

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ

**до лабораторних робіт
з дисципліни**

«Паралельні та розподілені обчислення»

для студентів спеціальності 123 «Комп'ютерна інженерія»

всіх форм навчання

Частина 1

2018

Методичні вказівки до лабораторних робіт з дисципліни "Паралельні та розподілені обчислення" для студентів спеціальності 123 "Комп'ютерна інженерія" всіх форм навчання. Частина 1 / Укл.: Р.К. Кудерметов, В.В. Шкарупило, О.В. Польська. – Запоріжжя: ЗНТУ, 2018. – 38 с.

Укладачі:

Р.К. Кудерметов, доцент, к.т.н.
В.В. Шкарупило, доцент, к.т.н.
О.В. Польська, ст. викладач

Рецензент:

М.Ю. Тягунова, доцент, к.т.н.

Відповідальний
за випуск:

Р.К. Кудерметов, доцент, к.т.н.

Затверджено:
на засіданні кафедри

"Комп'ютерні системи та мережі"
Протокол № 9 від 17.05.2018 р.

Рекомендовано до видання:

НМК факультету КНТ
Протокол № 10 від 28.05.2018 р.

ЗМІСТ

ВСТУП.....	5
ЛАБОРАТОРНА РОБОТА №1	
1 БАГАТОПОТОЧНЕ ПРОГРАМУВАННЯ У C++.....	6
1.1 Багатопоточність у C++.....	6
1.2 Розподілення обчислювального навантаження між потоками.....	8
1.3 Завдання до лабораторної роботи.....	15
1.4 Зміст звіту.....	16
1.5 Контрольні питання.....	17
ЛАБОРАТОРНА РОБОТА №2	
2 БАГАТОПОТОЧНЕ ПРОГРАМУВАННЯ У JAVA.....	18
2.1 Вступ до потоків Java.....	18
2.2 Використання потоків Java у паралельних і векторних обчисленнях.....	27
2.3 Завдання до лабораторної роботи.....	34
2.4 Зміст звіту.....	37
2.5 Контрольні питання.....	37
ЛІТЕРАТУРА.....	38

ВСТУП

Метою даних методичних вказівок є освоєння практичного курсу паралельного програмування для комп'ютерних систем з розподіленою і спільною пам'яттю.

Засвоєння матеріалів даних методичних вказівок потребує від студентів значного обсягу самостійної роботи, оскільки лише практичне програмування, вирішення конкретних проблем у кожній програмі може закласти міцний фундамент знань та навичок у паралельному програмуванні.

Певні деталі опису технологій паралельного програмування не наведено з метою спрощення викладення та сприйняття матеріалу. Тим не менш, значна частина відомостей, викладених у даній роботі, ґрунтується на кращих, сучасних і класичних літературних джерелах і стандартах.

Автори вважають, що матеріали, практичні приклади і завдання, викладені у даних методичних вказівках, під силу студентам, які бажають пов'язати свою діяльність із комп'ютерною інженерією та комп'ютерними науками, і будуть сприяти набуттю професійних знань і навичок у паралельному програмуванні.

Методичні вказівки складаються з двох частин. Частина 1 присвячена питанням багатопоточного програмування, у частині 2 розглядаються технології паралельних обчислень OpenMP та MPI.

ЛАБОРАТОРНА РОБОТА №1

1 БАГАТОПОТОЧНЕ ПРОГРАМУВАННЯ У C++

Мета роботи – вивчити основи багатопоточного програмування на мові C++ та його використання для реалізації паралельних алгоритмів і програм.

1.1 Багатопоточність у C++

Розглядаються деякі з основних засобів реалізації багатопоточності мовою C++, що увійшли до стандарту C++11 [1], і котрі необхідні для реалізації паралельних алгоритмів, що демонструють переваги паралельних обчислень у порівнянні із послідовними, перш за все, з позиції продуктивності.

Зауваження: для функціонування наведеного у роботі програмного коду необхідно у середовищі Dev-C++ у меню "Сервіс" > "Параметри компілятора" > "Компілятор" у верхньому вікні "Додати наступні команди до запуску компілятора:" додати команду

```
-std=c++11.
```

Програма 1.1.1 Створення і запуск потоків

Найпростіша багатопоточна програма наведена у лістингу 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` (`#include <thread>`). Конструктор цього класу приймає функцію та опційні параметри. Функція, що передається до конструктора, виконується у окремому потоці.

Кожний створений потік має власний ідентифікатор (ID), тип котрого – `thread::id`. Ідентифікатор потоку можна одержати шляхом виклику статичного методу класу `this_thread::get_id()` чи шляхом виклику методу `get_id()` екземпляру класу `thread`, наприклад, `th1.get_id()`.

Лістинг 1.1 – Проста багатопоточна програма на мові C++

```
1 #include <iostream>
2 #include <sstream>
```

```

3  #include <thread>
4  using namespace std;
5
6  void hello(string name)
7  {
8      cout << "Hello, " << name << " Thread ID = " <<
9          this_thread::get_id() << endl; }
10
11 void getid(int i)
12 {
13     thread::id id = this_thread::get_id();
14     stringstream str;
15     str << id << i << endl;
16     str >> i;
17     cout << i << endl;
18 }
19
20 int main() {
21     thread th1(hello, "csn!");
22     thread th2(hello, "scs!");
23     th1.join();
24     th2.join();
25     thread th[3];
26     for(int i = 0; i < 3; i++)
27         th[i] = thread(getid,i+1);
28     for(int i = 0; i < 3; i++)
29         th[i].join();
30     return 0;
31 }

```

У рядках 25–28 лістингу 1.1 показано як створювати масив потоків.

У результаті роботи програми одержимо на екрані повідомлення наступного змісту (не обов'язково у зазначеній послідовності рядків):

```

Hello, csn! Thread ID = 2
Hello, scs! Thread ID = 3
41
52
63

```

З наведеного результату роботи програми бачимо, що функція `main()` виконується у потоці з `ID = 1`. Для того, щоб потік, виконуючий функцію `main()`, не завершив свою роботу раніше за дочірні потоки `th1` і `th2`, використовуються виклики функції `join()`, що змушує головний (батьківський) потік очікувати на завершення роботи дочірніх потоків.

1.2 Розподілення обчислювального навантаження між потоками

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

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

Даний інтеграл можна обчислити методом прямокутників:

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

де $h=1/n$ – крок дискретизації; $x_i = h(i+1/2)$, $i=1, 2, \dots, n-1$; n – кількість підінтервалів.

Суть розпаралелювання обчислень для даної задачі полягає у тому, що підінтегральна функція $f(x)$ може обчислюватися у кожній точці дискретизації незалежно. Таким чином, обчислення функції у першій точці можна виконати у першому програмному потоці, обчислення у другій точці – у другому програмному потоці і т.д. Отже, за наявності у процесорі використовуваного комп'ютера декількох ядер, обчислення функції $f(x)$ у різних точках дискретизації можуть виконуватися на різних ядрах процесора.

Програма 1.2.1. Обчислення числа π

У даній програмі (лістинг 1.2) головний потік, що виконує функцію `main()` (`ID = 1`), створює два дочірніх потоки з `ID = 2` і

ID = 3 (рядок 48). Обчислення на парних підінтервалах ($i = 0, 2, \dots$) виконуються потоком з ID = 2, обчислення на непарних підінтервалах ($i = 1, 3, \dots$) – потоком з ID = 3.

У даній реалізації потоки мають доступ до спільного ресурсу – змінної `pi` (рядок 28). Для унеможливлення одночасних звернень незалежних потоків до даного ресурсу (цей процес називається гонкою даних) у програмі використовується примітив синхронізації, що носить назву "м'ютекс" (mutex – **m**utual **e**xclusion, взаємне виключення). Перед тим як звернутися до спільного ресурсу, потік захоплює (`lock()`) м'ютекс, а по завершенні роботи з ресурсом – вивільняє (`unlock()`) м'ютекс.

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]. Процес захоплення і вивільнення м'ютекса реалізовано у рядках 27 і 29, відповідно.

Відзначимо, що використання м'ютекса у даній програмі наведено виключно з навчальною метою, оскільки реалізувати дану програму можна було би і без м'ютекса, наприклад, накопичуючи суми результатів обчислень $f(x)$ у кожному з потоків, а результуюче складання цих результатів виконати у головному потоці, після завершення роботи дочірніх потоків.

Дана програма містить також функцію `computePi1()` (рядки 31–41), що призначена для обчислення значення π послідовним шляхом (в одному потоці) у функції `main()`, і демонструє один із можливих шляхів визначення часових витрат на проведення обчислення (див., наприклад, рядки 46, 52, 55).

Лістинг 1.2 – Розподілення обчислень за ітераціями

```

1  #include <iostream>
2  #include <mutex>
3  #include <thread>
4  #include <iomanip>
5  #include <cmath>
6
7  using namespace std ;

```

```

8
9  const double pi0 = 3.141592653589793238462643;
10 double pi = 0.0;
11 mutex m;
12 long intervals = 5000000;
13 int numThreads = 2;
14 long t1,t2;
15
16 void computePi(int num)
17 {
18     double x, width, localSum = 0.0;
19     width = 1.0/intervals;
20     for(int i=num; i<intervals; i+=numThreads)
21     {
22         x = (i +0.5)*width;
23         localSum += 4.0/(1.0 + x*x);
24     }
25     localSum *= width;
26
27     m.lock();
28     pi += localSum;
29     m.unlock();
30 }
31 void computePi1()
32 {
33     double x, width;
34     width = 1.0/intervals;
35     for(int i=0; i<intervals; i++)
36     {
37         x = (i+0.5)*width;
38         pi += 4.0/(1.0 + x*x);
39     }
40     pi *= width;
41 }
42
43 int main()
44 {
45     thread th[numThreads];
46     t1 = clock();
47     for(int i = 0; i < numThreads; i++)
48         th[i] = thread(computePi,i);
49

```

```

50     for(int i = 0; i < numThreads; i++)
51         th[i].join();
52     t2 = clock();
53     cout << setprecision(16) << pi << endl;
54     cout << setprecision(4) << "err: " <<
55         fabs(pi0 - pi) << endl;
56     cout << "time: " << t2-t1 << endl;
57     pi = 0.0;
58     t1 = clock();
59     computePi1();
60     t2 = clock();
61     cout << setprecision(16) << pi << endl;
62     cout << setprecision(4) << "err: " <<
63         fabs(pi0 - pi) << endl;
64     cout << "time: " << t2-t1 << endl;
65     return 0;
66 }

```

Програма 1.2.2. Розкладання числа на прості співмножники

У даній програмі розпаралелюється задача розкладання масиву чисел на прості співмножники. Нагадаємо, простими називаються числа, що діляться лише на одиницю і на самих себе. Ряд простих чисел має наступний вигляд: 2, 3, 5, 7, 11, 13, 17...

Перевірка, чи є число x простим, зводиться до перевірки того, чи існує на відріжку $[2; \sqrt{x}]$ число, на яке ділиться x . Реалізація алгоритму такої перевірки на мові C++ наведена у лістингу 1.3.

Лістинг 1.3

```

1  bool algorithm(int x)
2  {
3      for(int i = 2; i <= sqrt(x); i++)
4      {
5          if (x % i == 0)
6          {
7              return false;
8          }
9      }
10     return true;
11 }

```

Реалізація програми наведена у лістингу 1.4. Великі числа у програмі мають тип `unsigned long long`, Для цього у рядку 9 зроблено визначення макросу `ullong`. Числа можуть мати різну кількість простих співмножників, тому для кожного з них використовується тип `vector`. Тип `vector` – це клас, який призначений для збереження масивів даних змінного розміру у динамічній пам'яті. Цей клас містить набір функцій-членів для роботи з даними, які у ньому зберігаються: різноманітні функції доступу до елементів, визначення розміру масиву, додавання/видалення елементів і т.п.

Функцію обчислення розкладання (факторизації) наведено у рядках 13–30 коду лістингу 1.4. Для заповнення масиву чисел псевдовипадковими значеннями у рядках 32–43 визначено функцію, що генерує великі псевдовипадкові числа.

У даній програмі для оцінки числа апаратних потоків (процесорів, ядер) використовується функція `thread::hardware_concurrency()`. Обчислювальне навантаження полягає у розкладанні восьми великих псевдовипадкових чисел на прості співмножники, яке розподіляється між апаратними потоками порівну. Так, якщо у процесора є два ядра, то кожне з них буде обчислювати розкладання для чотирьох чисел. Для кожного числа визначається екземпляр класу `vector`, елементами якого є прості співмножники.

Лістинг 1.4 – Програма для розкладання великих чисел на прості співмножники

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <thread>
4  #include <vector>
5  #include <cmath>
6  #include <climits>
7  #include <ctime>
8
9  #define ullong unsigned long long
10
11 using namespace std;
```

```

12
13 void factoring(ulong arr[], int size,
    vector <ulong> vec[])
14 {
15     ulong x, div;
16     for(int i = 0; i < size; i++)
17     {
18         x = arr[i];
19         for(int j = 2; j <= sqrt(x); j++)
20         {
21             while (x % j == 0)
22             {
23                 vec[i].push_back(j);
24                 x /= j;
25             }
26         }
27         if (x != 1)
28             vec[i].push_back(x);
29     }
30 }
31
32 ulong ullrand()
33 {
34     ulong rnd = 0;
35     //ulong counter = ULLONG_MAX;
36     //ULLONG = 18446744073709551615
37     ulong counter = 1E+13;
38     while(counter > 0)
39     {
40         rnd = (rnd * (RAND_MAX + 1)) + rand();
41         counter /= (RAND_MAX + 1);
42     }
43     return rnd;
44 }
45 int main()
46 {
47     srand(time(NULL));
48     int n = 8;
49     long t;
50     int numThreads = thread::hardware_concurrency();
51

```

```

52  if(n%numThreads != 0)
53  {
54      cout << "The array size must be a multiple
55          of the threads number!\n";
56      exit(0);
57  }
58  thread th[numThreads];
59  int size_scatter = n/numThreads;
60  ullong* arr[numThreads];
61  vector<ullong> **vec =
        new vector<ullong>*[numThreads];
62
63  for(int i = 0; i < numThreads; i++)
64  {
65      arr[i] = new ullong[size_scatter];
66      vec[i] = new vector<ullong>[size_scatter];
67      for(int j = 0; j < size_scatter; j++)
68      {
69          arr[i][j] = ullrand();
70          vec[i][j].reserve(10);
71      }
72      t = clock();
73      th[i] = thread(factoring, arr[i],
        size_scatter, vec[i]);
74  }
75
76  for(int i = 0; i < numThreads; i++)
77      th[i].join();
78
79  long time_p = clock() - t;
80
81  for(int i = 0; i < numThreads; i++)
82  {
83      for(int j = 0; j < size_scatter; j++)
84      {
85          cout << "arr[" << i << "][" << j << "] = ";
86          cout << arr[i][j] << " = 1";
87          for(int k = 0; k < vec[i][j].size(); k++)
88              cout << "*" << vec[i][j][k];
89          cout << endl;
90      }

```

```

91     }
92     cout << "time_p = " << time_p << endl;
93
94     for(int i = 0; i < numThreads; i++)
95         delete [] arr[i];
96
97     return 0;
98 }

```

Результат роботи програми може мати наступний вигляд:

```

arr[0][0] = 15439557483638 = 1*2*103*74949308173
arr[0][1] = 28967148887428 = 1*2*2*499*14512599643
arr[0][2] = 17054992685100 = 1*2*2*3*5*5*56849975617
arr[0][3] = 21862690405418 = 1*2*557*82469*237973
arr[1][0] = 15133720203036 = 1*2*2*3*3*3*140127038917
arr[1][1] = 17330244595916 = 1*2*2*7*1709*3929*92177
arr[1][2] = 9600061402466 = 1*2*224351*21395183
arr[1][3] = 13629418694225 = 1*5*5*17*36497*878681
time_p = 31

```

1.3 Завдання до лабораторної роботи

1.3.1 Завдання до програми 1.1.1 (лістинг 1.1).

1.3.1.1 Реалізувати програму, наведену у лістингу 1.1, і пояснити результат її роботи.

1.3.1.2 Вивчити використані у програмі функції.

1.3.1.3 Розробити власну багатопоточну програму згідно наступних вимог:

- у програмі має створюватися масив з 10 потоків;
- до конструктора кожного потоку має передаватися функція із трьома аргументами: перший аргумент – ім'я, другий аргумент – прізвище, третій аргумент – випадкове ціле число у діапазоні від 0 до 9;
- функція має виводити на консоль лише ім'я, якщо третій аргумент парний, і лише прізвище, якщо третій аргумент непарний.

1.3.2 Завдання до програми 1.2.1 (лістинг 1.2).

1.3.2.1 Реалізувати програму, код якої наведено у лістингу 1.2, вивчити використаний спосіб розподілення задач між потоками.

Визначити, на яких підінтервалах будуть виконувати обчислення потоки, у випадку, якщо їх чотири.

1.3.2.2 Провести достатню кількість експериментів з програмою і визначити середнє значення прискорення для випадків двох та чотирьох потоків.

1.3.2.3 Допрацювати програму (лістинг 1.2) таким чином, щоб вона коректно виконувалася без використання примітиву синхронізації *mutex*.

1.3.2.4 Розробити програму, у якій є функції для паралельного та послідовного обчислень числа π за формулою на основі ряду Лейбніца:

$$\pi \approx 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right) = 4 \cdot \sum_{i=0}^{\infty} \frac{(-1)^i}{2 \cdot i + 1}. \quad (1.3)$$

1.3.2.5 Експериментально визначити значення кількості членів ряду Лейбніца, при якому число π обчислюється із похибкою $\varepsilon \leq 1 \cdot 10^{-7}$.

1.3.2.6 Визначити прискорення паралельного методу обчислення при розподіленні задач між двома та чотирма потоками.

1.3.3 Завдання до програми 1.2.2 (лістинг 1.4).

1.3.3.1 Реалізувати та вивчити програму, код якої подано у лістингу 1.4.

1.3.3.2 Допрацювати програму таким чином, щоб можна було оцінити прискорення обчислень за рахунок розпаралелювання.

1.3.3.3 Провести достатню кількість експериментів з програмою та визначити прискорення для випадків двох та чотирьох потоків.

1.4 Зміст звіту

1.4.1 Титульний лист.

1.4.2 Мета роботи.

1.4.3 Фрагменти програмного коду, розроблені самостійно.

1.4.4 Одержані експериментальні оцінки прискорення (див. завдання до програм). Вміти їх пояснювати.

1.4.5 Відповіді на контрольні питання. Вміти їх пояснювати в усній формі.

1.4.6 Виписати усі функції для організації багатопоточних обчислень на мові C++, використані у даній лабораторній роботі, навести їх короткі описи.

1.5 Контрольні питання

1.5.1 У чому полягає закон Мура? Навести приклади, що підтверджують його справедливість.

1.5.2 Що таке прискорення паралельного алгоритму?

1.5.3 Навести фактори, що обумовлюють швидкодію комп'ютерних систем.

1.5.4 Пояснити виникнення прискорення при конвексній, паралельній та векторній обробках даних.

1.5.5 Що таке рейтинг ТОП 500? Навести дані про перші п'ять найпотужніших систем.

1.5.6 Пояснити суть розпаралелювання за ітераціями у програмі 1.2.1, наведений у листингу 1.2.

1.5.7 Чому алгоритм множення матриць, наведений справа, працює швидше (табл. 1.1)?

Таблиця 1.1 – Порівняння реалізацій алгоритму множення матриць

Алгоритм "ijk"	Алгоритм "ikj"
<pre>for (i=0; i<n; i++) for (j=0; j<n; j++) for (k=0; k<n; k++) c[i][j] += a[i][k] * b[k][j];</pre>	<pre>for (i=0; i<n; i++) for (k=0; k<n; k++) for (j=0; j<n; j++) c[i][j] += a[i][k] * b[k][j];</pre>

ЛАБОРАТОРНА РОБОТА №2

2 БАГАТОПОТОЧНЕ ПРОГРАМУВАННЯ У JAVA

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

2.1 Вступ до потоків Java

У Java класи потоків визначаються як підкласи класу `Thread`, що інкапсулює (включає) основні методи роботи з потоками [2]. При запуску додатку 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 спадкувати (розширяти) інший клас.

У методі `run()` виконується уся робота потоку. Для запуску потоку першим способом необхідно створювати екземпляр класу потоку та викликати успадкований ним метод `start()`. Для запуску потоку, створений 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 ґрунтується на наступних чотирьох станах (рис. 2.1):

- `NEW` – потік створюється, але не виконується;
- `RUNNABLE` – потік виконується, в цей стан потік потрапляє після виконання методу `start()` та виклику ним методу `run()`;
- `BLOCKED` – потік заблоковано;

– DEAD – потік завершив свою роботу.

Визначити стан потоку можна за допомогою методу `getState()`. Звичайно потік завершує свою роботу, коли завершується виконання його методу `run()`. Хоча у Java є методи примусового завершення роботи потоку `stop()` і керування його блокуванням `suspend()` та `resume()`, ці методи вважаються застарілими і ними користуватися не рекомендується.

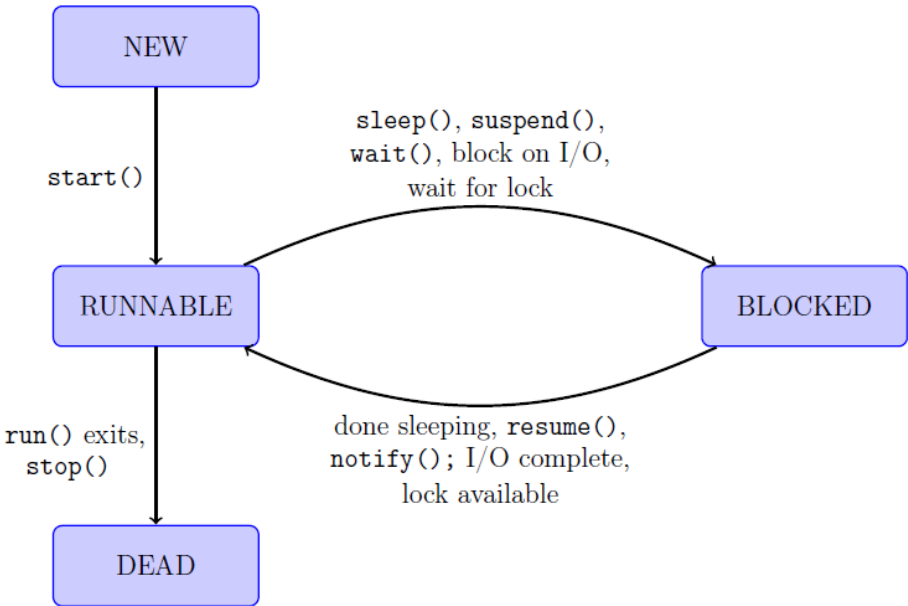


Рисунок 2.1 – Стани потоку

Програма 2.1.1. Створення і запуск потоків

Класи програми подано у лістингах 2.1–2.4. Програма демонструє створення і запуск потоків описаними вище двома способами. Потік класу `Letters` спадкує клас `Thread`, а потоки класів `Numbers` і `Symbols` реалізують інтерфейс `Runnable`.

У рядках 7 і 8 лістингу 2.4 показано, як запустити потоки класів `Letters` і `Numbers` у головному класі. Потік класу `Symbols` запускається у його конструкторі (рядок 5, лістинг 2.3). Метод `join()`

класу Thread (рядки 12–14 лістингу 2.4) використовується для очікування головним потоком завершення роботи дочірніх потоків.

Лістинг 2.1 – Клас потоку Letters спадкує клас Thread

```

1 public class Letters extends Thread {
2     public void run(){
3         for(int i = 'a'; i <= 'z'; i++){
4             System.out.print((char)i + " ");
5         }
6     }
7 }

```

Лістинг 2.2 – Клас потоку Numbers реалізує інтерфейс Runnable

```

1 public class Numbers implements Runnable {
2     public void run(){
3         for(int i = 0; i < 26; i++){
4             System.out.print(i + " ");
5         }
6     }
7 }

```

Лістинг 2.3 – Клас потоку Symbols реалізує інтерфейс Runnable

```

1 public class Symbols implements Runnable {
2     Thread th;
3     Symbols(){
4         th = new Thread(this);
5         th.start();
6     }
7     public void run(){
8         for(int i = 33; i <= 47; i++) {
9             System.out.print((char)i + " ");
10        }
11    }
12 }

```

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     }
19 }

```

Програма 2.1.2. Стани потоку

Сценарій даної програми побудовано таким чином, щоб за допомогою методу `getState()` класу `Thread` перевірити стани потоків, викликані різними ситуаціями. Головний клас `TControlDemo` програми подано у лістингу 2.5. У методі `main()` запускаються на виконання два потоки класу `TControl`, код якого подано у лістингу 2.6.

У програмі також показано, як можна виводити на консоль назву потоку (рядок 5, лістинг 2.5), встановлювати (рядок 8, лістинг 2.5) та одержувати (рядок 11, лістинг 2.6) значення пріоритету потоку.

Лістинг 2.5 – Клас TControlDemo

```

1  public class TControlDemo {
2
3      public static void main(String[] args) {
4

```

```

5      System.out.println("message 0: " +
6          Thread.currentThread());
7
8      Thread th1 = new Thread(new TControl("th1"));
9      th1.setPriority(Thread.MIN_PRIORITY+2);
10     Thread th2 = new Thread(new TControl("th2"));
11
12     System.out.println("message1: th1 is " +
13         th1.getState());
14
15     th1.start();
16
17     try {
18         Thread.sleep(50);
19     } catch (InterruptedException e) {
20         e.printStackTrace(System.err);
21     }
22
23     System.out.println("message2: th1 is " +
24         th1.getState());
25     System.out.println("message3: th2 is " +
26         th2.getState());
27
28     th2.start();
29
30     try {
31         Thread.sleep(10);
32     } catch (InterruptedException e) {
33         e.printStackTrace(System.err);
34     }
35
36     System.out.println("message4: th2 is " +
37         th2.getState());
38     System.out.println("message5: th1 is " +
39         th1.getState());
40
41     try {
42         th1.join();
43     }
44     System.out.println("message6: th1 is " +
45         th1.getState());
46     th2.join();
47     System.out.println("message7: th2 is " +

```

```

        th2.getState());
40     } catch (InterruptedException e) {
41         e.printStackTrace(System.err);
42     }
43 }
44 }

```

ЛІСТИНГ 2.6 – Клас TControl

```

1  public class TControl implements Runnable {
2
3      private String threadName;
4
5      TControl(String name){
6          threadName = name;
7      }
8
9      public void run() {
10         Thread.State state =
11             Thread.currentThread().getState();
12         int priority =
13             Thread.currentThread().getPriority();
14         System.out.println("messageA: " + threadName +
15             "priority: " + priority);
16         System.out.println("messageB: " + threadName +
17             " is " + state );
18         try {
19             Thread.sleep(100);
20         } catch(InterruptedException e) {
21             e.printStackTrace(System.err);
22         }
23     }
24
25     for(int i = 1; i < 6; i++) {
26         System.out.println("messageC: "+ threadName + " "+i);
27     }
28 }

```

Результат роботи програми 2.1.2 подано у лістингу 2.7.

Лістинг 2.7 – Пріоритети та стани потоків

```
message0: Thread [main,5,main]
message1: th1 is NEW
messageA: th1 priority: 3
messageB: th1 is RUNNABLE
message2: th1 is TIMED_WAITING
message3: th2 is NEW
messageA: th2 priority: 5
messageB: th2 is RUNNABLE
message4: th2 is TIMED_WAITING
message5: th1 is TIMED_WAITING
messageC: th1 1
messageC: th1 2
messageC: th1 3
messageC: th1 4
messageC: th1 5
message6: th1 is TERMINATED
messageC: th2 1
messageC: th2 2
messageC: th2 3
messageC: th2 4
messageC: th2 5
message7: th2 is TERMINATED
```

Програма 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, які викликали 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 оператор `synchronized`.

У даній програмі демонструється найпростіший спосіб синхронізації взаємодії потоків. У програмі створюються два потоки класу `Waiter` (лістинг 2.9), які у блоці синхронізації `msg` (рядки 10–18) намагаються одержати доступ до об'єкту `msg` класу `Message` (лістинг 2.8). Для цього вони у блоці синхронізації викликають метод `wait()` та очікують, коли потік класу `Notifier` (лістинг 2.10) завершить обробку об'єкту `msg` і викличе метод `notifyAll()`. Одержавши сповіщення про вивільнення об'єкту `msg`, потоки класу `Waiter`, що очікують, зможуть запропонувати свою роботу з об'єктом `msg`. Головний клас додатку `WaitNotifyDemo` наведено у лістингу 2.11.

Лістинг 2.8 – Клас `Message`

```

1  public class Message {
2      private String msg;
3
4      public void setMessage(String msg){
5          this.msg = msg;
6      }
7
8      public String getMessage(){
9          return msg;
10     }
11 }
```

Лістинг 2.9 – Клас `Waiter`

```

1  public class Waiter implements Runnable{
2      private Message msg;
3
4      public Waiter(Message msg){
5          this.msg = msg;
6      }
7 }
```

```

8      public void run(){
9          String name = Thread.currentThread().getName();
10         synchronized(msg){
11             try{
12                 System.out.println(name +
13                     " waits for notification" +
14                     System.currentTimeMillis());
15                 msg.wait();
16             } catch(InterruptedException e){
17                 e.printStackTrace();
18             }
19             System.out.println(name + " notified " +
20                 System.currentTimeMillis());
21             System.out.println(name +
22                 " received message: " + msg.getMessage());
23         }
24     }
25 }

```

Лістинг 2.10 – Клас Notifier

```

1  public class Notifier implements Runnable{
2      private Message msg;
3
4      public Notifier(Message msg){
5          this.msg = msg;
6      }
7
8      public void run() {
9          String name = Thread.currentThread().getName();
10         System.out.println(name + " started ");
11         try {
12             Thread.sleep(1500);
13             synchronized(msg) {
14                 msg.setMessage(name + " did his work ");
15                 //msg.notify();
16                 msg.notifyAll();
17             }
18         } catch ( InterruptedException e) {
19             e.printStackTrace();
20         }
21     }

```

22 }

Лістинг 2.11 – Клас WaitNotifyDemo

```

1  public class WaitNotifyDemo {
2
3      public static void main(String[] args) {
4          Message msg = new Message();
5          Waiter waiter1 = new Waiter(msg);
6          new Thread(waiter1, "waiter1").start();
7
8          Waiter waiter2 = new Waiter(msg);
9          new Thread(waiter2, "waiter2").start();
10
11         Notifier notifier = new Notifier(msg);
12         new Thread(notifier, "notifier").start();
13         System.out.println("all threads started ");
14     }
15 }
```

У результаті роботи програми на консолі можна спостерігати сценарій роботи синхронізованих потоків, наведений у лістингу 2.12.

Лістинг 2.12 – Можливий результат роботи програми

```

waiter1 waits for notification 1505064655417
waiter2 waits for notification 1505064655426
all threads started
notifier started
waiter2 notified 1505064656948
waiter2 received message: notifier did his work
waiter1 notified 1505064656948
waiter1 received message: notifier did his work
```

2.2 Використання потоків Java у паралельних і векторних обчисленнях

Потоки Java можна використовувати для організації ефективного вирішення багатьох науково-технічних задач. Ефективність полягає у прискоренні одержання рішення і досягається шляхом розпаралелювання та/чи векторизації алгоритмів вирішення.

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.

Програма 2.2.1. Паралельні обчислення

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

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

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 (до 100) членів ряду. Окрім того, у програмі використовується клас `java.math.BigDecimal`, який дозволяє працювати з дійсними числами будь-якої точності (розрядності).

Програма складається з трьох класів:

- `ComputeArcSin` (лістинг 2.13), у якому метод `bigArcSin()` обчислює значення функції $\arcsin(x)$;
- `ParallelThread` (лістинг 2.14), який спадкує клас `Thread`, а його метод `run()` викликає метод `bigArcSin()`;
- головний клас `DemoParallelThreads`, що містить функцію `main()` та керує потоками (лістинг 2.15).

Розпаралелювання задачі полягає в тому, що кожен член ряду розкладу (2.1) може обчислюватися незалежно від інших членів ряду. Отже, кожен член ряду можна обчислити в окремому потоці. У даній реалізації задачі число потоків визначається числом ядер процесора, а розподіл обчислень організовано наступним чином: якщо у комп'ютері використовується 4х-ядерний процесор, то потік на першому ядрі буде обчислювати члени ряду з номерами 0, 4, 8 і т.д.,

потік на другому ядрі – з номерами 1, 5, 9,..., потік на третьому – з номерами 2, 6, 10,..., потік на четвертому – з номерами 3, 7, 11,...

Лістинг 2.13 – Клас обчислення $\arcsin(x)$

```

1  import java.math.BigDecimal;
2
3  public class ComputeArcSin {
4      private BigDecimal arcsin;
5
6      private BigDecimal bigFact(BigDecimal k){
7          int a = k.compareTo(BigDecimal.ZERO);
8          return (a == 0) ? BigDecimal.ONE :
9              k.multiply(bigFact(k.subtract(BigDecimal.ONE)));
10     }
11
12     public void bigArcSin(int n, double x,
13         int threadNum, int numThreads){
14         BigDecimal fact1, fact2, numer, denom;
15         BigDecimal a = new BigDecimal(x);
16         BigDecimal arcsin = BigDecimal.ZERO;
17
18         for (int i = threadNum; i < n; i+= numThreads){
19             BigDecimal four = new BigDecimal(4);
20             fact1 = bigFact(new BigDecimal(i*2));
21             fact2 = bigFact(new BigDecimal(i));
22             fact2 = fact2.multiply(fact2);
23             numer = a.pow(i*2+1);
24             numer = numer.multiply(fact1);
25             denom = four.pow(i);
26             denom = denom.multiply(fact2);
27             denom = denom.multiply(new BigDecimal(2*i+1));
28             numer = numer.divide(denom,
29                 BigDecimal.ROUND_HALF_DOWN);
30             arcsin = arcsin.add(numer);
31         }
32         this.arcsin = arcsin;
33     }
34
35     public BigDecimal getResult(){
36         return arcsin;
37     }
38 }

```

```

34     }
35 }

```

Лістинг 2.14 – Клас потоку

```

1  public class ParallelThread extends Thread {
2      private ComputeArcSin compute;
3      private int n;
4      private double x;
5      private int threadNum;
6      private int numThreads;
7
8      public ParallelThread(ComputeArcSin compute, int n,
9                          double x, int threadNum, int numThreads) {
10         this.compute = compute;
11         this.n = n;
12         this.x = x;
13         this.threadNum = threadNum;
14         this.numThreads = numThreads;
15     }
16
17     public void run() {
18         compute.bigArcSin(n, x, threadNum, numThreads);
19     }

```

Лістинг 2.15 – Головний клас програми

```

1  import java.math.BigDecimal;
2  import java.util.Locale;
3
4  public class DemoParallelThreads {
5      public static final int TERMS_COUNT = 100;
6      public static final double X = 0.85;
7
8      public static void main(String[] args) {
9          int numThreads =
10             Runtime.getRuntime().availableProcessors();
11
12         ComputeArcSin[] compute =
13             new ComputeArcSin[numThreads];

```

```

12     ParallelThread[] pth =
13         new ParallelThread[numThreads];
14     for(int i = 0; i < numThreads; i++){
15         compute[i] = new ComputeArcSin();
16         pth[i] = new ParallelThread(compute[i],
17             TERMS_COUNT, X, i, numThreads );
18     }
19     long t1 = System.nanoTime();
20     for(int i = 0; i < numThreads; i++){
21         pth[i].start();
22     }
23     for(int i = 0; i < numThreads; i++){
24         try {
25             pth[i].join();
26         } catch (InterruptedException e) {
27             e.printStackTrace(System.err);
28         }
29     }
30     BigDecimal y = new BigDecimal (0.);
31     long t2 = System.nanoTime();
32     for(int i = 0; i < numThreads; i++){
33         y = y.add(compute[i].getResult());
34     }
35     BigDecimal err = new BigDecimal(Math.asin(X));
36     err = err.subtract(y).abs();
37     System.out.printf(Locale.ENGLISH, "x = %4.3f
38         arcsin = %20.19f err = %.3e\n", X, y, err);
39     System.out.printf(Locale.ENGLISH, "tp = %4.3f\n",
40         1.E -9*(double) (t2-t1));
41 }
42 }

```

Програма 2.2.2. Векторні обчислення

У даній програмі демонструється вирішення задачі обчислення функції $\arcsin(x_i)$ за формулою (2.1) для $x_i = 0.085, 0.170, \dots, 0.850$. Векторизація задачі полягає у тому, що для обчислення функції у кожній точці x_i використовується один потік. Таким чином, усі потоки виконують однакову роботу, але у різних точках x_i . Код

програми багато у чому аналогічний до коду програми 2.2.1 (лістинги 2.16–2.18).

Лістинг 2.16 – Клас апроксимації $\arcsin(x_i)$ для векторних обчислень

```

1  import java.math.BigDecimal;
2
3  public class ComputeArcSin {
4      private BigDecimal arcsin;
5
6      private BigDecimal bigFact(BigDecimal k){
7          int a = k.compareTo(BigDecimal.ZERO);
8          return (a == 0) ? BigDecimal.ONE:
9                  k.multiply(bigFact(k.subtract(BigDecimal.ONE)));
10     }
11
12     public void bigArcSin(int n, double x){
13         BigDecimal fact1, fact2, numer, denom;
14         BigDecimal a = new BigDecimal(x);
15         BigDecimal arcsin = BigDecimal.ZERO;
16
17         for (int i = 0; i < n; i++) {
18             BigDecimal four = new BigDecimal(4);
19             fact1 = bigFact(new BigDecimal(i*2));
20             fact2 = bigFact(new BigDecimal(i));
21             fact2 = fact2.multiply(fact2);
22             numer = a.pow(i*2+1);
23             numer = numer.multiply(fact1);
24             denom = four.pow(i);
25             denom = denom.multiply(fact2);
26             denom = denom.multiply(new BigDecimal(2*i+1));
27             numer = numer.divide(denom,
28                                 BigDecimal.ROUND_HALF_DOWN);
29             arcsin = arcsin.add(numer);
30         }
31         this.arcsin = arcsin;
32     }
33
34     public BigDecimal getResult(){
35         return arcsin;
36     }
37 }

```

```

34     }
35 }

```

Лістинг 2.17 – Клас потоку для векторних обчислень

```

1  public class VectorThread extends Thread {
2      private ComputeArcSin compute;
3      private int n;
4      private double x;
5
6      public VectorThread(ComputeArcSin compute, int n,
7                          double x){
8          this.compute = compute;
9          this.n = n;
10         this.x = x;
11     }
12
13     public void run(){
14         compute.bigArcSin(n, x);
15     }
16 }

```

Лістинг 2.18 – Головний клас для векторних обчислень

```

1  import java.math.BigDecimal;
2  import java.util.Locale;
3
4  public class DemoVectorThreads {
5      public static final int TERMS_COUNT = 100;
6      public static final int THREADS_COUNT = 10;
7      public static final double DX = 0.85/THREADS_COUNT;
8
9      public static void main(String[] args) {
10
11         double[] x = new double[THREADS_COUNT];
12         for(int i=0; i < THREADS_COUNT; i++){
13             x[i] = DX*(i+1);
14         }
15
16         ComputeArcSin[] compute = new
17             ComputeArcSin[THREADS_COUNT];

```

```

17     VectorThread[] vth =
18         new VectorThread[THREADS_COUNT];
19     for(int i = 0; i < THREADS_COUNT; i++){
20         compute[i] = new ComputeArcSin();
21         vth[i] = new
22             VectorThread(compute[i], TERMS_COUNT, x[i]);
23     }
24     long t1 = System.nanoTime();
25     for(int i = 0; i < THREADS_COUNT; i++){
26         vth[i].start();
27     }
28     for(int i = 0; i < THREADS_COUNT; i++){
29         try {
30             vth[i].join();
31         } catch (InterruptedException e) {
32             e.printStackTrace(System.err);
33         }
34     }
35     BigDecimal[] y = new BigDecimal[THREADS_COUNT];
36     BigDecimal[] err = new BigDecimal[THREADS_COUNT];
37     long t2 = System.nanoTime();
38     for(int i = 0; i < THREADS_COUNT; i++){
39         y[i] = compute[i].getResult();
40         err[i] = new BigDecimal(Math.asin(x[i]));
41         err[i] = err[i].subtract(y[i]).abs();
42         System.out.printf(Locale.ENGLISH, "x=%4.3f
43             Arcsin = %20.19f err = %.3e\n",
44                 x[i], y[i], err[i]);
45     }
46     System.out.printf(Locale.ENGLISH,
47         "tv = %4.3f\n", 1.E-9*(double)(t2-t1));
48 }

```

2.3 Завдання до лабораторної роботи

2.3.1 Завдання до програми 2.1.1 (лістинги 2.1–2.4).

2.3.1.1 Реалізувати програму, вивчити способи створення і запуску потоків у Java.

2.3.1.2 Доповнити та розробити нову програму, у якій виконати наступне:

- розробити два класи, перший з яких має спадкувати клас `Thread`, а другий – реалізовувати інтерфейс `Runnable`;
- класи повинні мати конструктори з параметрами;
- до конструкторів класів мають передаватися: ціле число, що ідентифікує номер створюваного екземпляру класу, та рядок – назва класу;
- у методах `run()` класів мають виводитися до друку номер екземпляру класу та назва класу;
- у головному методі у циклі створити масиви з 5 потоків кожного з розроблених класів, передавши до конструкторів кожного із екземплярів параметр циклу і назву класу;

– продемонструвати роботу програми та пояснити результат;

– лістинг розробленої програми занести до звіту.

2.3.2 Завдання до програми 2.1.2 (лістинги 2.5, 2.6).

2.3.2.1 Реалізувати програму.

2.3.2.2 Вивчити стани життєвого циклу потоків (рис. 2.1).

2.3.2.3 Пояснити кожне із повідомлень, що виводяться програмою на консоль.

2.3.2.4 Модифікувати програму наступним чином:

- розробити ще один (власний) клас, його назва повинна мати формат `ClassSurname`, де `Surname` – Ваше прізвище;
- у головному методі створити екземпляр класу `ClassSurname`, його назва повинна мати формат `objectName`, де `Name` – Ваше ім'я;
- встановити пріоритет об'єкту максимальним;
- побудувати сценарій роботи потоку таким чином, щоб він пройшов стани, аналогічні станам потоку `th1` класу `TControl`;
- розроблені фрагменти програми занести до звіту.

2.3.3 Завдання до програми 2.1.3 (лістинги 2.8–2.11).

2.3.3.1 Реалізувати програму.

2.3.3.2 Вивчити методи `wait()`, `notify()`, `notifyAll()`.

2.3.3.3 Замінити у програмі метод `notifyAll()` на метод `notify()`, запустити програму на виконання та пояснити результат її роботи.

2.3.4 Завдання до програми 2.2.1 (лістинги 2.13–2.15).

2.3.4.1 Реалізувати програму та довести правильність обчислення функції $\arcsin(x)$.

2.3.4.2 Допрацювати програму таким чином, щоб вона виконувала наступне:

- дозволяла задавати довільне число потоків;
- обчислювала прискорення.

2.3.4.3 Виконати достатню кількість прогонів допрацьованої програми для числа потоків, рівного числу ядер процесора, та оцінити її прискорення.

2.3.4.4 Виконати достатню кількість прогонів допрацьованої програми на 5 і 10 потоках, оцінити прискорення для цих випадків і прокоментувати одержувані результати.

2.3.5 Завдання до програми 2.2.2 (лістинги 2.16–2.18).

2.3.5.1 Реалізувати програму.

2.3.5.2 Допрацювати програму таким чином, щоб вона дозволяла обчислювати прискорення.

2.3.5.3 Оцінити прискорення програми для наступних випадків:

- кількість точок x_i та кількість потоків рівні числу ядер процесора;
- кількість точок x_i та кількість потоків рівні подвоєному числу ядер процесора.

2.3.6 Завдання до розділу 2.2.

2.3.6.1 Розробити програми для паралельних і векторних обчислень функції згідно варіанту:

- варіант 1:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2 \cdot n + 1)!} \cdot x^{2 \cdot n + 1}; \quad (2.2)$$

- варіант 2:

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

- варіант 3:

$$\arctan(x) = \sum_{n=0}^{\infty} \frac{(-1)^{n-1}}{(2 \cdot n + 1)!} \cdot x^{2 \cdot n + 1} . \quad (2.4)$$

Зауваження: варіант визначається як залишок від ділення номеру робочого місця на 3: 0 – 1-й варіант; 1 – 2-й; 2 – 3-й.

2.3.6.2 Виконати для цих програм завдання, аналогічні до завдань до програм 2.2.1 і 2.2.2.

2.3.6.3 Лістинги розроблених програм і одержані результати занести до звіту.

2.4 Зміст звіту

2.4.1 Титульний лист.

2.4.2 Мета роботи.

2.4.3 Фрагменти програмного коду, розроблені самостійно.

2.4.4 Одержані експериментальні оцінки прискорення (див. завдання до програм). Вміти їх інтерпретувати.

2.4.5 Виписати усі функції для організації багатопоточних обчислень на мові Java, використані у даній лабораторній роботі, навести їх короткі описи.

2.4.6 Відповіді на контрольні питання. Вміти їх пояснювати в усній формі.

2.5 Контрольні питання

2.5.1 Як у Java створюються потоки? Навести приклади способів створення потоків.

2.5.2 Перелічити і описати стани життєвого циклу потоку у Java.

2.5.3 Як завершити роботу потоку?

2.5.4 Як встановити та дізнатися пріоритет потоку?

2.5.5 Для чого використовується пріоритет потоку?

2.5.6 Який пріоритет потоку за замовчанням?

2.5.7 Навести приклад з використанням методу `sleep()`.

2.5.8 Як вивести на консоль інформацію про потік?

2.5.9 Для чого та як використовується оператор `synchronized`?

2.5.10 Описати призначення методів `wait()`, `notify()`, `notifyAll()`.

2.5.11 За допомогою якого методу можна визначити число доступних процесорів?

2.5.12 Перелічити вивчені методи класу `java.math.BigDecimal`.

2.5.13 Пояснити відмінність між паралельними і векторними обчисленнями.

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

2.5.15 Викласти формальне виведення закону Амдала.

ЛИТЕРАТУРА

1. Уильямс, Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ / Э. Уильямс. – ДМК Пресс, 2012.
2. Хорстман, К. С. Java. Библиотека профессионала, том 1. Основы. 9-е изд. / К. С. Хорстман, Г. Корнелл. – М : Вильямс, 2014.
3. Левин, М. П. Параллельное программирование с использованием OpenMP: учеб. пособие / М. П. Левин. – М : БИНОМ, 2012.
4. Антонов, А. С. Параллельное программирование с использованием технологии OpenMP: учеб. пособие / А. С. Антонов. – М : МГУ, 2009.
5. Гергель, В. П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем: учеб. пособие / В. П. Гергель. – Нижний Новгород : ННГУ им. Н.И. Лобачевского, 2010.
6. Седжвик, Р. Фундаментальные алгоритмы на C++ / Р. Седжвик. – СПб : ДиаСофтЮП, 2002.
7. Антонов, А. С. Параллельное программирование с использованием технологии MPI: учеб. пособие / А. С. Антонов. – М : МГУ, 2004.