МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних робіт з дисципліни
«Паралельні та розподілені обчислення»

для здобувачів вищої освіти першого (бакалаврського) рівня
спеціальності 123 Комп'ютерна інженерія
усіх форм навчання.

Частина 1.
Багатопотокове програмування у C++ та Java

2026

Методичні вказівки до лабораторних робіт з дисципліни «Паралельні та розподілені обчислення» для здобувачів вищої освіти першого (бакалаврського) рівня спеціальності 123 Комп'ютерна інженерія усіх форм навчання. Частина 1. Багатопотокове програмування у C++ та Java / Укл.: Р. К. Кудерметов, О. В. Польська, Т. С. Дьячук — Запоріжжя: НУ «Запорізька політехніка», 2026 — 52 с.

Укладачі: Р. К. Кудерметов, доцент, к.т.н.
О. В. Польська, ст. викладач
Т. С. Дьячук, ст. викладач

Рецензент: М. Ю. Тягунова, доцент, к.т.н.

Відповідальний за випуск: О. В. Польська, ст. викладач

Затверджено:
на засіданні кафедри
«Комп'ютерні системи та мережі»
Протокол № 6 від 26 січня 2026 р.

Рекомендовано до видання:
НМК факультету КНТ
Протокол № 6 від 29 січня 2026 р.

ЗМІСТ

ЛАБОРАТОРНА РОБОТА 1
Багатопотокове програмування на мові C++

Мета роботи – вивчити основи багатопотокового програмування на мові C++ та можливості його застосування для реалізації паралельних алгоритмів і програм.

1.1 Багатопотоковість у C++

Зазначимо, перш за все, що тільки починаючи зі стандарту версії C++11, мова C++ підтримує багатопотоковість. У даних методичних вказівках приклади виконані відповідно до стандарту C++17 (2017 р.). На сьогодні діє стандарт C++20, але ще не всі компілятори його підтримують [1].

У цій лабораторній роботі розглядаються деякі з основних засобів реалізації багатопотоковості мовою C++, які увійшли до стандарту C++17 та необхідні для реалізації паралельних алгоритмів [2], що демонструють переваги паралельних обчислень у порівнянні із послідовними.

1.1.1 Створення і запуск потоків

Найпростіша багатопотокова програма наведена у ліст. 1.1. У цій програмі використовується клас `thread`, що призначений для створення потоків засобами мови C++. Його визначення знаходиться у однойменному заголовному файлі `thread.h`. Конструктор класу `thread` приймає функцію та параметри цієї функції (нагадаємо, що ім'я функції – це покажчик на функцію). Функція, що передається до конструктора, виконується у окремому потоці (в екземплярі класу `thread`).

Кожний створений потік має власний ідентифікатор `ID`, тип якого – `thread::id`. Ідентифікатор потоку можна одержати шляхом виклику статичного методу класу `this_thread::get_id()` або шляхом виклику методу `get_id()` екземпляру класу `thread`, наприклад, `th_name.get_id()`.

Лістинг 1.1 – Проста багатопоточна програма на мові C++

```
1  #include <iostream>
2  #include <thread>
3
4  #define n 3
5  using namespace std;
6
7  void hello(char c) {
8     thread::id id = this_thread::get_id();
9     cout << "Hello from " << c << " id=" << id << endl;
10
11 }
12
13 void my_id(char c) {
14    thread::id id = this_thread::get_id();
15    cout << c <<": My id = " << id << endl;
16 }
17
18 int main() {
19    thread::id id_main = this_thread::get_id();
20    cout<< "The main() has id=" << id_main << endl;
21    thread thA(hello, 'A');
22    thread thB(hello, 'B');
23    thA.join();
24    thB.join();
25    thread th[n];
26    for(int i = 0; i < n; i++)
27       th[i] = thread(my_id, 67+i);
28    for(int i = 0; i < n; i++)
29       th[i].join();
30
31    return 0;
32 }
```

У рядках 7 і 14 ліст. 1.1 визначаються дві функції, призначені для передачі в екземпляри потоків. У кожній з цих функцій викликається функція `this_thread::get_id()`, щоб визначити ідентифікатор потоку, який виконує цю функцію. У головній функції `main()` також викликається функція `this_thread::get_id()` (рядок 19), щоб продемонструвати ідентифікатор головного потоку, тобто потоку, який виконує функцію `main()`.

У рядку 21 створюється потік `thA`, в який передається функція `hello()` і її аргумент (символ `'A'`). Аналогічно у наступному рядку створюється потік `thB`. Для того, щоб головний (батьківський) потік, що виконує функцію `main()`, не завершив свою роботу раніше за дочірні потоки `thA` і `thB`, використовуються виклики функції `join()` (рядки 23 і 24), що змушує головний потік очікувати на завершення роботи дочірніх потоків.

У рядку 25 продемонстровано, як створювати масив екземплярів потоків. У циклі `for` (рядок 26) у кожний потік масиву передається функція `my_id()` із власним аргументом. В іншому циклі `for` (рядок 28) викликається функція `join()` для кожного потоку із масиву потоків.

В результаті роботи програми одержимо на екрані повідомлення наступного змісту:

```
The main() has id=1
Hello from B id=3
Hello from A id=2
C: My id = 4
D: My id = 5
E: My id = 6
```

З наведеного результату роботи програми бачимо, що головна функція `main()` виконується у потоці з `ID = 1`. Інші потоки отримали ідентифікатори у порядку створення потоків. Крім того, зверніть увагу, що послідовність виведення на друк може бути неупорядкованою відповідно до їх ідентифікаторів (наприклад, потік `thB` повідомив про свій ідентифікатор раніше потоку `thA`), оскільки час виконання роботи потоків залежить від ресурсів, що виділяються операційною системою.

1.1.2 Паралельне обчислення числа π

У лістингу 1.2 демонструється один із способів розпаралелювання алгоритмів – розпаралелювання за ітераціями. У програмі обчислюється наближене значення числа π за відомою формулою:

$$\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan(x)|_0^1 = \pi. \tag{1.1}$$

Інтеграл у цій формулі можна обчислити методом прямокутників:

$$\int_0^1 f(x)dx \approx h\sum_{i=0}^{n-1} f(x_i) = h\sum_{i=0}^{n-1} \frac{4}{1+x_i^2}, \tag{1.2}$$

де h – крок дискретизації інтервалу інтегрування [0, 1] на n підінтервалів, $h = 1/n$.

Якщо для обчислення інтеграла використовується метод середніх прямокутників, то значення підінтегральної функції обчислюються в точках, що лежать у середині кожного з підінтервалів, тобто $x_i = h(i + 1/2)$.

Суть розпаралелювання обчислень для даної задачі полягає у тому, що підінтегральна функція $f(x)$ може обчислюватися у кожній точці дискретизації незалежно від обчислень цієї функції в інших точках.

Таким чином, обчислення функції у першій точці можна виконати у першому програмному потоці, обчислення у другій точці – у другому програмному потоці і так далі. Отже, за наявності у процесорі комп'ютера декількох ядер, обчислення функції $f(x)$ у різних точках дискретизації можуть виконуватися на різних ядрах процесора.

В програмі (ліст. 1.2) головний потік, що виконує функцію `main()` (`ID=1`), створює два дочірніх потоки з `ID=2` і `ID=3`. Обчислення на парних підінтервалах ($i = 0, 2, \ldots$) виконуються потоком з `ID=2`, обчислення на непарних підінтервалах ($i = 1, 3, \ldots$) – потоком з `ID=3`.

У даній реалізації потоки мають доступ до спільного ресурсу – змінної `pi` (рядок 23). При одночасному доступі кількох потоків до спільного ресурсу може виникнути конфлікт, який називається *перегонами даних* або умовою перегонів (race condition).

Для виключення одночасних звернень незалежних потоків до спільного ресурсу у багатопотокових програмах використовується примітив синхронізації, що носить назву *м'ютекс* (mutual exclusion, взаємне виключення).

Лістинг 1.2 – Паралельне обчислення числа π

```
#include <iostream>
#include <mutex>
#include <thread>
#include <iomanip>
#include <cmath>

using namespace std ;

double pi = 0.0;
mutex m;

void parallel(long n, int num_ths, int th_num) {
   double x;
   double local_pi = 0.0;
   double h = 1.0/n;
   long k = 0;
   for(long i=(long)th_num; i<n; i+=(long)num_ths) {
      x = h*(i + 0.5);
      local_pi += 4.0/(1.0 + x*x);
   }
   local_pi = h*local_pi;
   m.lock();
   pi += local_pi;
   m.unlock();
}

double serial(long n) {
   double x;
   double sum = 0.0;
   double h = 1.0/n;
   for(int i=0; i<n; i++) {
      x = h*(i+0.5);
      sum += 4.0/(1.0 + x*x);
   }
   return h*sum;
}

int main() {
   const double PI = 3.14159265358979323846264338;
   clock_t t1, t2;
   long it = 4.8E7; //кількість інтервалів
   int m = 4;    //кількість потоків
```

Кінець лістингу 1.2

```
43      thread th[m];
44      t1 = clock();
45      for(int i = 0; i < m; i++)
46         th[i] = thread(parallel, it, m, i);
47      for(int i = 0; i < m; i++)
48         th[i].join();
49      t2 = clock();
50      cout << "parallel: num-threads=" << m << endl;
51      cout << setprecision(16) << "pi=" << pi << endl;
52      cout << setprecision(4) << "err:" << fabs(PI - pi)
           << endl;
53      double t = double(t2 - t1) / CLOCKS_PER_SEC;
54      cout << "time: " << t << " s" << endl;
55
56      t1 = clock();
57      pi = serial(it);
58      t2 = clock();
59      cout << "\nserial:" << endl;
60      cout << setprecision(16) << "pi=" << pi << endl;
61      cout << setprecision(4) <<"err: "<<fabs(PI -
           pi)<<endl;
62      t = double(t2 - t1) / CLOCKS_PER_SEC;
63      cout << "time: " << t << " s" << endl;
64      return 0;
65  }
```

Екземпляр класу `mutex` у програмі створюється в рядку 10. Перед тим як звернутися до спільного ресурсу, потік захоплює м'ютекс за допомогою методу `lock()` (рядок 22), а по завершенні роботи з ресурсом – звільнює м'ютекс за допомогою методу `unlock()` (рядок 24). Бібліотека Thread Library гарантує, що, якщо один потік захопив певний м'ютекс, то всі інші потоки, які намагаються захопити той самий м'ютекс, будуть вимушені очікувати, доки м'ютекс не буде вивільнено [1]. Відзначимо, що використання м'ютекса у даній програмі наведено виключно з навчальною метою, оскільки реалізувати дану програму можна було б і без м'ютекса, наприклад, накопичуючи суми результатів обчислень $f(x_i)$ у кожному з потоків, а результуюче сумування цих результатів виконати у головному потоці, після завершення роботи дочірніх потоків.

Дана програма містить також функцію `serial()` (рядок 27), що призначена для обчислення значення π послідовним способом (в одному потоці) у функції `main()`. Програма також демонструє один із можливих шляхів визначення часових витрат на проведення обчислення (див., наприклад, рядки 44, 49, 54).

Один із можливих результатів виконання програми з ліст. 1.2 наступний:

```
parallel: num-threads=4
pi=3.14159265358984
err:4.663e-14
time: 0.28 s

serial:
pi=3.141592653589363
err: 4.299e-13
time: 0.671 s
```

Час виконання програми паралельного обчислення числа π залежить від продуктивності та кількості фізичних ядер і віртуальних потоків процесора, завантаженості процесора іншими задачами під час виконання даної програми, особливостей реалізації та інших факторів.

Однак зверніть увагу, що час виконання паралельної функції обчислення π на процесорі з двома ядрами приблизно вдвічі менше часу обчислення послідовним способом.

Особливістю даної задачі є те, що вона допускає майже ідеальне розпаралелювання, оскільки обчислення часткової суми у кожному потоці не залежить від обчислення часткових сум в інших потоках. Тому потоки під час обчислень власних часткових сум не обмінюються даними між собою, відсутні витрати часу на синхронізацію і пересилання даних.

Завдяки вище сказаному, ця задача є «класичною» для демонстрації переваг паралельних обчислень. У літературі ви можете зустріти багато паралельних реалізацій цієї задачі.

1.1.3 Факторизація цілого числа

Нагадаємо, що *просте число* – це натуральне число, яке має тільки два натуральних дільника – одиницю і саме це число.

Факторизація цілого числа – це задача розкладання цілого числа на прості множники. Ця задача є обчислювально складною і тому використовується в алгоритмах шифрування з відкритим ключем. Існує кілька алгоритмів факторизації, наприклад, решето Ерастофена, Ферма, Лемана, Полларда-Штрассена, та інші.

Найпростіший алгоритм факторизації – це *пробне ділення*. Основна ідея цього алгоритму полягає у пробному діленні цілого числа n на кожен можливий дільник m. Якщо n ділиться на певне $1 < m < n$, то n є *складене* число. Такий алгоритм належить до «переборних», тобто перевіряються усі числа від 2 до $n - 1$.

У даній лабораторній роботі використовується алгоритм Ібн аль-Банна, який також є перебірним, але перевірка простоти здійснюється в діапазоні чисел від 2 до $\sqrt{n}$. Цей алгоритм має часову складність $O(\sqrt{n}\log(n))$.

Переборні алгоритми є малоефективними тому, що потребують значних обчислювальних витрат. На практиці алгоритми на основі перебору використовуються для факторизації невеликих чисел, наприклад, до 2^{16}. Часова складність цього алгоритму є однією з мотивацій застосування паралельного розв'язання задачі факторизації в даній лабораторній роботі.

У лістингу 1.3 представлена програма, в якій задача факторизації восьми чисел розділена на чотири незалежні задачі, що виконуються чотирма потоками. У цій реалізації програми навантаження на потоки обрано умовно однаковими, тобто кожен потік виконує факторизацію двох чисел.

Наведемо деякі коментарі щодо реалізації програми з ліст. 1.3, які наведено після тексту лістингу. Пояснимо:

- використання типу `unsigned long long`;
- генерацію випадкових чисел;
- роботу функції факторизації `factoring()`;
- створення потоків у функції `main()`;
- використання ідентифікатору потоку `thread::id`.

Лістинг 1.3 – Паралельна факторізація чисел

```
#include <iostream>
#include <thread>
#include <vector>
#include <random>
#define ull unsigned long long

using namespace std;

const int m = 4;      //кількість потоків
const int n = 8;      //кількість чисел для факторизації
ull num[m][n];           //масив вихідних чисел
vector<ull> arr[m][n];  //масив векторів простих чисел

ull ullrand(){
   random_device rd;
   mt19937 gen(rd());
   const ull X_MIN = 1E18;
   const ull X_MAX = 1E19 - 1ULL;
   uniform_int_distribution<ull> distr(X_MIN, X_MAX);
   return distr(gen);
}

void factoring(int id, int n){
   for(int i=0;i<n;i++){
      vector<ull> fact; //вектор розкладання
      ull x = ullrand();
      num[id][i] = x;
      for(int j=2; j<=sqrt(x); j++){
         while(x % j == 0){
            fact.push_back(j);
            x = x/j;
         }
      }
      if(x != 1)
         fact.push_back(x);
      arr[id][i] = fact;
   }
}

int main(int argc, char** argv) {
   int k = n/m; //кількість чисел на один потік
   thread th[m]; //масив потоків
```

Кінець лістингу 1.3

```
43      clock_t t1, t2;
44      double t;
45
46      t1 = clock();
47      for(int i = 0; i < m; i++)
48         th[i] = thread(factoring,i,k);
49      for(int i = 0; i < m; i++)
50         th[i].join();
51      t2 = clock();
52
53      for(int i=0;i<m;i++){
54         for(int j=0;j<k;j++){
55            cout<<"th"<<i<<": "<<num[i][j]<<"=1";
56            for(auto f : arr[i][j]) //оператор foreach
57               cout<<"*"<< f;
58            cout << endl;
59         }
60      }
61      t = double(t2-t1)/CLOCKS_PER_SEC;
62      cout << "time=" << t << " s" <<endl;
63      return 0;
64   }
```

Програма дозволяє факторизувати числа, що представлені з 19 десятковими знаками. Для цього використовується тип цілого числа `unsigned long long` та для скороченого запису цього типу визначено макрос (ідентифікатор) `ull`.

Генерація випадкового числа із рівномірного розподілу здійснюється функцією `ullrand()`. У функції визначені константи `X_MIN` і `X_MAX`, які задають діапазон чисел, що генеруються. Під час налагодження програми цей діапазон можна зменшити, змінивши ці константи до потрібних значень. Для повторюваності результатів генерації (це зручно при налагоджуванні) можна задати аргументом функції `gen()` деяке постійне число, наприклад `gen(1234)`.

Функція факторизації `factoring()` викликає функцію генерації випадкового числа і розкладає це число на прості множники. Оскільки у кожного з чисел, що факторизуються, число множників може бути різним, у програмі використовується контейнер бібліотеки STL (Standard Tempale Library) – `vector`, і кожен із знайдених

простих множників записується у вектор за допомогою функції-члена контейнера `push_back()`. Отриманий у результаті розкладання вектор множників зберігається у двомірному масиві `arr[][]`, першим індексом якого є номер потоку, а другим – номер вектора. У головній функції створюються потоки, що асоціюються з функцією факторизації та її параметрами. Для дослідження продуктивності програми використовується функція вимірювання часу `clock()`.

Ідентифікатор потоку `thread::id` є об'єктом і може використовуватися, наприклад, для порівняння потоків, або ідентифікації потоків під час обміну даними між ними, виведення на друк *умовного* номера потоку, як це було зроблено в програмі з ліст. 1.1.

Оскільки ідентифікатор потоку `thread::id` є об'єктом, його не можна використовувати як індекс для масиву. Тому в цій і наступній програмі для позначення номера потоку використовуються числа 0, 1, 2 ..., які означають перший, другий, третій і інші потоки, створені головним потоком.

Нижче наведено можливий результат програми з ліст. 1.3. Як бачимо з результату, всі згенеровані в прикладі числа виявилися складеними, тобто не простими!

```
th0: 3442303384780969792=1*2*2*2*2*2*2*17*84137*37603929157
th0: 2291960608175916063=1*3*3*71*919*3902930156743
th1: 3404784566609683253=1*49697*68510867187349
th1: 4870413249749161710=1*2*3*5*10733*71263*212255683
th2: 3195749296185307793=1*7892953*404886396281
th2: 3054640988150592833=1*41*74503438735380313
th3: 9035993263327198753=1*13*13*73*191*3834713889359
th3: 7739605105232203923=1*3*17*19*2791*1636757*1748441
time=14.151 s
```

1.1.4 Паралельний тест простоти

Тестом простоти називають алгоритм, за допомогою якого перевіряється припущення у тому, що число є простим. Зазначимо, що робота з простими числами є надзвичайно важливою для багатьох практичних завдань, зокрема для формування електронного цифрового підпису, для якого необхідно знаходити прості числа розміром не менше 2^{1024}.

У програмі, наведеної в ліст. 1.4, реалізовано тест простоти, в основі якого використовується алгоритм, аналогічний алгоритму факторизації з попередньої програми.

Лістинг 1.4 – Паралельний тест простоти чисел

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
#include <random>

#define ull unsigned long long

using namespace std;
using namespace chrono;

int flag = 0;
const int flag_max = 2;
mutex mu;

ull ullrand(){
   random_device rd;
   mt19937 gen(rd());
   const ull X_MIN = 1E18;
   const ull X_MAX = 1E19 - 1ULL;
   uniform_int_distribution<ull> distr(X_MIN, X_MAX);
   return distr(gen);
}
void test_prime(int id, ull** prime){
   while(flag < flag_max){
      ull x = ullrand();
      bool b = true;
      for(ull i=2ULL; i<=sqrt(x); i++)
         if(x % i == 0ULL)
            b = false;
      if(b == true && flag != flag_max){
            prime[id][flag] = x;
            mu.lock();
               flag++;
            mu.unlock();
      }
   }
}
```

Кінець лістингу 1.4

```
39 int main() {
40    int m_th = thread::hardware_concurrency();
41    int n = 4;
42    cout << "hard_threads = " << m_th << endl;
43    int k = m_th/n;
44    if(n % m_th != 0) {
45       cout << "The array size must be a multiple\
46             of the threads number!\n";
47       exit(0);
48    }
49
50    ull** arr = new ull*[m_th];
51    for(int i=0;i<m_th;i++)
52       arr[i] = new ull[n];
53    for(int i=0;i<m_th;i++)
54       for(int j=0;j<n;j++)
55          arr[i][j] = 0ULL;
56    thread th[m_th];
57    auto start = high_resolution_clock::now();
58    for(int i = 0; i < m_th; i++)
59       th[i] = thread(test_prime, i, arr);
60    for(int i = 0; i < m_th; i++)
61       th[i].join();
62    auto stop = high_resolution_clock::now();
63    for(int i=0;i<m_th;i++){
64       cout << "th" << i <<": ";
65       for(int j=0;j<flag_max;j++)
66          cout << arr[i][j] <<"  ";
67       cout << endl;
68    }
69    auto time = duration_cast<milliseconds>(stop-start);
70    cout << "time=" << time.count() << " ms." <<endl;
71    for(int i=0;i<m_th;i++)
72       delete [] arr[i];
73    delete [] arr;
74    return 0;
75 }
```

Деякі коментарі до реалізації програми з ліст. 1.4:

– у програмі декількома потоками одночасно здійснюється пошук двох простих чисел. Як тільки ці два прості числа знайдені, потоки припиняють пошук;

– для досягнення максимальної продуктивності пошуку, за допомогою бібліотечної функції `thread::hardware_concurrency()`, визначається кількість потоків, які по-справжньому можуть працювати паралельно (наприклад, кількість ядер процесора) [1]. Далі ці потоки запускаються та виконують функцію `test_prime()`;

– функція `test_prime()` викликає функцію генерації випадкового числа та перевіряє це число на простоту. Якщо число виявилося простим, число записується у двовимірний масив `prime`, першим індексом якого є номер потоку, а другим – номер знайденого простого числа у програмі;

– у задачі необхідно знайти лише задану кількість простих чисел, тому у програмі введені дві глобальні змінні `flag` і `flag_max`, перша з яких є лічильником кількості знайдених усіма потоками простих чисел, друга – задає необхідну кількість простих чисел. Для виключення конфліктів при зміні потоками змінної `flag`, вона захищена примітивом синхронізації `mutex`. Оскільки процес перевірки числа на простоту обчислювально складний і може тривати довгий час і під час його виконання інші потоки цілком імовірно можуть змінити значення глобальної змінної `flag`, перед записом знайденого простого числа в результуючий масив `prime` додатково виконується перевірка значення змінної `flag`;

– для спрощення налагодження цієї програми під час роботи з великими числами можна скористатися рекомендаціями, запропонованими у коментарях до програми 1.3.

Нижче наведено приклад результату роботи програми (див. ліст. 1.4), з якого видно, що другий (`th1`) та третій (`th2`) потоки першими знайшли прості числа:

```
hard_threads = 4
th0: 0  0
th1: 5122503273555457117  0
th2: 0  7392808951772109017
th3: 0  0
time=23193 ms.
```

1.2 Завдання до лабораторної роботи

Для створення власних змістовних прикладів паралельних програм варто вивчити всі приклади даної лабораторної роботи.

1.2.1 Завдання до розділу 1.1.1

Реалізуйте багатопотокову програму для даного сценарію:

– у програмі визначено дві рядкові змінні: перша містить ім'я розробника, друга – його прізвище; наприклад, `string name = "Bogdan"; string sname = "Sidorenko";`

– у програмі створюється стільки потоків, скільки літер в імені, у даному випадку – 6;

– кожен потік асоціюється з функцією, яка ділить ASCII код літери, номер якої в імені дорівнює номеру потоку, на кількість літер у прізвищі та обчислює число Фібоначчі від результату поділу;

– обчислені у кожному потоці числа Фібоначчі повинні заноситися до загального масиву для подальшого виведення на консоль в головному потоці;

– виведення на друк має виконуватися в головному потоці. На друк слід виводити: номер потоку; літеру, з якою працює функція в потоці; ASCII код літери; результат поділу цього коду на число літер у прізвищі; обчислене число Фібоначчі.

Приклад виведення на друк наведено нижче:

```
0 B 66  7  13
1 o 111 12 144
2 g 103 11 89
3 d 100 11 89
4 a 97  10 55
5 n 110 12 144
```

1.2.2 Завдання до розділу 1.1.2

Для виконання цього завдання вивчіть та реалізуйте програму, наведену у ліст. 1.2. Використовуючи структуру та методи цієї програми, розробить власну програму, яка паралельно обчислює число π за модифікованою формулою Лейбниця:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \left(\frac{1}{4n+1} - \frac{1}{4n+3} \right). \tag{1.3}$$

Вимоги до програми:

– програма повинна містити функції для паралельного та послідовного обчислення числа π з використанням формули (1.3);

– кількість членів ряду формули (1.3) дорівнює $3.6 \cdot 10^7$;

– кількість паралельних потоків, що виконують обчислення, повинна дорівнювати кількості фізичних ядер вашого процесора, що рекомендовано визначити програмно за допомогою функції `thread::hardware_concurrency()`;

– розподіл членів ряду формули має бути аналогічним схемі розподілу в програмі з ліст 1.2, тобто перший дочірній потік обчислює члени ряду з n = 0, 2, ...; другий – члени ряду з n = 1, 3, ... і так далі;

– результат роботи програми має бути представлений у наступному вигляді:

```
parallel: num-threads=4
3.141592639646639
err:1.394e-08
time: 0.422 s

serial:
3.141592639490767
err: 1.41e-08
time: 0.836 s
```

1.2.3 Завдання до розділу 1.1.3

У цьому завданні розробити багатопотокову програму, яка факторизує цілі випадкові числа з 19 десятковими розрядами.

Вимоги до програми:

– у програмі має створюватися задана кількість дочірніх потоків;

– кожен потік повинен факторизувати одне число;

– числа для факторизації генеруються у головному потоці;

– у програмі необхідно забезпечити вимірювання часу роботи кожного дочірнього потоку;

– у програмі необхідно забезпечити вимірювання сумарного часу роботи всіх дочірніх потоків програми;

– результат роботи програми повинен мати вигляд, подібний до наведеного нижче:

```
number threads=3
th0: 7182904943418840859=1*191*449*29339*2854796159
    t=0.007 s
th1: 2759400573190355176=1*2*2*2*344925071648794397
    t=29.02 s
th2: 5153157974707382202=1*2*3*4974329*172658395223
    t=0.601 s
total t = 29.021 s
```

1.2.4 Завдання до розділу 1.1.4

У цьому завданні багатопотокове програмування використовується для роботи зі *взаємно простими числами* (*coprime integers*). Взаємно прості числа використовуються у криптографії, генераторах псевдовипадкових чисел та деяких шифрів.

Прикладом використання взаємно простих чисел у механіці є виготовлення ланцюгової передачі, при якому число зубів зірочки та кількість ланок ланцюга часто вибирають взаємно простими. У цьому випадку кожен зуб зірочки буде працювати з усіма ланками ланцюга, що забезпечує рівномірність зношування.

Нагадаємо, що два числа є взаємно простими, якщо немає цілого числа, більшого за одиницю, що ділить кожне з них. Наприклад, числа 49 і 71 є взаємно простими, а числа 36 і 48 не є взаємно простими, так як у них є спільні дільники 2, 3, 4, 6, 12.

Таким чином, два цілих числа є взаємно простими, якщо їх найбільший спільний дільник (НСД) дорівнює 1. Для знаходження взаємно простих чисел можна використовувати алгоритм Евкліда для обчислення НСД.

У завданні необхідно розробити багатопотокову програму пошуку взаємно простих чисел.

Вимоги до програми:

– головний потік програми має створювати задану кількість дочірніх потоків;

– кожен потік повинен визначати задану кількість пар взаємно простих чисел;

– числа-кандидати на взаємну простоту повинні складатися з щонайменше 19 десяткових розрядів і генеруватися в дочірніх потоках;

– для перевірки взаємної простоти має використовуватися рекурсивний алгоритм Евкліда для обчислення НСД;

– знайдені взаємно прості числа повинні записуватись у двовимірний масив, першим індексом якого є номер потоку, другим – номер пари взаємно простих чисел;

– результат роботи програми має бути представлений у наступному вигляді:

```
th0 2471065176703442369 1369614043714780144
th0 9860057061039892147 6117229864550349183
th0 5596119845150708927 2314283024086907319

th1 4901219794644525575 6645241529289570724
th1 8860381327766087739 8085154745475538964
th1 4144783170887639142 4704036400808801411

th2 2582544748208861925 7051997928890314039
th2 4781002675417857295 8740642724221031087
th2 6472487721303665413 3774371505001645571
```

1.3 Контрольні питання

1. У чому полягає закон Мура? Навести приклади, що підтверджують або ставлять під сумнів його справедливість.

2. Що таке прискорення паралельного алгоритму?

3. Навести фактори, що зумовлюють продуктивність комп'ютерних систем.

4. Пояснити виникнення прискорення при конвеєрній, паралельній та векторній обробці даних.

5. Навести основні характеристики перших п'яти комп'ютерних систем з останньої редакції рейтингу TOP 500 [3].

6. Поясніть у чому суть розпаралелювання за ітераціями у програмі, наведеній у ліст. 1.2.

7. Від чого залежить час виконання програми факторизації цілих чисел (ліст. 1.3)?

8. Яка ваша думка щодо того, що загальний час виконання програми у завданні 1.2.3 приблизно дорівнює часу роботи одного з потоків. Як це пов'язано із законом Амдала?

1.4 Зміст звіту

1. Титульний лист, оформлений відповідно до вимог.
2. Мета роботи.
3. Фрагменти програмного коду, розроблені самостійно.
4. Відповіді на контрольні питання.

ЛАБОРАТОРНА РОБОТА 2
Багатопотокове програмування у Java

Мета роботи – вивчити основи роботи з потоками на мові Java та можливості їх застосування для паралельних та векторних обчислень.

2.1 Багатопотоковість у Java

У Java класи потоків визначаються як підкласи класу `Thread`, що інкапсулює основні методи роботи з потоками [4–6]. При запуску програми у Java автоматично починає виконуватися головний потік, що дозволяє створювати дочірні потоки.

Визначити дочірні потоки можна двома способами:

– створити підклас класу `Thread` та перевизначити його метод `run()`;

– створити клас, що реалізує інтерфейс `Runnable`. Цей інтерфейс містить абстрактний метод `run()`, який клас потоку має реалізувати.

Вважається, що другий спосіб є більш гнучким у порівнянні із першим, оскільки клас, окрім реалізації інтерфейсу `Runnable`, може також успадковувати інший клас.

У методі `run()` виконується уся робота потоку. Для запуску потоку першим способом необхідно створити екземпляр класу потоку та викликати успадкований ним метод `start()`.

Для запуску потоку, створеного другим способом, необхідно створити об'єкт класу `Thread`, передати у його конструктор об'єкт класу, що реалізує інтерфейс `Runnable`, та викликати метод `start()` класу `Thread`. В обох випадках метод `start()` викликає метод `run()`, що виконує цільову задачу потоку. З цього моменту починається життєвий цикл потоку.

Життєвий цикл потоку [6] ґрунтується на наступних шести станах (рис. 2.1):

– `NEW` (новий) – потік створений, але не виконується;

– `RUNNABLE` (виконуваний) – потік виконується, в цей стан потік потрапляє після виконання методу `start()` та виклику ним методу `run()`;

– `BLOCKED` (заблокований). Якщо потік намагається отримати об'єкт блокування (`lock`), який зараз утримується іншим потоком, він блокується. Потік стає розблокованим, коли всі інші потоки звільняються від блокування, а планувальник потоків дозволяє цьому потоку утримувати його;

– `WAITING` (в очікуванні). Потік буде в стані очікування, коли він викликає метод `wait()`. Він перейде в робочий стан, коли інший потік викличе метод `notify()` або `notifyAll()`;

– `TIMED_WAITING` (тимчасово в очікуванні або заблокований за часом). У мові Java є кілька методів, які приймають як параметр час очікування (наприклад, `sleep(long timeout)`). Їх виклик вводить потік виконання в стан тимчасового очікування, яке зберігається доти, поки закінчиться заданий час очікування або отримано відповідне повідомлення;

– `TERMINATED` (завершений). Потік завершує існування, коли він повністю виконаний програмою або сталася якась незвичайна помилкова подія, наприклад, помилка сегментації або необроблена виключна ситуація.

За допомогою методу `getState()` можна дізнатись про стан потоку. Зазвичай потік завершує свою роботу, коли завершується виконання його методу `run()`. Хоча у Java є метод `stop()` для примусового завершення роботи та методи `suspend()` і `resume()` для керування його блокуванням, але ці методи вважаються застарілими.

Кожен потік Java має *пріоритет*. За замовчуванням потік успадковує пріоритет потоку, що його створив [4]. За допомогою методу `setPriority()` можна встановити значення пріоритету потоку в діапазоні від `MIN_PRIORITY` до `MAX_PRIORITY`, які відповідно дорівнюють 1 і 10. Константа `NORM_PRIORITY` (дорівнює 5) визначає нормальний пріоритет потоку. Коли Java-віртуальна машина планує потік для виконання, вона надає ресурси потокам із найвищим пріоритетом. У звичайних багатопотокових програмах не рекомендується будувати логіку роботи програми, залежну від пріоритетів.

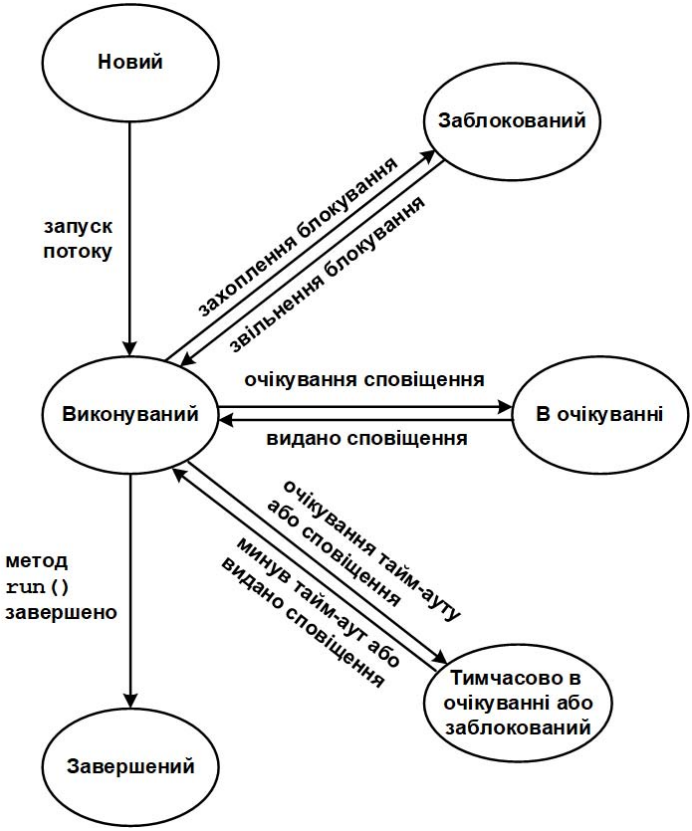

Рисунок 2.1 – Стани потоку

2.1.1 Створення і запуск потоків

Розглянемо на прикладі програми (див. ліст. 2.1-2.4), в якій застосовуються три потоки виконання, створення та способи запуску потоків.

У програмі визначено три класи: `Letters`, `Numbers` та `Symbols`. Кожен з цих класів містить власний метод `run()`, який виконує деяку задачу. У прикладі – це дуже прості методи, які записують у рядок символи алфавіту у класі `Letters`, символи цифр у класі `Numbers` і деякі символи операцій і роздільників у класі `Symbols`.

У програмі реалізовані обидва способи запуску потоків: потік класу `Letters` успадковує (`extends`) клас `Thread`, а потоки класів `Numbers` і `Symbols` реалізують (`implements`) інтерфейс `Runnable`.

Лістинг 2.1 – Клас потоку `Letters` успадковує клас `Thread`

```
1  public class Letters extends Thread {
2     public String letters = "";
3     public void run(){
4        for(int i = 'a'; i <= 'z'; i++){
5           letters = letters + (char)i + " ";
6        }
7     }
8  }
```

Лістинг 2.2 – Клас потоку `Numbers`

```
1  public class Numbers implements Runnable {
2     public String numbers = "";
3     public void run(){
4        for(int i = 0; i < 10; i++){
5           numbers = numbers + i + " ";
6        }
7     }
8  }
```

Лістинг 2.3 – Клас потоку `Symbols`

```
1  public class Symbols implements Runnable {
2     public String symbols = "";
3     public Thread th;
4     public Symbols(){
5        th = new Thread(this);
6        th.start();
7     }
8     public void run(){
9        for(int i = 36; i <= 47; i++) {
10          symbols = symbols + (char)i + " ";
11       }
12    }
13 }
```

Лістинг 2.4 – Головний клас `DemoThreadsStart`

```
1  public class DemoThreadsStart {
2     public static void main(String[] args) {
3        Letters letters = new Letters();
4        Numbers numbers = new Numbers();
5        Thread thread = new Thread(numbers);
6
7        letters.start();
8        thread.start();
9        Symbols symbols = new Symbols();
10
11       try {
12          letters.join();
13          thread.join();
14          symbols.th.join();
15       } catch(InterruptedException e) {
16             e.printStackTrace(System.err);
17       }
18       System.out.println(letters.letters);
19       System.out.println(numbers.numbers);
20       System.out.println(symbols.symbols);
21    }
22 }
```

Об'єкти класів потоків створюються у методі `main()` головного класу `DemoThreadsStart` (ліст. 2.4). Потоки класів `Letters` і `Numbers` запускаються у методі `main()` (рядки 7-8), а потік класу `Symbols` – у його конструкторі (рядок 6, ліст 2.3). При нормальному завершенні роботи методу `run()` потік звичайним чином завершує своє існування.

Метод `join()` класу `Thread` (рядки 12-14, ліст. 2.4) використовується для очікування головним потоком завершення роботи дочірніх потоків.

В результаті роботи програми на консолі одержимо повідомлення наступного змісту:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
$ % & ' ( ) * + , - . /
```

2.1.2 Перевірка станів потоків

У ліст. 2.5, 2.6 представлена програма, в якій за допомогою методу `getState()` класу `Thread` перевіряються деякі стани потоків, викликані різними ситуаціями. За допомогою методу `getPriority()` можна ідентифікувати пріоритети потоків.

Клас `ThreadState` (ліст. 2.5) призначений для створення двох потоків. У методі `main()` головного потоку (ліст. 2.6) створюються та запускаються на виконання два потоки класу `ThreadState`.

Лістинг 2.5 – Клас `ThreadState`

```
1  public class ThreadState implements Runnable {
2     private String threadName;
3     public ThreadState(String name){
4        threadName = name;
5     }
6     public void run() {
7        Thread.State state =
             Thread.currentThread().getState();
8        int priority =
             Thread.currentThread().getPriority();
9        System.out.println("msgA: " +threadName+ "
             priority: " + priority);
10       System.out.println("msgB: "+threadName+" is
             "+state);
11       try {
12          System.out.println("msgC: " + threadName + "
                fell asleep");
13          Thread.sleep(100);
14       } catch(InterruptedException e) {
15          e.printStackTrace(System.err);
16       }
17       for(int i = 1; i < 4; i++) {
18          System.out.println("msgD: "+ threadName + " is
                working");
19       }
20    }
21    public String getName(){
22       return threadName;
23    }
24 }
```

Лістинг 2.6 – Клас DemoThreadState

```
1  public class DemoThreadState {
2     public static void main(String[] args) {
3        System.out.println("msg0: " +
             Thread.currentThread());
4        Thread th1 = new Thread(new ThreadState("th1"));
5        th1.setPriority(Thread.MIN_PRIORITY+2);
6        Thread th2 = new Thread(new ThreadState("th2"));
7        System.out.println("msg1: th1 is " +
             th1.getState());
8        th1.start();
9        try {
10          Thread.sleep(50);
11       } catch (InterruptedException e) {
12          e.printStackTrace(System.err);
13       }
14
15       System.out.println("msg2: th1 is " +
             th1.getState());
16       System.out.println("msg3: th2 is " +
             th2.getState());
17       th2.start();
18       try {
19          Thread.sleep(10);
20       } catch (InterruptedException e) {
21          e.printStackTrace(System.err);
22       }
23       System.out.println("msg4: th2 is " +
             th2.getState());
24       System.out.println("msg5: th1 is " +
             th1.getState());
25       try {
26          th1.join();
27          System.out.println("msg6: th1 is " +
                th1.getState());
28          th2.join();
29          System.out.println("msg7: th2 is " +
                th2.getState());
30       } catch (InterruptedException e) {
31          e.printStackTrace(System.err);
32       }
33    }
34 }
```

У класі `ThreadState` показано як можна дізнатися стан потоку (рядок 7) та перевірити значення пріоритету потоку (рядок 8). У головному класі у рядку 5 заданий пріоритет одного із потоків.

Нижче наведено результат роботи програми з ліст. 2.5, 2.6.

```
msg0: Thread[main,5,main]
msg1: th1 is NEW
msgA: th1 priority: 3
msgB: th1 is RUNNABLE
msgC: th1 fell asleep
msg2: th1 is TIMED_WAITING
msg3: th2 is NEW
msgA: th2 priority: 5
msgB: th2 is RUNNABLE
msgC: th2 fell asleep
msg4: th2 is TIMED_WAITING
msg5: th1 is TIMED_WAITING
msgD: th1 is working
msgD: th1 is working
msgD: th1 is working
msg6: th1 is TERMINATED
msgD: th2 is working
msgD: th2 is working
msgD: th2 is working
msg7: th2 is TERMINATED
```

2.1.3 Взаємодія потоків

У деяких випадках необхідно, щоб певні потоки очікували результати виконання роботи іншими потоками. Для організації такої взаємодії кожен об'єкт класу `Object` має власний «монітор» – високорівневий механізм організації взаємодії і синхронізації потоків, що забезпечує доступ до загального ресурсу. Такий механізм ґрунтується на використанні наступних методів:

– `wait()` – потік, що викликав цей метод, переходить до стану очікування, доки інший потік не викличе метод `notify()`;

– `notify()` – розблоковує один з потоків, якій викликав метод `wait()`;

– `notifyAll()` – розблоковує усі потоки, які викликали метод `wait()`.

Розглянуті методи можуть бути викликані лише із синхронізованого методу чи блоку. Для синхронізації методу і створення синхронізованого блоку використовується ключове слово `synchronized` – *модифікатор*, який блокує метод або блок, щоб його міг використовувати лише один потік одночасно.

У програмі (див. ліст. 2.7-2.10) демонструється найпростіший спосіб синхронізації взаємодії потоків.

Головний клас програми `DemoSynchro` наведено у ліст. 2.7.

У програмі створюються три потоки класу `Recipient` (одержувач) (ліст. 2.8), які у блоці синхронізації `msg` (рядки 17-30) намагаються одержати доступ до об'єкту `msg` класу `Message` (ліст. 2.9).

Для цього вони у блоці синхронізації `msg` викликають метод `wait()` (рядок 23) та очікують, коли потік класу `Sender` (відправник) (ліст. 2.10) завершить обробку об'єкту `msg` і викличе метод `notifyAll()` (рядок 21).

Одержавши сповіщення про вивільнення об'єкту `msg`, потоки класу `Recipient`, що очікують, зможуть продовжити свою роботу з об'єктом `msg`.

Лістинг 2.7 – Головний клас `DemoSynchro`

```
1  public class DemoSynchro {
2    public static void main(String[] args) {
3      Message msg = new Message();
4
5      Recipient rcp1 = new Recipient(msg);
6      new Thread(rcp1,"Recipient1: ").start();
7      Recipient rcp2 = new Recipient(msg);
8      new Thread(rcp2,"Recipient2: ").start();
9      Recipient rcp3 = new Recipient(msg);
10     new Thread(rcp3,"Recipient3: ").start();
11
12     Sender sender = new Sender(msg);
13     new Thread(sender,"Sender: ").start();
14     System.out.println("All participants of the
           cooperation have started");
15   }
16 }
```

Лістинг 2.8 – Клас Recipient

```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3  import java.util.Random;
4
5  public class Recipient implements Runnable {
6    private Message msg;
7    private String time;
8    public Recipient(Message msg) {
9      this.msg = msg;
10   }
11   SimpleDateFormat localDateFormat = new
         SimpleDateFormat("HH:mm:ss:ms");
12   public void run() {
13     Random random = new Random();
14     final long minDT = 500L;
15     final long maxDT = 1500L;
16     String name = Thread.currentThread().getName();
17     synchronized(msg){
18       try{
19         long randDT = random.nextLong(minDT, maxDT);
20         Thread.sleep(1000L + randDT);
21         time = localDateFormat.format(new Date());
22         System.out.println(name + " waits for
               notification at " + time);
23         msg.wait();
24         }catch(InterruptedException e){
25           e.printStackTrace();
26         }
27       time = localDateFormat.format(new Date());
28       System.out.println(name+" notified at "+time);
29       System.out.println("\t msg:"+msg.getMessage());
30     }
31   }
32 }
```

Лістинг 2.9 – Клас Message

```
1  public class Message {
2    private String msg;
3    public void setMessage(String msg){ this.msg = msg;}
4    public String getMessage(){ return msg;}
5  }
```

Лістинг 2.10 – Клас Sender

```
1  import java.text.SimpleDateFormat;
2  import java.util.Date;
3  import java.util.Random;
4  public class Sender implements Runnable {
5    private Message msg;
6    public Sender(Message msg) {
7      this.msg = msg;
8    }
9    public void run() {
10     Random random = new Random();
11     long maxSleepTime = 1000L;
12     String name = Thread.currentThread().getName();
13     SimpleDateFormat localDateFormat = new
           SimpleDateFormat("HH:mm:ss:ms");
14     String time = localDateFormat.format(new Date());
15     System.out.println(name + " begun preparing message
           at " + time);
16     try {
17       int randomSleepTime = random.nextInt((int)
             maxSleepTime);
18       Thread.sleep(3000 + randomSleepTime);
19       synchronized(msg) {
20       msg.setMessage(" 'The message has been sent to
           you.' ");
21       msg.notifyAll();
22       }
23     } catch (InterruptedException e) {
24       e.printStackTrace();
25 } } }
```

Нижче наведено можливий результат роботи програми.

```
All participants of the cooperation have started
Sender:  begun preparing message at 11:22:14:2214
Recipient1:  waits for notification at 11:22:16:2216
Recipient3:  waits for notification at 11:22:17:2217
Recipient2:  waits for notification at 11:22:19:2219
Recipient1:  notified at 11:22:19:2219
         msg: 'The message has been sent to you.'
Recipient2:  notified at 11:22:19:2219
         msg: 'The message has been sent to you.'
Recipient3:  notified at 11:22:19:2219
         msg: 'The message has been sent to you.'
```

2.2 Використання потоків Java

Потоки Java можна використовувати для організації ефективного вирішення багатьох науково-технічних задач. Ефективність полягає у прискоренні одержання рішення і досягається шляхом розпаралелювання та/або векторизації алгоритмів розв'язання. Особливо помітне прискорення можна одержати при використанні багатоядерних процесорів, коли кожен потік виконується на окремому ядрі процесора. У даному розділі розглядається використання потоків для паралельних та векторних обчислень.

2.2.1 Паралельні обчислення

У даній програмі (ліст. 2.11-2.13) потоки Java використовуються для організації паралельного обчислення значення функції $\arcsin(x)$ у заданій точці x_0. Для обчислення використовується розкладання цієї функції у ряд Маклорена:

$$\arcsin(x) = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n \cdot (n!)^2} \cdot \frac{x^{2n+1}}{2n+1}. \tag{2.1}$$

Оскільки значення факторіалу може перевищити діапазон представлення цілих чисел у комп'ютері, у цій програмі обчислюється факторіал як дійсне число типу `double`. Для цього використовується формула:

$$n! = \prod_{k=1}^{\infty} \frac{(1+\frac{1}{k})^n}{1+\frac{n}{k}}. \tag{2.2}$$

З метою імітації великого обчислювального навантаження на процесор і демонстрації ефективності розпаралелювання задачі, обчислення виконується з високою точністю. Висока точність досягається шляхом використання для апроксимації функції достатньо великого числа (більше 20) членів ряду.

Лістинг 2.11 – Головний клас

```
1  import java.util.Locale;
2
3  public class DriveParallelArcsin {
4     public static void main(String[] args) {
5        int n = 48;         //кількість членів ряду 48
6        double x = 0.0174524;
7        double result = 0;
8        int numThreads = 4; //кількість потоків
9        Runtime runtime = Runtime.getRuntime();
10       int cores = runtime.availableProcessors();
11       System.out.println("Intel(R) Core(TM) i5-9400 CPU
             2.9 GHz");
12       System.out.println("Available cores: " + cores);
13       System.out.println("Used threads: " + numThreads);
14       ThreadExecutor [] ths = new
             ThreadExecutor[numThreads];
15       for(int i=0; i<numThreads; i++) {
16          ths[i] = new ThreadExecutor(n,x,i,numThreads);
17       }
18       long t1 = System.nanoTime();
19       for(int i=0; i<numThreads; i++) {
20          ths[i].start();
21       }
22       try {
23          for(int i=0; i<numThreads; i++) {
24             ths[i].join();
25             result += ths[i].getResult();
26          }
27       } catch (InterruptedException e) {
28          e.printStackTrace(System.err);
29       }
30       long t2 = System.nanoTime();
31       double reference = Math.asin(x);
32       System.out.println("x=" + x);
33       System.out.println("math.asin(x)=" + reference);
34       System.out.println("result=" + result);
35       double err = Math.abs(reference-result);
36       System.out.printf(Locale.UK, "error=%.3E\n", err);
37       double t = (double)(t2-t1)*1.0E-9;
38       System.out.printf(Locale.UK, "time=%.3f s", t);
39    }
40 }
```

Лістинг 2.12 – Клас обчислення арксинусу

```
1  public class ArcsinParallel {
2     private int nTerms; //кількість членів ряду на потік
3     private int thNum;  //номер потоку
4     private int numThs; //кількість потоків
5
6     public ArcsinParallel(int n, int thNum, int numThs) {
7        this.thNum = thNum;
8        this.numThs = numThs;
9        nTerms = n / numThs;
10    }
11    private double factorial(int k) {
12       long n = 10_000_000;
13       double x = (double)k;
14       double f = 1.0;
15       for(long i=1;i<=n;i++) {
16          f = f*Math.pow(1.0+1.0/i, x)/(1+x/i);
17       }
18       return f;
19    }
20       public double arcsin(double x) {
21       double as = 0.0;
22       double [] fact2i = new double[nTerms];
23       double [] fact2s = new double[nTerms];
24       int p = thNum;
25       for(int i=0; i<nTerms; i++) {
26          double f = factorial(p);
27          fact2s[i] = f*f;
28          fact2i[i] = factorial(2*p);
29          p += numThs;
30       }
31       int k = thNum;
32       for(int i=0;i<nTerms; i++) {
33          double numenator = fact2i[i]*Math.pow(x,
               2*k+1);
34          double
               denominator=Math.pow(4,k)*fact2s[i]*(2*k+1);
35          as += numenator/denominator;
36          k += numThs;
37       }
38       return as;
39    }
40 }
```

Лістинг 2.13 – Клас потоку

```
1  public class ThreadExecutor extends Thread{
2     private double res;
3     private ArcsinParallel as;
4     private double x;
5     public ThreadExecutor(int n, double x, int
          threadNum, int numThreads) {
6        this.x = x;
7        as = new ArcsinParallel(n, threadNum, numThreads);
8     }
9     public void run() {
10       res = as.arcsin(x);
11    }
12    public double getResult() {
13       return res;
14    }
15 }
```

Як бачимо, програма (ліст. 2.11-2.13) складається з трьох класів: головного класу `DriveParallelArcsin`, класу обчислення арксинусу `ArcsinParallel` і класу потоку `ThreadExecutor`.

Розпаралелювання задачі полягає в тому, що кожен член ряду розкладання (2.1) може обчислюватися незалежно від інших членів ряду. Отже, кожен член ряду можна обчислити в окремому потоці. У даній реалізації задачі розподіл обчислень організовано так: якщо, наприклад, використовується чотири потоки, то перший потік буде обчислювати члени ряду з номерами 0, 4, 8, ...; другий – члени ряду 1, 5, 9, ... ; третій – 2, 6, 10, ...; четвертий – 3, 7, 11, ...

Нижче наведено можливий результат роботи програми та графік залежності прискорення від кількості використовуваних потоків для конкретної моделі процесора (рис. 2.2).

```
Intel(R) Core(TM) i5-9400 CPU 2.9 GHz
Available cores: 6
Used threads: 2
x=0.0174524
math.asin(x)=0.017453286081679203
result=0.017453286081590552
error=8.865E-14
time=4.958 s
```

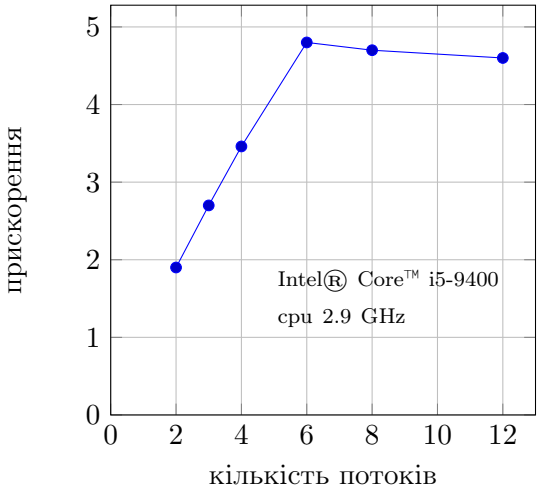

Рисунок 2.2 – Прискорення обчислень

2.2.2 Векторні обчислення

У даній програмі (ліст. 2.14-2.16) демонструється розв'язання задачі обчислення функції $\arcsin(x_i)$ за формулою (2.1) для наступних значень $x_i = 0.0174, 0.0259, 0.0342, 0.0423$. Векторизація задачі полягає у тому, що для обчислення функції у кожній точці x_i використовується один потік. Отже, всі потоки виконують однакову за обчислювальною складністю роботу, але у різних точках x_i.

Лістинг 2.14 – Клас потоку

```
1  public class ThreadExecutorV extends Thread{
2     private double res;
3     private ArcsinVector as;
4     private double x;
5     public ThreadExecutorV(int n, double x) {
6        this.x = x;
7        as = new ArcsinVector(n);
8     }
9     public void run() { res = as.arcsin(x);}
10    public double getResult() {
11       return res;
12 }  }
```

Лістинг 2.15 – Головний клас

```
1  import java.util.Locale;
2  public class DriveVectorThread {
3    public static void main(String[] args) {
4      int n = 48;   //кількість членів ряду на потік
5      double [] x = {0.0174, 0.0259, 0.0342, 0.0423};
6      double [] result = new double[x.length];
7      int numThreads = x.length; //кількість потоків
8      Runtime runtime = Runtime.getRuntime();
9      int cores = runtime.availableProcessors();
10     System.out.println("Intel(R) Core(TM) i3-2310M CPU
           2.1 GHz");
11     System.out.println("Available cores: " + cores);
12     System.out.println("Used threads: " + numThreads);
13     ThreadExecutorV [] ths=new
           ThreadExecutorV[numThreads];
14     for(int i=0; i<numThreads; i++) {
15       ths[i] = new ThreadExecutorV(n, x[i]);
16     }
17     long t1 = System.nanoTime();
18     for(int i=0; i<numThreads; i++) {
19       ths[i].start();
20     }
21     try {
22       for(int i=0; i<numThreads; i++) {
23         ths[i].join();
24         result[i] = ths[i].getResult();
25       }
26     } catch (InterruptedException e) {
27       e.printStackTrace(System.err);
28     }
29     long t2 = System.nanoTime();
30     for(int i=0; i<numThreads;i++) {
31       double reference = Math.asin(x[i]);
32       System.out.print("x=" + x[i] + "; ");
33       System.out.println("math.asin(x)=" + reference);
34       System.out.print("result=" + result[i] + "; ");
35       double err = Math.abs(reference-result[i]);
36       System.out.printf(Locale.UK, "error=%.3E\n", err);
37     }
38     double t = (double)(t2-t1)*1.0E-9;
39     System.out.printf(Locale.UK, "time=%.3f s", t);
40 } }
```

Лістинг 2.16 – Клас обчислення арксинусу

```
1  public class ArcsinVector {
2     private int nTerms;
3
4     public ArcsinVector(int n) {
5        nTerms = n;
6     }
7     private double factorial(int k) {
8        long n = 10_000_000;
9        double x = (double)k;
10       double f = 1.0;
11       for(long i=1;i<=n;i++) {
12          f = f*Math.pow(1.0+1.0/i, x)/(1+x/i);
13       }
14       return f;
15    }
16
17    public double arcsin(double x) {
18       double as = 0.0;
19       double [] fact2i = new double[nTerms];
20       double [] fact2s = new double[nTerms];
21       for(int i=0; i<nTerms; i++) {
22          double f = factorial(i);
23          fact2s[i] = f*f;
24          fact2i[i] = factorial(2*i);
25       }
26       for(int i=0;i<nTerms; i++) {
27          double numenator = fact2i[i]*Math.pow(x,
               2*i+1);
28          double denominator = Math.pow(4,
               i)*fact2s[i]*(2*i+1);
29          as += numenator/denominator;
30       }
31       return as;
32    }
33 }
```

Результат роботи програми може бути таким:

```
Intel(R) Core(TM) i3-2310M CPU 2.1 GHz
Available cores: 4
Used threads: 4
x=0.0174; math.asin(x)=0.01740087812364258
result=0.017400878123554724; error=8.786E-14
```

```
x=0.0259; math.asin(x)=0.02590289653761376
result=0.025902896537323816; error=2.899E-13
x=0.0342; math.asin(x)=0.03420667045951309
result=0.03420667045884492; error=6.682E-13
x=0.0423; math.asin(x)=0.042312624662275866
result=0.042312624661010226; error=1.266E-12
time=91.343 s
```

2.2.3 Задача «виробник-споживач»

Однією із класичних задач синхронізації паралельних та розподілених процесів є задача «виробник-споживач». Найпростішим варіантом цієї задачі є випадок, коли взаємодіють лише два процеси: один виробник і один споживач. Обмін об'єктами (повідомленнями, даними) між виробником та споживачем здійснюється через деякий спільний ресурс (об'єкт, потік, файл, область пам'яті), наприклад, буфер. До такого варіанту задачі можна віднести, наприклад, обмін між процесором комп'ютера та зовнішнім накопичувачем.

Найбільш простими, але обов'язковими вимогами до такої системи є наступні:

– у кожний момент часу лише один процес може мати доступ до спільного ресурсу;

– процеси повинні працювати із спільним ресурсом у певній послідовності;

– споживач може зчитувати дані із спільного ресурсу лише за умови, що виробник вже помістив дані до спільного ресурсу;

– виробник може записувати чергові дані до спільного ресурсу за умови, що попередні дані вже зчитані;

– процеси не можуть володіти спільним ресурсом нескінченно довго.

Програма (див. ліст. 2.17-2.20) моделює систему «виробник-споживач» і складається з класів: `Buffer`, `Producer`, `Consumer` і головного класу потоку `DemoProducerConsumer`.

Спільним ресурсом у цій програмі є поле `resource` екземпляра класу `Buffer`, який передається як параметр у потік класу `Producer` (виробник) та потік класу `Consumer` (споживач).

Потік `Producer` за допомогою методу `put()` записує число у змінну `resource`, а потік `Consumer` за допомогою методу `get()` зчитує значення цієї змінної. Для виключення одночасного доступу до змінної `resource` методи `put()` та `get()` оголошені як `synchronized`.

При виклику потоком `Producer` методу `put()` об'єкт класу `Buffer` стає недоступним для інших потоків після виклику у методі `put()` методу `wait()`.

Після запису числа у змінну `resource` викликається метод `notify()`, який робить доступним спільний ресурс. Аналогічно працює із спільним ресурсом і потік `Consumer`.

Лістинг 2.17 – Клас `Buffer`

```
1  public class Buffer {
2     private int resource;
3     private boolean flag = false;
4     public synchronized int get(String who) {
5        if(!flag)
6           try {
7              wait();
8           } catch (InterruptedException e) {
9              e.printStackTrace (System.err);
10          }
11       System.out.println(", "+who+" read: "+resource);
12       flag = false;
13       notify();
14       return resource;
15    }
16    public synchronized void put(int resource) {
17       if(flag)
18          try {
19             wait();
20          } catch(InterruptedException e) {
21             e.printStackTrace (System.err);
22          }
23       this.resource = resource;
24       System.out.print("Producer: written: "+resource);
25       flag = true;
26       notify();
27    }
28 }
```

Послідовність запису та зчитування даних регулюється за допомогою змінної `flag`. Після запису числа у змінну `resource` змінна `flag` отримує значення `true`, а після зчитування цього числа змінна `flag` отримує значення `false`.

У головному потоці (клас `DemoProducerConsumer`) створюються екземпляр класу `Buffer` та екземпляри потоків `Producer` і `Consumer`. Кількість звернень до загального ресурсу визначається змінною `n`.

Лістинг 2.18 – Клас потоку `Producer`

```
1  import java.util.Random;
2
3  public class Producer implements Runnable{
4     private Buffer buffer;
5     private int n;
6     public Producer(Buffer buffer, int n) {
7        this.buffer = buffer;
8        this.n = n;
9        new Thread(this, "Producer").start();
10    }
11    public void run() {
12       Random rand = new Random();
13       final long MIN_D = 1000L;
14       final long MAX_D = 2000L;
15       int i = 0;
16       while (i < n) {
17          int r = rand.nextInt(100, 199);
18          long delay = rand.nextLong(MIN_D, MAX_D);
19          try {
20             Thread.sleep(delay);
21          } catch(InterruptedException e) {
22             e.printStackTrace();
23          }
24          buffer.put(r);
25          i++;
26       }
27       System.out.println("\nProducer has finished
             writing");
28    }
29 }
```

Лістинг 2.19 – Клас потоку Consumer

```
1  import java.util.Random;
2
3  public class Consumer implements Runnable {
4     private String name;
5     private Buffer buffer;
6     private int n;
7     public Consumer(Buffer buffer, String name, int n) {
8        this.buffer = buffer;
9        this.name = name;
10       this.n = n;
11       new Thread(this, "Consumer").start();
12    }
13    public void run() {
14       Random rand = new Random();
15       final long MIN_D = 100L;
16       final long MAX_D = 500L;
17       int i=0;
18       while (i<n) {
19          long delay = rand.nextLong(MIN_D, MAX_D);
20          try {
21             Thread.sleep(delay);
22          } catch(InterruptedException e) {
23             e.printStackTrace();
24          }
25          buffer.get(name);
26          i++;
27       }
28       System.out.println("\n" + name + " has finished
             reading");
29    }
30 }
```

Лістинг 2.20 – Головний клас потоку DemoProducerConsumer

```
1  public class DemoProducerConsumer {
2     public static void main(String args[]) {
3        Buffer buffer = new Buffer();
4        int n = 5;
5        new Producer(buffer, n);
6        new Consumer(buffer, "Consumer", n);
7     }
8  }
```

Зазначимо, що розглянутий спосіб синхронізації потоків не єдиний, але найпростіший у Java. У пакеті `java.util.concurrent` містяться класи, які забезпечують високу продуктивність, масштабованість, і вдосконалені утиліти синхронізації [4].

Нижче наведено можливий результат роботи програми.

```
Producer: written: 170, Consumer read: 170
Producer: written: 141, Consumer read: 141
Producer: written: 140, Consumer read: 140
Producer: written: 142, Consumer read: 142
Producer: written: 136, Consumer read: 136

Producer has finished writing
Consumer has finished reading
```

2.3 Завдання до лабораторної роботи

Для створення власних змістовних багатопотокових програм рекомендується спочатку вивчити та реалізувати всі приклади програм даної лабораторної роботи.

2.3.1 Завдання до розділу 2.2.1

У даному завданні необхідно реалізувати багатопотокову програму для обчислення функції згідно з варіантом з табл. 2.1. Кожна функція з таблиці є складовою двох інших функцій.

Таблиця 2.1 – Вихідні дані для варіантів завдання

№	$f(x)$	x	x_i
1	$\ln(1+\sin(x))$	0.0174	$0.0174 \cdot i$
2	$\exp(\arctan(x))$	-0.02	$-0.02 \cdot i$
3	$\cos(\ln(1+x))$	-0.015	$-0.015 \cdot i$
4	$\arctan(\exp(x))$	-0.174	$-(0.174 + 0.02 \cdot i)$
5	$\ln(1+\cos(x))$	1.5	$1.5 + 0.01 \cdot i$

Для обчислення цих функцій у заданій точці x використовуються формули їх розкладання до ряду Маклорена (2.3)–(2.7):

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} \cdot x^{2n+1}; \tag{2.3}$$

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} \cdot x^{2n}; \tag{2.4}$$

$$\arctan(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \cdot x^{2n+1}, \quad -1 < x < 1; \tag{2.5}$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}; \tag{2.6}$$

$$\ln(1+x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{n+1} \cdot x^{n+1}, \quad -1 < x < 1. \tag{2.7}$$

Значення аргументу x для кожної функції наведено у третьому стовпчику таблиці[1].

Як випливає з табл. 2.1, у кожному варіанті необхідно обчислити дві функції. Позначимо «внутрішню» функцію через $y(x)$, а «зовнішню» – через $f(y)$. Наприклад, для першого варіанта можна записати $y(x) = 1 + \sin(x)$, $f(y) = \ln(y(x))$.

У такому випадку головний клас програми умовно складатиметься із двох частин: у першій частині обчислюється функція $y(x)$, у другій частині – функція $f(y)$.

Кожна частина повинна виконувати обчислення з використанням кількох потоків.

Таким чином, головний клас `DriveParallelMyVariant` можна представити у вигляді, який наведено нижче.

[1] У четвертому стовпчику таблиці наведено дані для завдання 2.3.2

```
public class DriveParallelMyVariant{
  public static void main{String[] args}{
    <ініціалізація полів класу>

    <створення екземплярів класу потоку>
    <багатопотокове обчислення функції y(x)>
    <очікування завершення роботи потоків>

    <створення екземплярів класу потоку>
    <багатопотокове обчислення функції f(y)>
    <очікування завершення роботи потоків>

    <виведення на консоль результатів обчислень>
  }
}
```

Вимоги до виконання завдання:

– функції $y(x)$ і $f(y(x))$ повинні обчислюватися за допомогою декількох потоків з використанням розподілу роботи за ітераціями циклу (див. п. 2.2.1);

– для обчислення функцій $y(x)$ і $f(y(x))$ потрібно використовувати однакову кількість потоків;

– програма повинна виводити на консоль результат роботи у вигляді, наведеному на стор. 37;

– виконайте обчислення прискорення програми при використанні 2, 3, 4, 6, 8 і 12 потоків та побудуйте графік прискорення залежно від кількості потоків (див. рис. 2.2).

2.3.2 Завдання до розділу 2.2.2

У даному завданні необхідно розробити програму, яка виконує багатопотокові векторні обчислення.

Всі дочірні потоки виконують однакову роботу, але з різними даними. Обчислювальним навантаженням для потоків є функція згідно з варіантом з табл. 2.1 і даними з четвертого стовпчика цієї таблиці.

Вимоги до програми:

– кількість дочірніх потоків повинна дорівнювати числу фізичних ядер процесора;

– кожен потік повинен обчислювати значення функції для аргументу, визначеного у четвертому стовпчику таблиці. Аргумент функції визначається виразом, у якому i є номером потоку. Тобто, у варіанті 4 перший потік обчислює для $x_1 = -(0.174 + 0.02 \cdot 1)$, другий потік – для $x_2 = -(0.174 + 0.02 \cdot 2)$ і так далі;

– програма повинна виводити результат роботи на консоль у вигляді, представленому на стор. 40.

2.3.3 Завдання до розділу 2.2.3

У завданні необхідно розробити багатопотокову програму, що моделює систему «виробник-споживачі». У цій системі має бути чотири класи: `Broker` (брокер), `Producer` (виробник), `Consumer` (споживач) і головний клас, який створює об'єкти та потоки.

Програма має моделювати наступний сценарій:

– брокер отримує від виробника два види товару, нехай це буде чоловічий та жіночий одяг;

– у сценарії беруть участь два види споживачів: чоловічого одягу та жіночого (які мають імена, наприклад, `Consumer-M` та `Consumer-W`);

– брокер відправляє товар за призначенням: чоловічий одяг споживачеві чоловічого одягу, жіночий – споживачеві жіночого одягу.

Програма повинна відповідати таким вимогам:

– у програмі повинні створюватися: об'єкт типу `Broker`; потік типу `Producer`; два потоки типу `Consumer`;

– об'єкт класу `Broker` має передаватися у конструктори потоків;

– «товаром» моделі можуть бути повідомлення зі значеннями, які мають відрізнятися залежно від типу товару (чоловічий чи жіночий) та номерів повідомлень;

– повідомленнями можуть бути об'єкти типу `String` (рядкові змінні) або екземпляри придуманого вами класу;

– номери повідомлень повинні бути випадковими цілими числами, наприклад, для чоловічого одягу в діапазоні від 200 до 299, для жіночого – в діапазоні від 100 до 199;

– моменти часу надсилання та отримання повідомлень повинні бути випадковими, наприклад, в діапазоні від 1000 до 3000 мілісекунд;

– функції надсилання та отримання повідомлень повинні бути синхронізованими (`synchronized`) методами класу Broker;

– номери повідомлень, що надсилаються і одержуються, повинні бути впорядковані, тобто, якщо надіслано, наприклад, повідомлення `m212` і `w131`, то споживач `Consumer-M` повинен отримати повідомлення `m212`, а споживач `Consumer-W` – повідомлення `w131`.

Нижче наведено приклад можливого результату роботи програми, в якому потік з ім'ям `Consuner-M`, що імітує споживача чоловічого одягу, отримує повідомлення, позначене символом `m`, а потік з ім'ям `Consumer-W`, що імітує споживача жіночого одягу, отримує повідомлення, позначене символом `w`.

```
Producer: put: m212 w131
Consumer-W got: w131 Consumer-M got: m212
Producer: put: m295 w178
Consumer-M got: m295 Consumer-W got: w178
Producer: put: m213 w157
Consumer-W got: w157 Consumer-M got: m213
Producer: put: m231 w116
Consumer-M got: m231 Consumer-W got: w116
Producer: put: m264 w159
Producer has stopped supplying
Consumer-M got: m264
Consumer-M has finished receiving
Consumer-W got: w159
Consumer-W has finished receiving
```

2.4 Контрольні питання

1. Навести приклади способів створення дочірніх потоків у Java.
2. Перелічити і описати стани життєвого циклу потоку у Java.
3. Як завершити роботу потоку?
4. Як встановити та дізнатися пріоритет потоку?
5. Яким є пріоритет потоку за замовчуванням?

6. Який метод можна використовувати, щоб зупинити виконання потоку на заданий час?

7. Яке призначення модифікатора `synchronized`?

8. Описати роботу методів `wait()`, `notify()` і `notifyAll()`.

9. За допомогою якого методу можна дізнатись про кількість доступних процесорів (ядер)?

10. Поясніть відмінність між паралельними і векторними обчисленнями.

11. Що означає термін «гіперпотоковість»?

12. Дати визначення основним характеристикам паралельних алгоритмів: ступінь паралелізму, середній ступінь паралелізму, прискорення та ефективність.

13. Викласти формальне виведення закону Амдала.

2.5 Зміст звіту

1. Титульний лист, оформлений відповідно до вимог.
2. Мета роботи.
3. Фрагменти програмного коду, розроблені самостійно.
4. Відповіді на контрольні питання.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Williams A. C++ Concurrency in Action. 2-nd ed. / A. Williams. – Manning Publications Co., 2019.

2. Hirvonen J. Distributed Algorithms / J. Hirvonen, J. Suomela. — Aalto: Aalto University, 2023. – 221 p. URL: https://jukkasuomela.fi/da2020/da2020.pdf.

3. TOP 500. URL: https://top500.org/lists/top500/2025/11/.

4. Horstmann C. S. Core Java. Volume I: Fundametals. 13-th ed. / C. S. Horstmann. – Pearson Education, 2024.

5. Horstmann C. S. Core Java. Volume II: Advanced Features. 13-th ed. / C. S. Horstmann. – Pearson Education, 2024.

6. Соловей О. Л. Технології розподілених систем та паралельних обчислень / О. Л. Соловей. – Київ : КНУБА, 2024. – 100 с. URL: https://library.knuba.edu.ua/books/23_1_24_1.pdf.

ДОДАТОК А
Список лістингів