Міністерство освіти і науки України
Центральноукраїнський національний технічний університет
Механіко-технологічний факультет
Кафедра кібербезпеки та програмного забезпечення

Комп’ютерні системи

Методичні вказівки до виконання лабораторних робіт для студентів денної форми навчання за спеціальністю 122 «Комп’ютерні науки», 123 «Комп’ютерні інженерія»

ЗАТВЕРДЖЕНО
на засіданні кафедри кібербезпеки та програмного забезпечення, протокол №10 від 25.03.2025 року.

Кропивницький
2025

УДК 004.9

Комп’ютерні системи: методичні вказівки до виконання лабораторних робіт для студентів за спеціальностями 122 «Комп’ютерні науки», 123 «Комп’ютерні інженерія»/ М-во освіти і науки України, Центральноукраїнський нац. техн. ун-т; [уклад. П.С. Усік, Н.Л. Козірова, Г.М. Дєєва, О.С Улічев] – Кропивницький: ЦНТУ, 2025. – 44с.

Укладачі: Усік П. С., доктор філософії, старший викладач;
Козірова Н.Л., викладач;
Дєєва Г.М., доктор філософії, викладач;
Улічев О.С. кандидат технічних наук, доцент

Рецензенти: Смірнов О. А., докт. техн. наук, професор;
Коваленко О. В., докт. техн. наук, професор.

ЗМІСТ

ВСТУП

Мета: «Комп’ютерні системи» є набуття систематизованих знань про структуру та принципи роботи комп'ютерних систем різного призначення. Вироблення навичок оцінки техніко-експлуатаційних можливостей засобів комп’ютерної техніки, ефективності різних режимів роботи комп’ютерів та комп’ютерних систем. Придбання практичних навичок вибору і використання комп’ютерних систем для обробки інформації (даних) різноманітного призначення.

Завдання:

– Вивчення теоретичних основ функціонування комп’ютерних систем.

– Вивчення архітектури та принципів роботи сучасних комп’ютерних систем і їх компонентів.

– Набуття загальних навичок аналізу та вибору комп’ютерних систем відповідно до вимог користувача.

У результаті вивчення навчальної дисципліни студент повинен:

– знати новітні технології в галузі інформаційний технологій. Знати і розуміти наукові положення, що лежать в основі функціонування комп’ютерних засобів, систем та мереж.

– вміти застосовувати знання для ідентифікації, формулювання і розв’язування технічних задач спеціальності, використовуючи методи, що є найбільш придатними для досягнення поставлених цілей. Вміти ідентифікувати, класифікувати та описувати роботу комп’ютерних систем та їх компонентів.

Структурно логічна схема підготовки бакалавра.

Враховуючи послідовність накопичення знань та інформації, дисципліна вивчається після викладання наступних дисциплін: «Архітектура комп’ютерів», «Комп’ютерні мережі».

Для опанування матеріалу дисципліни «Комп'ютерні системи» окрім лекційних та лабораторних занять, тобто аудиторного навантаження, значна увага приділяється самостійній роботі.

До основних видів самостійної роботи студента відносимо:

1. Вивчення лекційного матеріалу.

2. Робота з літературними джерелами.

3. Розв'язання практичних задач.

4. Підготовка до модульних, підсумкового контролю, екзамену (денна) та заліку (заочна).

5. Виконання контрольної роботи для заочної форми навчання.

В ході викладання дисципліни викладачем застосовуються види занять, які згідно з програмою навчальної дисципліни передбачають лекційні, та лабораторні заняття, а також виконання самостійної роботи.

Основна мета лекції – дати систематизовані основи знань з навчальної дисципліни, зосередити увагу студентів на найбільш складних та ключових питаннях.

Основна мета лабораторної роботи – закріплення й деталізація знань, а головне – формування навичок і вмінь.

Шкала оцінювання: національна та ЄКТС

Сума балів за всі види навчальної діяльності	Оцінка ЄКТС	Оцінка за національною шкалою
		для екзамену
90-100	A	відмінно
82-89	B	добре
74-81	C	
64-73	D	задовільно
60-63	E	
35-59	FX	незадовільно з можливістю повторного складання
1-34	F	незадовільно з обов'язковим повторним вивченням дисципліни

Вибравши предметну область, над якою ви будете працювати, ви повинні виконати завдання до лабораторних робіт, а також відповісти на питання в кінці кожної лабораторної роботи. Звіт повинен містити хід виконання завдань а також графічні матеріали, що підтверджують виконання цих завдань.

Лабораторна робота 1: Основи компіляції та оптимізації коду

Мета: Навчитися базовим прийомам компіляції та оптимізації програмного коду за допомогою GCC.

Теоретична частина

GCC (GNU Compiler Collection) — це набір компіляторів для різних мов програмування, таких як C, C++, Fortran та інших. Він широко використовується для компіляції програм на різних операційних системах, включаючи Linux та Windows. На Windows найчастіше встановлюють GCC через **MinGW** або **Cygwin**.

Етапи інсталяції GCC:

- На Windows: завантажте MinGW або Cygwin, інсталюйте їх і додайте шлях до компілятора в змінні середовища (PATH).
- На Linux: GCC зазвичай встановлений за замовчуванням. Якщо ні, його можна встановити за допомогою менеджера пакетів (наприклад, sudo apt install gcc для Debian-based систем).

Після інсталяції перевірка версії виконується командою:

gcc --version

Ця команда покаже версію компілятора та доступні компілятори, такі як gcc (для C), g++ (для C++), та gfortran (для Fortran).

Написання програми на C

C - це структурна мова програмування низького рівня, яка дозволяє ефективно працювати з ресурсами системи. Для реалізації базових математичних операцій можна використовувати такі оператори:

Додавання: +

Віднімання: -

*Множення: **

Ділення: /

Програма повинна читати вхідні дані з файлу, виконувати обчислення і записувати результат у файл. Це робиться за допомогою стандартних функцій для роботи з файлами:

fopen() — відкриття файлу.

fscanf() — читання з файлу.

fprintf() — запис у файл.

fclose() — закриття файлу.

Базова компіляція

Компіляція полягає в перетворенні вихідного коду на машинний код. Це здійснюється за допомогою GCC, використовуючи команду:

```
gcc -o my_program my_program.c
```

Ця команда створить виконуваний файл my_program. Під час компіляції також може бути створено об'єктний файл (.o), який містить проміжний код.

Рівні оптимізації в GCC

GCC надає кілька рівнів оптимізації:

- -O0: відсутність оптимізації (стандартний режим).
- -O1: базовий рівень оптимізації, що зменшує обсяг коду.
- -O2: більший рівень оптимізації з покращенням продуктивності.
- -O3: максимальна оптимізація продуктивності з деякими додатковими агресивними методами.

- -Os: оптимізація для зменшення розміру виконуваного файлу.
- -Ofast: агресивна оптимізація без дотримання стандартів точності обчислень.

Команда для компіляції з різними рівнями оптимізації:

```
gcc -O2 -o my_program my_program.c
```

Тестування програми та вимірювання часу виконання

Час виконання програми можна вимірювати за допомогою функцій:

- **clock()** — функція з бібліотеки time.h, яка вимірює час виконання програми в тиках процесора. Щоб отримати час у секундах, кількість тиків треба поділити на значення CLOCKS_PER_SEC.

Приклад вимірювання часу:

```
#include <time.h>
#include <stdio.h>

int main() {
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    // код програми
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Час виконання: %.2f секунд\n", cpu_time_used);
    return 0;
}
```

Аналіз результатів оптимізації

Оптимізація впливає на кілька аспектів роботи програми:

- **Продуктивність**: більш високі рівні оптимізації (-O2, -O3) можуть значно пришвидшити виконання програм.
- **Розмір виконуваного файлу**: параметр -Os зменшує розмір файлу.
- **Час виконання**: під час тестування програм з різними рівнями оптимізації можна побачити, як змінюється час виконання. Важливо вимірювати час кілька разів для отримання точних результатів

Завдання:

1. Інсталяція та налаштування GCC:

- Встановити GCC на власну систему (якщо ще не встановлено).
- Перевірити версію GCC та створити короткий звіт про доступні компілятори (gcc, g++, gfortran і т.д.).

1. Написання програми на C: Напишіть програму на мові C, яка реалізує базові математичні операції (додавання, віднімання, множення, ділення). Програма повинна читати дані з файлу і виводити результати обчислень у файл.

2. Базова компіляція: Скомпілюйте програму за допомогою GCC без використання жодних спеціальних параметрів. Дослідіть отримані файли (.o та виконуваний файл).

3. Виконати компіляцію програми з різними рівнями оптимізації (O0, O1, O2, O3, Os, Ofast). Використати команду gcc -O[level] -o sort_program sort_program.c.

4. Провести тестування програми з різними рівнями оптимізації, вимірюючи час виконання кожного алгоритму. Використати функції clock() або gettimeofday() для вимірювання часу виконання.

5. Проаналізувати отримані результати та зробити висновки щодо впливу оптимізації на продуктивність програми.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.
- Висновок.

Лабораторна робота 2: Профілювання та налагодження програм

Мета: Навчитися профілювати та налагоджувати програми на C/C++ з використанням GCC та інших інструментів.

Алгоритм швидкого сортування (Quick Sort) є одним із найбільш ефективних методів для сортування великих масивів даних. Він використовує підхід "розділяй і володарюй", де масив розділяється на дві частини, які рекурсивно сортуються, і далі об'єднуються.

Компілятор GCC підтримує різні прапори, які дозволяють налаштовувати компіляцію для певних цілей, наприклад, для налагодження або профілювання.

Компіляція програми з підтримкою налагодження

Для того, щоб додати інформацію для налагодження в програму, використовується прапор -g. Він дозволяє компілятору зберегти додаткові дані про вихідний код у виконуваному файлі, що допоможе під час налагодження та профілювання.

Команда для компіляції:

```
g++ -g -o quicksort quicksort.cpp
```

Профілювання з використанням gprof

Профілювання — це аналіз виконання програми для визначення, де вона витрачає найбільше часу або ресурсів. Щоб дозволити профілювання, програму потрібно компілювати з прапором -pg, який додає спеціальні інструкції для збору статистики під час виконання.

Компіляція для профілювання:

```
g++ -pg -o quicksort quicksort.cpp
```

Після запуску програми:

```
./quicksort
```

Створюється файл gmon.out, який містить профіль виконання. Для аналізу цього файлу використовується команда gprof:

```
gprof quicksort gmon.out > profile_report.txt
```

Результат профілювання буде збережений у файлі profile_report.txt, де можна побачити, які функції споживають найбільше часу.

GDB (GNU Debugger) — це інструмент для налагодження програм, написаних на мовах C і C++. Він дозволяє встановлювати точки зупину, переглядати значення змінних, стежити за виконанням програми та аналізувати стек викликів.

Для запуску програми в режимі налагодження можна використати команду:

```
gdb ./quicksort
```

Щоб встановити точку зупину в коді (наприклад, у функції partition) напишіть:

```
break partition
```

Для запуску програми після встановлення точок зупину:

```
run
```

Крокове виконання

Під час зупинки програми можна використовувати команди для покрокового виконання:

step — виконує один рядок коду та входить у функції.

next — виконує один рядок коду, не заходячи у функції.

Для перевірки значень змінних використовується команда print:

```
print variable_name
```

Для перегляду стека викликів:

```
backtrace
```

Після виявлення помилок або невідповідностей у логіці програми їх можна виправити та повторно запустити програму для перевірки.

Оптимізація програм на основі профілювання

Після проведення профілювання можна визначити найбільш "важкі" ділянки коду, які потребують оптимізації. Це дозволяє підвищити ефективність програми за рахунок оптимізації найбільш ресурсомістких операцій.

Для оптимізації продуктивності програми компілятор GCC підтримує кілька рівнів оптимізації про них ви пам'ятаєте з минулої лабораторної:

-O1 — базовий рівень оптимізації.

-O2 — більш агресивна оптимізація, що покращує продуктивність без значного збільшення часу компіляції.

-O3 — максимальна оптимізація, що включає всі можливі поліпшення.

-Ofast — агресивна оптимізація, яка може порушити стандарти точності обчислень.

Команда для компіляції з оптимізацією:

```
g++ -O2 -o quicksort_opt quicksort.cpp
```

Повторне профілювання

Після внесення змін у код і компіляції з оптимізацією необхідно повторити процес профілювання, щоб порівняти результати до і після оптимізації:

```
./quicksort_opt
gprof quicksort_opt gmon.out > optimized_profile_report.txt
```

Це дозволяє оцінити, наскільки продуктивність програми покращилася після оптимізації. Важливо пам'ятати, що оптимізація повинна базуватися на результатах профілювання, щоб уникнути "передчасної оптимізації", яка може не дати значних результатів.

Завдання:

1. Написання більш складної програми: Напишіть програму на C++, яка реалізує алгоритм швидкого сортування (Quick Sort) для великих масивів даних.

2. Компіляція та профілювання: Скомпілюйте програму з використанням GCC і опції -g для налагодження. Запустіть програму і за допомогою gprof проведіть профілювання виконання. Визначте найбільш "важкі" ділянки коду.

3. Налагодження: Використайте gdb для налагодження програми. Наприклад, встановіть точки зупину, дослідіть стек викликів та змінні в процесі виконання програми. Виправте будь-які знайдені помилки.

4. Оптимізація на основі профілювання: На основі отриманих даних профілювання, оптимізуйте код для підвищення його продуктивності. Заново профілюйте програму, щоб оцінити покращення.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.
- Висновок.

Лабораторна робота 3: Створення та використання статичних і динамічних бібліотек

Мета: Навчитися створювати та використовувати статичні та динамічні бібліотеки на C/C++.

Бібліотека — це набір функцій або об'єктних файлів, які можуть бути використані в різних програмах. Вони дозволяють повторно використовувати код без необхідності його переписувати або включати у кожен проєкт окремо. Бібліотеки полегшують процес розробки, спрощують підтримку та сприяють більш структурованому підходу до програмування.

Існують два основних типи бібліотек:

Статичні бібліотеки — це бібліотеки, які підключаються до програми під час компіляції. Після цього бібліотека стає частиною виконуваного файлу програми.

Динамічні бібліотеки — ці бібліотеки підключаються під час виконання програми, що зменшує розмір виконуваного файлу, але вимагає наявності бібліотеки на системі під час запуску.

Статична бібліотека (зазвичай має розширення .a в Linux або .lib в Windows) — це архів об'єктних файлів, які підключаються до виконуваної програми під час компіляції. Усі функції зі статичної бібліотеки копіюються у виконуваний файл, тому для його роботи не потрібна наявність бібліотеки на комп'ютері користувача.

Для створення статичної бібліотеки на C слід:

- Написати кілька функцій у файлах .c.
- Скомпілювати ці файли у об'єктні файли за допомогою GCC.
- Створити архів цих об'єктних файлів за допомогою утиліти ar.

Приклад команд:

Компіляція вихідних файлів у об'єктні файли:

```
gcc -c add.c sub.c mul.c div.c
```

Це створить файли add.o, sub.o, mul.o, div.o.

Створення статичної бібліотеки:

```
ar rcs libmath.a add.o sub.o mul.o div.o
```

Ця команда створить архів libmath.a, що містить об'єктні файли.

Використання статичної бібліотеки: Щоб використовувати статичну бібліотеку у програмі, потрібно скомпілювати програму з посиланням на бібліотеку:

```
gcc -o myprogram myprogram.c -L. -lmath
```

Тут -L. вказує компілятору шукати бібліотеки в поточній директорії, а -lmath підключає бібліотеку libmath.a.

Динамічна бібліотека (розширення .so на Linux або .dll на Windows) підключається до програми під час її виконання. На відміну від статичних бібліотек, код із динамічної бібліотеки не копіюється у виконуваний файл, що зменшує його розмір. Однак під час запуску програми операційна система повинна мати доступ до цієї бібліотеки.

Для створення динамічної бібліотеки використовуються схожі принципи, але під час компіляції додається прапор -fPIC для створення позиційно-незалежного коду та -shared для створення бібліотеки.

Приклад команд:

Компіляція з позиційно-незалежним кодом (Position Independent Code, PIC):

```
gcc -c -fPIC add.c sub.c mul.c div.c
```

Це створить об'єктні файли для динамічної бібліотеки.

```
gcc -shared -o libmath.so add.o sub.o mul.o div.o
```

Ця команда створить динамічну бібліотеку libmath.so.

Використання динамічної бібліотеки:

Щоб використовувати динамічну бібліотеку у програмі:

```
gcc -o myprogram myprogram.c -L. -lmath
```

Програма під час виконання звернеться до libmath.so.

Налаштування динамічного завантаження: Щоб операційна система могла знайти динамічну бібліотеку під час виконання, може знадобитися налаштування змінної середовища LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=.
```

Це додасть поточну директорію до списку шляхів, де операційна система шукає динамічні бібліотеки.

Для порівняння бібліотек можна використовувати утиліту ldd, яка показує залежності програми від динамічних бібліотек:

```
ldd myprogram
```

Автоматизація компіляції за допомогою Makefile

Makefile — це текстовий файл, що містить правила автоматизації компіляції та створення проекту. Використання Makefile значно спрощує процес управління великими проектами, де є багато файлів і бібліотек. У ньому можна визначити залежності між файлами та автоматизувати процес компіляції бібліотек і програм.

Приклад базового Makefile:

CC = gcc

CFLAGS = -Wall

```
# Створення статичної бібліотеки
libmath.a: add.o sub.o mul.o div.o
     ar rcs libmath.a add.o sub.o mul.o div.o

# Створення об'єктних файлів
add.o: add.c
     $(CC) $(CFLAGS) -c add.c
sub.o: sub.c
     $(CC) $(CFLAGS) -c sub.c
mul.o: mul.c
     $(CC) $(CFLAGS) -c mul.c
div.o: div.c
     $(CC) $(CFLAGS) -c div.c

# Створення виконуваного файлу з використанням бібліотеки
myprogram: myprogram.o libmath.a
     $(CC) -o myprogram myprogram.o -L. -lmath
```

```
clean:
	rm *.o *.a myprogram
```

Основні команди Makefile:

make — запускає процес компіляції згідно з правилами, визначеними у Makefile.

make clean — видаляє всі проміжні файли (об'єктні файли та бібліотеки), щоб підготувати проект до нової компіляції.

Завдання:

5. Створення статичної бібліотеки: Напишіть декілька функцій на C, які реалізують основні математичні операції (додавання, віднімання, множення, ділення). Створіть статичну бібліотеку (libmath.a) з цими функціями. Напишіть програму, яка використовує цю бібліотеку.

6. Створення динамічної бібліотеки: Створіть динамічну бібліотеку (libmath.so) з тими ж функціями. Напишіть програму, яка використовує цю динамічну бібліотеку. Перевірте правильність завантаження бібліотеки під час виконання програми.

7. Використання бібліотек: Порівняйте використання статичних та динамічних бібліотек з точки зору розміру виконуваних файлів та швидкодії. Дослідіть, як змінюються залежності виконуваних файлів від бібліотек за допомогою інструмента ldd.

8. Автоматизація компіляції: Створіть Makefile для автоматизації процесу компіляції, створення бібліотек та виконуваних файлів. Оцініть переваги використання Makefile для великих проектів.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.
- Висновок.

Лабораторна робота 4: Вступ до паралельних обчислень з OpenMP

Мета: Ознайомитися з основами паралельних обчислень за допомогою бібліотеки OpenMP.

OpenMP (Open Multi-Processing) — це набір директив, бібліотек і змінних середовища, що дозволяють реалізовувати паралельне програмування в мовах C, C++ і Fortran. OpenMP забезпечує можливість багатопотокового виконання на багатоядерних процесорах за допомогою простих директив, які вставляються безпосередньо у вихідний код програми.

Паралелізація — це процес розподілу обчислювального завдання на кілька частин, які можуть виконуватись одночасно (паралельно) на різних процесорах, ядрах або потоках.

Паралелізація за допомогою OpenMP дозволяє:

- Підвищити продуктивність програм шляхом розподілу обчислень між кількома потоками.

- Мінімізувати зміни в коді: за допомогою кількох директив можна додати паралельність у вже існуючий код.

- Контролювати кількість потоків і розподіл роботи між ними.

OpenMP надає кілька основних директив для керування паралельним виконанням, наприклад:

- #pragma omp parallel: створює групу потоків для паралельного виконання коду.

- #pragma omp for: використовується для паралелізації циклів.

- #pragma omp critical: забезпечує виконання блоку коду тільки одним потоком у певний момент часу.

- #pragma omp barrier: синхронізує потоки, змушуючи їх чекати один на одного.

Послідовна програма

У послідовній версії програма виконується на одному потоці, де цикл ітерує по кожному елементу масиву і додає його до змінної суми.

Паралельна програма з OpenMP

За допомогою OpenMP можна розподілити виконання циклу між кількома потоками. Кожен потік обчислює часткову суму, яка потім об'єднується для отримання кінцевого результату.

Компіляція програми з OpenMP: Щоб використовувати OpenMP, потрібно додати прапор *-fopenmp* під час компіляції:

```
gcc -fopenmp -o sum_parallel sum_parallel.c
```

Використання директив OpenMP

Однією з найпоширеніших директив є *#pragma omp parallel*, яка дозволяє створювати групу потоків для виконання певного коду. Для розподілу обчислень у циклі використовується директива *#pragma omp for*.

Директива *#pragma omp parallel for* - поєднує дві директиви - створення паралельного блоку і паралелізацію циклу. Ця директива автоматично розподіляє ітерації циклу між потоками.

Приклад додавання директив:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    sum += array[i];
}
```

OpenMP розподіляє ітерації циклу між потоками, що прискорює обчислення, особливо при великих масивах даних.

OpenMP дозволяє вказати кількість потоків, які будуть задіяні в обчисленнях, за допомогою функції omp_set_num_threads() або змінної середовища OMP_NUM_THREADS. Наприклад:

```
omp_set_num_threads(4);
```

Ця команда встановить 4 потоки для виконання паралельних операцій.

Приклад запуску з різною кількістю потоків:

```
export OMP_NUM_THREADS=4
./sum_parallel
```

Або встановлення кількості потоків у самому коді:

```
omp_set_num_threads(8);
```

Для порівняння продуктивності послідовної та паралельної версій програми необхідно виміряти час виконання кожної з них.

Вимірювання часу виконання:

У мові С для вимірювання часу використовується функція *clock()* з бібліотеки *time.h.* За допомогою цієї функції можна виміряти, скільки часу витратила програма на обчислення.

Графіки часу виконання:

Для аналізу результатів продуктивності можна створити графіки часу виконання для різних розмірів масиву та кількостей потоків. Час виконання вимірюється для кожного варіанту, і потім можна вивести графіки, які показують, як змінюється продуктивність залежно від кількості потоків або об'єму даних.

Приклад аналізу:

Послідовна версія може бути швидкою для малих масивів, але при збільшенні кількості елементів її продуктивність суттєво знижується.

Паралельна версія повинна показувати краще масштабування з ростом розміру масиву і кількості потоків, особливо на багатоядерних процесорах.

Побудова графіків часу виконання

Щоб візуалізувати результати вимірювання часу, можна використати інструменти для побудови графіків, наприклад, gnuplot або програми на Python з використанням бібліотек на кшталт matplotlib. Для цього можна зберігати час виконання для кожного тесту у файл, а потім на основі цих даних будувати графіки.

Приклад збереження результатів у файл:

```
echo "Threads: 4, Time: 0.01" >> results.txt
```

Після цього можна побудувати графік, який покаже, як змінюється час виконання програми з ростом кількості потоків або розміру масиву.

Завдання:

1. Систематизація базових понять: Прочитайте документацію OpenMP та розберіться з основними директивами.

2. Паралельне сумування: Напишіть програму на C, яка обчислює суму елементів масиву. Реалізуйте версію програми без паралелізації та з паралелізацією за допомогою OpenMP. Порівняйте час виконання.

3. Використання директив: Додайте директиви OpenMP для паралелізації циклів. Вивчіть, як змінюється продуктивність при різних кількостях потоків.

4. Аналіз результатів: Створіть графіки часу виконання для різних розмірів масиву та кількостей потоків. Проаналізуйте результати.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.
- Висновок.

Лабораторна робота 5: Паралельні алгоритми з MPI

Мета: Ознайомитися з основами паралельних обчислень за допомогою MPI.

Паралельне програмування — це підхід до написання програмного забезпечення, в якому завдання розбивається на кілька підзадач, що можуть виконуватись одночасно на різних процесорах, ядрах або комп'ютерах. Мета — скоротити час виконання завдяки одночасному обчисленню.

Чому воно потрібне

- Зростання обсягів даних — обробка великих масивів даних потребує більше часу при послідовному виконанні.
- Фізичні обмеження процесорів — зростання тактової частоти уповільнилось, тому збільшення кількості ядер стало основним шляхом до підвищення продуктивності.
- Ефективне використання ресурсів — розподілені обчислення дозволяють повністю завантажувати доступне обладнання.

Основні ідеї паралелізму

- Розбиття задачі (decomposition) — завдання ділиться на підзадачі.
- Розподіл задач (assignment) — кожна підзадача призначається окремому виконавцю (потоку або процесу).
- Виконання (execution) — підзадачі виконуються паралельно.
- Синхронізація (synchronization) — узгодження результатів виконання підзадач.
- Комунікація (communication) — обмін даними між потоками або процесами.

Типи паралелізму

- Data Parallelism (паралелізм даних): одна і та ж операція виконується над різними частинами даних.

Наприклад: обчислення середнього значення великого масиву.

- Task Parallelism (паралелізм задач): різні завдання виконуються паралельно.

Наприклад: один процес обробляє вхідні дані, інший виконує обчислення, ще інший зберігає результати.

Рівні паралельного програмування

- Рівень інструкцій (Instruction Level) — паралелізм реалізується на апаратному рівні CPU.
- Рівень потоків (Thread Level) — реалізується багатопоточність, як-от в OpenMP.
- Рівень процесів (Process Level) — використовується окремі процеси, часто з розподіленою пам’яттю, як у MPI.
- Кластерний рівень — паралельна обробка на багатьох комп’ютерах через мережу.

Види пам’яті

- Shared Memory (спільна пам'ять) — всі потоки мають доступ до однієї пам’яті (наприклад, OpenMP).
- Distributed Memory (розподілена пам'ять) — кожен процес має власну пам’ять, дані передаються через повідомлення (MPI).

Виклики паралельного програмування

- Складність розробки (синхронізація, змагання за ресурси).
- Балансування навантаження між процесами.
- Надмірні витрати на комунікацію між процесами.
- Відсутність детермінізму — результат може залежати від порядку виконання процесів.

MPI (Message Passing Interface) — це стандарт програмного інтерфейсу, який дозволяє процесам обмінюватися повідомленнями під час паралельного виконання. Він особливо корисний у середовищах з розподіленою пам’яттю, де кожен процес має власну пам’ять, а обмін даними відбувається через передачу повідомлень.

MPI дозволяє:

- передавати дані між процесами;
- синхронізувати виконання;
- розподіляти обчислення між вузлами кластера або ядрами багатоядерного процесора.

Основні характеристики MPI

- **Стандартизований** — є відкритим стандартом (MPI-1, MPI-2, MPI-3).
- **Портативний** — підтримується на більшості платформ (Windows, Linux, macOS).
- **Масштабований** — підходить як для невеликих програм, так і для суперкомп'ютерів.

Ініціалізація та завершення MPI-програми

Щоб почати роботу з MPI, потрібно викликати функції ініціалізації та завершення:

```
MPI_Init(&argc, &argv);         // Ініціалізація MPI
MPI_Finalize();                 // Завершення MPI
```

Ідентифікація процесів

Усі процеси, що беруть участь у програмі, об'єднані в комунікатор (найчастіше MPI_COMM_WORLD). Кожен процес має унікальний ідентифікатор (ранг):

```
int size, rank;
MPI_Comm_size(MPI_COMM_WORLD, &size); // Кількість процесів
MPI_Comm_rank(MPI_COMM_WORLD,  &rank);  //  Ранг  (ідентифікатор)
поточного процесу
```

Точка-точка (point-to-point) комунікація

MPI_Send — надсилання повідомлення

```
MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm);
```

MPI_Recv — прийом повідомлення

```
MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status* status);
```

Пояснення параметрів:

- buf — буфер для даних;
- count — кількість елементів;
- datatype — тип даних (наприклад, MPI_INT, MPI_FLOAT);
- dest / source — ранг процесу-приймача/відправника;
- tag — мітка повідомлення;
- comm — комунікатор (найчастіше MPI_COMM_WORLD);
- status — структура для зберігання інформації про прийом (може бути MPI_STATUS_IGNORE).

Колективні операції (коротко)

Окрім обміну між окремими процесами, MPI підтримує колективні операції, які залучають усі процеси комунікатора:

- **MPI_Bcast** — розсилання повідомлення від одного процесу всім іншим.
- **MPI_Scatter / MPI_Gather** — розподіл / збирання частин масиву.
- **MPI_Reduce** — зведення (наприклад, сума) даних від усіх процесів до одного.
- **MPI_Allreduce** — зведення з розсиланням результату всім.

Синхронізація процесів

- MPI_Barrier(MPI_COMM_WORLD); — змушує всі процеси дочекатися одне одного перед виконанням наступної інструкції.

Компіляція та запуск

Для компіляції програм з MPI зазвичай використовують спеціальний компілятор:

```
mpicc prog.c -o prog
mpirun -np 4 ./prog      # запуск з 4 процесами
```

Приклад найпростішої програми MP

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello from process %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Розподіл даних — це процес поділу великого обсягу інформації (наприклад, масиву або матриці) між кількома паралельними процесами з метою одночасного оброблення різних частин. Це дозволяє ефективніше використовувати ресурси і скоротити загальний час виконання обчислень.

Методи розподілу даних

а) Ручне розбиття і передача

- Головний процес сам ділить дані на частини.
- Передає кожну частину іншим процесам через MPI_Send.
- Отримує результати через MPI_Recv.

б) Колективні операції MPI (автоматизоване)

- **MPI_Scatter** — автоматично розподіляє частини масиву між процесами.

```
MPI_Scatter(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype,
            root, comm);
```

- **MPI_Gather** — збирає частини результату назад до головного процесу.

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype,
           root, comm);
```

Приклад розподілу з MPI_Scatter та MPI_Gather

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int N = 8; // розмір масиву (припустимо, ділиться нацело)
    int data[N], local[2]; // масив для розподілу

    if (rank == 0) {
        for (int i = 0; i < N; i++) data[i] = i + 1;
    }

    MPI_Scatter(data, 2, MPI_INT, local, 2, MPI_INT, 0,
MPI_COMM_WORLD);

    // Кожен процес виводить свої дані
    printf("Process %d received: %d %d\n", rank, local[0],
local[1]);

    MPI_Finalize();
    return 0;
}
```

Поради для ефективного розподілу

- Намагайтеся **рівномірно** ділити дані між процесами.

- Якщо кількість елементів не ділиться нацело, обробляйте залишок окремо (наприклад, головний процес).
- Уникайте зайвих передач даних — вони можуть стати вузьким місцем продуктивності.
- Використовуйте буфери мінімального розміру, які потрібні кожному процесу.

Типові сценарії

Завдання	**Метод розподілу**
Обчислення середнього, суми	MPI_Scatter, MPI_Gather або MPI_Reduce
Пошук максимуму або мінімуму	MPI_Reduce
Паралельна обробка блоків зображення	Ручне розбиття з MPI_Send
Розподіл елементів матриці	1D або 2D блокове розбиття

Колективні операції в MPI — це функції, які передбачають участь усіх процесів у вказаному комунікаторі (зазвичай MPI_COMM_WORLD). Вони виконують одночасні дії з передачею або обробкою даних, забезпечуючи ефективну взаємодію між усіма процесами.

Такі операції автоматизують багато поширених шаблонів комунікації, зменшуючи кількість коду та потенційних помилок, і зазвичай реалізовані більш ефективно, ніж набір MPI_Send / MPI_Recv.

Основні типи колективних операцій MPI

а) Розсилання (broadcast): MPI_Bcast

Розсилає одне повідомлення від одного (кореневого) процесу всім іншим у комунікаторі.

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

Застосування: наприклад, передати однаковий параметр усім процесам.

б) Збирання даних (gather): MPI_Gather

Збирає дані з усіх процесів у один (кореневий) процес.

```
MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype,
           void *recvbuf, int recvcount, MPI_Datatype recvtype,
           int root, MPI_Comm comm);
```

Застосування: отримати частини масиву, обчислені різними процесами.

в) Розподіл даних (scatter): MPI_Scatter

Розподіляє масив із одного процесу на частини, які отримують усі процеси.

```
MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype,
            void *recvbuf, int recvcount, MPI_Datatype recvtype,
            int root, MPI_Comm comm);
```

Застосування: роздати шматки великого масиву на обробку.

г) Зведення (reduce): MPI_Reduce

Виконує операцію (наприклад, суму, добуток, максимум, мінімум) над даними з усіх процесів і надсилає результат одному процесу.

```
MPI_Reduce(void *sendbuf, void *recvbuf, int count,
           MPI_Datatype datatype, MPI_Op op,
           int root, MPI_Comm comm);
```

Приклад операцій: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD

ґ) Повне зведення (all-reduce): MPI_Allreduce

Те саме, що MPI_Reduce, але результат отримують усі процеси.

```
MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
              MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

Застосування: коли всі процеси потребують спільного результату (наприклад, глобальна сума).

д) Бар'єрна синхронізація: MPI_Barrier

Зупиняє виконання всіх процесів, доки кожен не дійде до виклику цієї функції. Використовується для синхронізації.

```
MPI_Barrier(MPI_Comm comm);
```

Приклад: Використання MPI_Reduce **для обчислення середнього**

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int local_val = rank + 1; // кожен процес має своє число

    int sum;
    MPI_Reduce(&local_val,   &sum,   1,   MPI_INT,   MPI_SUM,   0,
MPI_COMM_WORLD);

    if (rank == 0) {
        float avg = (float)sum / size;
        printf("Сума: %d, Середнє: %.2f\n", sum, avg);
    }

    MPI_Finalize();
    return 0;
}
```

***Пер*еваги використання колективних операцій**

- Простота: менше коду, менше помилок.
- Оптимізація: реалізації MPI мають високоефективні алгоритми.
- Масштабованість: краще працюють при збільшенні кількості процесів.
- Безпечність: обробка синхронізації та передавання даних вбудована.

Аналіз продуктивності — це процес вимірювання, порівняння та оцінки ефективності роботи паралельного алгоритму. Метою є визначення, наскільки

добре алгоритм масштабується, як використовуються ресурси, де виникають затримки, і які підходи забезпечують найкращу швидкодію.

Основні метрики продуктивності

а) Час виконання (Execution Time)

Загальний час, який витрачається програмою на виконання. Основна метрика, яку найчастіше вимірюють у секундах або мілісекундах.

б) Прискорення (Speedup)

Відношення часу виконання послідовної програми до часу виконання паралельної при певній кількості процесів:

Speedup=ТпаралТпосл

Ідеальне прискорення дорівнює кількості процесів, але на практиці досягається рідко через витрати на комунікацію.

в) Ефективність (Efficiency)

Відношення прискорення до кількості процесів:

Efficiency=PSpeedup

де P — кількість процесів. Вимірюється у відсотках. Показує, наскільки ефективно використовуються обчислювальні ресурси.

г) Масштабованість (Scalability)

Здатність алгоритму ефективно працювати при збільшенні кількості процесів або обсягу даних. Розрізняють сильну масштабованість (фіксований обсяг даних, більше процесів) та слабку масштабованість (збільшення обсягу даних пропорційно кількості процесів).

Фактори, що впливають на продуктивність

- Витрати на комунікацію між процесами (наприклад, у MPI_Send, MPI_Recv).
- Навантаження: рівномірний чи нерівномірний розподіл даних.
- Синхронізація: затримки через очікування інших процесів (наприклад, у MPI_Barrier).

- Інтенсивність обчислень у порівнянні з обсягом переданих даних.
- Тип колективної операції: деякі, як MPI_Reduce, можуть бути оптимізовані краще, ніж ручна передача.

Приклад аналізу продуктивності

Завдання: обчислення середнього значення великого масиву.

- Варіант 1: з використанням MPI_Send та MPI_Recv для збору результатів.
- Варіант 2: з використанням MPI_Reduce.

План аналізу:

1. Вимірюємо час виконання обох варіантів при 2, 4, 8, 16 процесах.
2. Обчислюємо прискорення та ефективність для кожного варіанту.
3. Створюємо графіки залежності часу від кількості процесів.
4. Робимо висновки: де менші затримки, краща масштабованість і чому.

Як вимірювати час

У MPI використовується функція:

```
double MPI_Wtime(void);
```

Ця функція повертає час у секундах від моменту ініціалізації MPI. Її можна використати так:

```
double start = MPI_Wtime();
// код для аналізу
double end = MPI_Wtime();
if (rank == 0) {
    printf("Час виконання: %f секунд\n", end - start);
}
```

Приклад графіка

Кількість процесів	Час (MPI_Send/Recv)	Час (MPI_Reduce)	Speedup (Reduce)
1	2.4 с	2.4 с	1.00
2	1.5 с	1.3 с	1.85

Кількість процесів	Час (MPI_Send/Recv)	Час (MPI_Reduce)	Speedup (Reduce)
4	1.1 с	0.8 с	3.00
8	0.9 с	0.5 с	4.80

Висновок: MPI_Reduce дозволяє зменшити комунікаційні витрати та досягти кращого прискорення.

Завдання:

1. Вступ до MPI: Ознайомтесь із основами MPI. Напишіть просту програму, яка використовує MPI_Send та MPI_Recv для обміну даними між двома процесами.

2. Паралельне обчислення середнього: Реалізуйте програму, яка обчислює середнє значення елементів великого масиву, розподіляючи частини масиву між процесами. Кожен процес повинен обчислити свою частину та надіслати результати назад до головного процесу.

3. Оптимізація комунікації: Досліджуйте, як змінюється час виконання при різних стратегіях обміну даними (використання MPI_Reduce для обчислення загальної суми).

4. Аналіз продуктивності: Зробіть порівняння продуктивності між версією з обміном даними та версією з використанням MPI_Reduce. Створіть графіки, що демонструють зміни часу виконання залежно від кількості процесів.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.

- Висновок.

Лабораторна робота 6: Використання паралельних бібліотек для обробки даних

Мета: Ознайомитися з основами паралельних бібліотек для обробки даних.

Потік - це окремий шлях виконання в межах однієї програми. На відміну від процесів, потоки одного процесу розділяють одну і ту ж пам’ять, що дає змогу ефективно обмінюватися даними між ними, але також вимагає контролю синхронізації.

Потоки в C++ (C++ Thread Library)

У C++11 була введена стандартна бібліотека потоків (<thread>), яка дозволяє створювати багатопоточні програми на високому рівні абстракції.

Створення потоку

```
#include <thread>
void task() {
    // код потоку
}
int main() {
    std::thread t(task); // створення потоку
    t.join();            // очікування завершення потоку
    return 0;
}
```

Передача параметрів у потік

```
void task(int x) {
    // ...
}
std::thread t(task, 5);
```

Очікування завершення потоку

- join() — очікує завершення потоку.
- detach() — від'єднує потік для незалежного виконання.

Добуток матриць

Множення матриць — це обчислювально складна операція з часовою складністю O(n3). Розпаралелювання дозволяє прискорити виконання, розподіливши обчислення елементів результатної матриці між потоками.

Послідовний алгоритм:

*C[i][j] = sum(A[i][k] * B[k][j])*

Паралельний підхід:

- Розподіл обчислення рядків або блоків між потоками.
- Кожен потік обчислює свою частину результатної матриці.

Синхронізація потоків

У багатопоточних програмах може виникнути **гонка даних**, якщо кілька потоків одночасно змінюють одні й ті самі змінні.

М'ютекси (std::mutex)

```
#include <mutex>
std::mutex m;
void safe_function() {
    m.lock();
    // критична секція
    m.unlock();
}
```

Або за допомогою *std::lock_guard:*

```
void safe_function() {
    std::lock_guard<std::mutex> guard(m);
    // автоматичне розблокування при виході з функції
}
```

Синхронізація потрібна лише тоді, коли потоки спільно змінюють загальні змінні. При обчисленні добутку матриць, якщо кожен потік працює над унікальними елементами результату, м’ютекси можуть не знадобитися.

Продуктивність: Порівняння одно- та багатопотокового підходу

Метричні показники:

- Час виконання — скільки секунд триває обчислення.
- Speedup = ТбагатопотоковийТоднопотоковий
- Ефективність = потоківКількість потоківSpeedup

Залежність продуктивності:

- Від розміру матриці.
- Від кількості потоків.
- Від архітектури процесора (кількість фізичних ядер).
- Від витрат на створення та керування потоками.

Якщо потоків більше, ніж ядер, продуктивність може навіть знизитись через перевантаження.

Побудова графіків продуктивності

Графіки дозволяють візуалізувати:

- **Час виконання** залежно від кількості потоків.
- **Використання пам'яті**.
- **Прискорення**.

Приклад побудови графіка:

Кількість потоків	**Час виконання (с)**	**Speedup**
1	4.5	1.0
2	2.4	1.875
4	1.3	3.46
8	1.0	4.5

Можна використати Python (наприклад, matplotlib) або Excel для побудови графіків.

Поради з реалізації

- Не створюйте більше потоків, ніж ядер.

- Краще обробляти **рядки** результатної матриці окремо — кожному потоку по блоку рядків.
- Мінімізуйте використання м’ютексів — уникайте спільного запису в одні й ті самі змінні.
- Використовуйте std::vector або динамічні масиви для гнучкого розміру матриць.

Завдання:

1. Розробка програми з використанням потоків: Напишіть програму на C++, яка обчислює добуток двох великих матриць за допомогою потоків. Реалізуйте метод, що розподіляє обчислення між кількома потоками.

2. Керування потоками: Використайте стандартні бібліотеки C++ для створення та управління потоками. Реалізуйте синхронізацію потоків за допомогою м'ютексів, щоб уникнути гонок даних.

3.Паралельне виконання: Порівняйте продуктивність програми з версією, що виконує добуток матриць в однопотоковому режимі. Проаналізуйте, як зміна кількості потоків впливає на час виконання.

4.Графічний аналіз: Створіть графіки для візуалізації часу виконання та використання пам'яті. Проаналізуйте, які фактори впливають на ефективність паралелізації.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.
- Висновок.

Лабораторна робота 7: Основи конвеєрної обробки з використанням потоків

Мета: Ознайомитися з принципами конвеєрної обробки, використовуючи потоки.

Конвеєрна обробка — це метод організації виконання задачі, який поділяє її на кілька етапів (стадій), кожен з яких обробляє дані послідовно, але незалежно. Результати одного етапу передаються наступному. Таким чином, кілька частин даних можуть одночасно оброблятися різними етапами.

У програмуванні це дозволяє підвищити ефективність за рахунок паралельного виконання різних частин завдання. Наприклад:

- Поки один потік читає наступний блок даних,
- другий обробляє попередній,
- третій — записує ще раніше оброблений.

Архітектура програмного конвеєра

Конвеєр зазвичай складається з:

1. **Джерела даних** (перший етап) — зчитує або генерує дані.
2. **Обробника даних** (другий етап) — виконує певні обчислення або перетворення.
3. **Споживача даних** (третій етап) — записує або виводить результати.

Кожен етап реалізується як окремий **потік**, а дані передаються між потоками через **черги** або **умовні змінні**.

Потоки в C++

У C++ створення та керування потоками здійснюється за допомогою стандартної бібліотеки <thread>. Кожен потік може виконувати певну функцію:

```
std::thread t1(функція1);
std::thread t2(функція2);
```

Для завершення програми потрібно викликати .join() для кожного потоку.

Передача даних між потоками

Щоб передати дані від одного етапу до іншого, використовуються:

Блокуючі черги (Blocking Queue)

Це структура даних, яка дозволяє безпечно передавати об’єкти між потоками. Вона блокує потік-отримувач, поки черга не стане непорожньою, і блокує потік-відправник, якщо черга переповнена.

У C++ можна реалізувати блокуючу чергу за допомогою *std::queue, std::mutex, std::condition_variable.*

Умовні змінні (Condition Variables)

Умовна змінна (std::condition_variable) дозволяє потоку "заснути", поки інший потік не повідомить, що деяка умова виконалась. Це корисно для координації обміну даними між етапами.

Приклад використання:

```
std::mutex mtx;
std::condition_variable cv;
bool ready = false;
cv.wait(lock, []{ return ready; });
```

Синхронізація потоків

Синхронізація потрібна, щоб уникнути гонок даних (race conditions). Для цього використовують:

- *std::mutex — для захисту критичних секцій.*
- *std::lock_guard або std::unique_lock — автоматичне управління м’ютексами.*
- *std::condition_variable — для сигналізації між потоками про готовність даних.*

Продуктивність конвеєра

Конвеєрна обробка дозволяє досягти припливного паралелізму, коли декілька елементів даних одночасно перебувають на різних етапах обробки. Це зменшує загальний час виконання при обробці великих обсягів даних.

🔶 **Оцінка ефективності**

- **Послідовне виконання** — кожен етап чекає на завершення попереднього.

- **Конвеєрне виконання** — всі етапи працюють паралельно над різними блоками даних.

Ефективність залежить від:

- часу, який займає кожен етап,
- об’єму даних,
- накладних витрат на синхронізацію,
- швидкості передачі даних між потоками.

Графічний аналіз

Для повного аналізу ефективності слід побудувати графіки:

- **часу виконання** залежно від розміру даних (x — кількість елементів, y — час у мілісекундах),
- **прискорення** (Speedup) — відношення часу послідовної обробки до часу конвеєрної.

Завдання:

1. Розробка програми конвеєра: Напишіть програму на C++, яка реалізує простий конвеєр для обробки даних. Конвеєр повинен мати три етапи:

- Читання даних (з файлу або генерування масиву).
- Обробка даних (наприклад, виконання математичних операцій).
- Запис результатів (виведення в файл або консоль).

2. Використання потоків: Реалізуйте кожен етап конвеєра в окремому потоці. Забезпечте синхронізацію між потоками за допомогою черг або умовних змінних.

3. Тестування продуктивності: Порівняйте час виконання конвеєрної обробки з версією, що виконує всі етапи послідовно. Оцініть переваги та недоліки конвеєрного підходу.

4. Графічний аналіз: Створіть графіки для візуалізації часу виконання для різних обсягів даних. Проаналізуйте, як зміна розміру даних впливає на продуктивність конвеєра.

Результати:

- Вихідний код програми.
- Скриншоти виконання програми.
- Висновок.

Список використаної літератури

1. B. Stroustrup: The C++ Programming Language (Fourth Edition). May 2013. Addison Wesley. Reading Mass. USA. May 2013. ISBN 0-321-56384-0. 1360 pages. Softcover, hardcover, and electronic versions.

1. Ira Pohl: Object-oriented programming using C++. 1997. Addison-Wesley. ISBN 978-0201895506. 543 pages.

2. B. Stroustrup: Programming -- Principles and Practice Using C++. December 2008. Addison-Wesley. ISBN 978-0321543721. 1264 pages. Softcover.

3. B. Stroustrup: Programming -- Principles and Practice Using C++ (Second Edition). May 2014. Addison-Wesley. ISBN 978-0321992789. 1312 pages. Softcover and electronic versions.

4. Гергель В. П.:Основи паралельного програмування. — Харків: ХНУ, 2005.

5. William Gropp, Ewing Lusk, Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. — MIT Press, 1999.

6. Blaise Barney.:Introduction to Parallel Computing, Lawrence Livermore National Laboratory. https://hpc.llnl.gov