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:

Алгоритми арифметичних операцій над великими числами над скінченними полями та групами: бібліотека GNU GMP у паралельних обчислювальних середовищах

Оформлення звіту:
Юрчук Олексій, ФІ-52мн

2 жовтня 2025 р.
м. Київ

1 Опис теми дослідження

1.1 Вступ

Багатоточні арифметичні операції над великими числами є фундаментальними для численних обчислювальних областей, включаючи криптографію, теорію чисел, систем комп'ютерної алгебри та розподілені обчислювальні додатки. Бібліотека GNU Multiple Precision Arithmetic Library (GMP) надає високооптимізовані реалізації арифметичних операцій, що перевищують вбудовану цілочисельну точність стандартних комп'ютерних архітектур.

1.2 Research Context

Моє дослідження впродовж лабораторної спрямовано на вивчення ефективності арифметичних операцій GMP при паралельних обчислювальних середовищах, що характеризуються:

- Наявністю декількох процесорів (можливо, багатоядерних)
- 64-бітна архітектура
- Об'єм пам'яті to 128 GB RAM
- Сценарії застосування: сервери обробки транзакцій та обладнання постачальників хмарних послуг

1.3 Значимість багаторозрядної арифметики

Сучасні криптографічні протоколи, такі як RSA, криптографія на еліптичних кривих (ECC) та постквантові криптографічні алгоритми, вимагають арифметичних операцій над цілими числами з тисячами бітів. Стандартні 64-бітні типи цілих чисел є недостатніми для цих застосувань, що вимагає використання спеціалізованих бібліотек, таких як GMP, що можуть обробляти цілі числа з довільною точністю.

1.4 Загальний огляд бібліотеки GNU GMP

GMP – це безкоштовна бібліотека для арифметики довільної точності, що працює з цілими числами зі знаком, раціональними числами та числами з плаваючою крапкою. Основні функції якої це:

- Відсутність практичних обмежень по точності (обмеження лише за доступною пам'яттю)
- Високооптимізовані асемблерні реалізації для багатьох архітектур процесорів
- Широкий набір функцій для арифметичних, теоретико-чисельних та бітових операцій
- Добре зарекомендована бібілотека по продуктивності (одна з найшвидших бібліотек з багаторазовою точністю)

1.5 Міркування про паралельні обчислення

Хоча GMP сама по собі не є розпаралеленою на рівні функцій, паралельні обчислювальні середовища можуть використовувати GMP за допомогою:

- Task-level parallelism: розподіл незалежних арифметичних операцій між декількома ядрами
- Data parallelism: одночасне виконання однакових операцій над різними наборами даних
- Pipeline parallelism: перекриття різних етапів складних обчислень

1.6 Мета мого дослідження

Основними цілями цього дослідження є:

1. Аналіз алгоритмічних основ основних арифметичних функцій GMP
2. Оцінка обчислювальної складності з точки зору часу та пам’яті
3. Оцінка характеристик продуктивності на 64-бітних багатоядерних архітектурах
4. Визначення можливостей оптимізації для сценаріїв паралельних обчислень
5. Надання практичних рекомендацій щодо впровадження GMP у високопродуктивних обчислювальних середовищах

2 Арифметичні функції з багаторозрядною точністю

2.1 Integer Initialization and Assignment Functions

2.1.1 mpz_init

Опис: Ініціалізує цілочисельну змінну та встановлює її значення на 0.
Алгоритм: Виділяє пам'ять для limb array (зазвичай спочатку один limb) та ініціалізує метадані.

Зауваження:. *Лімб є основним будівельним блоком для представлення цілих чисел довільної точності в GMP. Його можна уявити як "цифру" в арифметиці з основою* 2^{64}*. Так само, як десяткове число 12345 може бути представлене як масив цифр* $[1, 2, 3, 4, 5]$*, велике ціле число в GMP представлене як масив лімбів.*
Приклад: *192-бітне ціле число потребує трьох лімбів (3 × 64 = 192 біти):*

- *limb 0: біти 0-63 (найменш значущі)*
- *limb 1: біти 64-127*
- *limb 2: біти 128-191 (найстарші)*

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

Складність: $O(1)$ time, $O(1)$ space
Вхід: Неініціалізована змінна mpz_t (передається за посиланням)
Вихід: Ініціалізована змінна mpz_t із значенням 0
Return Code: void (no return value)

2.1.2 mpz_set

Опис: Присвоює значення одного цілого числа іншому.
Алгоритм: Копіює limbs з source в destination, перерозподіляючи їх у разі необхідності.

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

Складність: $O(n)$ time, where n кількість limbs in op, $O(n)$ space
Вхід:

- rop: destination integer
- op: source integer

Вихід: rop містить копію op
Return Code: void (no return value)

2.2 Основні арифметичні операції

2.2.1 mpz_add

Опис: Обчислює суму двох цілих чисел: $rop = op1 + op2$
Алгоритм: "Шкільне" додавання з перенесення carry. Для кожного limb position i:

Algorithm 1 Multi-precision Addition

carry $\leftarrow 0$
for $i = 0$ to $\max(n_1, n_2)$ **do**
    sum $\leftarrow$ op1[i] + op2[i] + carry
    rop[i] $\leftarrow$ sum mod 2^{64}
    carry $\leftarrow$ sum div 2^{64}
end for
if carry $\neq 0$ **then**
    rop[$\max(n_1, n_2)$] $\leftarrow$ carry
end if

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

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

- rop: змінна-результат цілого типу
- op1: перший операнд
- op2: другий операнд

Вихід: rop містить суму $op1 + op2$
Return Code: void (no return value)

2.2.2 mpz_sub

Опис: Обчислює різницю: $rop = op1 - op2$
Алгоритм: Подібний до додавання, але з позичанням borrow. Також обробляє зміну знака.

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

Складність: $O(n)$ time, $O(n)$ space
Вхід: rop, op1, op2 (такий же як в mpz_add)
Вихід: rop містить op1 - op2
Return Code: void (no return value)

2.2.3 mpz_mul

Опис: Обчислює добуток: $rop = op1 \times op2$
Алгоритм: GMP використовує кілька алгоритмів залежно від operand size:

- **Basecase multiplication** (малі операнди): ”щкільне” множення, алгоритм $O(n^2)$
- **Karatsuba algorithm** (середні операнди): підхід ”розділяй і володарюй”
- **Toom-Cook algorithm** (великі операнди): узагальнений алгоритм Карацуби
- **FFT-based multiplication** (дуже великі операнди): використання швидкого перетворення Фур’є

Algorithm 2 Karatsuba Multiplication

function KARATSUBA(x, y, n)
  if $n \leq$ threshold **then**
    return Basecase-Multiply(x, y)
  end if
  $m \leftarrow \lceil n/2 \rceil$
  Split $x = x_1 \cdot 2^m + x_0$, $y = y_1 \cdot 2^m + y_0$
  $z_0 \leftarrow$ Karatsuba(x_0, y_0, m)
  $z_2 \leftarrow$ Karatsuba($x_1, y_1, n - m$)
  $z_1 \leftarrow$ Karatsuba($x_0 + x_1, y_0 + y_1, m + 1$) - z_0 - z_2
  return $z_2 \cdot 2^{2m} + z_1 \cdot 2^m + z_0$
end function

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

Складність:

- Базовий випадок: $O(n^2)$
- Карацуба: $O(n^{\log_2 3}) \approx O(n^{1.585})$
- Тоом-Кук: $O(n^{\log_3 5}) \approx O(n^{1.465})$
- FFT: $O(n \log n \log \log n)$

Space Complexity: $O(n)$ where n – сума of operand sizes
Вхід: rop, op1, op2
Вихід: rop містить результат op1 $\times$ op2
Return Code: void (no return value)

2.2.4 mpz_div (mpz_tdiv_q)

Опис: Обчислення частки: $q = \lfloor n/d \rfloor$
Алгоритм: GMP використовує кілька алгоритмів ділення:

- **”Шкільне” ділення**: для малих дільників
- **Ділення методом ”розділяй і володарюй”**: на основі рекурсивних методів
- **Редукція за Барретом**: для модулярної арифметики
- **Точне ділення**: оптимізоване, коли лишок рівний нулю

```
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);
```

Складність: $O(n \cdot m)$ для ”шкільного”, де n – dividend size та m – divisor size; вдосконалені алгоритми досягають складності $O(M(n))$, де $M(n)$ – час множення
Вхід:

- q: результат ділення

- r: лишок (для варіанту qr)
- n: ділене
- d: дільник

Вихід: q містить quotient, r містить remainder
Return Code: void (no return value)

2.3 Модулярні арифметичні операції

2.3.1 mpz_mod

Опис: Обчислює $r = n \mod d$ з невід'ємним результатом

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

Алгоритм: Виконує ділення і повертає лишок від ділення, коригуючи знак, якщо необхідно
Складність: $O(n \cdot m)$ подібно до звичаного ділення
Вхід: r (result), n (dividend), d (divisor)
Вихід: r містить n mod d
Return Code: void (no return value)

2.3.2 mpz_powm

Опис: Модулярне піднесення до степеня: $rop = base^{exp} \mod mod$
Алгоритм: Бінарне піднесення до степеня (square-and-multiply) з модулярною редукцією

Algorithm 3 Modular Exponentiation

```
function ModPow(base, exp, mod)
    result ← 1
    base ← base mod mod
    while exp > 0 do
        if exp is odd then
            result ← (result × base) mod mod
        end if
        exp ← exp ≫ 1
        base ← (base × base) mod mod
    end while
    return result
end function
```

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

Складність: $O(k \cdot M(n))$, де k – бітова довжина експоненти та $M(n)$ – складність операції множення
Вхід: rop (result), base, exp (exponent), mod (modulus)
Вихід: rop містить $base^{exp}$ за mod
Return Code: void (no return value)

2.3.3 mpz_invert

Опис: Обчислює обернене за операцією множення: $rop \cdot op \equiv 1 \pmod{mod}$
Алгоритм: Розширений алгоритм Евкліда

Algorithm 4 Extended Euclidean Algorithm

```
function ExtGCD(a, b)
    if b = 0 then
        return (a, 1, 0)
    end if
    (g, x', y') ← ExtGCD(b, a mod b)
    x ← y'
    y ← x' - ⌊a/b⌋ · y'
    return (g, x, y)
end function
```

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

Складність: $O(n^2)$, де n – size of modulus
Вхід: rop (result), op (value to invert), mod (modulus)
Вихід: rop містить обернене значення до op, якщо воно взагалі існує
Return Code: Відмінне від нуля, якщо обернене існує, та 0 якщо не існує (gcd(op, mod) $\neq 1$)

2.4 Числові теоретичні функції

2.4.1 mpz_gcd

Опис: Обчислює найбільший спільний дільник: $rop = \gcd(op1, op2)$
Алгоритм: Бінарний GCD (алгоритм Стейна) or алгоритм Евкліда

Algorithm 5 Euclidean Algorithm

```
function GCD(a, b)
    while b ≠ 0 do
        temp ← b
        b ← a mod b
        a ← temp
    end while
    return a
end function
```

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

Складність: $O(n^2)$ для алгоритму Евкліда, де n – довжина в бітах
Вхід: rop (result), op1, op2
Вихід: rop містить gcd(op1, op2)
Return Code: void (no return value)

2.4.2 mpz_probab_prime_p

Опис: Імовірнісний тест на простоту

Алгоритм: Тест на простоту Міллера-Рабіна із заданою наперед кількістю раундів

Algorithm 6 Miller-Rabin Primality Test

```
1:  function MillerRabin(n, k)
2:      Write n − 1 = 2^r · d with d odd
3:      for i = 1 to k do
4:          Choose random a ∈ [2, n − 2]
5:          x ← 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 ← 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
```

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

Складність: $O(k \cdot \log^3 n)$, де k – кількість раундів
Вхід: n (число для перевірки), reps (кількість раундів у схемі Міллера-Рабіна)
Вихід: Цілочисельний результат
Return Code:

- 2: однозначно просте
- 1: ймовірно просте
- 0: однозначно складене

2.5 Бітові та логічні операції

2.5.1 mpz_and, mpz_ior, mpz_xor

Опис: Bitwise AND, OR, XOR operations

Алгоритм: Limb-by-limb побітові операції з обробкою знаків

```c
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);
```

Складність: $O(n)$, де $n = \max(\text{size}(op1), \text{size}(op2))$
Вхід: rop (result), op1, op2
Вихід: rop місить результат побітової інформації
Return Code: void (no return value)

2.5.2 mpz_popcount

Опис: Підраховує кількість одиничних бітів в двійковому представлені (Hamming weight)
Алгоритм: Використовує специфічну для процесора інструкцію POPCNT або таблицю перегляду

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

Складність: $O(n)$, де n – кількість of limbs
Вхід: op (ціле число)
Вихід: Кількість of 1-bits
Return Code: mp_bitcnt_t (unsigned long)

2.6 Функції порівняння

2.6.1 mpz_cmp

Опис: Порівнює два цілих числа
Алгоритм: Спочатку порівнює знаки, потім порівнює limbs(розряди), починаючи з найстаршого

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

Складність: $O(n)$ – у найгіршому випадку, але часто $O(1)$, якщо ромір або ранні limbs відрізняються
Вхід: op1, op2
Вихід: Цілочисельне значеня
Return Code:

- Позитивне значення, якщо op1 $>$ op2
- 0, якщо op1 $=$ op2
- Негативне значення, якщо op1 $<$ op2

3 Summary по оцінках складності

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 $O(n)$

Таблиця 1: Часова складність операцій GMP

Де:

- n = кількість of limbs (64-bit слів) в операндах
- m = кількість of limbs у другому операнді (для асиметричних операцій)
- k = кількість бітів в експоненті або кількості тестових раундів
- $M(n)$ = складність множення (варіюється залежно від алгоритму)

3.1 Space Complexity

Більшість операцій GMP мають space complexity of $O(n)$, де n – розмір результату. Множення та ділення можуть вимагати тимчасового зберігання додаткових limbs $O(n)$.

3.2 Роздуми про паралельні обчислення

3.2.1 Task-Level Parallelism

Для серверів обробки транзакцій, що обробляють кілька незалежних криптографічних операцій:

- Незалежні операції GMP можуть виконуватися одночасно на різних ядрах
- Кожен потік повинен підтримувати власні змінні mpz_t, щоб уникнути конфліктів
- Memory allocation patterns мають бути оптимізовані для NUMA architectures

3.2.2 Продуктивність на 64-бітній архітектурі

GMP є високооптимізованим для 64-бітних систем:

- Розмір Native limb відповідає ширині регістра (64 біти)
- Оптимізовані під асемблер процедури для x86-64, ARM64 тощо
- Ефективне використання інструкцій SIMD, де це можливо

3.2.3 Управління пам’яттю з 128 GB RAM

Для великорозмірних операцій:

- GMP може обробляти цілі числа з мільйонами бітів
- Розподіл пам’яті стає значним для дуже великих чисел
- Custom allocators можуть покращити продуктивність у сценаріях, що вимагають великого обсягу пам’яті

4 Специфікація вхідних і вихідних даних

4.1 Тип данних: mpz_t

Основним типом даних, що використовується в цілочисельних операціях GMP, є `mpz_t`, який визначено так:

```
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 Limbs представлення

На 64-бітних архітектурах `mp_limb_t` виглядає так:

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

4.3 Вимоги до Input даних

4.3.1 Вимоги до ініціалізації

Всі змінні типу `mpz_t` повинні бути ініціалізовані перед використанням:

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

4.3.2 Формати введення

GMP підтримує кілька форматів введення:

- Пряме присвоювання з типів з C++: `mpz_set_ui`, `mpz_set_si`
- String conversion: `mpz_set_str` (підтримує системи числення з основами 2-62)
- Імпорт з binary data: `mpz_import`

Приклад:

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

4.4 Формати виведення

4.4.1 Конвертація в стандартні дла C++ типи

```
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

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

4.4.3 Binary Export

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

4.5 Управління пам'яттю

4.5.1 Очищення

Усі ініціалізовані змінні `mpz_t` необхідно очистити, щоб звільнити пам'ять:

```
mpz_clear(x);
```

4.5.2 Оцінка використання пам'яті

Для цілого числа з b бітами:

- Кількість of limbs: $n = \lceil b/64 \rceil$
- Memory usage: $\approx 8n + 16$ байт (у 64-бітних системах)

5 Return Codes та обробка помилок

5.1 Категорії of Return Codes

5.1.1 Void Functions (No Return)

Більшість арифметичних функцій GMP повертають `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);
```

Ці функції завжди виконуються успішно (за умови правильної ініціалізації та наявності достатнього обсягу пам'яті).

5.1.2 Integer Return Values

mpz_invert

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

Return:

- Не нуль (істина) (true), якщо оберенене існує
- 0 (false), якщо обернене не існує (gcd(op, mod) $\neq 1$)

mpz_probab_prime_p

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

Return:

- 2, якщо n точно просте число
- 1, якщо n можливо просте число (велика ймовірність)
- 0, якщо n однозначно складене число

mpz_cmp (and variants)

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

Return:

- Позитивне значеня, якщо op1 $>$ op2
- 0, якщо op1 $=$ op2
- Негативне значення, якщо op1 $<$ op2

mpz_sgn

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

Return:

- $+1$, якщо op > 0
- 0, якшо op $= 0$
- -1, якшо op < 0

5.1.3 Size and Count Returns

mpz_sizeinbase

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

Return: Кількість цифр у заданій системі числення (приблизна кількість, якщо не є степенем числа 2)

mpz_popcount

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

Return: Кількість 1-біт у двійковому представленні

5.2 Обробка помилок

5.2.1 Memory Allocation Failures

GMP використовує настроюваний механізм розподілу пам'яті. За замовчуванням використовується `malloc/realloc/free`. Якщо розподіл пам'яті не вдається:

- GMP викликає функцію обробки помилок
- Стандартна поведінка: виводить повідомлення про помилку і викликає `abort()`
- Custom handlers можна встановити за допомогою `mp_set_memory_functions`

5.2.2 Ділення на нуль

Операції, такі як `mpz_div`, `mpz_mod`, and `mpz_invert` із нульовим дільником:

- Поведінка GMP: викликає обробник помилок (зазвичай перериває роботу)
- Рекомендація: перевіряйте дільники перед операціями в робочому коді

5.2.3 Недійсні аргументи функції

GMP припускає, що:

- Всі змінні `mpz_t` правильно ініціалізовані
- Виконані попередні умови функції (наприклад, додатний модуль `mpz_mod`)
- Порушення умов призводить до невизначеної поведінки

5.3 Return Code Summary Table

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

Таблиця 2: Типи повернення функцій GMP

5.4 Розгляд безпеки потоків

- Функції GMP, як правило, є безпечними для потоків при роботі з окремими змінними `mpz_t`
- Відсутність внутрішнього глобального стану (крім користувацьких розподільників пам'яті)
- Кожен потік повинен підтримувати власний набір змінних
- Синхронізація потрібна тільки при спільному використанні змінних `mpz_t` між потоками

6 Аналіз продуктивності для паралельних обчислювальних середовищ

6.1 Міркування стосовно архітектури апаратного забезпечення

6.1.1 Переваги 64-bit Architecture

- Native 64-bit limb операції без програмної емуляції
- Повне використання 64-бітних регістрів (RAX, RBX тощо)
- Ефективні multiply-with-carry інструкції (MUL, IMUL, ADC)
- Великий адресний простір для обробки великих цілих чисел

6.1.2 Використання багатоядерних процесорів

Для серверів обробки транзакцій:

Scenario 1: Незалежні криптографічні операції

```
// 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;
}
```

Масштабованість: Майже лінійна залежно від кількості ядер для незалежних операцій

6.1.3 Оптимізація архітектури NUMA

Для систем з 128 GB RAM оперативної пам'яті, розподіленої між NUMA nodes:

- Allocate локальні для потоку змінні mpz_t на локальних NUMA nodes
- Використовувати `numa_alloc_local()` для власного розподільника GMP
- Мінімізуйте міжвузловий доступ до пам'яті
- Pin threads до конкретних ядер/вузлів для стабільнішої продуктивності

6.2 Оптимізація кешу

6.2.1 L1/L2/L3 Cache Behavior

- Small integers (< 1024 bits): зазвичай вміщуються в кеш L1/L2
- Medium integers (1024-8192 bits): підходять для кешу L3
- Large integers (> 8192 bits): домінують за пропускною здатністю пам'яті

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 Огляд пропускної здатності пам'яті

Для дуже великих цілих чисел (мільйони бітів) продуктивність залежить від пам'яті:

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

Таблиця 3: Приблизна продуктивність на сучасному 64-бітному сервері

6.4 Практичні рекомендації по впровадженню даної бібліотеки

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 Рекомендації щодо тестування (Benchmark Recommendations)

Для оцінки продуктивності GMP у конкретному середовищі необхідно:

1. **Single-threaded baseline:** Виміряти час роботи ядра
2. **Multi-threaded scaling:** Тестування на 1, 2, 4, 8, 16+ потоках
3. **Memory pressure:** Тестування з різними розмірами цілих чисел
4. **NUMA effects:** Порівняння локального та віддаленого доступу до пам’яті
5. **Cache behavior:** Вимірювання продуктивності з ”гарячими” та ”холодними” даними

7 Висновок

7.1 Підсумок результатів дослідження

В моєму міні-дослідженні проведено комплексний аналіз функцій бібліотеки GNU GMP для арифметики з багатократною точністю в паралельних обчислювальних середовищах. Основні висновки такі:

- GMP надає асимптотично оптимальні алгоритми для всіх основних арифметичних операцій
- Складність варіюється від $O(n)$ для додавання, до $O(n \log n \log \log n)$ для множення на основі швидкого перетворення Фур'є
- Бібліотека є високооптимізованою для 64-бітних архітектур з асемблерними реалізаціями
- Task-level parallelism забезпечує майже лінійне масштабування для незалежних операцій
- Управління пам'яттю має вирішальне значення для продуктивності у великомасштабних проєктах

7.2 Придатність для обробки транзакцій

Для серверів обробки транзакцій та постачальників хмарних послуг:

Переваги:

- Зріла, добре перевірена бібліотека з десятиліттями оптимізації
- Відмінна однопотокова продуктивність
- Проста паралелізація незалежних операцій
- Ефективне використання пам'яті для типових криптографічних операцій

Недоліки:

- Відсутність вбудованого паралелізму на рівні функцій (наприклад, паралельне множення)
- Накладні витрати на розподіл пам'яті для дуже частих невеликих операцій
- Для підтримки NUMA потрібен спеціальний розподільник пам'яті

7.3 Рекомендації щодо оптимізації

7.3.1 Для сценаріїв з високою пропускною здатністю

1. Використовувати пули потоків із попередньо ініціалізованими змінними mpz_t
2. Пакетні операції зі спільними модулями для поліпшення використання кешу
3. Розгляньти можливість використання спеціальних розподільників пам'яті для систем NUMA
4. Профілювання шаблонів доступу до пам'яті для мінімізації міжвузлового трафіку

7.3.2 Для сценаріїв з низькою затримкою

1. Попередньо виділять змінні mpz_t з очікуваним максимальним розміром
2. Зберігайти часто використовувані значення (наприклад, модулі, відкриті ключі) в кеші
3. Використовувати цілі числа фіксованого розміру, якщо відомі вимоги до точності
4. Уникайти динамічного розподілу пам'яті in critical paths

7.4 Загальні висновки

GNU GMP залишається золотим стандартом для арифметики з багатократною точністю в високопродуктивних обчислювальних середовищах. Поєднання алгоритмічної досконалості, оптимізації під конкретну архітектуру та гнучкості робить його ідеальним для серверів обробки транзакцій та хмарних служб, що вимагають криптографічних операцій з великими цілими числами. За допомогою відповідних стратегій паралельних обчислень GMP може ефективно використовувати сучасні багатоядерні 64-бітні архітектури з великими обсягами пам'яті, забезпечуючи високу пропускну здатність і низьку затримку для вимогливих додатків.

Література

[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.