabilistic primality test
Algorithm: Miller-Rabin primality test with specified number of rounds

```c
int mpz_probab_prime_p(const mpz_t n, int reps);
```

Complexity: $O(k \cdot \log^3 n)$ where k is number of rounds
Input: n (number to test), reps (number of Miller-Rabin rounds)
Output: Integer result
Return Code:

- 2: definitely prime
- 1: probably prime
- 0: definitely composite

Algorithm 6 Miller-Rabin Primality Test

```
1: function MillerRabin(n, k)
2:     Write $n - 1 = 2^r \cdot d$ with d odd
3:     for $i = 1$ to $k$ do
4:         Choose random $a \in [2, n - 2]$
5:         $x \leftarrow a^d \mod n$
6:         if $x = 1$ or $x = n - 1$ then
7:             continue
8:         end if
9:         for $j = 1$ to $r - 1$ do
10:            $x \leftarrow x^2 \mod n$
11:            if $x = n - 1$ then
12:                continue outer loop
13:            end if
14:        end for
15:        return composite
16:    end for
17:    return probably prime
18: end function
```

2.5 Bitwise and Logical Operations

2.5.1 mpz_and, mpz_ior, mpz_xor

Description: Bitwise AND, OR, XOR operations
Algorithm: Limb-by-limb bitwise operations with sign handling

```
void mpz_and(mpz_t rop, const mpz_t op1, const mpz_t op2);
void mpz_ior(mpz_t rop, const mpz_t op1, const mpz_t op2);
void mpz_xor(mpz_t rop, const mpz_t op1, const mpz_t op2);
```

Complexity: $O(n)$ where $n = \max(\text{size}(op1), \text{size}(op2))$
Input: rop (result), op1, op2
Output: rop contains bitwise operation result
Return Code: void

2.5.2 mpz_popcount

Description: Counts the number of 1-bits (population count)
Algorithm: Uses processor-specific POPCNT instruction or lookup table

```
mp_bitcnt_t mpz_popcount(const mpz_t op);
```

Complexity: $O(n)$ where n is number of limbs
Input: op (integer to count)
Output: Number of 1-bits
Return Code: mp_bitcnt_t (unsigned long)

2.6 Comparison Functions

2.6.1 mpz_cmp

Description: Compares two integers
Algorithm: Compares signs first, then compares limbs from most significant

```c
int mpz_cmp(const mpz_t op1, const mpz_t op2);
```

Complexity: $O(n)$ worst case, but often $O(1)$ if sizes or early limbs differ
Input: op1, op2
Output: Integer comparison result
Return Code:

- Positive value if op1 $>$ op2
- 0 if op1 $=$ op2
- Negative value if op1 $<$ op2

3 Complexity Estimates Summary

Table 1: Time Complexity of GMP Operations

Operation	Function	Time Complexity
Addition	mpz_add	$O(n)$
Subtraction	mpz_sub	$O(n)$
Multiplication	mpz_mul	$O(n^{1.465})$ to $O(n \log n \log \log n)$
Division	mpz_tdiv_q	$O(n \cdot m)$ to $O(M(n))$
Modulo	mpz_mod	$O(n \cdot m)$
Mod. Exponentiation	mpz_powm	$O(k \cdot M(n))$
Mod. Inverse	mpz_invert	$O(n^2)$
GCD	mpz_gcd	$O(n^2)$
Primality Test	mpz_probab_prime_p	$O(k \cdot \log^3 n)$
Bitwise Operations	mpz_and/ior/xor	$O(n)$
Comparison	mpz_cmp	$O(1)$ to $O(n)$

Where:

- n = number of limbs (64-bit words) in operands
- m = number of limbs in second operand (for asymmetric operations)
- k = number of bits in exponent or number of test rounds
- $M(n)$ = complexity of multiplication (varies by algorithm)

3.1 Space Complexity

Most GMP operations have space complexity of $O(n)$ where n is the size of the result. Multiplication and division may require temporary storage of $O(n)$ additional limbs.

3.2 Parallel Computing Considerations

3.2.1 Task-Level Parallelism

For transaction processing servers handling multiple independent cryptographic operations:

- Independent GMP operations can execute simultaneously on different cores
- Each thread should maintain its own mpz_t variables to avoid contention
- Memory allocation patterns should be optimized for NUMA architectures

3.2.2 Performance on 64-bit Architecture

GMP is highly optimized for 64-bit systems:

- Native limb size matches register width (64 bits)
- Assembly-optimized routines for x86-64, ARM64, etc.
- Efficient use of SIMD instructions where applicable

3.2.3 Memory Management with 128 GB RAM

For large-scale operations:

- GMP can handle integers with millions of bits
- Memory allocation overhead becomes significant for very large numbers
- Custom allocators can improve performance in memory-intensive scenarios

4 Input and Output Data Specifications

4.1 Data Type: mpz_t

The primary data type used in GMP integer operations is `mpz_t`, which is defined as:

```c
typedef struct {
    int _mp_alloc;  // Number of limbs allocated
    int _mp_size;   // Number of limbs used (sign in sign bit)
    mp_limb_t *_mp_d;  // Pointer to limb array
} __mpz_struct;

typedef __mpz_struct mpz_t[1];
```

4.2 Limb Representation

On 64-bit architectures, `mp_limb_t` is typically:

```c
typedef unsigned long mp_limb_t;  // 64 bits
```

4.3 Input Data Requirements

4.3.1 Initialization Requirements

All `mpz_t` variables must be initialized before use:

```c
mpz_t x;
mpz_init(x);  // Must be called before any operations
```

4.3.2 Input Formats

GMP supports multiple input formats:

- Direct assignment from C types: `mpz_set_ui`, `mpz_set_si`
- String conversion: `mpz_set_str` (supports bases 2-62)
- Import from binary data: `mpz_import`

  Example:

```c
mpz_t n;
mpz_init(n);
mpz_set_str(n, "123456789012345678901234567890", 10);
```

4.4 Output Data Formats

4.4.1 Conversion to C Types

```c
unsigned long mpz_get_ui(const mpz_t op);
long mpz_get_si(const mpz_t op);
double mpz_get_d(const mpz_t op);
```

4.4.2 String Output

```c
char* mpz_get_str(char *str, int base, const mpz_t op);
```

4.4.3 Binary Export

```c
void mpz_export(void *rop, size_t *countp, int order,
                size_t size, int endian, size_t nails,
                const mpz_t op);
```

4.5 Memory Management

4.5.1 Cleanup

All initialized `mpz_t` variables must be cleared to free memory:

```c
mpz_clear(x);
```

4.5.2 Memory Usage Estimation

For an integer with b bits:

- Number of limbs: $n = \lceil b/64 \rceil$
- Memory usage: approximately $8n + 16$ bytes (on 64-bit systems)

5 Return Codes and Error Handling

5.1 Return Code Categories

5.1.1 Void Functions (No Return)

Most GMP arithmetic functions return `void`:

```
void mpz_add(mpz_t rop, const mpz_t op1, const mpz_t op2);
void mpz_mul(mpz_t rop, const mpz_t op1, const mpz_t op2);
```

These functions always succeed (assuming proper initialization and sufficient memory).

5.1.2 Integer Return Values

mpz_invert

```
int mpz_invert(mpz_t rop, const mpz_t op, const mpz_t mod);
```

Returns:

- Non-zero (true) if inverse exists
- 0 (false) if inverse does not exist (gcd(op, mod) $\neq$ 1)

mpz_probab_prime_p

```
int mpz_probab_prime_p(const mpz_t n, int reps);
```

Returns:

- 2 if n is definitely prime
- 1 if n is probably prime (high probability)
- 0 if n is definitely composite

mpz_cmp (and variants)

```
int mpz_cmp(const mpz_t op1, const mpz_t op2);
```

Returns:

- Positive value if op1 $>$ op2
- 0 if op1 $=$ op2
- Negative value if op1 $<$ op2

mpz_sgn

```
int mpz_sgn(const mpz_t op);
```

Returns:

- $+1$ if op > 0
- 0 if op $= 0$
- -1 if op < 0

5.1.3 Size and Count Returns

mpz_sizeinbase

```
size_t mpz_sizeinbase(const mpz_t op, int base);
```

Returns: Number of digits in given base (approximate for non-power-of-2 bases)

mpz_popcount

```
mp_bitcnt_t mpz_popcount(const mpz_t op);
```

Returns: Number of 1-bits in the binary representation

5.2 Error Handling

5.2.1 Memory Allocation Failures

GMP uses a configurable memory allocation mechanism. By default, it uses `malloc/realloc/free`. If allocation fails:

- GMP calls an error handler function
- Default behavior: prints error message and calls `abort()`
- Custom handlers can be installed via `mp_set_memory_functions`

5.2.2 Division by Zero

Operations like `mpz_div`, `mpz_mod`, and `mpz_invert` with zero divisor:

- GMP behavior: calls error handler (typically aborts)
- Recommendation: validate divisors before operations in production code

5.2.3 Invalid Function Arguments

GMP assumes:

- All `mpz_t` variables are properly initialized
- Function preconditions are met (e.g., positive modulus for `mpz_mod`)
- Violation leads to undefined behavior

5.3 Return Code Summary Table

5.4 Thread Safety Considerations

- GMP functions are generally thread-safe when operating on separate `mpz_t` variables
- No internal global state (except custom memory allocators)
- Each thread should maintain its own set of variables
- Synchronization required only when sharing `mpz_t` variables across threads

Table 2: GMP Function Return Types

Function	Return Type	Return Values
mpz_add, mpz_sub, mpz_mul	void	N/A
mpz_div, mpz_mod	void	N/A
mpz_invert	int	0 (no inverse) or non-zero (success)
mpz_probab_prime_p	int	0 (composite), 1 (probable), 2 (prime)
mpz_cmp	int	$< 0, = 0, > 0$
mpz_sgn	int	$-1, 0, +1$
mpz_sizeinbase	size_t	Number of digits
mpz_popcount	mp_bitcnt_t	Number of 1-bits
mpz_get_ui	unsigned long	Integer value
mpz_get_si	long	Integer value
mpz_get_str	char*	Pointer to string

6 Performance Analysis for Parallel Computing Environments

6.1 Hardware Architecture Considerations

6.1.1 64-bit Architecture Advantages

- Native 64-bit limb operations without software emulation
- Full utilization of 64-bit registers (RAX, RBX, etc.)
- Efficient multiply-with-carry instructions (MUL, IMUL, ADC)
- Large address space for handling massive integers

6.1.2 Multi-Core Utilization

For transaction processing servers:

Scenario 1: Independent Cryptographic Operations

```c
// Thread function for parallel RSA signature verification
void* verify_signature(void* arg) {
    signature_data* data = (signature_data*)arg;
    mpz_t signature, message, n, e, result;

    mpz_init(signature);
    mpz_init(message);
    mpz_init(n);
    mpz_init(e);
    mpz_init(result);

    // Load signature and public key
    mpz_import(signature, data->sig_len, 1, 1, 0, 0, data->sig);
    mpz_set(n, data->modulus);
    mpz_set(e, data->exponent);

    // Verify: result = signature^{e} mod n
    mpz_powm(result, signature, e, n);

    // Compare with message hash
    int valid = (mpz_cmp(result, data->hash) == 0);

    mpz_clear(signature);
    mpz_clear(message);
    mpz_clear(n);
    mpz_clear(e);
    mpz_clear(result);

    return (void*)(long)valid;
}
```

Scalability: Nearly linear with number of cores for independent operations

6.1.3 NUMA Architecture Optimization

For systems with 128 GB RAM distributed across NUMA nodes:

- Allocate thread-local mpz_t variables on local NUMA node
- Use `numa_alloc_local()` for custom GMP allocator
- Minimize cross-node memory access
- Pin threads to specific cores/nodes for consistent performance

6.2 Cache Optimization

6.2.1 L1/L2/L3 Cache Behavior

- Small integers ($<$ 1024 bits): typically fit in L1/L2 cache
- Medium integers (1024-8192 bits): L3 cache friendly
- Large integers ($>$ 8192 bits): dominated by memory bandwidth

6.2.2 Cache-Friendly Patterns

```
// Good: Process batch of operations with same modulus
mpz_t mod, base[1000], result[1000];
mpz_init(mod);
mpz_set_str(mod, "large_prime_modulus", 10);

for (int i = 0; i < 1000; i++) {
    mpz_init(base[i]);
    mpz_init(result[i]);
    // Set base[i] values
}

// Modulus stays hot in cache
for (int i = 0; i < 1000; i++) {
    mpz_powm_ui(result[i], base[i], 65537, mod);
}
```

6.3 Memory Bandwidth Considerations

For very large integers (millions of bits), performance is memory-bound:

Table 3: Approximate Performance on Modern 64-bit Server

Integer Size	Operation	Approx. Time
1024 bits	Multiplication	$\sim$1 μs
2048 bits	Multiplication	$\sim$3 μs
4096 bits	Multiplication	$\sim$10 μs
1024 bits	Modular Exponentiation	$\sim$500 μs
2048 bits	Modular Exponentiation	$\sim$3 ms
4096 bits	Modular Exponentiation	$\sim$20 ms

6.4 Practical Implementation Guidelines

6.4.1 Thread Pool Architecture

```c
typedef struct {
    mpz_t* work_items;
    int count;
    mpz_t modulus;
} thread_task;

void* worker_thread(void* arg) {
    thread_task* task = (thread_task*)arg;

    // Thread-local temporary variables
    mpz_t temp;
    mpz_init(temp);

    for (int i = 0; i < task->count; i++) {
        // Process each work item
        mpz_powm_ui(task->work_items[i],
                    task->work_items[i],
                    65537,
                    task->modulus);
    }

    mpz_clear(temp);
    return NULL;
}
```

6.4.2 Memory Pre-allocation Strategy

```c
// Pre-allocate to avoid reallocation during computation
mpz_t large_number;
mpz_init2(large_number, 8192);  // Reserve space for 8192 bits
```

6.5 Benchmark Recommendations

For evaluating GMP performance in your specific environment:

1. **Single-threaded baseline:** Measure core operation times
2. **Multi-threaded scaling:** Test with 1, 2, 4, 8, 16+ threads
3. **Memory pressure:** Test with varying integer sizes
4. **NUMA effects:** Compare local vs. remote memory access
5. **Cache behavior:** Measure performance with hot vs. cold data

7 Conclusion

7.1 Summary of Findings

This research has presented a comprehensive analysis of GNU GMP library functions for multi-precision arithmetic in parallel computing environments. Key findings include:

- GMP provides asymptotically optimal algorithms for all major arithmetic operations
- Complexity ranges from $O(n)$ for addition to $O(n \log n \log \log n)$ for FFT-based multiplication
- The library is highly optimized for 64-bit architectures with assembly implementations
- Task-level parallelism enables near-linear scaling for independent operations
- Memory management is critical for performance in large-scale deployments

7.2 Suitability for Transaction Processing

For transaction processing servers and cloud service providers:

Advantages:

- Mature, well-tested library with decades of optimization
- Excellent single-threaded performance
- Straightforward parallelization of independent operations
- Efficient memory usage for typical cryptographic operations

Considerations:

- No built-in function-level parallelism (e.g., parallel multiplication)
- Memory allocation overhead for very frequent small operations
- NUMA-awareness requires custom memory allocator

7.3 Optimization Recommendations

7.3.1 For High-Throughput Scenarios

1. Use thread pools with pre-initialized mpz_t variables
2. Batch operations with common moduli to improve cache utilization
3. Consider custom memory allocators for NUMA systems
4. Profile memory access patterns to minimize cross-node traffic

7.3.2 For Low-Latency Scenarios

1. Pre-allocate mpz_t variables with expected maximum size
2. Keep frequently used values (e.g., moduli, public keys) in cache
3. Use fixed-size integers when precision requirements are known
4. Avoid dynamic memory allocation in critical paths

7.4 Future Research Directions

- Investigation of GPU-accelerated multi-precision arithmetic
- Comparative analysis with other libraries (MPFR, Boost.Multiprecision)
- Development of GMP-based parallel algorithms for advanced operations
- Integration with hardware cryptographic accelerators
- Performance modeling for cloud-native deployments

7.5 Conclusion

GNU GMP remains the gold standard for multi-precision arithmetic in high-performance computing environments. Its combination of algorithmic sophistication, architecture-specific optimization, and flexibility makes it ideally suited for transaction processing servers and cloud services requiring cryptographic operations on large integers. With proper parallel computing strategies, GMP can efficiently utilize modern multi-core 64-bit architectures with large memory capacities, delivering both high throughput and low latency for demanding applications.

References

[1] Granlund, T., and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. Edition 6.3.0, 2023. https://gmplib.org/

[2] Knuth, D. E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd edition, Addison-Wesley, 1997.

[3] Karatsuba, A., and Ofman, Y. "Multiplication of Multidigit Numbers on Automata." *Soviet Physics Doklady*, Vol. 7, 1963, pp. 595-596.

[4] Toom, A. L. "The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers." *Soviet Mathematics Doklady*, Vol. 3, 1963, pp. 714-716.

[5] Schönhage, A., and Strassen, V. "Schnelle Multiplikation großer Zahlen." *Computing*, Vol. 7, 1971, pp. 281-292.

[6] Miller, G. L. "Riemann's Hypothesis and Tests for Primality." *Journal of Computer and System Sciences*, Vol. 13, 1976, pp. 300-317.

[7] Rabin, M. O. "Probabilistic Algorithm for Testing Primality." *Journal of Number Theory*, Vol. 12, 1980, pp. 128-138.

[8] Montgomery, P. L. "Modular Multiplication Without Trial Division." *Mathematics of Computation*, Vol. 44, 1985, pp. 519-521.

[9] Knuth, D. E. "Analysis of Euclid's Algorithm." *The Art of Computer Programming, Volume 2*, Section 4.5.3, 1997.

[10] Brent, R. P., and Zimmermann, P. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[11] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of Applied Cryptography*. CRC Press, 1996.

[12] Lameter, C. "NUMA (Non-Uniform Memory Access): An Overview." *ACM Queue*, Vol. 11, 2013.