МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені
ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт
з дисципліни
"Організація високопродуктивних обчислень"

Розробник: доцент, канд. техн. наук, доцент Павлов В.Г.
(посада, вчена ступінь та звання П.І.Б.)

Затверджено на засіданні кафедри
Протокол № 10 від 25 травня 2022 р.

Завідувач кафедри ОТ
_____ Стіренко С.Г.
(підпис) (прізвище, ініціали)

Київ – 2022

ЗМІСТ

ВСТУП

Дисципліна «Організація високопродуктивних обчислень» відноситься до вибіркових дисциплін підготовки фахівців рівня магістр з напрямку 121 «Інженерія програмного забезпечення» та 123 «Комп'ютерна інженерія». Для успішного вивчення курсу «Високопродуктивні обчислення» студенти повинні засвоїти матеріал та мати певні знання. вміння та навички з таких дисциплін, як «Теорія алгоритмів», «Алгоритми та методи обчислень», «Системне програмування», «Компоненти програмної інженерії», «Програмне забезпечення комп'ютерних систем».

Цикл лабораторних робіт складається з п'яти робіт і призначений для покриття частини комп'ютерного практикуму кредитного модулю дисципліни «Організація високопродуктивних обчислень» студентів денної форми навчання. Роботи дозволяють отримати практичні навички написання елементів системних програм та їх доповнення за індивідуальними завданнями з використанням механізмів сучасних мов та бібліотек системного програмування.

Методичні вказівки включають для кожної роботи:

- теоретичний матеріал, необхідний для їх виконання, з посиланнями на список рекомендованих джерел;
- інструкція з порядку підготовки до роботи та її виконання у формі доповнення шаблонів програмних проектів;
- варіанти індивідуальних завдань для виконання кожної лабораторної роботи з циклу.

1. МЕТОДИЧНІ ВКАЗІВКИ ЩО ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

1.1 Мета циклу лабораторних робіт

Метою проведення лабораторних занять з кредитного модулю "Високопродуктивні обчислення" надбання практичних навичок по роботі з методами та засобам програмування процесів (потоків) в сучасних мовах та бібліотеках паралельного програмування.

1. Обмін даними за участю декількох задач в MPI.
2. Користувацькі типи даних в MPI.
3. Приклад розробки складної програми за допомогою MPI

1.2 Зміст протоколу та оформлення лабораторних робіт

Протокол виконання кожної лабораторної роботи має містити:

- титульний лист;
- завдання на лабораторну роботу згідно варіанта;
- лістинг програми з коментаріями.
- результати експериментального виконання роботи;
- висновки по роботі (аналіз одержаних результатів).

Під час оформлення роботи слід звернути увагу на лістинг програми, в якому за допомогою відступів і коментарів відокремлювати основні частини програми, забезпечити легке читання лістинга. Програма повинна розпочинатися з заголовка, в якому треба вказати назву та варіант роботи, прізвище виконавця, дату, групу.

2. ЛАБОРАТОРНІ РОБОТИ

Цикл лабораторних робіт включає 5 робіт. Варіанти завдань видаються викладачем на підставі таблиць в описах робіт.

2.1 Лабораторна робота 1
Блокуючі передачі в MPI

Мета роботи:

- ознайомитись з бібліотекою розробки паралельних програм для систем з локальною пам'яттю − MPI;
- навчитись створювати програми, що використовують MPI;
- вивчити блокуючі функції MPI для передачі повідомлень в режимі один-до-одного;
- навчитись аналізувати алгоритми щодо необхідності передач даних.

Завдання: розробити паралельну програму для обчислення значення заданої по варіанту функції в будь-якій точці з заданою точністю ″ за допомогою обчислення суми ряду.

1.1. MPI

На сьогоднішній день найбільш поширеною технологією програмування для паралельних систем з локальною пам'яттю є інтерфейс MPI. MPI відповідає моделі, що базується на відправці повідомлень. MPI не додає в мову програмування нові керуючі конструкції, а містить тільки функції, що забезпечують взаємодію запущених задач.

MPI передбачає написання однієї програми, незалежно від кількості обчислювальних вузлів. Паралельне виконання програми в MPI досягається одночасним запуском програми на декількох обчислювальних вузлах. Утиліта запуску MPI (***mpiexec***) запускає програму на вказаних вузлах з локальною пам'яттю та виконує налаштування для передачі даних між запущеними програмами. Кожну запущену копію MPI програми будемо називати *задачею*.

Група − впорядкована множина задач. Кожній задачі в групі присвоюється порядковий номер, починаючи з 0. Цей номер називається **рангом** (або ідентифікатором) задачі.

Комунікатор − об'єкт MPI, який дозволяє задачам відправляти та отримувати повідомлення.

Інтра-комінукатор − комунікатор, пов'язаний з деякою групою задач. Дозволяє задачам відправляти повідомлення в межах цієї групи. Надалі будемо називати інтра-комунікатори просто «комунікаторами».
Інтер-комунікатор − комунікатор, призначений для відправки повідомлень між двома або більшою кількістю груп задач.
Після ініціалізації MPI для програми доступний комунікатор **MPI_COMM_WORLD**. Цей комунікатор пов'язаний з групою, яка включає всі запущені задачі.
Для роботи з MPI в програмі необхідно підключити заголовочний файл mpi.h.

MPI_Init

Ініціалізація MPI. Всі інші функції MPI дозволяється викликати тільки після MPI_Init.

```
int MPI_Init(int *argc, char ***argv);
```

Параметри argc та argv − покажчики на відповідні аргументи функції main().

MPI_Finalize

Очищення (деінціалізація) MPI.

```
int MPI_Finalize();
```

Перед тим, як завершити виконання, програма повинна викликати **MPI_Finalize.** Перед викликом MPI_Finalize задача має впевнитись, що її участь в обміні повідомлень завершена. Після виклику цієї функції заборонено викликати будь-які інші функції MPI.

MPI_Abort

Аварійне завершення роботи MPI програми (всіх запущених задач).

```
int MPI_Abort(MPI_Comm comm, int errorcode);
```

- ***comm*** − комунікатор.
- ***errorcode*** − код помилки, який буде повернутий процесу, що викликав MPI програму.

MPI_Comm_size

Визначає кількість задач в групі, що пов'язана з вказаним комунікатором.

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- ***comm*** − комунікатор.
- [OUT] ***size*** − покажчик на змінну цілого типу, куди буде записана кількість задач в групі, що пов'язана з комунікатором comm.

MPI_Comm_rank

Визначає ранг поточної задачі в групі, що пов'язана з вказаним комунікатором.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

- ***comm*** − комунікатор групи процесів.

- [OUT] ***rank*** − покажчик на змінну цілого типу, куди буде записаний ранг поточної задачі в групі, що пов'язана з комунікатором ***comm***.

Приклад 1. Розглянемо наступний код:
Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_series/mpi_first.c

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
/* Ініціалізація середовища MPI */
MPI_Init(&argc, &argv);
/* Отримання рангу даної задачі */
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Отримання загальної кількості задач */
int np;
MPI_Comm_size(MPI_COMM_WORLD, &np);
/* Кожна задача виконає цей оператор */
printf("I am task %d. There are %d tasks total.\n", rank, np);
if(rank == 1)
{
/* Цей оператор виконає тільки задача з рангом 1 */
printf("I am task 1 and I do things differently.\n");
}
/* Де-ініціалізація середовища MPI та вихід з програми */
MPI_Finalize();
return 0;
}
```

Компілюємо та запускаємо:

```
user@n001$ mpicc -W -Wall -O2 -std=c99 mpi_first.c
user@n001$ mpiexec -np 4 ./a.out
I am task 0. There are 4 tasks total.
I am task 1. There are 4 tasks total.
I am task 3. There are 4 tasks total.
I am task 1 and I do things differently.
I am task 2. There are 4 tasks total.
```

mpiexec запустив 4 екземпляри даної програми та призначив їм ранги. Кожна задача отримала власний ранг та загальну кількість задач. Кожна задача вивела ці дані на консоль. Задача з рангом 1 окрім цього вивела ще один рядок.

1.2. Повідомлення

Кожне повідомлення складається з **конверту** та **буфера даних**. До конверту
повідомлення відносяться:

- ранг задачі-відправника;
- ранг задачі-отримувача;
- тег повідомлення;
- комунікатор, в рамках якого відбувається взаємодія вказаних задач.

Функції відправки та прийому повідомлень MPI оперують масивами однотипних елементів − так званими **буферами**. Розмір буферів вказується не в байтах, а в елементах.

Кожна функція MPI, що працює з буферами, має аргумент, в якому вказується тип елементів буфера. Для кожного базового типу мови С передбачена константа MPI (табл. 1.1). Програміст може реєструвати в MPI свої власні типи даних, наприклад, структури, після чого MPI зможе обробляти їх нарівні з базовими (ця можливість буде розглянута більш детально далі).

Програміст має забезпечити, щоб фактичний тип змінних та тип, вказаний в аргументах функції MPI, збігались. Компілятор такої перевірки *не виконує* та не видає попереджень у випадку якщо в аргументах функції MPI вказано тип, що не відповідає типу змінної.

Програміст може вважати, що передача відбувається за три кроки, і має забезпечити, щоб на кожному з цих етапів типи даних збігались:

- відправник зчитує дані з буфера та формує повідомлення (тип змінних в буфері відправки співпадає з типом, вказаним MPI функції відправки);
- повідомлення передається від задачі-відправника до задачі-отримувача (тип, що вказано в функції відправки, співпадає з типом, вказаним при прийомі);
- отримувач записує дані з повідомлення до буфера (тип, що вказано в функції прийому, співпадає з типом змінних в буфері прийому).

З кожним повідомленням, що передається в режимі один-до-одного, асоціюється певний тег. Тег − це ціле число, яке обирає програміст на власний розсуд. Діапазон допустимих значень тегів − від 0 до як мінімум

32767 включно (реалізаціям MPI дозволено обирати максимальне значення тегу). Звичайно теги використовуються програмістами для того, щоб розрізняти різні види повідомлень. Задача-відправник повідомлення встановлює його тег, а задача-отримувач може обрати такий режим прийому, щоб отримувати тільки повідомлення із заданим тегом.

1.3. Блокуючі передачі

В даній лабораторній роботі будемо розглядати передачу повідомлень між двома задачами (так звана передача один-до-одного, точка-точка, англ. point-to-point). Передача повідомлень між двома задачами MPI реалізується на основі функції **відправки** та функції **прийому**. Ці функції реалізовані в декількох варіантах: блокуючі та неблокуючі. Розглянемо блокуючі функції, так як їх використання є більш простим.

Блокуючі функції під час виклику блокують задачу до тих пір, доки буфери, задіяні в передачі будуть необхідні для копіювання повідомлення. Блокуюча функція відправки повідомлення блокує задачу до тих пір, доки дані не будуть скопійовані з буфера користувача в проміжний буфер MPI або будуть відправлені. Блокуюча функція прийому повідомлення блокує задачу до тих пір, доки прийняті дані не будуть повністю записані в буфер користувача.

Табл. 1.1. Відповідність констант опису типів MPI стандартним типам змінних мови С

Тип мови С	**Константа MPI**
int	MPI_INT
short int	MPI_SHORT
char	MPI_CHAR
long int	MPI_LONG
unsigned int	MPI_UNSIGNED
unsigned short int	MPI_UNSIGNED_SHORT
unsigned char	MPI_UNSIGNED_CHAR
unsigned long int	MPI_UNSIGNED_LONG
float	MPI_FLOAT
double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE

MPI_Send

Блокуюча відправка повідомлення вказаній задачі.

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

- ***buf*** − покажчик буферу відправки; розмір буферу має бути не менший, ніж ***count*** елементів.

- ***count*** − кількість елементів, що будуть відправлені задачі з рангом dest.
- ***datatype*** − тип елементів, що відправляються (табл. 1.1).
- ***dest*** − ранг задачі, що отримає дані.
- ***tag*** − тег повідомлення.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.

Функція MPI_Send повертає управління програмі тоді, коли дані вже скопійовані з буферу buf − саме це і називається блокуючою передачею.
В момент, коли функція MPI_Send повертає управління програмі, не гарантується, що повідомлення вже отримано задачею-отримувачем. Тим не менше, звичайно це не потрібно знати.

MPI_Recv

Блокуючий прийом повідомлення від вказаної задачі з вказаним тегом.

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

- [OUT] ***buf*** − покажчик буферу прийому; розмір буферу має бути не менше, ніж ***count*** елементів.
- ***count*** − кількість елементів, що будуть прийняті від задачі з рангом ***source***.
- ***datatype*** − тип елементів, що приймаються (табл. 1.1).
- ***source*** − ранг задачі, що відправляє дані або **MPI_ANY_SOURCE** для прийому повідомлення від будь-якої задачі.
- ***tag*** − тег повідомлення, яке необхідно прийняти, або **MPI_ANY_TAG** для прийому повідомлення з будь-яким тегом.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.
- [OUT] ***status*** − покажчик на структуру **MPI_Status**, в яку буде записано інформацію про прийняте повідомлення або **MPI_STATUS_IGNORE**, якщо програмі не потрібна ця інформація.

Функція MPI_Recv обирає повідомлення порівнюючи значення його конверту з вказаними аргументами: ранг задачі-відправника, тег, комунікатор. Якщо ці значення виявляються рівними, то повідомлення приймається повністю та функція **MPI_Recv** повертає управління програмі.
Коли **MPI_Recv** повернула управління програмі, гарантується, що прийняті дані
вже повністю записані в буфер прийому buf.
Структура **MPI_Status** містить наступні поля:

- **MPI_SOURCE** − ранг задачі-відправника;
- **MPI_TAG** − тег повідомлення;
- **MPI_ERROR** − код помилки.

Ця інформація може бути потрібна задачі, якщо вона під час прийому скористалась константами MPI_ANY_SOURCE або MPI_ANY_TAG і не знає рангу відправника і/або тегу повідомлення.

Приклад 2. Розглянемо наступний код:
Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.
Файл code/lab_series/mpi_send_recv.c

```
#include <stdio.h>
#include <mpi.h>
const int TAG_SOME_DATA = 10;
int main(int argc, char *argv[])
{
MPI_Init(&argc, &argv);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int np;
MPI_Comm_size(MPI_COMM_WORLD, &np);
if(rank == 0)
{
int x = 42;
/* Послідовна передача x кожній задачі 1..np */
for(int i = 1; i < np; i++)
{
MPI_Send(&x, 1, MPI_INT, i, TAG_SOME_DATA, MPI_COMM_WORLD);
}
}
else
{
int x = 0;
/* Прийом x від задачі 0 */
MPI_Recv(&x, 1, MPI_INT, 0, TAG_SOME_DATA, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
printf("Task %d. Got data = %d\n", rank, x);
}
MPI_Finalize();
return 0;
}
```

Запускаємо програму:

```
user@n001$ mpirun -np 4 ./a.out
Task 1. Got data = 42
```

Task 3. Got data = 42
Task 2. Got data = 42
Задача з рангом 0 передала значення змінної x всім іншим задачам, які це значення прийняли та вивели на консоль.

Приклад 3. Розглянемо наступний код:
Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.
Файл code/lab_series/mpi_recv_any_source.c

```
#include <stdio.h>
#include <mpi.h>
const int TAG_DATA1 = 10;
const int TAG_DATA2 = 20;
const int TAG_DATA3 = 25;
int main(int argc, char *argv[])
{
MPI_Init(&argc, &argv);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0)
{
int x;
/* Прийом повідомлення з тегом TAG_DATA3 */
MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG_DATA3,
MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
printf("Got %d with tag TAG_DATA3\n", x);
/* Прийом повідомлення з тегом TAG_DATA1 */
MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG_DATA1,
MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
printf("Got %d with tag TAG_DATA1\n", x);
/* Прийом повідомлення з тегом TAG_DATA2 */
MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, TAG_DATA2,
MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
printf("Got %d with tag TAG_DATA2\n", x);
}
else if(rank == 1)
{
int x = 1000;
```

```
MPI_Send(&x, 1, MPI_INT, 0, TAG_DATA1, MPI_COMM_WORLD);
}
else if(rank == 2)
{
int x = 2000;
MPI_Send(&x, 1, MPI_INT, 0, TAG_DATA2, MPI_COMM_WORLD);
}
else
{
int x = 3000;
MPI_Send(&x, 1, MPI_INT, 0, TAG_DATA3, MPI_COMM_WORLD);
}
MPI_Finalize();
return 0;
}
```

Запускаємо:

user@n001$ mpirun -np 4 ./a.out
Got 3000 with tag TAG_DATA3
Got 1000 with tag TAG_DATA1
Got 2000 with tag TAG_DATA2

Даний приклад демонструє наступне:

- теги − будь-які числа, що обираються програмістом;
- використання константи MPI_ANY_SOURCE для відбору повідомлень тільки по тегу.

1.4. Математичний апарат

Задача: розробити паралельну програму знаходження значення $f(x)$ заданої функції f у заданій точці $x0$ із заданою точністю $''$.

Рядом Тейлора називається розклад функції в нескінченну суму степеневих функцій. Загальний вигляд ряду для функції $f(x)$ в точці a має вигляд:

$$\sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

Розклад функції існує в точці a, якщо в цій точці вона є нескінченно диференційовною, тобто в даній точці можна обчислити похідну будь-якого порядку від $f(x)$. Ряд Тейлора в точці $a = 0$ називається рядом Маклорена. Якщо $f(x)$ є аналітичною функцією, то її ряд Тейлора в будь-якій точці x області визначення збігається до значення $f(x)$:

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

Ряди Тейлора та Маклорена застосовуються при апроксимації функції многочленами, лінеаризації рівнянь та обчисленні математичних констант. Наведемо основні відомі ряди.

Експонента

$$\mathrm{e}^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \quad x \in \mathbb{C}.$$

Натуральний логарифм

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}x^n}{n}, \quad |x| < 1.$$

Квадратний корінь

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} + \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n(2n)!}{(1-2n)(n!)^2 4^n}, \quad |x| < 1.$$

Геометричний ряд

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots = \sum_{n=0}^{\infty} x^n, \quad |x| < 1.$$

Синус

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}x^{2n+1}, \quad x \in \mathbb{C}.$$

Косинус

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!}x^{2n}, \quad x \in \mathbb{C}.$$

Арксинус

$$\arcsin x = x + \frac{x^3}{6} + \frac{3x^5}{40} + \cdots = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n(n!)^2(2n+1)}x^{2n+1}, \quad |x| < 1.$$

Арктангенс

$$\operatorname{arctg} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}x^{2n+1}, \quad |x| < 1.$$

Гіперболічний синус

$$\operatorname{sh} x = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots = \sum_{n=0}^{\infty} \frac{1}{(2n+1)!} x^{2n+1}, \quad x \in \mathbb{C}.$$

Гіперболічний косинус

$$\operatorname{ch} x = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots = \sum_{n=0}^{\infty} \frac{1}{(2n)!} x^{2n}, \quad x \in \mathbb{C}.$$

Арксинус гіперболічний

$$\operatorname{arsh} x = x - \frac{x^3}{6} + \frac{3x^5}{40} - \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n (2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}, \quad |x| < 1.$$

Арктангенс гіперболічний

$$\operatorname{arth} x = x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots = \sum_{n=0}^{\infty} \frac{1}{2n+1} x^{2n+1}, \quad |x| < 1.$$

Точність та критерій зупинки сумування. Звичайно, програмувати сумування нескінченної кількості членів ряду не має смислу − таке обчислення ніколи не завершиться. Тому необхідно мати критерій, за яким визначати, коли зупиняти сумування.

Відомо, що для знакопочергових рядів Лейбніца абсолютне значення суми залишку ряду (відкинутих членів ряду, що не приймають участь в сумі) менша за абсолютне значення першого відкинутого члену. Не всі ряди, представлені вище, є знакопочерговими рядами Лейбніца, але для спрощення будемо застосовувати цей критерій і до них.

Отже, нехай задана точність ε. Необхідно просумувати всі члени ряду, що за абсолютним значенням більші за ε. Як тільки знайдено член ряду, що менший за абсолютним значенням за ε, можна припиняти сумування (але якщо буде просумовано кілька «зайвих членів», то це не погіршить точність).

1.5. Паралельне представлення задачі

Особливістю даної задачі є те, що нам невідомо, скільки членів ряду необхідно сумувати взагалі та ми не можемо заздалегідь визначити, скільки членів ряду має вирахувати кожна задача. Тому застосуємо наступний підхід. Нехай є P задач. Задача з номером i буде обчислювати члени ряду з номерами i, $P + i$, $2P + i$ і так далі. Все обчислення буде розбите на кроки, під час кожного з яких буде паралельно обчислено P членів ряду:

$f(x0) = |a_1 + a_2 + \ldots + a_P| + a_{P+1} + a_{P+2} + \ldots + a_{2P}| + \ldots$

крок 1 ***крок 2***

Після виконання кожного кроку всі задачі передають значення, обчислене на поточному кроці, головній задачі. Головна задача сумує всі прийняті результати та перевіряє критерій зупинки.

1.6. Повний текст програми

Обчислення значення експоненти e*x* в точці *x* з точністю ***ε***. Вхідні дані: число ***x*** знаходиться в файлі in.txt. Результат буде записано в файл out.txt.
Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.
Файл code/lab_series/ex.c

```
#include <mpi.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
const double EPSILON = 1E-100;                  // Точність обчислення значення
const int VALUE_TAG = 1;                    // Тег показнику ступеня числа E
const int TERM_NUMBER_TAG = 2;              // Тег номера поточного члену ряду
const int TERM_TAG = 3;                     // Тег значення поточного члену ряду
const int BREAK_TAG = 4;                    // Тег сигналу про завершення обчислень
const char *input_file_name = ”in.txt”;         // Ім’я файла вхідних даних
const char *output_file_name = ”out.txt”;       // Ім’я файла результату
/* Функція обчислення факторіалу */
double factorial(int value)
{
/* Факторіал від’ємного числа не визначений */
if(value < 0)
{
return NAN;
}
/* 0! = 1 за визначенням */
else if(value == 0)
{
return 1.;
}
/* обчислення факторіалу N як добутку всіх натуральних чисел від 1 до N */
```

```
else
{
double fact = 1.;
for(int i = 2; i <= value; i++)
{
fact *= i;
}
return fact;
}
}
/* Функція обчислення члену ряду за його номером в точці value
* Для exp(x) член ряду дорівнює x^n / n! */
double calc_series_term(int term_number, double value)
{
return pow(value, term_number) / factorial(term_number);
}
/* Основна функція (програма обчислення e^x) */
int main(int argc, char *argv[])
{
/* Ініціалізація середовища MPI */
MPI_Init(&argc, &argv);
/* Отримання номеру даної задачі */
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Отримання загальної кількості задач */
int np;
MPI_Comm_size(MPI_COMM_WORLD, &np);
/* Значення x для обчислення exp(x) */
double exponent;
/* Введення x в задачі 0 з файла */
if(rank == 0)
{
FILE *input_file = fopen(input_file_name, "r");
/* Аварійне завершення всіх задач, якщо не вдається відкрити вхідний
файл */
if(!input_file)
{
fprintf(stderr, "Can't open input file!\n\n");
MPI_Abort(MPI_COMM_WORLD, 1);
return 1;
}
```

```
/* Зчитування x з файла */
fscanf(input_file, "%lf", &exponent);
fclose(input_file);
}
/* Розсилка x з задачі 0 всім іншим задачам */
if(rank == 0)
{
/* Послідовна передача x кожній задачі 1..np */
for(int i = 1; i < np; i++)
{
MPI_Send(&exponent, 1, MPI_DOUBLE, i, VALUE_TAG,
MPI_COMM_WORLD);
}
}
else
{
/* Прийом x від задачі 0 */
MPI_Recv(&exponent, 1, MPI_DOUBLE, 0, VALUE_TAG,
MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
/* Номер останнього обчисленого члену ряду */
int last_term_number = 0;
/* Сума членів ряду */
double sum = .0;
/* Основний цикл ітерації */
for(int step = 0; step < 1000; step++)
{
/* Номер члену ряду, що обчислюється на поточному кроці в даній задачі
*/
int term_number;
/* Пересилка з задачі 0 всім іншим задачам номерів членів, які вони
* мають обчислити на поточному кроці */
if(rank == 0)
{
term_number = last_term_number++;
int current_term_number = last_term_number;
for(int i = 1; i < np; i++)
{
MPI_Send(&current_term_number, 1, MPI_INT, i, TERM_NUMBER_TAG,
MPI_COMM_WORLD);
```

```
current_term_number++;
}
last_term_number = current_term_number;
}
else
{
MPI_Recv(&term_number, 1, MPI_INT, 0, TERM_NUMBER_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
/* Обчислення поточного члену ряду */
double term = calc_series_term(term_number, exponent);
/* Прапорець ”ітерація завершена, так як досягнуто необхідну точ-
ність” */
int need_break = false;
/* Обчислення суми членів ряду */
if(rank == 0)
{
double current_term = term;
/* Додавання до загальної суми члену, обчисленого в задачі 0 */
sum += current_term;
/* Оскільки ряд для e^x є монотонно спадаючим, члени ряду
* обчислюються задачами за зростанням рангу та прийом членів ряду
* від задач ведеться за зростанням рангу, то якщо останній прийнятий
* член менше константи EPSILON, то і всі наступні члени також
* менше цієї константи. Після додавання такого члену необхідна
* точність досягнута і можна завершувати ітерацію */
if(current_term < EPSILON)
{
need_break = true;
}
for(int i = 1; i < np; i++)
{
/* Прийом члену від і-тої задачі, додавання його до загальної суми */
MPI_Recv(&current_term, 1, MPI_DOUBLE, i, TERM_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
sum += current_term;
/* Перевірка умови завершення ітерації (див. вище) */
if(current_term < EPSILON)
{
need_break = true;
break;
```

```
}
}
/* Передача сигналу про необхідність завершення ітерації з задачі 0 всім
* іншим задачам */
for(int i = 1; i < np; i++)
{
MPI_Send(&need_break, 1, MPI_INT, i, BREAK_TAG,
MPI_COMM_WORLD);
}
}
else
{
/* Передача обчисленого члену ряду в задачу 0 */
MPI_Send(&term, 1, MPI_DOUBLE, 0, TERM_TAG,
MPI_COMM_WORLD);
/* Прийом від задачі 0 сигналу про необхідність завершення ітерації */
MPI_Recv(&need_break, 1, MPI_INT, 0, BREAK_TAG,
MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}
/* Завершення ітерації, якщо досягнута необхідна точність */
if(need_break)
{ break;
}
}
/* Вивід результату в задачі 0 */
if(rank == 0)
{
FILE *output_file = fopen(output_file_name, "w");
/* Аварійне завершення, якщо не вдається відкрити файл результату */
if(!output_file)
{
fprintf(stderr, "Can't open output file!\n\n");
MPI_Abort(MPI_COMM_WORLD, 2);
return 2;
}
fprintf(output_file, "%.15lf\n", sum);
} /* Де-ініціалізація середовища MPI та вихід з програми */
MPI_Finalize();
return 0;
}
```

1.7. Варіанти

Варіант визначається по табл. 1.2.

Табл. 1.2. Варіанти завдання

№	Функція	Точність обчислення
1	Експонента	10^{-8}
2	Експонента	10^{-11}
3	Натуральний логарифм	10^{-8}
4	Натуральний логарифм	10^{-11}
5	Корінь квадратний	10^{-8}
6	Корінь квадратний	10^{-11}
7	Геометричний ряд	10^{-8}
8	Геометричний ряд	10^{-11}
9	Синус	10^{-8}
10	Синус	10^{-11}
11	Косинус	10^{-8}
12	Косинус	10^{-11}
13	Синус гіперболічний	10^{-8}
14	Синус гіперболічний	10^{-11}
15	Косинус гіперболічний	10^{-8}
16	Косинус гіперболічний	10^{-11}
17	Арксинус	10^{-8}
18	Арксинус	10^{-11}
19	Арктангенс	10^{-8}
20	Арктангенс	10^{-11}
21	Арксинус гіперболічний	10^{-8}
22	Арксинус гіперболічний	10^{-11}
23	Арктангенс гіперболічний	10^{-8}
24	Арктангенс гіперболічний	10^{-11}

1.8. Питання для самоконтролю

1. Якій моделі взаємодії задач відповідає MPI?
2. Що представляє собою задача в MPI?
3. Що таке комунікатор? Який комунікатор створюється автоматично під час ініціалізації MPI?
4. Наведіть приклади функцій MPI, які реалізують передачу повідомлень між задачами в режимі один-до-одного?
5. Для чого потрібен тег повідомлення?
6. Як задача може прийняти будь-яке повідомлення (повідомлення з будь-яким тегом від будь-якої задачі)?
7. Чи може програма зчитувати чи записувати якісь дані в буфер зразу ж після виклику MPI_Send?
8. За допомогою якої функції можна аварійно завершити роботу MPI програми?
9. В функціях прийому повідомлень структура MPI_Status використовується для:
   а) визначення кількості прийнятих байт;
   б) визначення коректності завершення операції прийому;
   в) визначення рангу задачі, що відправила дані;
   г) визначення тегу прийнятого повідомлення.

1.9. Література

1. Специфікація MPI 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

2.2 Лабораторна робота 2
Неблокуючі та широкомовні передачі в MPI

Мета роботи:

- вивчити спосіб неблокуючої передачі даних;
- вивчити спосіб передачі даних за допомогою широкомовних повідомлень;
- вивчити метод розпаралелювання задачі шляхом розбиття задачі на під задачі на прикладі чисельного інтегрування.

Завдання: розробити паралельну програму чисельного інтегрування функції $f(x)$ заданим методом інтегрування за допомогою заданих засобів MPI (залежно від варіанту).

2.1. Неблокуючі передачі

Особливістю неблокуючої передачі повідомлень є те, що виконання програми не призупиняється для виконання прийому чи передачі даних. Такий спосіб передачі є більш складним для програмування, але при правильному застосуванні може в значній мірі зменшити втрату ефективності паралельних обчислень через повільну (у порівнянні зі швидкістю процесору) передачу даних. Неблокуюча передача повідомлень дозволяє суміщати в часі відправку чи прийом повідомлень та обчислення.

MPI забезпечує можливість неблокуючої передачі даних між двома задачами. Назва неблокуючих аналогів для відповідних блокуючих функцій створюється з назви відповідної функції шляхом додавання префіксу «I» (від англ. immediate).

Неблокуючі функції приймають ті ж самі параметри, що і блокуючі, і ще один додатковий параметр request з типом **MPI_Request**. У функції прийому **MPI_Irecv** відсутній параметр ***status***.

В параметрі request передається дескриптор запиту відправки або прийому. Цей дескриптор можна передати функціям **MPI_Wait** або **MPI_Waitall** для того, щоб дочекатись завершення передачі.

Усі буфери, що використовуються в неблокуючих передачах стають недоступними користувачу після запуску передачі чи прийому інформації. Буфери стануть знову доступними після виконання функцій MPI_Wait або MPI_Waitall для цих передач.

Зверніть увагу! *Вибір способу відправки повідомлень (блокуючий/неблокуючий) не впливає на вибір способу прийому. Повідомлення, які були відправлені блокуючими функціями, можна приймати неблокуючими функціями, і навпаки.*

MPI_Isend

Неблокуюча відправка повідомлення вказаній задачі.

int MPI_Isend(**void** *buf, **int** count, **MPI_Datatype** datatype,
int dest, **int** tag, **MPI_Comm** comm,
MPI_Request *request);

- ***buf*** − покажчик буфера відправки; розмір буфера має бути не менший, ніж ***count*** елементів. Після виконання функції буфер буде тимчасово недоступний.
- ***count*** − кількість елементів, що будуть відправлені задачі з рангом dest.
- ***datatype*** − тип елементів, що відправляються (табл. 1.1).
- ***dest*** − ранг задачі, що отримає дані.
- ***tag*** − тег повідомлення.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.
- [OUT] ***request*** − покажчик на змінну типу **MPI_Request**, в яку буде записано дескриптор передачі, розпочатої цим викликом **MPI_Isend**.

MPI_Irecv

Неблокуючий прийом повідомлення від вказаної задачі з вказаним тегом.
int MPI_Irecv(**void** *buf, **int** count, **MPI_Datatype** datatype,
int source, **int** tag, **MPI_Comm** comm, **MPI_Request** *request);

- [OUT] ***buf*** − покажчик буфера прийому; розмір буфера має бути не менше, ніж count елементів. Після виконання функції буфер буде тимчасово недоступний.
- ***count*** − кількість елементів, що будуть прийняті від задачі з рангом ***source***.
- ***datatype*** − тип елементів, що приймаються (табл. 1.1).
- ***source*** − ранг задачі, що відправляє дані або **MPI_ANY_SOURCE** для прийому повідомлення від будь-якої задачі.
- ***tag*** − тег повідомлення, яке треба прийняти або **MPI_ANY_TAG** для прийому повідомлення з будь-яким тегом.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.
- [OUT] request − покажчик на змінну типу **MPI_Request**, в яку буде записано дескриптор передачі, розпочатої цим викликом **MPI_Irecv**.

MPI_Wait

Очікування завершення вказаної неблокуючої передачі. Після виконання функції буфер, задіяний в передачі, знову стає доступним.
int MPI_Wait(**MPI_Request** *request, **MPI_Status** *status);

- ***request*** − покажчик на дескриптор передачі.
- [OUT] ***status*** − покажчик на структуру **MPI_Status**, в яку буде записано інформацію про прийняте повідомлення або

MPI_STATUS_IGNORE, якщо програмі не потрібна ця інформація.

MPI_Waitall

Очікує завершення вказаних неблокуючих передач. Після виконання функції буфери, задіяний в передачах, знову стають доступними.

```
int MPI_Waitall(int count, MPI_Request array_of_requests[],
MPI_Status array_of_statuses[]);
```

- ***count*** − кількість дескрипторів в масиві array_of_requests.
- ***array_of_requests*** − масив дескрипторів передач.
- [OUT] ***array_of_statuses*** − масив станів передач.

Приклад 4. В якості простого прикладу неблокуючої передачі даних можна привести програму, яка передає масив з п’яти цілих чисел з однієї задачі в іншу:

Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_integrate/mpi_nonblock.c

```
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
const int MY_TAG = 42;
/*
* Програма розрахована рівно на 2 процеси. Від процесу 0 до
* процесу 1 передається масив з 5 цілих чисел за допомогою
* неблокуючої передачі. При цьому процес 1 починає прийом
* даних раніше ніж процес 0 починає їх передачу.
*/
int main (int argc, char* argv[])
{
int np;
int rank;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if(rank == 0)
{
sleep(5); /* Затримка. В реальній програмі -- обчислення. */
int data[] = { 1, 2, 3, 4, 5}; /* Дані для передачі */
MPI_Request send_req; /* Дескриптор передачі даних */
printf("Task 0. MPI_Isend()\n");
```

```
MPI_Isend(&data, 5, MPI_INT, 1, MY_TAG, MPI_COMM_WORLD,
&send_req);
/* Тепер буфер data недоступний */
printf("Task 0. Sending data...\n");
sleep(5); /* Затримка. В реальній програмі -- обчислення. */
printf("Task 0. Computation finished.\n");
/* Очікування завершення передачі */
MPI_Wait(&send_req, MPI_STATUS_IGNORE);
}
if(rank == 1)
{
int data[5]; /* Буфер для прийому даних */
MPI_Request recv_req; /* Дескриптор прийому даних */
printf("Task 1. MPI_Irecv()\n");
MPI_Irecv(&data, 5, MPI_INT, 0, MY_TAG, MPI_COMM_WORLD,
&recv_req);
/* Тепер буфер data недоступний */
printf("Task 1. Recieving data...\n");
sleep(2); /* Затримка. В реальній програмі -- обчислення. */
/* Очікування на завершення передачі */
MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
printf("Task 1. Recieved data:");
/* Вивід даних */
for(int i = 0; i < 5; i++)
{
printf(" %d", data[i]);
}
printf("\n");
}
MPI_Finalize();
return 0;
}
```

2.2. Широкомовні передачі

MPI_Bcast

MPI_Bcast реалізує передачу «один до багатьох»: пересилає повідомлення від задачі з рангом ***root*** до усіх інших задач в комунікаторі ***comm***.

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

- [IN/OUT] ***buffer*** − покажчик буферу відправки (для задачі root), для інших − покажчик буферу прийому.

- ***count*** − кількість елементів в буфері.
- ***datatype*** − тип елементів в буфері.
- ***root*** − ранг задачі, що відправляє дані.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.

2.3. Передачі багато-до-одного MPI_Reduce

Операція згортки. Задача з номером root отримує дані від усіх задач і виконує над даними операцію ***op***.

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

- ***sendbuf*** − покажчик буфера відправки.
- ***count*** − кількість елементів в буфері.
- ***datatype*** − тип елементів в буфері.
- ***op*** − операція згортки даних (табл. 2.1).
- ***root*** − ранг задачі, яка отримає результат згортки.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.
- [OUT] ***recvbuf*** − покажчик буферу прийому (вказує тільки задача з рангом ***root***, для інших допустимо **NULL**).

В MPI є декілька вбудованих операцій для згортки (табл. 2.1). MPI також дозволяє програмісту визначати власні операції згортки.

Табл. 2.1. Операції MPI_OP

MPI_Name	**Операція**
MPI_MAX	Визначення максимального значення
MPI_MIN	Визначення мінімального значення
MPI_PROD	Добуток
MPI_SUM	Сума
MPI_LAND	Логічне І
MPI_LOR	Логічне АБО
MPI_LXOR	Логічна сума по модулю два
MPI_BAND	Порозрядне І
MPI_BOR	Порозрядне АБО
MPI_BXOR	Порозрядна сума по модулю два

2.4. Математичний апарат

Чисельне інтегрування − наближене обчислення визначеного інтегралу. Методи чисельного інтегрування, що розглядаються в даній роботі, найпростіше пояснити з використанням графічної інтерпретації визначеного інтегралу (рис. 2.1): визначений інтеграл чисельно дорівнює площі криволінійної трапеції, обмеженої віссю абсцис, графіком функції і відрізками прямих $x = a$, $x = b$, де a і b − межі інтегрування.

Апроксимація − науковий метод, що полягає у заміні одних об'єктів іншими, більш простими (наприклад, кривих ліній − ламаними, неперервних функцій − многочленами, плоских фігур − кількома прямокутниками).

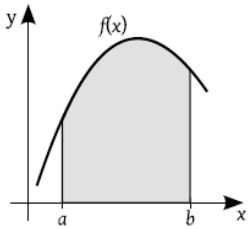


Рис. 2.1. Графічна інтерпретація визначеного інтегралу

Головна ідея більшості методів інтегрування полягає в апроксимації криволінійної трапеції фігурами, площу яких можна точно (і просто) розрахувати. При цьому сам інтеграл можна представити у вигляді формули:

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{i=1}^{n} w_i f(x_i),$$

де:

- n − кількість точок, в яких визначається значення підінтегральної функції;
- x_i − вузол методу;
- w_i − вага вузла (коефіцієнт, що має формулу для обчислення в кожному методі).

Метод лівих прямокутників

Проміжок інтегрування $[a; b]$ розбивають точками $a = x_0; x_1; \ldots; b = x_{n-1}$ на n рівних частин довжи$b - a\Delta x = h =$ кожна (рис. 2.2). На кожній частині будується пямокуn, верхній лівий кут якого лежить на графіку функції. Наближене значення інтегралу дорівнює сумі площ прямокутників:

$$\int_a^b f(x)\,\mathrm{d}x \approx h\sum_{i=0}^{n-1} f(x_i) = h\sum_{i=0}^{n-1} f(a+ih).$$

Метод правих прямокутників

Метод правих прямокутників відрізняється від методу лівих тим, що у прямокутників праві верхні кути лежать на графіку функції (рис. 2.3). Наближене значення інтегралу в даному випадку:

$$\int_a^b f(x)\,\mathrm{d}x \approx h\sum_{i=1}^{n} f(x_i) = h\sum_{i=1}^{n} f(a+ih)$$

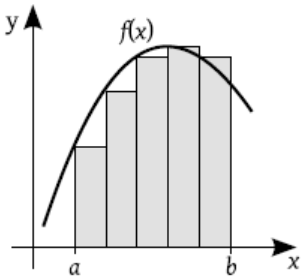


Рис. 2.2. Апроксимація лівими прямокутниками

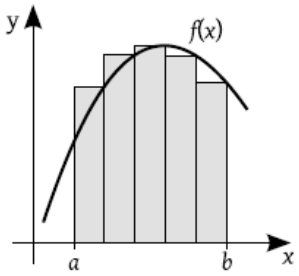


Рис. 2.3. Апроксимація правими прямокутниками

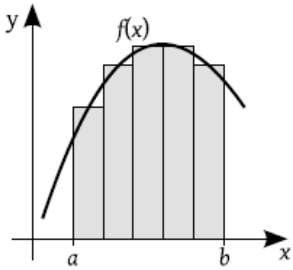


Рис. 2.4. Апроксимація середніми прямокутниками

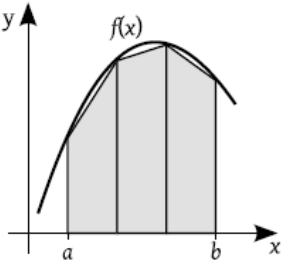


Рис. 2.5. Апроксимація трапеціями

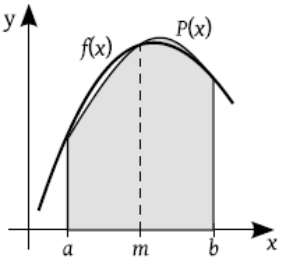


Рис. 2.6. Апроксимація параболами

Метод середніх прямокутників

Метод середніх прямокутників відрізняється від методу лівих тим, що у прямокутників середні точки верхньої сторони лежать на графіку функції (рис. 2.4).

Наближене значення інтегралу в даному випадку:

$$\int_a^b f(x)\,\mathrm{d}x \approx h\sum_{i=0}^{n-1} f\left(x_i + \frac{h}{2}\right) = h\sum_{i=0}^{n-1} f\left(a + ih + \frac{h}{2}\right).$$

Метод трапецій

Метод трапецій передбачає апроксимацію функції на кожному з відрізків [*xi*; *xi*+1] лінійною функцією. Таким чином, на кожному з відрізків будується трапеція (рис. 2.5). Площа кожної окремо взятої трапеції обчислюється за формулою:

$$S = h\frac{f(x_i) + f(x_{i+1})}{2}.$$

Оскільки інтеграл є сумою площ *n* трапецій, то наближене значення інтегралу можна обчислити за формулою:

$$\int_a^b f(x)\,\mathrm{d}x \approx h\left(\frac{f(x_0) + f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i)\right) =$$

$$h\left(\frac{f(a) + f(b)}{2} + \sum_{i=1}^{n-1} f(a + ih)\right).$$

Метод парабол (метод Сімпсона)

Для обчислення значення інтегралу на відрізку [*a*; *b*] методом Сімпсона необхідно виконати апроксимацію заданої функції параболою *P*(*x*), або кількома параболами для збільшення точності (рис. 2.6). Вихідна функція

і апроксимуюча мають 3 спільні точки, зазвичай це кінці і середина проміжку інтегрування (*a; b;m*). Наближене значення інтегралу при апроксимуванні однією параболою обчислюється за формулою:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right).$$

При апроксимації *n* параболами формула має вигляд:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6n}\left(\frac{1}{2}f(x_0)+\sum_{i=1}^{n-1}f(x_i)+\sum_{i=0}^{n-1}f\left(\frac{x_i+x_{i+1}}{2}\right)+\frac{1}{2}f(x_n)\right)=$$

$$=\frac{h}{3}\left(\frac{1}{2}f(a)+\sum_{i=1}^{n-1}f(a+ih)+\sum_{i=0}^{n-1}f\left(a+ih+\frac{h}{2}\right)+\frac{1}{2}f(b)\right).$$

2.4.1. Визначення похибки. Правило Рунге

Чисельні методи обчислення інтегралу дозволяють знайти лише наближене значення інтегралу з деякою точністю. Для більш точного обчислення інтегралу необхідно збільшити кількість проміжків інтегрування, зазвичай кількість проміжків інтегрування на кожному наступному кроці збільшується вдвічі.

Для визначення похибки ″ при обчисленні методом трапецій можна використовувати формулу:

$$|\varepsilon| \le \frac{(b-a)^3}{12n^2}M_2,$$

де

$$M_2 = \max_{x\in[a,b]} |f''(x)|.$$

Але визначення похибки за допомогою даної формули звичайно не виконують через велику кількість обчислень.

У загальному випадку для визначення похибки чисельних методів інтегрування використовують правило Рунге. Головна ідея полягає в обчисленні заданим методом з кроком *h* (отримаємо значення інтегралу *In*), а потім з кроком *h/2* (вдвічі більше кроків, отримаємо значення I_{2n}) і в подальшій оцінці різниці цих двох обчислень. Тому обчислення виконуються для 1*;* 2*;* 4*;* 8*; . . . ;* 2^N кроків. Похибку обчислення інтегралу при цьому можливо оцінити за формулою:

$$\Delta 2n \approx \Theta\,(I_{2n} - I_n)$$

де Θ = 1/3 для методів прямокутників і трапецій, Θ = 1/15 для методу парабол (Сімпсона).

Критерій зупинки. Збільшення кількості кроків виконується до тих пір, поки похибка обчислення не стане меншою або рівною заданій: $\varepsilon \le \Delta_{2n}$.

2.5. Паралельне представлення задачі

Для розпаралелювання задачі чисельного інтегрування можна використовувати метод розбиття задачі на менші підзадачі такого самого виду. Тобто скориставшись тим, що:

$$\int_a^b f(x)\,\mathrm{d}x = \underbrace{\int_a^{a_1} f(x)\,\mathrm{d}x}_{\text{задача 0}} + \underbrace{\int_{a_1}^{a_2} f(x)\,\mathrm{d}x}_{\text{задача 1}} + \cdots + \underbrace{\int_{a_n}^{b} f(x)\,\mathrm{d}x}_{\text{задача } P-1} = I_0 + I_1 + \ldots + I_{P-1}:$$

можна розпаралелити задачу інтегрування на P задач. При даній реалізації обмін даними необхідний лише для передачі вхідних даних від головної задачі до підлеглих і для передачі часткових результатів від підлеглих задач до головної.

В якості вхідних даних для кожної задачі будуть виступати кінці проміжку інтегрування та точність. Кожна задача буде обчислювати одне значення I_i.

2.6. Повний текст програми

2.6.1. Передача з використанням MPI_Isend, MPI_Irecv

Обчислення наближеного значення інтегралу функції $f(x) = x^3$ методом лівих прямокутників. Використовуються функції **MPI_Isend**, **MPI_Irecv**. Вхідні дані: початок, кінець проміжку інтегрування, точність знаходяться в файлі ***input.txt***. Результат буде записано в файл ***output.txt***.

Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_integrate/integrate.c

```
#include <stdio.h>
#include <math.h>
#include <stdbool.h>
#include <mpi.h>
/* Обчислення підінтегральної функції */
double function(double x)
{
/* x^3 */
return x*x*x;
}
/* Перевірка правила Рунге */
```

```
bool check_Runge(double I2, double I, double epsilon)
{
return (fabs(I2 - I) / 3.) < epsilon;
}
/* Інтегрування методом лівих прямокутників */
double integrate_left_rectangle(double start, double finish, double epsilon)
{
int num_iterations = 1; /* Початкова кількість ітерацій */
double last_res = 0.; /* Результат інтегрування на попередньому кроці */
double res = -1.; /* Поточний результат інтегрування */
double h = 0.; /* Ширина прямокутників */
while(!check_Runge(res, last_res, epsilon))
{
num_iterations *= 2;
last_res = res;
res = 0.;
h = (finish - start) / num_iterations;
for(int i = 0; i < num_iterations; i++)
{
res += function(start + i * h) * h;
}
}
return res;
}
/* Запис одного числа типу double в файл з ім'ям filename */
void write_double_to_file(const char* filename, double data)
{
/* Відкриття файлу на запис */
FILE *fp = fopen(filename, "w");
/* Перевірка правильності відкриття файлу */
if(fp == NULL)
{
printf("Failed to open the file\n");
/* Аварійне завершення всіх задач */
MPI_Abort(MPI_COMM_WORLD, 1);
}
/* Запис 1 числа типу double в файл */
fprintf(fp, "%lg\n", data);
/* Закриття файлу */
fclose(fp);
}
```

```
int main(int argc, char* argv[])
{
int np; /* кількість задач */
int rank; /* ранг цієї задачі */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Введення даних з файлу в масив з 3-х змінних.
* Відбувається у задачі 0.
* input[0] -- нижня межа інтегрування
* input[1] -- верхня межа інтегрування
* input[2] -- допустима абсолютна похибка */
double input[3];
if(rank == 0)
{
/* Відкриття файлу input.txt у режимі лише для читання */
FILE *fp = fopen("input.txt", "r");
/* Перевірка правильності відкриття файлу */
if(fp == NULL)
{
printf("Failed to open the file\n");
/* Аварійне завершення всіх задач */
MPI_Abort(MPI_COMM_WORLD, 1);
}
/* Зчитування 3 чисел типу double */
for(int i = 0; i < 3; i++)
fscanf(fp, "%lg", &input[i]);
/* Закриття файлу */
fclose(fp);
}
/* Передача введених даних від задачі 0 до всіх інших задач */
MPI_Bcast(input, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
double start = input[0];
double finish = input[1];
double epsilon = input[2];
double step = (finish - start) / np;
double res = integrate_left_rectangle(start + rank * step, start +
(rank + 1) * step, epsilon / np);
/* Передача проміжного результату інтегрування від усіх задач,
* окрім задачі 0, до задачі 0 */
if(rank != 0)
```

```
{
MPI_Send(&res, 1, MPI_DOUBLE, 0, rank, MPI_COMM_WORLD);
}
/* Прийом задачею 0 проміжних результатів інтегрування
* від усіх інших задач */
if(rank == 0)
{
MPI_Request recv_reqs[np - 1];
MPI_Status status[np - 1];
double resall[np - 1];
for(int i = 0; i < (np - 1); i++)
{
/* Прийом проміжного результату інтегрування без блокування виконан-
ня
* програми. Після виконання операції масив resall стане недоступним
* для використання */
MPI_Irecv(&resall[i], 1, MPI_DOUBLE, (i+1), (i+1), MPI_COMM_WORLD,
&recv_reqs[i]);
}
/* Очікування на завершення прийому усіх проміжних результатів
* інтегрування. Після виконання операції масив resall знову
* стане доступним для використання */
MPI_Waitall(np - 1, recv_reqs, status);
for(int i = 0; i < (np - 1); i++)
{
res += resall[i];
}
/* Виведення задачею 0 результату роботи програми у вихідний файл з
ім'ям
* output.txt */
write_double_to_file("output.txt", res);
}
MPI_Finalize();
return 0;
}
```

2.6.2. Передача з використанням MPI_Bcast, MPI_Reduce

Обчислення наближеного значення інтегралу функції $f(x) = x^3$ методом лівих прямокутників. Використовуються функції **MPI_Bcast**, **MPI_Reduce**.

Вхідні дані: початок, кінець проміжку інтегрування, точність знаходяться в файлі ***input.txt***. Результат буде записано в файл ***output.txt***.

Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_integrate/integrate2.c

```
#include <stdio.h>
#include <math.h>
#include <stdbool.h>
#include <mpi.h>
/* Обчислення функції */
double function(double x)
{
/* x^3 */
return x*x*x;
}
/* Перевірка правила Рунге */
bool check_Runge(double I2, double I, double epsilon)
{
return (fabs(I2 - I) / 3.) < epsilon;
}
/* Інтегрування методом лівих прямокутників */
double integrate_left_rectangle(double start, double finish, double epsilon)
{
int num_iterations = 1; /* Початкова кількість ітерацій */
double last_res = 0.; /* Результат інтугрування на попередньому кроці */
double res = -1.; /* Поточний результат інтегрування */
double h = 0.; /* Ширина прямокутників */
while (!check_Runge(res, last_res, epsilon))
{
num_iterations *= 2;
last_res = res;
res = 0.;
h = (finish - start) / num_iterations;
for (int i = 0; i < num_iterations; i++)
{
res += function(start + i * h) * h;
}
}
return res;
}
/* Запис одного числа типу double в файл з ім'ям filename */
void write_double_to_file(const char* filename, double data)
{
```

```
/* Відкриття файлу на запис */
FILE *fp = fopen(filename, "w");
/* Перевірка правильності відкриття файлу */
if (fp == NULL)
{
printf("Failed to open the file");
/* Завершення усіх процесів, що включені в комунікатор
MPI_COMM_WORLD */
MPI_Abort(MPI_COMM_WORLD, 1);
}
/* Запис 1 числа типу double в файл */
fprintf(fp, "%lg\n", data);
/* Закриття файлу */
fclose(fp);
}
int main(int argc, char* argv[])
{
int np;
int rank;
/* Ініціалізація MPI */
MPI_Init(&argc,&argv);
/* Отримати кількість процесів */
MPI_Comm_size(MPI_COMM_WORLD, &np);
/* Отримати індивідуальний ідентифікатор процесу */
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Введення даних з файлу в масив з 3ох змінних. Відбувається у
* процесі з ідентифікатором 0.
* 1 - нижня межа інтегрування
* 2 - верхня межа інтегрування
* 3 - допустима похибка */
double input[3];
if (rank == 0)
{
/* Відкриття файлу input.txt у режимі лише для читання */
FILE *fp = fopen("input.txt", "r");
/* Перевірка правильності відкриття файлу */
if (fp == NULL)
{
printf("Failed to open the file");
/* Завершення усіх процесів, що включені в комунікатор
MPI_COMM_WORLD */
```

```
MPI_Abort(MPI_COMM_WORLD, 1);
}
/* Зчитування 3 чисел типу double */
for (int i = 0; i < 3; i++)
fscanf(fp, "%lg", &input[i]);
/* Закриття файлу */
fclose(fp);
}
/* Передача введених даних від процесу 0 до всіх інших процесів, що
* включені у комунікатор MPI_COMM_WORLD */
MPI_Bcast(input, 3, MPI_DOUBLE, 0, MPI_COMM_WORLD);
double start = input[0];
double finish = input[1];
double epsilon = input[2];
double step = (finish - start) / np;
double res = integrate_left_rectangle(start + rank * step, start +
(rank + 1) * step, epsilon / np);
double result = res;
/* Передача проміжного результату інтегрування від усіх процесів до
процесу 0
* та збереження суми проміжних результатів у змінній result */
MPI_Reduce (&res, &result, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
/* Виведення процесом 0 результату роботи програми у вихідний файл з
ім'ям
* output.txt */
if (rank == 0)
write_double_to_file("output.txt", result);
/* Закінчити роботу з MPI */
MPI_Finalize();
return 0;
}
```

2.7. Варіанти

Варіант визначається по табл. 2.2.

Табл. 2.2. Варіанти завдання

№	Метод інтегр.	Метод передач	Функція	Межі	Точність
1	Лівих прям.	MPI_Isend, MPI_Irecv	$f(x) = x^{x+3}$	[0; 3.5]	0.0001
2	Лівих прям.	MPI_Bcast, MPI_Reduce	$f(x) = 4^{2x+3}$	[-3; 2]	0.001
3	Правих прям.	MPI_Isend, MPI_Irecv	$f(x) = \log_2 x^3$	[5; 1500]	0.0001
4	Правих прям.	MPI_Bcast, MPI_Reduce	$f(x) = x \lvert x \rvert \sin x$	[0; 100]	0.001
5	Середніх прям.	MPI_Isend, MPI_Irecv	$f(x) = \log_{10} x \lvert \sin x \rvert$	[1; 250]	0.00001
6	Середніх прям.	MPI_Bcast, MPI_Reduce	$f(x) = x^3 \cos x$	[0; 15]	0.0001
7	Трапецій	MPI_Isend, MPI_Irecv	$f(x) = 2^x \sin x$	[-3; 10]	0.0001
8	Трапецій	MPI_Bcast, MPI_Reduce	$f(x) = \lvert x \rvert^{\lvert x \rvert}$	[0; 5]	0.0001
9	Парабол	MPI_Isend, MPI_Irecv	$f(x) = 5x - \sin^2 x$	[0; 100]	0.0001
10	Парабол	MPI_Bcast, MPI_Reduce	$f(x) = \sin x / x^3$	[0.03; 10]	0.00001
11	Лівих прям.	MPI_Isend, MPI_Irecv	$f(x) = 5x - \sin^2 x$	[0; 100]	0.0001
12	Лівих прям.	MPI_Bcast, MPI_Reduce	$f(x) = \sin x / x^3$	[0.03; 10]	0.00001
13	Правих прям.	MPI_Isend, MPI_Irecv	$f(x) = x^{x+3}$	[0; 3.5]	0.0001
14	Правих прям.	MPI_Bcast, MPI_Reduce	$f(x) = 4^{2x+3}$	[-3; 2]	0.001
15	Середніх прям.	MPI_Isend, MPI_Irecv	$f(x) = \log_2 x^3$	[5; 1500]	0.000001
16	Середніх прям.	MPI_Bcast, MPI_Reduce	$f(x) = x \lvert x \rvert \sin x$	[0; 100]	0.00001
17	Трапецій	MPI_Isend, MPI_Irecv	$f(x) = \log_{10} x \lvert \sin x \rvert$	[1; 250]	0.00001
18	Трапецій	MPI_Bcast, MPI_Reduce	$f(x) = x^3 \cos x$	[0; 15]	0.0001
19	Парабол	MPI_Isend, MPI_Irecv	$f(x) = 2^x \sin x$	[-3; 10]	0.0001
20	Парабол	MPI_Bcast, MPI_Reduce	$f(x) = \lvert x \rvert^{\lvert x \rvert}$	[0; 5]	0.0001
21	Лівих прям.	MPI_Isend, MPI_Irecv	$f(x) = 2^x \sin x$	[-3; 10]	0.0001
22	Лівих прям.	MPI_Bcast, MPI_Reduce	$f(x) = \lvert x \rvert^{\lvert x \rvert}$	[0; 5]	0.0001
23	Правих прям.	MPI_Isend, MPI_Irecv	$f(x) = 5x - \sin^2 x$	[0; 100]	0.0001
24	Правих прям.	MPI_Bcast, MPI_Reduce	$f(x) = \sin x / x^3$	[0.03; 10]	0.00001
25	Середніх прям.	MPI_Isend, MPI_Irecv	$f(x) = x^{x+3}$	[0; 3.5]	0.0001
26	Середніх прям.	MPI_Bcast, MPI_Reduce	$f(x) = 4^{2x+3}$	[-3; 2]	0.001
27	Трапецій	MPI_Isend, MPI_Irecv	$f(x) = \log_2 x^3$	[5; 150]	0.0001
28	Трапецій	MPI_Bcast, MPI_Reduce	$f(x) = x \lvert x \rvert \sin x$	[0; 28]	0.0001
29	Парабол	MPI_Isend, MPI_Irecv	$f(x) = \log_{10} x \lvert \sin x \rvert$	[1; 250]	0.00001
30	Парабол	MPI_Bcast, MPI_Reduce	$f(x) = x^3 \cos x$	[0; 15]	0.0001

2.8. Питання для самоконтролю

1. Що таке неблокуюча передача даних?
2. Які переваги і недоліки є у неблокуючих передач?
3. Чи можна за допомогою блокуючої функції **MPI_Recv** прийняти повідомлення, яке було відправлене за допомогою функції неблокуючої відправки **MPI_Isend**?
4. Що таке передача один до багатьох?
5. Що таке операція згортки? Яка функція MPI призначена для її виконання?
6. Які існують стандартні операції згортки в MPI?

2.3 Лабораторна робота 3
Обмін даними за участю декількох задач в MPI

Мета роботи:

- вивчити види передач MPI «один до багатьох» та «багато до багатьох» та їх різновиди;
- навчитися аналізувати алгоритми паралельних програм щодо необхідних типів передачі даних між паралельно виконуваними частинами.

Завдання: розробити паралельну програму для розв'язання СЛАР методом Якобі або з застосуванням LU-розкладу.

3.1. Типи пересилок в MPI

Більшості паралельних програм необхідно не лише обмінюватися даними між парами задач, а й виконувати більш складні види обміну, в яких бере участь більше, ніж дві задачі. Такі види обміну в MPI називаються колективними. В колективних взаємодіях беруть участь всі задачі, що входять до деякого комунікатору.

Такі види обміну в цілому можна розділити на три умовні групи:

1. одна задача відправляє дані декільком; (one-to-many);
2. одна задача отримує дані від декількох інших (many-to-one);
3. декілька задач обмінюються даними між собою (many-to-many).

Так, вже розглянута функція **MPI_Bcast** належить до першої групи.

Як і у попередніх роботах, завжди будемо використовувати комунікатор **MPI_COMM_WORLD** і тому для всіх функцій «декілька» задач буде означати всі запущені задачі.

В даній роботі розглянуті основні функції всіх типів:

1. one-to-many: **MPI_Scatter**;
2. many-to-one: **MPI_Gather**, **MPI_Reduce**;
3. many-to-many: **MPI_Allgather**, **MPI_Allreduce**, **MPI_Reduce_scatter**, **MPI_Alltoall**.

Цей перелік функцій колективного обміну не є вичерпним: в MPI визначені інші функції обміну, але вони розширеними варіантами вищевказаних, наприклад, виконують розбиття даних на блоки різного розміру під час розсилки.

Розглянемо також два основних методи організації взаємодії задач на основі пересилання повідомлень. Перший з них називається одноранговим (англ. **peer to peer**). Це не означає, що задачі мають один ранг в середовищі MPI. Задачі є «рівноправними» між собою, тобто не мають централізованого управління з боку деякої задачі. Протилежний підхід називається майстер-робітник (англ. **master/slave**) та вимагає створення окремої задачі

з особливою логікою роботи, яка є «головною» (англ. **master**) або диспетчеризуючою (англ. **dispatch**). Ця задача виконує централізоване управління обміном даних між іншими. Наприклад, якщо задачам-робітникам необхідно обмінятися даними, то вони всі дані відправляють головній задачі, після чого вона розсилає згруповані дані між ними.

MPI_Scatter

Розсилає всім задачам окремі незалежні частини однакового розміру вихідного буфера.

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

- ***sendbuf*** − покажчик буфера відправки; розмір буфера має бути не менший за
- ***sendcnt*** · P елементів, де P − кількість задач.
- ***sendcnt*** − кількість елементів, що будуть відправлені кожній з задач.
- ***sendtype*** − тип елементів, що відправляються.
- ***recvbuf*** − покажчик буферу прийому; розмір буферу має бути не менше, ніж ***recvcnt*** елементів. Задача ***root***, що відправляє дані, також має вказати буфер прийому: в нього буде скопійована частина даних, яка виділена для даної задачі.
- ***recvcnt*** − кількість елементів, що будуть отримані задачею; має збігатися з кількістю відправлених елементів.
- ***recvtype*** − тип елементів, що отримуються.
- ***root*** − ранг задачі, що відправляє дані.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.

Для виконання розсилки даних функцію має викликати кожна задача з однаковими значеннями ***root*** та ***comm*** та відповідними значеннями ***recvcnt*** та ***recvtype*** (розмір даних, що передаються, має збігатися). Аргументи ***sendbuf***, ***sendcnt***, ***sendtype*** є необхідними тільки для відправляючої дані задачі.

У випадках, коли аргументи виклику не є значущими, можна встановити їх наступним чином. Покажчик на буфер як **NULL**, розмірність 0, тип даних як спеціальну константу **MPI_DATATYPE_NULL**. Деякі реалізації MPI можуть виконувати перевірку коректності розмірності (більше нуля) і типів (є зареєстрованим типом в середовищі MPI) навіть тих аргументів, які для даної задачі ігноруються.

Приклад 5. Необхідно додати дві матриці. Значення кожного елемента результату не залежить від інших результатів, тому можна будь-яким чином розбити матриці на частини та в кожній задачі виконувати додавання

цих частин. Нехай матриця оголошена як одномірний масив **int MA[N*N]**, причому **N = k · P**, та була зчитана в задачі 0. Тоді можемо легко розіслати всім задачам по k рядків матриці в буфер **int MAh[k*N]** наступним викликом:

MPI_Scatter(MA, k*N, MPI_INT, MAh, k*N, MPI_INT, 0, MPI_COMM_WORLD);

Але створювати об'єкт великого розміру в кожній задачі недоцільно, тому, як правило, матриця **MA** існуватиме лише в задачі 0. Тоді в ньому необхідно виконати такий самий виклик:

MPI_Scatter(MA, k*N, MPI_INT, MAh, k*N, MPI_INT, 0, MPI_COMM_WORLD);

а в усіх інших −

MPI_Scatter(NULL, 0, MPI_DATATYPE_NULL, MAh, k*N, MPI_INT, 0, MPI_COMM_WORLD);

Передача даних відбудеться, тому що перші три аргументи в задачах, що приймають дані ігноруються, а кількість та тип отримуваних елементів, номер задачі-відправника та комунікатор збігаються.

Задача P_1	Задача P_2	Задача P_3
1 2 3 4 5 6 7 8 9	−	−
[1 2 3]	[4 5 6]	[7 8 9]

MPI_Gather

Отримує дані з усіх задач та об'єднує їх в один блок відповідно до рангів задач.

MPI_Gather(**void** *sendbuf, **int** sendcnt, **MPI_Datatype** sendtype, **void** *recvbuf, **int** recvcnt, **MPI_Datatype recvtype, int root, MPI_Comm comm);**

Призначення параметрів та умови виконання обміну такі ж самі, як і для **MPI_Scatter**. Параметри ***recvbuf***, ***recvcnt***, ***recvtype*** мають значення тільки для задачі, що отримує дані. Буфер відправки в кожній задачі має бути розміром не менше за sendcnt, буфер прийому в приймаючій задачі не менше за ***recvcnt*** **· P**, де **P** − кількість задач.

Функція може використовувати один і той самий буфер як буфер відправки та прийому у відправляючій задачі. Для цього необхідно аргумент ***sendbuf*** встановити рівним константі **MPI_IN_PLACE**. При цьому аргументи ***sendcnt*** та ***sendtype*** ігноруються. Бажано дотримуватися попередніх рекомендацій щодо встановлення значень ігнорованих аргументів.

Приклад 6. Після додавання матриць у попередньому прикладі необхідно вивести результат в задачі 2. Для цього в ньому виділено буфер **int MR[N*N]**. Дана задача має зробити виклик:
MPI_Gather(MAh, k*N, MPI_INT, MR, k*N, MPI_INT, 2, MPI_COMM_WORLD);
а всі інші задачі –
MPI_Gather(MAh, k*N, MPI_INT, NULL, 0, MPI_INT, 2, MPI_COMM_WORLD);
Так як вказано однакову кількість та тип елементів, що передаються, однаковий ранг задачі, що збирає дані, та комунікатор, то такий виклик є коректним та передача відбудеться.

Задача P_1	Задача P_2	Задача P_3
[1 2 3]	[4 5 6]	[7 8 9]
1 2 3 4 5 6 7 8 9	–	–

MPI_Allgather
Отримує дані з усіх задач, об'єднує їх в один блок відповідно до рангів та розсилає об'єднаний блок в усі задачі.
MPI_Allgather(**void** *sendbuf, **int** sendcnt, **MPI_Datatype** sendtype, **void** *recvbuf, **int** recvcnt, **MPI_Datatype** recvtype, **MPI_Comm** comm);
Призначення параметрів таке ж саме, як і в функції **MPI_Scatter**. Кожна задача має виділити буфер для прийому всього блоку даних. Обмін відбудеться якщо всі задачі в одному комунікаторі викличуть функцію та вкажуть однаковий розмір даних. Фактично, ця функція реалізує обмін багато до багатьох, тому що кожна окрема задача розсилає свою частину даних усім іншим задачам.
Ця функція використовується при одноранговій організації обміну. У випадку організації «майстер-робітник» замість **MPI_Allgather** виконується спочатку операція **MPI_Gather**, яка збирає всі дані в головну задачу, після чого вона за допомогою широкомовної передачі **MPI_Bcast** розсилає об'єднані дані всім іншим задачам.
Якщо необхідно отримати дані в той самий буфер, що використовується при відправці, то можна скористатися константою **MPI_IN_PLACE** аналогічно до функції **MPI_Gather**. В даному випадку дані, що відсилаються, мають бути розташовані в тій частині буфера прийому, в якій вони б опинилися після прийому за звичайних умов. Тобто якщо відбувається збір

матриці за рядками, то для **i**-тої задачі дані мають зберігатися в **i**-му рядку матриці, яка є буфером прийому.

Приклад 7. Нехай після додавання матриць необхідно не вивести результат, а помножити на нього іншу матрицю MK. Найпростіший алгоритм паралельного множення матриць вимагає для отримання k рядків результату в кожній задачі **k** рядків матриці, що є першим операндом, та повну матрицю, що є другим операндом.
В такому разі кожній з задач необхідна ціла матриця MR, але вона має лише її частину. Для цього всі задачі виділяють буфер **int MR[N*N]** та виконують виклик:
MPI_Allgather(MAh, k*N, MPI_INT, MR, k*N, MPI_INT, MPI_COMM_WORLD);
Якщо необхідно організувати аналогічні дії через головну задачу з рангом 1, то в ній необхідно викликати:
MPI_Gather(MAh, k*N, MPI_INT, MR, k*N, MPI_INT, 1, MPI_COMM_WORLD);
а в усіх інших −
MPI_Gather(MAh, k*N, MPI_INT, NULL, 0, MPI_INT, 1, MPI_COMM_WORLD);
після чого виконати розсилку за допомогою:
MPI_Bcast(MR, N*N, MPI_INT, 1 MPI_COMM_WORLD);

Задача P_1	Задача P_2	Задача P_3
[1 2 3]	[4 5 6]	[7 8 9]
1 2 3	1 2 3	1 2 3
4 5 6	4 5 6	4 5 6
7 8 9	7 8 9	7 8 9

MPI_Reduce
Виконує згортку (тобто операцію зведення багатьох значень до одного) поелементно над буферами з усіх задач, результат зберігається в буфері вказаної задачі.
MPI_Reduce(**void** *sendbuf, **void** *recvbuf, **int** count, **MPI_Datatype** datatype, MPI_Op op, **int** root, **int** comm);

- ***sendbuf*** − покажчик буферу відправки;
- ***recvbuf*** − покажчик буферу прийому;
- ***count*** − розмірність буферів відправки та прийому; згортка буде проводитись над елементами з однаковими індексами в кожному буфері, її результат буде розміщений в буфері прийому з тим же індексом;
- ***datatype*** − тип даних, що передаються;

- ***op*** − операція, за якою відбувається згортка (табл. 3.1);
- ***root*** − задача, що приймає результат редукції;
- ***comm*** − комунікатор, в рамках якого виконується взаємодія.

Аргумент ***recvbuf*** є значущим тільки для задачі, яка приймає результат, та має бути виділений розміром не менше count перед викликом функції. Для виконання згортки необхідно, щоб всі задачі в одному комунікаторі викликали функцію **MPI_Reduce** з однією приймаючою задачею. Обмін може бути невдалим, якщо розмір даних, що передаються, не збігається з очікуваним розміром при прийомі.

Використання константи **MPI_IN_PLACE** замість буфера відправки повідомлює MPI про те, що необхідні дані знаходяться в буфері прийому головної задачі та мають бути замінені на результат після завершення його обчислення.

В MPI вбудовані найбільш поширені операцій згортки над стандартними типами даних (табл. 3.1).

Користувач може визначити власні функції згортки, які відповідають наступному прототипу:

MPI_User_function(**void** *a, **void** *b, **int** *len, **MPI_Datatype** *datatype);

Причому функція, яка має виконувати операцію #, працює за наступним шаблоном:

Табл. 3.1. Вбудовані операції згортки в MPI

Константа	**Операція**	**Типи даних**
MPI_MAX	Максимум	int, float
MPI_MIN	Мінімум	int, float
MPI_SUM	Сума	int, float
MPI_PROD	Добуток	int, float
MPI_LAND	Логічне «І»	int
MPI_BAND	Побітове «І»	int
MPI_LOR	Логічне «АБО»	int
MPI_BOR	Побітове «АБО»	int
MPI_LXOR	Логічне виключне «АБО»	int
MPI_BXOR	Побітове виключне «АБО»	int

```
for(int i = 0; i < *len; i++)
{
b[i] = b[i] # a[i];
}
```

Таким чином буфер **a** є операндом, а буфер **b** − накопичувачем результату. Операція, яка реалізується, має бути асоціативною, тобто допускати групування послідовно виконуваних операцій (наприклад множення). Оскільки більшість загальновживаних асоціативних операцій вже визна-

чені в MPI, то визначені користувачем функції згортки здебільшого використовуються для типів, що також визначені користувачем. Отримати дескриптор користувацької функції згортки в системі MPI можливо за допомогою виклику:
MPI_Op_create(MPI_User_function *func, bool commute, MPI_Op *op);

- ***func*** − покажчик на функцію користувача, якою буде виконуватися згортка;
- ***commute*** − прапорець комутативності функції; якщо встановлений, то функція вважається комутативною; якщо ні операнди будуть розміщені за зростанням рангу задачі, з якої вони були отримані;
- ***op*** − покажчик на змінну, в якій буде збережено дескриптор даної операції.

Приклад 8. Нехай в кожній задачі є матриця **int MA[N*N]**. Задача 0 має вивести суму всіх матриць. Для цього вона виділяє буфер int MR[N*N] та викликає функцію згортки:
*MPI_Reduce(MA, MR, N*N, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);*
Всі інші задачі також викликають цю функцію, але не вказують буфер прийому:
MPI_Reduce(MA, NULL, N*N, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

Задача P_1	Задача P_2	Задача P_3
1 2 3 4	5 6 7 8	9 10 11 12
15 18 21 24	_	_

MPI_Allreduce
Виконує згортку поелементно над буферами усіх задач та розсилає результат усім задачам.
MPI_Allreduce(**void** *sendbuf, **void** *recvbuf, **int** count, **MPI_Datatype** datatype, MPI_op op, **MPI_Comm** comm);
Призначення аргументів таке ж саме, як і для **MPI_Reduce**. Кожна задача має виділити буфер прийому розміром не менше ***count***. Ця функція, як правило, використовується при одноранговому обміні. Аналогічно до **MPI_Reduce** існує можливість використати один і той самий буфер для прийому та відправки в кожній з задач із використанням замість покажчика буфера відправки константи **MPI_IN_PLACE**.

При організації взаємодії через головний задачу даний виклик аналогічний до послідовного виклику **MPI_Reduce** та запису результату у головній задач та **MPI_Bcast**, який розсилає результат з головної в усі інші задачі.

Приклад 9. Скористаємося умовами попереднього прикладу, але змінимо їх таким чином, що результат додавання матриць є необхідним для кожної задачі. Тоді кожна задача виділяє буфер прийому int **MR[N*N]** та викликає функцію:
MPI_Allreduce(MA, MR, N*N, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

Задача P_1	Задача P_2	Задача P_3
1 2	5 6	9 10
3 4	7 8	11 12
15 18	15 18	15 18
21 24	21 24	21 24

MPI_Reduce_scatter
Виконує згортку поелементно над векторами з усіх задач, розбиває результат на незалежні частини різного розміру та розсилає їх всім задачам.
MPI_Reduce_scatter(**void** *sendbuf, **void** *recvbuf, **int** *recvcnts, **MPI_Datatype** datatype, MPI_Op op, **MPI_Comm** comm);

- ***sendbuf*** − покажчик буферу відправки;
- ***recvbuf*** − покажчик буферу прийому;
- ***recvcnts*** − масив кількостей елементів, які необхідно отримати кожній задачі (впорядкований за рангами задач);
- ***datatype*** − тип даних, які передаються;
- ***op*** − операція, за якою виконується згортка;
- ***comm*** − комунікатор, в межах якого відбувається взаємодія.

Для проведення обміну всі задачі мають викликати функцію з однаковою операцією та комунікатором. Для вдалого обміну розміри даних мають збігатися. Кожна задача має виділити буфер на кількість елементів не меншу, ніж визначена відповідним елементом масиву recvcnts. Відмітимо, що ця функція має більше можливостей, ніж централізований збір за допомогою MPI_Reduce та наступна за ним розсилка через **MPI_Scatter**, тому що дозволяє виконувати розбиття на частині різних розмірів.
Приклад 10. Нехай в кожній задачі є матриця **int MAi[N*N]** та однакова для всіх задач матриця MB. Необхідно обчислити добуток суми матриць MAi на матрицю MB.

При найбільш простому алгоритмі паралельного обчислення добутку матриць кожній задачі необхідно мати цілу матрицю MB та частину рядків матриці MR для отримання частини рядків матриці-результату. Тому необхідно спочатку виконати поелементне додавання матриць MA усіх задач, а після цього розіслати рядки MR, що не перекриваються, усім задачам. Щоб продемонструвати можливості даної функції приймемо, що маємо np = 3 запущених задач (по одній задачі на процесорі), причому процесор, на якому запущена задача з рангом 1, має вдвічі більшу продуктивність. Тоді доцільно більш продуктивним процесорам відіслати більшу кількість рядків. Всі задачі мають виконати наступні дії:

```
int recvcnts[] = { N/4, N/2, N/4 }; // всього N елементів
int alloc_size;
if(rank == 0 || rank == 2) alloc_size = N/4;
else if(rank == 1) alloc_size = N/2;
int *MP = malloc(alloc_size * sizeof(int));
MPI_Reduce_scatter(MA, MP, recvcnts, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

MPI_Alltoall

В кожній з викликаючих задач розбиває блок даних на рівні частини та розсилає їх іншим задачам.

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

- ***sendbuf*** − покажчик буфера відправки; розмір буфера має бути не менший, ніж **sendcnt · P** елементів, де **P** − кількість задач.
- ***sendcnt*** − кількість елементів, що будуть відправлені кожній із задач.

Задача P_1	Задача P_2	Задача P_3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
3 6 9 12	15 18 21 24 27 30 33 36	39 42 45 48

- ***sendtype*** − тип елементів, що відправляються.
- ***recvbuf*** − покажчик буферу прийому; розмір буферу має бути не менший, ніж **recvcnt · P** елементів. Задача ***root***, що відправляє да-

ні, також має вказати буфер прийому: в нього буде скопійована частина даних, яка виділена для даної задачі.

- ***recvcnt*** − кількість елементів, що будуть отримані від кожної задачі; має збігатися з кількістю відправлених елементів.
- ***recvtype*** − тип елементів, що отримуються.
- ***comm*** − комунікатор, в рамках якого відбувається взаємодія.

Всі буфери мають бути виділені та мати розмір необхідний розмір для виконання передачі. Безпосередньо обмін даними відбувається, коли всі задачі викличуть функцію з однаковим розміром передач, що відправляються та приймаються, типом елементів та комунікатором. Ця функція аналогічна до послідовного виклику **MPI_Scatter** в кожній задачі.
Дана функція також дозволяє використовувати той самий буфер для прийому і відправки даних через механізм **MPI_IN_PLACE.**

Приклад 11. Нехай в пам'яті кожної задачі зберігається один рядок матриці. Необхідно транспонувати матрицю, але залишити зберігання за рядками. Для того, щоб виконати транспонування, необхідно рядки зробити стовпцями матриці. Програмно це значить, що для будь-якого рядка **i**-тий елемент необхідно передати в **i**-ту задачу. Якщо розбити рядки по одному елементу та виконати **MPI_Alltoall**, то матрицю буде транспоновано.
MPI_Alltoall(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, MA, 1, MPI_INT, MPI_COMM_WORL D);

Задача P_1	Задача P_2	Задача P_3
[1 2 3]	[4 5 6]	[7 8 9]
[1 4 7]	[4 5 6]	[4 6 9]

3.2. Математичний апарат

3.2.1. Метод Якобі

Метод Якобі − ітеративний метод для розв'язання систем лінійних алгебраїчних рівнянь. СЛАР наступного виду:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\cdot \quad \cdot \quad \cdot \quad \cdot$$
$$\cdot \quad \cdot \quad \cdot \quad \cdot$$
$$\cdot \quad \cdot \quad \cdot \quad \cdot$$
$$a_{m1}x_1 + a_{m2}x2 + \ldots + a_{mn}x_n = b_m$$

можна записати у вигляді матричного рівняння:

$$A\mathbf{x} = \mathbf{b}; \qquad (3.1)$$

де:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

Матрицю **A** можна представити у вигляді суми двох матриць **D** та **R**, де **D** − діагональна матриця, а R містить нулі на головній діагоналі:

$A = D + R$;

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

Тоді рівняння (3.1) можна переписати так:

(D + R)x = b;
Dx + Rx = b;
Dx = b - Rx:

Із відомого наближення розв'язку **x**(k) можна отримати більш точне наближення $\mathbf{x}^{(k+1)}$

$$\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)}):$$

(3.2)

В якості початкового наближення $\mathbf{x}^{(0)}$ можна взяти одиничний вектор:

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Нагадаємо, що обернену матрицю для діагональної D можна знайти тривіально:

$$D^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{22}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{a_{nn}} \end{bmatrix}$$

Не важко впевнитись, що $D^{-1}D = I$.

Критерій зупинки ітерації. Звичайно для оцінки точності розв'язку обчислюють значення норми вектору нев'язки:

$\| \mathbf{r}^{(k)} \| = \| \mathbf{b} - A\mathbf{x}^{(k)} \|$;

а в якості норми використовують евклідову норму:

$$\|\mathbf{v}\| = \sqrt{v_1^2 + \cdots + v_n^2}.$$

Абсолютне значення норми вектору нев'язки відображає похибку наближення розв'язку. Але краще використовувати відносне значення, так як для матриць з малими коефіцієнтами навіть одиничний вектор в якості розв'язку може дати малу $\| \mathbf{r}^{(k)} \|$ (але звичайно одиничний вектор не є розв'язком), а для матриць з великими коефіцієнтами в деяких випадках неможливо отримати мале значення $\| \mathbf{r}^{(k)} \|$ через обмежену розрядну сітку машини.

Тому відношення норми вектору нев'язки до норми вектору **b** можна інтерпретувати як точність розв'язку. Тоді критерій зупинки ітерації:

$$\frac{\left\|\mathbf{r}^{(k)}\right\|}{\|\mathbf{b}\|} = \frac{\left\|\mathbf{b} - A\mathbf{x}^{(k)}\right\|}{\|\mathbf{b}\|} < \varepsilon,$$

. 3)

де ε − задана точність.

Збіжність. Послідовність наближених значень розв'язків $\mathbf{x}^{(k)}$ є збіжною якщо матриця A має домінуючу головну діагональ, тобто:

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}| .$$

Для деяких інших матриць метод Якобі також є збіжним.

3.2.2. LU-розклад

LU-розкладом (англ. LU-decomposition) прямокутної матриці A називається представлення матриці у вигляді добутку $A = L \cdot U$, де L − нижня трикутна матриця з головною діагоналлю з одиниць, U − верхня трикутна матриця.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

3.2. Математичний апарат

Алгоритм Дулітла для знаходження LU-розкладу. Зведення матриці до верхньотрикутного виду можливе аналогічно алгоритму Гауса. Фактично, необхідно занулити всі елементи матриці, для яких i > j, де i − номер рядка, j – номер стовпця. Існують перетворення матриці, які не змінюють ранг та визначник матриці. До них належить операція алгебраїчної суми

рядків з деяким коефіцієнтом. Таким чином, щоб занулити всі потрібні елементи першого стовпця необхідно з усіх рядків окрім першого відняти перший, помножений на відповідний коефіцієнт. Коефіцієнт може бути легко визначений з формули $a_{ii} - k \cdot a_{1i} = 0$:

$$A^{(1)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ \frac{a_{21}}{a_{11}} \cdot a_{11} & \frac{a_{21}}{a_{11}} \cdot a_{12} & \frac{a_{21}}{a_{11}} \cdot a_{13} \\ \frac{a_{31}}{a_{11}} \cdot a_{11} & \frac{a_{31}}{a_{11}} \cdot a_{12} & \frac{a_{31}}{a_{11}} \cdot a_{13} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \\ 0 & a_{22} - \frac{a_{21}}{a_{11}} \cdot a_{12} & \\ 0 & a_{32} - \frac{a_{31}}{a_{11}} \cdot a_{12} & \end{bmatrix}$$

Збережемо використані коефіцієнти в матриці L:

$$L = \begin{bmatrix} x & x & x \\ \frac{a_{21}}{a_{11}} & x & x \\ \frac{a_{31}}{a_{11}} & x & x \end{bmatrix}$$

Аналогічно можна занулити елементи другого стовпця, для яких i > j, віднімаючи з усіх рядків другий, помножений на відповідний коефіцієнт:

$$A^{(2)} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} - \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{a'_{32}}{a'_{22}} \cdot 0 & \frac{a'_{32}}{a'_{22}} \cdot a'_{22} & \frac{a'_{32}}{a'_{22}} \cdot a'_{23} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \\ 0 & a'_{22} & \\ 0 & 0 & a'_{33} - \end{bmatrix}$$

$$L = \begin{bmatrix} x & x & x \\ \frac{a_{21}}{a_{11}} & x & x \\ \frac{a_{31}}{a_{11}} & \frac{a'_{32}}{a'_{22}} & x \end{bmatrix}$$

Процес необхідно продовжувати доти, поки всі елементи нижче головної діагоналі не стануть рівними нулю. Виконуючи такі перетворення послідовно при занулянні j-го стовпця немає необхідності виконувати дії над елементами стовпців 1 : : : (j - 1). Також при цьому не будуть змінюватися рядки 1 . . . j. Таким чином можна побудувати наступний ітеративний алгоритм. На k-му кроці обираємо ведучій елемент a_{kk}. Для кожного рядка i > k знаходимо коефіцієнт $l_{ik} = a_{ik}/ a_{kk}$ та записуємо його у матрицю L, після чого з рядка i матриці A віднімаємо поелементно рядок k помножений на

l_{ik}. Зверніть увагу, що на кожному кроці алгоритм використовує матрицю A, що була модифікована в процесі всіх попередніх кроків. Ітерація закінчується, після k = n □ 1 кроків, де n − розмірність матриці. Після завершення ітерації діагональні елементи матриці L встановлюються рівними 1, а всі інші невстановлені − рівними 0. Таким чином маємо нижню трикутну матрицю L та верхню трикутну матрицю A(n) = U. Перевірити, що їх добуток дорівнює вихідній матриці A(0) можна виконавши безпосереднє множення.

Формально алгоритм можна записати системою рівнянь для ітераційного процесу: □

$$\begin{cases} l_{ik} = 0, & \text{при } i = \overline{(1, k-1)}; \\ l_{kk} = 1; & \\ l_{ik} = \dfrac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, & \text{при } i = \overline{(k+1, n)}; \\ a_{ij}^{(k+1)} = a_{ij}^{(k)}, & \text{при } i = \overline{(1, k)} \quad j = \overline{(1, n)}; \\ a_{ij}^{(k+1)} = 0, & \text{при } i = \overline{(k+1, n)} \quad j = \overline{(1, k)}; \\ a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} \cdot a_{kj}^{(k)}, & \text{при } i = \overline{(k+1, n)} \quad j = \overline{(k+1, n)}. \end{cases}$$

Де також необхідно завершити обчислення L матриці у додатковому кроці за формулами {

$$\begin{cases} l_{in} = 0, & \text{при } i = \overline{(1, n-1)}; \\ l_{nn} = 1; & \end{cases}$$

(3.5)

Існування та єдиність розкладу. LU-розклад існує тоді і тільки тоді, коли всі ведучі головні доповнювальні мінори матриці не дорівнюють нулю. Нагадаємо, що доповнювальним мінором M^k k-го порядку матриці A називається визначник квадратної підматриці матриці A розмірністю k, отриманої вилученням з матриці одного або декількох рядків та стовпців. Головним мінором називається такий мінор, в якому номери вилучених рядків та стовпців збігаються. Ведучім називається мінор, якому відповідає прямокутна верхня ліва підматриця даної матриці A. Найпростішою перевіркою існування розкладу є порівняння на k-тому кроці ітерації Дулітла значення a_{kk} на рівність нулю. Якщо дорівнює, то LU-розклад даної матриці не існує. Розклад є єдиним, якщо головна діагональ матриці L складається з одиниць. Таким чином якщо LU-розклад існує, то він є єди-

ним. Також можна довести, що якщо розклад існує та визначник матриці не дорівнює нулю, то СЛАР з такими коефіцієнтами має єдиний розв'язок.

Розв'язання СЛАР за допомогою LU-розкладу. Підставимо формулу розкладу у (3.1)

A**x** = LU**x** = **b**:

Якщо позначити U**x** = **y**, то від розв'язку СЛАР загального вигляду можна перейти до розв'язку двох СЛАР трикутного вигляду

$$\mathbf{Ly} = \mathbf{b};$$
$$\mathbf{Ux} = \mathbf{y};$$

де спочатку необхідно розв'язати перше рівняння відносно **y**, а потім друге відносно **x**. Розв'язування трикутних СЛАР є тривіальним та зводиться до прямої та зворотньої підстановки подібно методу Гауса. Для нижньої трикутної матриці L визначити **y** можливо за формулою

$$y_i = \frac{1}{l_{ii}} \left(b_i - \sum_{j=1}^{i-1} y_j l_{ij} \right) = b_i - \sum_{j=1}^{i-1} y_j l_{ij},$$

а для верхньої трикутної матриці U за формулою

$$x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{j=i+1}^{n} x_j u_{ij} \right),$$

при чому для визначення $\mathbf{y_i}$ підстановку потрібно починати з першого рівняння, а для визначення $\mathbf{x_i}$ – з останнього.

Фактично виконання LU-розкладу та розв'язування двох СЛАР, що описуються трикутними матрицями, є тими ж самими діями, що й при використанні методу Гауса для розв'язку СЛАР. Єдиною відмінністю є те, що у методі Гауса необхідно відразу оперувати з матрицею коефіцієнтів A та вектор-стовпцем вільних членів **b** для отримання верхньої трикутної матриці. А при застосуванні LU-розкладу обчислюється верхня трикутна матриця U, та матриця L, яка дозволить перетворити значення **b**. Коефіцієнти матриці L є множниками рядків, які використовувалися для зануляння певних елементів при відніманні, тобто якщо провести ті самі дії над вектором **b**, він буде модифікований як і у методі Гауса. Таким чином застосування LU-розкладу дозволяє розв'язувати СЛАР з однаковими коефіцієнтами та різними векторами вільних членів без перетворень. Така властивість корисна, наприклад, при обробці результатів повторень одного експерименту.

Обчислення визначника. LU-розклад може бути застосований до обчислення визначника матриці. За правилами знаходження визначника:

$$\det A = \det(L \cdot U) = (\det L) \cdot (\det U). \quad (3.6)$$

Для будь-якої трикутної матриці визначник дорівнює добутку елементів головної діагоналі (дане твердження можна довести, застосувавши будь-який метод обчислення визначника), тому вираз (3.6) можна переписати у вигляді:

$$\det A = (\det L) \cdot (\det U) = \prod_{i=1}^{n} l_{ii} \cdot \prod_{i=1}^{n} u_{ii} = \prod_{i=1}^{n} u_{ii} \quad (3.7)$$

3.3. Аналіз задачі з точки зору програмування

3.3.1. Метод Якобі

Загальний вигляд алгоритму. Програма має обрати початкове наближення **x**(0) та за формулою (3.2) обчислювати наступні наближення до тих пір, доки не буде виконано критерій зупинки за формулою (3.3).

Еквівалентні перетворення математичних формул. Перевірити критерій зупинки (3.3) − доволі дорога операція. Кількість обчислень, необхідна для обчислення за цією формулою, приблизно така ж, що і для обчислення наступного наближення. Тому, якщо реалізувати алгоритм у такому вигляді, на перевірку критерію запинки буде витрачено половину машинного часу. З цієї ситуації є два виходи:

- перетворити критерій зупинки та/або основну обчислювальну формулу так, щоб формули критерію зупинки використовували проміжні дані основного розрахунку;
- перевіряти критерій зупинки не після кожної ітерації, а після кожних M = 10; 20; 50; 100.

Розглянемо критерій зупинки (3.3): знаменник є константним, його можна обчислити один раз після старту програми. Чисельник:

$$\begin{aligned} \left\|\mathbf{r}^{(k)}\right\| &= \left\|\mathbf{b} - A\mathbf{x}^{(k)}\right\| = \\ &= \left\|\mathbf{b} - (D+R)\mathbf{x}^{(k)}\right\| = \\ &= \left\|D\mathbf{x}^{(k)} + \left\{\mathbf{b} - R\mathbf{x}^{(k)}\right\}\right\|. \end{aligned} \quad (3.8)$$

Вираз у фігурних дужках є підвиразом основної формули (3.2). Тому під час обчислення наступного наближення **x**(k+1) можна доволі дешево розрахувати нев'язку **r**(k) для попереднього наближення **x**(k). Якщо нев'язка для k-го наближення задовольняє критерій зупинки, то необхідно взяти k-те (а не останнє розраховане (k + 1)-е) наближення в якості розв'язку.

3.3.2. LU-розклад

Загальний вигляд алгоритму. Програма має обчислити матриці L та U. Нехай у загальному випадку маємо розмірність матриці n, та систему з локальною пам'яттю на p = n процесорів.

Розглянемо випадок, коли кожний рядок матриці розміщено в локальній пам'яті окремого процесора. На першому кроці процесор P_1 розсилає рядок 1 всім іншим процесорам. На другому процесори P_2 . . . P_p обчислюють коефіцієнт l_{p1} та виконують віднімання рядків матриці, відповідно до формул (3.4). На третьому кроці процесор P_2 розсилає рядок 2 процесорам P_3 . . . P_p. Зауважимо, що розсилати достатньо елементи рядка, починаючи з другого, тому як перший вже дорівнює нулю за формулами та відніманя нуля з елементів інших рядків не призведе до їх зміни. Оскільки процесору P_1 більше модифікувати свій рядок не потрібно, то він не отримує даних та не виконує обчислень. На четвертому кроці процесори P_3 . . . P_p виконують обчислення l_{p2} та модифікують a_{pj} відповідно до (3.4). Наступні кроки виконуються аналогічно, зі зменшенням кількості процесорів, яким відбувається розсилка, та кількості елементів, що розсилаються. Виконувані кожним процесором на перших кроках дії наведені у табл. 3.2.

Табл. 3.2. Перші кроки алгоритму при зберіганні по одному рядку

Крок	Процесор P_1	Процесор P_2	Процесор P_3	…	Процесор P_p
1	$P_1 \xrightarrow{a_{11} \dots a_{1n}} P_2 \dots P_p$				
2	—	$l_{21} = a_{21}/a_{11}$ $a_{2j} = a_{2j} - l_{21} \cdot a_{1j}$	$l_{31} = a_{31}/a_{11}$ $a_{3j} = a_{3j} - l_{31} \cdot a_{1j}$	… …	$l_{p1} = a_{p1}/a_{11}$ $a_{pj} = a_{pj} - l_{p1} \cdot a_{1j}$
3	—	$P_2 \xrightarrow{a_{22} \dots a_{2n}} P_3 \dots P_p$			
4	—	—	$l_{32} = a_{32}/a_{22}$ $a_{3j} = a_{3j} - l_{32} \cdot a_{2j}$	… …	$l_{p2} = a_{p2}/a_{22}$ $a_{pj} = a_{pj} - l_{p2} \cdot a_{2j}$
5	—	—	$P_3 \xrightarrow{a_{33} \dots a_{3n}} P_4 \dots P_p$		

Після обчислення за формулами (3.4) першого кроку ітерації процесор P_1 простоює до повного завершення розкладу. Процесор P_2 починає простоювати після 3 кроку і так далі. Постійно працюватиме тільки останній процесор. Таке використання є неефективним, навантаження не є збалансованим, тому необхідно запропонувати спосіб розбиття вихідної матриці, який би зменшив час простою процесорів та дисбаланс навантаження.

Розглянемо випадок, коли в локальній пам'яті кожного процесора розміщено один стовпець матриці. На першому кроці процесор P_1 обчислює всі значення коефіцієнтів l_{i1}. На другому − розсилає їх іншим процесорам. На третьому кроці всі процесори віднімають від кожного елемента в стовпці, що зберігається, значення першого помножене на l_{p1}. Після цих кроків

елементи першого стовпця стали рівними нулю та їх подальша модифікація не потрібна. На четвертому кроці процесор P_2 розраховує значення всіх коефіцієнтів l_{i2} та на наступному розсилає їх процесорам $P_3 \ldots P_p$. Далі аналогічно, з роботи виключаються процесори один за одним.
Виконувані на кожному кроці дії наведені у табл. 3.3.

Табл. 3.3. Перші кроки алгоритму при зберіганні по одному стовпцю

Крок	Процесор P_1	Процесор P_2	Процесор P_3	…	Процесор P_p
1	$l_{i1} = a_{i1}/a_{11}$	—	—	…	—
2	$P_1 \xrightarrow{l_{21} \ldots l_{n1}} P_2 \ldots P_p$				
3	$a_{i1} = a_{i1} - l_{i1} \cdot a_{11}$	$a_{i2} = a_{i2} - l_{i2} \cdot a_{12}$	$a_{i3} = a_{i3} - l_{i3} \cdot a_{13}$	…	$a_{ip} = a_{ip} - l_{ip} \cdot a_{1p}$
4	—	$l_{i2} = a_{i2}/a_{22}$	—	…	—
5	—	$P_2 \xrightarrow{l_{32} \ldots l_{n2}} P_3 \ldots P_p$			
6		$a_{i2} = a_{i2} - l_{i2} \cdot a_{22}$	$a_{i3} = a_{i3} - l_{i3} \cdot a_{23}$	…	$a_{ip} = a_{ip} - l_{ip} \cdot a_{2p}$
7	—	—	$l_{i3} = a_{i3}/a_{33}$	…	—
8	—	—	$P_3 \xrightarrow{l_{43} \ldots l_{n3}} P_4 \ldots P_p$		

В даному випадку процесори починають простоювати після більшої кількості кроків, але проблема нерівномірності навантаження зберігається. Також з'являються кроки, в яких простоюють і останні процесори. Тому такий підхід не вирішує проблему ефективного використання.

В реальних випадках, як правило $p \le n$. Приймемо для спрощення $n = s \cdot p$. Тоді можливо запропонувати дві схеми розбиття матриці.

Блочна схема. В даній схемі в локальній пам'яті i-го процесора зберігаються рядки (або стовпці) з $i \cdot s + 1$ по $(i + 1) \cdot s$ включно. Наведемо приклад такої схеми зберігання за рядками та за стовпцями для матриці розмірністю $n = 6$ в системі з локальною пам'яттю на $p = 3$ процесорів. Замість кожного елемента матриці вказано номер процесора, в пам'яті якого буде зберігатися даний елемент.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 2 & 2 & 3 & 3 \\ 1 & 1 & 2 & 2 & 3 & 3 \\ 1 & 1 & 2 & 2 & 3 & 3 \\ 1 & 1 & 2 & 2 & 3 & 3 \\ 1 & 1 & 2 & 2 & 3 & 3 \\ 1 & 1 & 2 & 2 & 3 & 3 \end{bmatrix}$$

Таким чином після розсилки елементів першого рядка на першому кроці ітерації процесор 1 може продовжувати роботу над наступними своїми рядками паралельно з іншими процесорами. Але після завершення роботи буде очікувати, доки не буде обчислено останній рядок останнім процесором, тобто для нього час очікування еквівалентний $\mathbf{s \cdot (p - 1)}$. В такому випадку загальний час простою зменшується та зменшується час на розсилку даних всім процесорам; зі збільшенням s зменшується загальний коефіцієнт простою процесорів.

Циклічна схема. В даній схемі в локальній пам'яті і-го процесора зберігається кожний s-тий рядок (або стовпець) починаючи з і-го (тобто ***i, i + s, i + 2s, i + 3s …***) Наведемо приклад такої схеми зберігання за рядками та за стовпцями для матриці розмірністю n = 6 в системі з локальною пам'яттю на p = 3 процесорів. Замість кожного елемента матриці вказано номер процесора, в пам'яті якого буде зберігатися даний елемент.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \\ 1 & 2 & 3 & 1 & 2 & 3 \end{bmatrix}$$

Після розрахунку значень першого рядка на першому кроці ітерації процесор 1 може продовжувати роботу над наступним своїм рядком. Після завершення обробки останнього рядка він має очікувати завершення обробки всіх рядків, але через обраний спосіб зберігання кожному з процесорів необхідно буде обробити лише один рядок. Швидкість даної схеми еквівалентна швидкості блочної лише у випадку n = p. В усіх інших дана схема є більш ефективною, але потребує складнішої організації розсилки даних.

3.4. Програмна реалізація

3.4.1. Метод Якобі

Послідовний алгоритм:

Вхідні дані: A, **b**

```
1  begin
2      bnorm ← ‖b‖ /* обчислюємо один раз, так як b не змінюється */
3      обрати x′ (наприклад, одиничний вектор)
4      for Iter ← 0 to 1000 do
5          rnorm ← 0
6          for i ← 1 to dim x do
               s ← b_i − Σ_{j=1, j≠i}^{dim x} a_ij x′_j
7
               x_i ← s / a_ii
8
               r_i ← −s + a_ii x′_i
9
               rnorm ← rnorm + r_i^2
10
11         end
12         rnorm ← √rnorm
13         if rnorm/bnorm < ε then
14             вивести вектор x як результат
15             закінчити роботу
16         end
17         x′ ← x
18     end
19 end
```

3.4.2. LU-розклад

Схема, орієнтована на рядки:

Вхідні дані: A, **b**

Результат : L, A = U, **x**

```
1  begin
2  |  for k ← 1 to n − 1 do                                   /* ітерація */
3  |  |  for i ← k + 1 to n do                              /* цикл за рядками */
4  |  |  |  l_ik ← a_ik / a_kk
5  |  |  |  for j ← k + 1 to n do          /* цикл за елементами у рядку */
6  |  |  |  |  a_ij ← a_ij − l_ik · a_kj
7  |  |  |  end
8  |  |  end
9  |  end
   |  /* розв'язування з прямим використанням LU-розкладу                */
10 |  for i ← 1 to n do
11 |  |  y_i ← b_i
12 |  |  for s ← 1 to i − 1 do
13 |  |  |  y_i ← y_i − y_s · l_is
14 |  |  end
15 |  end
16 |  for i ← n downto 1 do
17 |  |  x_i ← y_i
18 |  |  for s ← i + 1 to n do
19 |  |  |  x_i ← x_i − x_s · a_is
20 |  |  end
21 |  |  x_i ← x_i / a_ii
22 |  end
23 end
```

Схема, орієнтована на стовпці:

Вхідні дані: A, **b**

Результат : L, A = U, **x**

```
1  begin
2  |  for k ← 1 to n − 1 do                                  /* ітерація */
3  |  |  for i ← k + 1 to n do                             /* цикл за рядками */
4  |  |  |  l_ik ← a_ik / a_kk
5  |  |  end
6  |  |  for j ← k + 1 to n do                          /* цикл за стовпцями */
7  |  |  |  for i ← k + 1 to n do          /* цикл за елементами у стовпці */
8  |  |  |  |  a_ij ← a_ij − l_ik · a_kj
9  |  |  |  end
10 |  |  end
11 |  end
   |  /* розв'язування з прямим використанням LU-розкладу                */
12 |  for j ← 1 to n do
13 |  |  y_i ← b_i
14 |  |  for s ← 1 to j − 1 do
15 |  |  |  y_i ← y_i − y_s · l_is
16 |  |  end
17 |  end
18 |  for j ← n downto 1 do
19 |  |  x_j = y_j / a_jj
20 |  |  for i ← 1 to j − 1 do
21 |  |  |  y_i ← y_i − x_j · a_ij
22 |  |  end
23 |  end
24 end
```

3.5. Повний текст програми

Програма реалізує розв'язання СЛАР методом Якобі.
Вхідні дані: матриця A знаходиться в файлі ***MA.txt***, вектор **b** в файлі ***b.txt***.
Результат (вектор x) буде записано в файл x.txt.
Приклад вводу: файл MA.txt:

```
5 5
10 1 1 1 1
1 10 1 1 1
1 1 10 1 1
1 1 1 10 1
1 1 1 1 10
```

файл b.txt:

```
5
1 1 1 1 1
```

Програма виведе в x.txt:
0.071447296000000 0.071447296000000 0.071447296000000
0.071447296000000 0.071447296000000

Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_slae/jacobi.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include ”linalg.h”
/* Точність обчислення коренів */
const double epsilon = 0.001;
/* Функція обчислення наступного наближення ітераційного процесу
Якобі */
void jacobi_iteration(
int start_row,                    // Номер першого рядка частини матриці
struct my_matrix *mat_A_part,     // Частина рядків матриці коефіцієнтів
struct my_vector *b,              // Вектор вільних членів
struct my_vector *vec_prev_x,     // Вектор попереднього наближення
struct my_vector *vec_next_x_part, // Частина вектору наступного наближення
                                  // (встановлюється в функції)
double *residue_norm_part)        // Значення норми на попередньому кроці
                                  // обчислень (встановлюється в функції)
{
/* Акумулятор значення норми даної частини обчислень */
double my_residue_norm_part = 0.0;
/* Поелементне обчислення частини вектору наступного наближення */
for(int i = 0; i < vec_next_x_part->size; i++)
{
double sum = 0.0;
for(int j = 0; j < mat_A_part->cols; j++)
{
if(i + start_row != j)
```

```
{
sum += mat_A_part->data[i * mat_A_part->cols + j] * vec_prev_x->data[j];
}
}
sum = b->data[i + start_row] - sum;
vec_next_x_part->data[i] = sum / mat_A_part->data[i * mat_A_part->cols + i
+ start_row];
/* Обчислення норми на попередньому кроці */
sum = -sum + mat_A_part->data[i * mat_A_part->cols + i + start_row] *
vec_prev_x->data[i + start_row];
my_residue_norm_part += sum * sum;
}
*residue_norm_part = my_residue_norm_part;
}
/* Основна функція */
int main(int argc, char *argv[])
{
const char *input_file_MA = ”MA.txt”;
const char *input_file_b = ”b.txt”;
const char *output_file_x = ”x.txt”;
/* Ініціалізація MPI */
MPI_Init(&argc, &argv);
/* Отримання загальної кількості задач та рангу поточної задачі */
int np, rank;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
/* Зчитування даних в задачі 0 */
struct my_matrix *MA;
struct my_vector *b;
int N;
if(rank == 0)
{
MA = read_matrix(input_file_MA);
b = read_vector(input_file_b);
if(MA->rows != MA->cols) {
fatal_error(”Matrix is not square!”, 4);
}
if(MA->rows != b->size) {
fatal_error(”Dimensions of matrix and vector don’t match!”, 5);
}
N = b->size;
```

```
}
/* Розсилка всім задачам розмірності матриць та векторів */
MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
/* Виділення пам'яті для зберігання вектора вільних членів */
if(rank != 0)
{
b = vector_alloc(N, .0);
}
/* Обчислення частини векторів та матриці, яка буде зберігатися в кожній
* задачі, вважаємо що N = k*np. Виділення пам'яті для зберігання частин
* векторів та матриць в кожній задачі та встановлення їх початкових значень */
int part = N / np;
struct my_matrix *MAh = matrix_alloc(part, N, .0);
struct my_vector *oldx = vector_alloc(N, .0);
struct my_vector *newx = vector_alloc(part, .0);
/* Розбиття вихідної матриці MA на частини по part рядків та розсилка частин
* у всі задачі. Звільнення пам'яті, виділеної для матриці MA. */
if(rank == 0)
{
MPI_Scatter(MA->data, N*part, MPI_DOUBLE, MAh->data, N*part,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
free(MA);
}
else
{
MPI_Scatter(NULL, 0, MPI_DATATYPE_NULL, MAh->data, N*part,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
/* Розсилка вектора вільних членів */
MPI_Bcast(b->data, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* Обчислення норми вектору вільних членів в задачі 0 та розсилка її значення
* у всі задачі */
double b_norm = 0.0;
if(rank == 0)
{
for(int i = 0; i < b->size; i++)
{
```

```
b_norm = b->data[i] * b->data[i];
}
b_norm = sqrt(b_norm);
}
MPI_Bcast(&b_norm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* Значення критерію зупинки ітерації */
double last_stop_criteria;
/* Основний цикл ітерації Якобі */
for(int i = 0; i < 1000; i++)
{
double residue_norm_part;
double residue_norm;
jacobi_iteration(rank * part, MAh, b, oldx, newx, &residue_norm_part);
/* Обчислення сумарного значення нев'язки */
MPI_Allreduce(&residue_norm_part, &residue_norm, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
residue_norm = sqrt(residue_norm);
/* Перевірка критерію зупинки ітерації. Оскільки на поточному кроці
* обчислюється значення норми для попереднього кроку, то результатом
* обчислення є дані попереднього кроку */
last_stop_criteria = residue_norm / b_norm;
if(last_stop_criteria < epsilon)
{
break;
}
/* Збір значень поточного наближення вектору невідомих */
MPI_Allgather(newx->data, part, MPI_DOUBLE, oldx->data, part,
MPI_DOUBLE, MPI_COMM_WORLD);
}
/* Вивід результату */
if(rank == 0)
{
write_vector(output_file_x, oldx);
}
/* Повернення виділених ресурсів системі та фіналізація середовища MPI
free(MAh);
free(oldx);
free(newx);
free(b);
return MPI_Finalize();
}
```

3.6. Варіанти

Варіант визначається по табл. 3.4.

3.7. Питання для самоконтролю

1. Колективні операції в MPI − це:
   а) операції над комунікаторами;
   б) пересилки, в яких приймають участь всі процеси деякого комунікатору;
   в) операції над групами процесів.
2. На які групи можна умовно розділити види пересилок, в яких беруть участь більше, ніж 2 задачі?
3. Які є два основних методи організації взаємодії задач на основі пересилання повідомлень?
4. В чому різниця між функціями MPI_Bcast та MPI_Scatter?
5. Необхідно виконати пошук у великому масиві даних, що був зчитаний однією задачею. Існує функція пошуку, яка може шукати в будь-якому обсязі даних. Яким чином можна пришвидшити пошук та як необхідно оперувати з даними?
6. Кожна задача згенерувала декілька послідовних членів ряду, причому номер першого згенерованого члену є пропорційним рангу задачі. Необхідно вивести весь ряд користувачу, але доступ до пристрою виводу має лише задача 4. За допомогою якої функції найпростіше підготувати дані до виводу?
7. Кожна задача згенерувала декілька послідовних членів ряду як і у попередньому питанні, але ряд необхідний для обчислення значень функції (яка буда розкладена в ряд) в деяких точках. Великий масив точок зберігається в задачі 2. Як забезпечити максимальне розпаралелювання такої роботи?
8. Запропонуйте ще один спосіб для попереднього питання.
9. В кожній задачі є певна кількість чисел, що є результатами експерименту. Як знайти мінімальне та максимальне значення результату всього експерименту?
10. Кожна задача моделювала фізичний процес при певних параметрах. Якщо одне з декількох явищ мало місце впродовж експерименту – встановлювався відповідний прапорець. Необхідно визначити, що на даному наборі параметрів хоча б в одному з експериментів мало місце кожне явище.
11. В кожній задачі зберігається масив додатних чисел. Яким чином нормалізувати їх до проміжку [0; 1], де 0 відповідатиме найменшому значенню, а 1 − найбільшому?
12. В системі з P задачами необхідно чисельно обчислити P визначених інтегралів на однаковому проміжку. Нехай існує функція, що може

швидко обчислювати значення інтегралів декількох функцій на однаковому проміжку. Кожна задача розбила весь проміжок інтегрування на P частин. Яким чином розіслати дані для ефективного розпаралелювання?

13. Яким чином забезпечити отримання результатів інтегрування задачами з попереднього питання?

Табл. 3.4. Варіанти завдання

№	Метод	Розбиття даних	Організація задач
1	Розв'язання СЛАР методом Якобі	—	майстер-робітник
2	Розв'язання СЛАР методом Якобі	—	однорангова
3	Розв'язання СЛАР методом LU-розкладу	блоки рядків	майстер-робітник
4	Розв'язання СЛАР методом LU-розкладу	блоки рядків	однорангова
5	Розв'язання СЛАР методом LU-розкладу	блоки стовпців	майстер-робітник
6	Розв'язання СЛАР методом LU-розкладу	блоки стовпців	однорангова
7	Розв'язання СЛАР методом LU-розкладу	цикли рядків	майстер-робітник
8	Розв'язання СЛАР методом LU-розкладу	цикли рядків	однорангова
9	Розв'язання СЛАР методом LU-розкладу	цикли стовпців	майстер-робітник
10	Розв'язання СЛАР методом LU-розкладу	цикли стовпців	однорангова
11	Обчислення визначника	блоки рядків	майстер-робітник
12	Обчислення визначника	блоки рядків	однорангова
13	Обчислення визначника	блоки стовпців	майстер-робітник
14	Обчислення визначника	блоки стовпців	однорангова
15	Обчислення визначника	цикли рядків	майстер-робітник
16	Обчислення визначника	цикли рядків	однорангова
17	Обчислення визначника	цикли стовпців	майстер-робітник
18	Обчислення визначника	цикли стовпців	однорангова

2.4 Лабораторна робота 4
Користувацькі типи даних в MPI

Мета роботи:

- вивчити засоби середовища MPI для опису користувацьких типів і можливості щодо їх передачі;
- навчитися аналізувати алгоритм програми щодо типів даних, які будуть пересилатися між задачами;
- навчитися використовувати користувацькі типи разом з функціями обміну даними.

Завдання: реалізувати паралельну програму для розв'язання СЛАР методом Якобі або із застосуванням LU-розкладу. В даній роботі використовуються той самий математичний апарат, що й в попередній роботі.

4.1. Типи даних в MPI

Будь-яка функція передачі даних в MPI має обов'язковий аргумент, який вказує тип даних, що передаються. Це викликано тим, що передача даних на фізичному рівні відбувається за байтами або навіть бітами. Інтерпретувати послідовність байт можливо різним чином, тому важливо знати спосіб кодування типу даних, який ними представлений, та їх розмір. Наприклад, в деяких комп'ютерах тип ***long long*** вдвічі більший за ***int***, тому одне значення першого може бути помилково інтерпретоване як два значення int. Також в комп'ютерах може відрізнятись послідовність байтів в багатобайтовому значенні: спочатку молодший або спочатку старший (так звані ***little endian*** та ***big endian*** архітектури). Щоб врахувати все це в MPI існує спеціальний механізм опису типів даних через змінні типу **MPI_Datatype**. Для основних стандартних типів мови С визначені спеціальні константи цього типу (розглянуті раніше, див. табл. 1.1.).

Також визначений спеціальний тип **MPI_B**YTE для передачі одного байта даних та тип **MPI_PACKED** для даних, які були упаковані за допомогою **MPI_Pack()**.

MPI надає можливість конструювати опис користувацьких типів даних за допомогою вже визначених типів. Для опису типу в середовищі MPI існує тип змінної-дескриптора **MPI_Datatype**. Будь-який тип перед безпосередньою відсилкою повідомлень має бути зареєстрований за допомогою функції

MPI_Type_commit(**MPI_Datatype** *type);

де ***type*** − попередньо сконструйований тип. При реєстрації типу MPI виділяє додаткові ресурси для зберігання дескриптору, тому після завершення використання типу або перед завершенням програми необхідно повернути ці ресурси операційній системі. Для цього призначена функція

MPI_Type_free(**MPI_Datatype** *type);

де ***type*** − зареєстрований тип.

З базових типів можна сконструювати послідовні (англ. ***contiguous***), векторні (англ. ***vector***), індексовані (англ. ***indexed***) типи та структури. Будь-який тип може бути сконструйований з вже зареєстрованих, тобто якщо існує зареєстрована структура ***struct st***, то можна створити дескриптор структури, яка має елементи типу ***struct st***. MPI дозволяє будь-який рівень вкладеності типів даних. Деякі функції опису типів вимагають вказання розміру типів елементів, що входять до описуваного типу, в байтах. Оскільки розмір типу в MPI може відрізнятися від розміру в мові програмування (який повертає sizeof) через наявність службових даних, то для отримання розміру зареєстрованого типу в байтах необхідно користуватися функцією

MPI_Type_extent(**MPI_Datatype** type, **MPI_Aint** *extent);

• type − дескриптор зареєстрованого типу даних;

• extent − покажчик змінної, в яку буде записано розмір в байтах.

MPI_Aint − це спеціальний тип цілого числа, яке може вмістити максимальну можливу адресу комірки пам'яті в даному комп'ютері, фактично це MPI аналог стандартного типу size_t.

Функції розсилок, які виконують розбиття, об'єднання або згортку даних (описані в попередній роботі) оперують з типами даних в цілому та мають доступу до їх внутрішньої структури. Якщо оголошений тип матриці як двовимірного масиву елементів, то функція MPI_Scatter може виконати тільки розбиття масиву з декількох матриць та розіслати окремі матриці задачам. Вона не може розбити одну матрицю, якщо вона відправляється як один об'єкт типу «матриця», на рядки або елементи.

Послідовні типи

Послідовним називається тип, що складається з елементів одного базового типу, розміщених послідовно один за одним. Фактично це масив елементів базового типу фіксованої довжини.

MPI_Type_contiguous(**int** count, **MPI_Datatype** oldtype, **MPI_Datatype** *newtype);

- ***count*** − кількість елементів базового типу;

- ***oldtype*** − дескриптор базового типу;
- ***newtype*** − покажчик на дескриптор конструйованого типу.

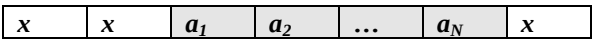
Функція має бути викликана з однаковими параметрами в усіх задачах, що будуть використовувати даний тип.

Приклад 12. Програма оперує з векторами довжини **N**, що описані як ***double *vec***. Між задачами необхідно передати декілька векторів. Очевидна реалізація складається з декількох послідовних передач **N** елементів типу **MPI_DOUBLE**. Але краще створити спеціальний тип:

```
MPI_Datatype mytype;
MPI_Type_contiguous(N, MPI_DOUBLE, &mytype);
MPI_Type_commit(&mytype);
```

Тоді з'являється можливість пересилати цілі вектори, або створити в програмі масив з декількох векторів та передавати його одним викликом. Зверніть увагу, що після опису типу та перед його використанням, тип було зареєстровано в середовищі MPI. Після завершення передач векторів, якщо надалі вони не будуть передаватись, необхідно звільнити ресурси дескриптору:

```
MPI_Type_free(&mytype);
```



Векторні типи

Векторний тип являє собою послідовність блоків даних фіксованої довжини, між якими можуть бути проміжки фіксованої довжини.

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

- ***count*** − кількість блоків;
- ***blocklength*** − довжина одного блока;
- ***stride*** − кількість елементів між початками блоків;
- ***oldtype*** − дескриптор типу елементів в блоці;
- ***newtype*** − покажчик на дескриптор конструйованого типу.

В векторному типі всі блоки однакового розміру та відстань між початками також однакова. Блоки можуть перекриватися, тобто відстань між початками блоків може бути менше довжини блока. Якщо відстань між початками дорівнює довжині блока, то векторний тип повністю аналогічний послідовному з довжиною ***count _ blocklength***.

```
MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype);
```

Аналогічний до **MPI_Type_vector**, але відстань між початками stride визначається в байтах (через **MPI_Type_extent**).

Одною з корисних особливостей використання власних типів даних є те, що при передачі тип даних, які відправляються, не обов'язково має збігатися з типом даних, що приймаються. Таким чином можна відправити векторний тип даних, а прийняти послідовний і надалі працювати в приймаючій задачі з масивом.

Приклад 13. Трьохдіагональною називається матриця вигляду

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & \dots & 0 \\ 0 & 0 & a_4 & b_4 & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & c_{N-1} \\ 0 & 0 & 0 & 0 & a_N & b_N \end{bmatrix}$$

Існує швидкий метод розв'язку СЛАР, коефіцієнти якої записуються у вигляді такої матриці. Нехай задача 2 має розіслати таку матрицю всім іншим задачам, які мають отримати матрицю в такій самій формі представлення. В тридіагональній матриці ненульових елементів **3N-2**, що при великих **N** значно менше за $\mathbf{N^2}$. Доцільно розсилати тільки ненульові елементи, а всі інші ініціалізувати нулем в задачах.

Кожен рядок, окрім першого та останнього містить три ненульових елементи. Для того, щоб структура даних стала однорідною, виділимо буфер розмірністю $\mathbf{N^2 + 2}$ та розташуємо елементи матриці починаючи з індексу **1** (а елемент **0** залишимо неініціалізованим). Тоді можна описати векторний тип з **N** блоків довжиною **3** елементи. Між елементами одного стовпця матриці при зберіганні в одновимірному масиві знаходиться (**N - 1**) елементів. А для того, щоб перейти в наступний стовпчик необхідно пропустити ще один. Враховуючи нульовий та останній елементи масиву, виділені додатково, регулярність структури даних зберігається.

```
int *MA = malloc(sizeof(int)*(N*N + 2));
if(rank == 2)
get_matrix(&MA[1], "MA.dat"); // зчитування елементів, починаючи з першого
MPI_Datatype tdm;
MPI_Type_vector(N, 3, N+1, MPI_INT, &tdm);
MPI_Type_commit(&tdm);
MPI_Bcast(MA, 1, tdm, 2, MPI_COMM_WORLD);
MPI_Type_free(&tdm);
```

x	b_1	c_1	***0***	
	a_2	b_2	c_2	
	0	a_3	b_3	***x***

Приклад 14. При використанні блочного алгоритму матриця розбивається на підматриці, що зберігаються в кожній задачі. Нехай розмірність кожної під матриці **N*N**. За алгоритмом необхідно розіслати головну діагональ матриці всім задачам. Невигідно виділяти в усіх задачах місце для зберігання всієї матриці, від якої відома лише головна діагональ. Програмувати подвійну індексацію елементів також важче, тому доцільно зберігати діагональ як вектор. Оскільки кожній задачі потрібна головна діагональ кожної матриці, то можна організувати матрицю, в якій ***i***-тий рядок відповідатиме головній діагоналі ***i***-го стовпця. Між елементами одного стовпця матриці при зберіганні в одновимірному масиві знаходиться (**N-1**) елементів. А для того, щоб перейти в наступний стовпчик необхідно пропустити ще один. Тобто між елементами головної діагоналі міститься N елементів. Якщо виділяти «блоки» довжиною в один елемент, то відстань між початками блоків становитиме (**N+1**). Необхідна розсилка «багато до багатьох», тому використання використання **MPI_Allgather** доцільніше, ніж безпосередній обмін.

```
double *MA = malloc(sizeof(double)*N*N);          // підматриця
double *diags = malloc(sizeof(double)*N*N);       // матриця головних діагоналей
MPI_Datatype type_row, type_diagonal;
MPI_Type_contiguous(N, MPI_DOUBLE, &type_row);
MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &type_diagonal);
MPI_Type_commit(&type_row);
MPI_Type_commit(&type_diagonal);
MPI_Allgather(MA, 1, type_diagonal, diags, 1, type_row, MPI_COMM_WORLD);
// ... використання ...
MPI_Type_free(&type_row);
MPI_Type_free(&type_diagonal);
```

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

Індексовані типи

Індексований тип є послідовністю блоків даних одного типу, причому довжина кожного блока та відстань між кожною парою блоків може бути різною.

MPI_Type_indexed(**int** count, **int** blocklengths[], **int** indices[],**MPI_Datatype** oldtype, **MPI_Datatype** &newtype);

- ***count*** − кількість блоків;
- ***blocklengths*** − масив розміру count довжин блоків;

- ***indices*** − масив розміру count відстані від початку буфера;
- ***oldtype*** − дескриптор базового типу даних;
- ***newtype*** − покажчик на дескриптор конструйованого типу.

Як правило перший блок даних починається з початку буфера, тобто відстань від початку буфера до першого блока становитиме 0. Наприклад, якщо необхідно відправити 1, 3 та 7 елементи послідовності (нумерація з одиниці), то відстані становитимуть 0, 2 та 6 відповідно. Але таке розміщення не є обов'язковим. Наприклад, для відправки елементів 4, 13 та 21 необхідно вказати відстані 3, 12 та 20 відповідно.

MPI_Type_hindexed(**int** count, **int** blocklengths[], **MPI_Aint** indices[], **MPI_Datatype** oldtype, **MPI_Datatype** &newtype);

Аналогічний до **MPI_Type_indexed**, але відстані від початку буферу ***indices***[] визначається в байтах (через **MPI_Extent**).

Приклад 15. Задача 0 після деяких дій гарантовано отримує тридіагональну матрицю та має розіслати її всім іншим задачам. Зберігати нульові елементи недоцільно, тому алгоритми роботи задач побудовані таким чином, що використовують наступну структуру даних:

[b_1 c_1 a_2 b_2 c_2 . . . a_N b_N] ;

тобто зберігає лише ненульові елементи. Оскільки кількість ненульових елементів в першому і останньому та інших рядках різна, то доцільно використати індексований тип. Його параметри мають бути наступними:

blocklengths[N] = {2, 3, 3, 3, 3, ... 3, 3, 2}
indices[N] = {0, N, 2*N+1, 3*N+2, ... (N-2)*N+(N-3), (N-1)*N+(N-2)}

Скористаємося також можливістю приймати інший тип даних, ніж був надісланий, при забезпеченні того ж розміру передачі.

```
double *MA;
if(rank == 0)
   MA = malloc(N*N*sizeof(double));
else
   MA = malloc((3*N-2)*sizeof(double));
int blocklengths[N];
int indices[N];
indices[0] = 0;
blocklengths[i] = 2;
for(int i = 1; i < N; i++)
{
   blocklengths[i] = 3;
   indices[i] = i*N + i - 1;
}
```

```
blocklengths[N-1] = 2;
MPI_Datatype tdm;
MPI_Datatype tdm_compact;
MPI_Type_indexed(N, blocklengths, indices, MPI_DOUBLE, &tdm);
MPI_Type_contiguous(3*N-2, MPI_DOUBLE, &tdm_compact);
MPI_Type_commit(&tdm);
MPI_Type_commit(&tdm_compact);
if(rank == 0)
   for(int i = 1; i < np; i++)
         MPI_Send(MA, 1, tdm, i, 1, MPI_COMM_WORLD
else
   MPI_Recv(MA, 1, tdm_compact, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Type_free(&tdm);
MPI_Type_free(&tdm_compact);
```

b_1	c_1	0	0
a_2	b_2	c_2	0
0	a_3	b_3	c_3
0	0	a_4	b_4

Структури

Структура в MPI є послідовністю блоків даних різних типів та різного розміру з можливими проміжками різної довжини між ними.

MPI_Type_struct(**int** count, **int** blocklength[], **MPI_Aint** offsets[], **MPI_Datatype**[] oldtypes, **MPI_Datatype** *newtype);

- ***count*** − кількість блоків (елементів) структури;
- ***blocklength*** − масив розміру count довжин блоків;
- ***offsets*** − масив розміру count відстаней від початку структури до початку блока;
- ***oldtypes*** − масив розміру count дескрипторів типів блоків (елементів) структури;
- ***newtype*** − покажчик на дескриптор конструйованого типу.

Дані описуваного типу необов'язково мають становити структуру в мові С, достатньо, щоб вони були розміщені в пам'яті послідовно. Якщо дескриптор конструюється для вже існуючої структури, то необов'язково, щоб він включав всі елементи структури. Послідовно розташовані елементи структури однакового типу можна об'єднувати в блоки, довжина яких дорівнюватиме кількості елементів.

Приклад 16. Програма моделює частки в Великому Адронному колайдері. Кожна частка описується трьома координатами (з плаваючою комою), швидкістю (з плаваючою комою), типом (ціле) та вагою у відносних одиницях маси (ціле). Для зручності опису частки описано відповідну структуру.

```
struct particle
{
   double x, y, z;
   int type, mass;
};
```

Кожна задача моделює поведінку N часток. Після завершення моделювання задача 3 має вивести результат моделювання всіх часток. Кількість задач P. Опишемо тип для даної структури в середовищі MPI, після чого зможемо скористатися процедурою MPI_Gather для збору даних в одну задачу.

```
MPI_Datatype type_particle;
int extent;
MPI_Type_extent(MPI_DOUBLE, &extent);
int blocklength[] = {3, 2};
MPI_Aint offsets[] = {0, 3*extent};
MPI_Datatype oldtypes[] = {MPI_DOUBLE, MPI_INT};
MPI_Type_struct(2, blocklength, offsets, oldtypes, &type_particle);
MPI_Type_commit(&type_particle);
if(rank == 3)
{
   struct particle *results = malloc(sizeof(struct particle)*N*P);
   MPI_Gather(model, N, type_particle, results, N, type_particle, 3,
MPI_COMM_WORLD);
} else {
   MPI_Gather(model, N, type_particle, NULL, 0, MPI_DATATYPE_NULL,
3,
   MPI_COMM_WORLD);
}
MPI_Type_free(type_particle);
```

Приклад 17. Ускладнимо попередню задачу. Тепер координати частки зберігаються в спеціальній структурі «координати» В процесі моделювання кожна задача використовує службові поля структури «останній давач, що зафіксував частку» (ціле), «кількість фіксацій частки давачами» (ціле), «час останньої фіксації» (структура).

```
struct point
{
```

```
    double x, y, z;
};
struct timer
{
    short hours, minutes, seconds, msecs, usecs, nanosecs;
};
struct particle_adv
{
    struct point last_point;
    int type, detector;
    struct timer last_detected;
    int mass, detection_number;
}
```

Однак службові дані не потрібні користувачу, тому для виводу використовується структура particle з попереднього прикладу. Вважатимемо, що вона вже описана та зареєстрована в MPI і має дескриптор type_particle. Необхідно описати нову структуру таким чином, щоб відповідні дані знаходилися на однаковій відстані. Таке використання можливе лише тоді, коли дані в обох структурах розташовані в одному порядку. Оскільки структура point містить лише дані одного типу, то можемо легко представити її у вигляді послідовного типу.

```
MPI_Datatype type_particle_adv, type_point, type_timer;
MPI_Type_contiguous(3, MPI_DOUBLE, &type_point);
MPI_Type_contiguous(6, MPI_SHORT, &type_timer);
MPI_Type_commit(&type_point);
MPI_Type_commit(&type_timer);
int point_extent, int_extent, timer_extent;
MPI_Type_extent(type_point, &point_extent);
MPI_Type_extent(MPI_INT, &int_extent);
MPI_Type_extent(type_timer, &timer_extent);
MPI_Type_free(type_timer);
int blocklengths[] = {1, 1, 1};
MPI_Aint offsets[] = {0, point_extent, point_extent + 2*int_extent +
timer_extent};
MPI_Datatype oldtypes[] = {type_point, MPI_INT, MPI_INT};
MPI_Type_struct(3, blocklengths, offsets, oldtypes, &type_particle_adv);
MPI_Type_commit(&type_particle_adv);
MPI_Gather(model, N, type_particle_adv results, N, type_particle, 3,
MPI_COMM_WORLD);
MPI_Type_free(type_particle_adv);
MPI_Type_free(type_point);
```

Структури з покажчиками. При описі структур необхідно бути особливо уважними, якщо структура містить покажчики. Оскільки MPI працює у системі з локальною пам'яттю, та покажчик є адресою комірки локальної пам'яті, то передача покажчика не передає дані, на які він вказує. Якщо покажчик було помилково передано, то в більшості випадків програму буде аварійно зупинено через неправильний доступ до пам'яті. В деяких випадках (наприклад, структура містить елемент − покажчик на іншу структуру, − та вказано дескриптор вкладеної структури в якості базового типу) передача може не відбутися через різний очікуваний та реальний розмір елемента. Якщо структура містить покажчик, то дані, на які він вказує, мають бути передані окремо.

Приклад 18. Маємо просту структуру, що відображає вектор змінної довжини. Перше значення − довжина, друге − покажчик на елементи вектора. Необхідно розіслати вектор всім задачам. В першу чергу необхідно розіслати розмірність вектора, тому що при прийому елементів вона використовується як аргумент функції MPI, і якщо її значення не збігатиметься з кількістю відправлених елементів, передача може не відбутися.

```
struct vect
{
   int size;
   double *elements;
};
struct vect v;
MPI_Bcast(v.size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(v.elements, v.size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Структури змінної довжини. Часто в програмах на С використовують так звані структури змінної довжини. Це структури, в кінці яких знаходиться «масив» нульового розміру, який фактично є адресою кінця структури. Під час виділення пам'яті для структури виділяється більше пам'яті: вона виділяється відразу після кінця структури, і може бути адресована як наступні елементи масиву.

```
struct my_vector
{
   int size;
   double elems[0];
};
struct my_vector *v = malloc(sizeof(struct my_vector) + N*sizeof(double));
v->elems[N-1] = 10;
```

Якщо дані в структурі змінної довжини мають обмеження зверху по кількості або використовуються тільки типові розміри, то для них можна описати користувацький тип.

Приклад 19. Програма використовує бібліотеку лінійної алгебри, в якій вектор описується структурою змінної довжини, яка показана вище як my_vector. Програма оперує лише векторами довжини **N** та **M**. Опишемо типи для цих типових випадків.

```
MPI_Datatype vector_n, vector_m;
int extent;
MPI_Type_extent(MPI_INT, &extent);
int blocklengths[] = {1, N};
MPI_Aint offsets[] = {0, extent};
MPI_Datatype oldtypes[] = {MPI_INT, MPI_DOUBLE};
MPI_Type_struct(2, blocklengths, offsets, oldtypes, &vector_n);
blocklengths[1] = M;
MPI_Type_struct(2, blocklengths, offsets, oldtypes, &vector_m);
MPI_Type_commit(&vector_n);
MPI_Type_commit(&vector_m);
```

4.2. Аналіз структур даних задачі

4.2.1. Метод Якобі

Необхідно організувати передачу вектор-стовпця вільних членів, розбиття матриці за рядками та передачу рядків, збір вектора поточного наближення за елементами.

Нагадаємо, що в запропонованій бібліотеці для роботи з векторами та матрицями вони представлені структурою змінної довжини. На початку структури зберігаються розмірності. Доцільно описати окремий послідовний тип «рядок матриці», який буде мати доступ тільки до елементів матриці. Розмірності можна передавати перед елементами, або розраховувати, оскільки за алгоритмом кожна задача отримує фіксоване число рядків матриці. Також необхідно описати тип для вектора вільних членів. Це може бути структура, яка міститиме розмірність та елемент.

А може бути послідовний тип, що містить лише елементи, а розмірність передається окремо або обчислюється. Для збору з частин вектора поточного наближення можна використати послідовний тип, який відображатиме декілька елементів вектора. Але якщо розмірність дорівнює кількості задач, такий тип буде дублювати MPI_DOUBLE.

4.2.2. LU-розклад

Розбиття за рядками. Необхідно організувати передачу ведучого рядка матриці. Для цього можна сконструювати послідовний тип з її елементів.

Оскільки починаючи з другого кроку перші елементи рядка, що передається, нульові, то при великих **N** доцільно створити декілька типів різних розмірностей, а в задачах-отримувачах заповнювати інші елементи нулями. Також можна організувати поелементну передачу для максимально можливої економії пропускної здатності каналу.

Розбиття за стовпцями. Спочатку необхідно організувати розсилку стовпців матриці по задачам. Для цього можна використати векторний тип, що складається з **N** блоків по одному елементу, розміщених з проміжком **N.** Таким чином вказуючи в якості початку буфера перший елемент ***i***-го стовпця (тобто ***i***-тий елемент матриці) можна передати весь стовпець. Для зберігання стовпця як одновимірного масиву в задачах необхідно отримувати дані поелементно в такий масив. Використати **MPI_Scatter** для розсилки даних в таких умовах неможливо, так як дана функція визначає початок наступного об'єкту даного типу після кінця поточного. Таким чином перший об'єкт для MPI_Scatter буде першим стовпцем, а другий − починатиметься з другого елемента останнього рядка та виходитиме за буфер. Розсилку в такому разі необхідно організувати за допомогою **MPI_Send/MPI_Recv.**

Також необхідно розсилати вектор поточних значень l, причому на кожному наступному кроці кількість нульових елементів на початку вектора збільшується. Сконструювати тип для нього можна аналогічно до вектора у розбитті за рядками.

Блочна організація. Достатньо передавати декілька послідовних об'єктів вищеописаного типу, в залежності від розбиття. Можна також описати на базі існуючого векторного типу новий тип для відображення декількох рядків або стовпців. Для рядків такий тип буде послідовним. Для стовпців − векторний тип з проміжком **1.**

Циклічна організація. Існує два можливих варіанти: створити новий тип, який буде описувати відразу всі рядки (стовпці), що відправляються одній задачі. Або зробити вкладений тип, що складається з вищеописаних типів рядка та стовпця.

Розглянемо перший варіант для рядків. Нехай part = **N / np**. Необхідно створити векторний тип з part блоків даних типу базового елемента, довжиною **N** та проміжком **N*(part-1)** між ними. Таким чином буде виділено один рядок з **part** та пропущено **(part-1)** наступних за ним. При розсилці до ***i***-тої задачі початок буфера необхідно вказувати на початку ***i***-го рядка вихідної матриці.

Для стовпців необхідно створити векторний тип з **part*N** блоків даних довжиною **1**, проміжок між блоками визначатиметься як **part**. Таким чином буде обрано всі елементи кожного part-го стовпця. Пересилку даних ***i***-тій задачі необхідно починати з ***i***-го елемента першого рядка. В разі та-

кої організації в задачі стовпці зберігатимуться як матриця (необхідна відповідна індексація) з **N** рядків по **part** елементів.
Розглянемо другий варіант для рядків. Необхідно створити векторний тип з **part** блоків даних типу рядка, довжиною **1** та проміжком (**part-1**) між ними. Очевидно, що довжина та проміжок стали в **N** разів менше через використання типу з **N** елементів базового типу.
Для стовпців також потрібен векторний тип з part блоків даних типу стовпця, довжиною **1** та проміжком між ними part. Розбиття по задачам аналогічне до першого варіанта.
В жодному варіанті немає можливості використати функцію **MPI_Scatter**

4.3. Повний текст програми

Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_slae2/lu.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include ”linalg.h”
/* Ім’я вхідного файлу */
const char *input_file_MA = ”MA.txt”;
/* Тег повідомленя, що містить стовпець матриці */
const int COLUMN_TAG = 0x1;
/* Основна функція (програма обчислення визначника) */
int main(int argc, char *argv[])
{
   /* Ініціалізація MPI */
   MPI_Init(&argc, &argv);
   /* Отримання загальної кількості задач та рангу поточної задачі */
   int np, rank;
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   /* Зчитування даних в задачі 0 */
   struct my_matrix *MA;
   int N;
   if(rank == 0)
   {
         MA = read_matrix(input_file_MA);
         if(MA->rows != MA->cols) {
            fatal_error(”Matrix is not square!”, 4);
```

```
    }
    N = MA->rows;
  }
  /* Розсилка всім задачам розмірності матриць та векторів */
  MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
  /* Обчислення кількості стовпців, які будуть зберігатися в кожній задачі та
  * виділення пам'яті для їх зберігання */
  int part = N / np;
  struct my_matrix *MAh = matrix_alloc(N, part, .0);
  /* Створення та реєстрація типу даних для стовпця елементів матриці */
  MPI_Datatype matrix_columns;
  MPI_Type_vector(N*part, 1, np, MPI_DOUBLE, &matrix_columns);
  MPI_Type_commit(&matrix_columns);
  /* Створення та реєстрація типу даних для структури вектора */
  MPI_Datatype vector_struct;
  MPI_Aint extent;
  MPI_Type_extent(MPI_INT, &extent);          // визначення розміру в байтах
  MPI_Aint offsets[] = {0, extent};
  int lengths[] = {1, N+1};
  MPI_Datatype oldtypes[] = {MPI_INT, MPI_DOUBLE};
  MPI_Type_struct(2, lengths, offsets, oldtypes, &vector_struct);
  MPI_Type_commit(&vector_struct);
  /* Розсилка стовпців матриці з задачі 0 в інші задачі */
  if(rank == 0)
  {
    for(int i = 1; i < np; i++)
    {
                MPI_Send(&(MA->data[i]), 1, matrix_columns, i,
      COLUMN_TAG, MPI_COMM_WORLD);
    }
    /* Копіювання елементів стовпців даної задачі */
    for(int i = 0; i < part; i++)
    {
        int col_index = i*np;
      for(int j = 0; j < N; j++)
      {
          MAh->data[j*part + i] = MA->data[j*N + col_index];
      }
```

```
        }
        free(MA);
    }
    else
    {
        MPI_Recv(MAh->data, N*part, MPI_DOUBLE, 0, COLUMN_TAG,
        MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
    /* Поточне значення вектору l_i */
    struct my_vector *current_l = vector_alloc(N, .0);
    /* Частина стовпців матриці L */
    struct my_matrix *MLh = matrix_alloc(N, part, .0);
    /* Основний цикл ітерації (кроки) */
    for(int step = 0; step < N-1; step++)
            {
        /* Вибір задачі, що містить стовпець з ведучім елементом та обчис-
лення
        * поточних значень вектору l_i */
        if (step % np == rank)
        {
            int col_index = (step - (step % np)) / np;
            MLh->data[step*part + col_index] = 1.;
            for(int i = step+1; i < N; i++)
            {
                MLh->data[i*part + col_index] = MAh->data[i*part + col_index]
        /
                    MAh->data[step*part + col_index];
            }
                    for(int i = 0; i < N; i++)
            {
                current_l->data[i] = MLh->data[i*part + col_index];
            }
        }
        /* Розсилка поточних значень l_i */
        MPI_Bcast(current_l,        1,        vector_struct,        step        %        np,
MPI_COMM_WORLD);
        /* Модифікація стовпців матриці MA відповідно до поточного l_i */
        for(int i = step+1; i < N; i++)
        {
            for(int j = 0; j < part; j++)
            {
```

```
                MAh->data[i*part + j] -= MAh->data[step*part + j] * current_l-
            >data[i];
                }
                    }
    }
    /* Обислення добутку елементів, які знаходяться на головній діагоналі
    * основної матриці (з урахуванням номеру стовпця в задачі) */
    double prod = 1.;
    for(int i = 0; i < part; i++)
    {
        int row_index = i*np + rank;
        prod *= MAh->data[row_index*part + i];
    }
    /* Згортка добутків елементів головної діагоналі та вивід результату
в задачі 0 */
                if(rank == 0)
    {
        MPI_Reduce(MPI_IN_PLACE, &prod, 1, MPI_DOUBLE, MPI_PROD,
0, MPI_COMM_WORLD);
        printf("%lf", prod);
    }
    else
    {
        MPI_Reduce(&prod, NULL, 1, MPI_DOUBLE, MPI_PROD, 0,
MPI_COMM_WORLD);
    }
    /* Повернення виділених ресурсів */
    MPI_Type_free(&matrix_columns);
    MPI_Type_free(&vector_struct);
    return MPI_Finalize();
}
```

4.4. Варіанти

Варіант визначається по табл. 4.1.

Табл. 4.1. Варіанти завдання

№	Метод	Розбиття даних	Організація задач
1	Розв’язання СЛАР методом LU-розкладу	блоки рядків	однорангова
2	Розв’язання СЛАР методом LU-розкладу	блоки рядків	майстер-робітник
3	Розв’язання СЛАР методом LU-розкладу	цикли стовпців	однорангова
4	Розв’язання СЛАР методом LU-розкладу	цикли стовпців	майстер-робітник
5	Розв’язання СЛАР методом LU-розкладу	цикли рядків	однорангова
6	Розв’язання СЛАР методом LU-розкладу	цикли рядків	майстер-робітник
7	Розв’язання СЛАР методом LU-розкладу	блоки стовпців	однорангова
8	Розв’язання СЛАР методом LU-розкладу	блоки стовпців	майстер-робітник
9	Обчислення визначника	блоки рядків	однорангова
10	Обчислення визначника	блоки рядків	майстер-робітник
11	Обчислення визначника	цикли стовпців	однорангова
12	Обчислення визначника	цикли стовпців	майстер-робітник
13	Обчислення визначника	цикли рядків	однорангова
14	Обчислення визначника	цикли рядків	майстер-робітник
15	Обчислення визначника	блоки стовпців	однорангова
16	Обчислення визначника	блоки стовпців	майстер-робітник
17	Розв’язання СЛАР методом Якобі	послідовний	однорангова
18	Розв’язання СЛАР методом Якобі	послідовний	майстер-робітник
19	Розв’язання СЛАР методом Якобі	структура	однорангова
20	Розв’язання СЛАР методом Якобі	структура	майстер-робітник

4.5. Питання для самоконтролю

1. Яка необхідна послідовність дій для створення дескриптору типу даних в середовищі MPI?
2. Чому необхідно вказувати тип даних при передачі повідомлень?
3. Чим відрізняються між собою послідовні, векторні та індексовані типи?
4. Чим структура відрізняється від усіх інших типів даних, які можливо сконструювати в MPI?
5. Розташуйте функції конструювання дескрипторів типів даних за зростанням можливої складності структури описуваних даних.
6. Існує процедура зниження розмірності обчислюваного визначника через розклад за елементами рядка (стовпця). Для цього необхідно обчислити алгебраїчну суму добутків визначників доповнювальних мінорів на елемент, до якого ведеться доповнення. Причому елементи, в яких номер сума номера рядка та стовпця

парна беруться додатними, а інші − від'ємними. Дані зберігаються в задачі 0. Попередній аналіз будуть виконувати задача 1 всіх «додатних» значень та задача 2 всіх «від'ємних» значень. Запропонуйте тип даних для ефективної розсилки при парних N.

7. Запропонуйте тип даних для ефективної розсилки у попередньому питанні при непарних N.
8. Однозв'язний список − це структура даних, що складається з елементів, кожний з яких є структурою з поля даних та покажчика на наступний елемент. Зберігається лише перший елемент. В останньому елементі покажчик дорівнює NULL. Запропонуйте методику пересилки однозв'язного списку.
9. Яким чином переслати матрицю, що описана як структура змінної довжини ***struct matrix {int columns, rows; double elems[0]};*** ?
10. За яких умов можна передавати дані одного послідовного, індексованого або векторного типу, а приймати − іншого?
11. Яким чином і за яких умов можна передати дані з однієї структури до іншої?

Лабораторна робота 5
Приклад розробки складної програми за допомогою MPI

Мета роботи:

- продемонструвати процес створення програми для вирішення реальної математичної задачі на кластерній системі;
- показати комплексне використання раніше вивчених функцій MPI.

5.1. Математичний апарат

5.1.1. Постановка задачі

Двовимірне рівняння Пуассона являє собою диференційне рівняння у часткових похідних виду:

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = f(x, y).$$

Очевидно, що розв'язати його у аналітичному вигляді немає можливості. Одним із можливих методів розв'язання є застосування методу кінцевих різниць.

5.1.2. Метод кінцевих різниць

Метод кінцевих різниць передбачає дискретизацію диференціальних рівнянь на прямокутних координатних сітках. Для двовимірних задач елементарні комірки таких сіток є прямокутниками, а для тривимірних задач комірки представляють собою паралелепіпеди.

Кінцево-різничні сітки. Розглянемо одновимірну область Θ, що являє собою відрізок [**0; s**]. Розіб'ємо цей відрізок точками $\mathbf{x_i = ih; i = 0; 1; 2; \ldots ; n}$ на **n** рівних частин довжиною $\mathbf{h = s / n}$ кожна. Множина точок $\mathbf{G = \{x_i = ih | i = 0; 1; 2; \ldots ; n\}}$ називається рівномірною одномірною координатною сіткою, а число **h** – кроком сітки.

Відрізок [**0; s**] можна розбити на **n** частин, вводячи довільні точки $\mathbf{0 < x_1 < x_2 < \ldots < x_{i-1} < xi < x_{i+1} < \ldots < x_{n-1} < s}$.

Координатна сітка $\mathbf{G = \{x_i | i = 0; 1; 2; \ldots ; n; x_0 = 0; x_n = s\}}$ буде мати крок $\mathbf{h_i = x_i - x_{i-1}}$, що залежить від номера ***i*** вузла x_i. Якщо $\mathbf{h_i \neq h_{i+1}}$ хоча б для одного номера ***i***, координатна сітка **G** називається нерівномірною (рис. 5.1).

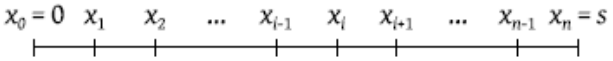


Рис. 5.1. Одновимірна сітка

Аналогічно, двовимірною сіткою називають множину точок

$$G = \{(x_i = ih_1; y_j = jh_2) | i = 0; 1; 2; \ldots ; n; j = 0; 1; 2; \ldots ; m\}.$$

Нерівномірна двовимірна сітка являє собою множину

$$G = \{(x_i; y_j) | i = 0; 1; 2; \ldots ; n; j = 0; 1; 2; \ldots ; m; x_0 = 0; x_n = s_x; y_0 = 0; y_m = s_y\}$$

(рис. 5.2).

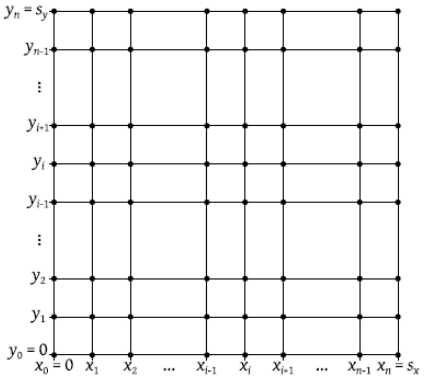


Рис. 5.2. Двовимірна сітка

Сіткові функції, кінцеві різниці. Введення для області _ координатної сітки G передбачає що значення всіх змінних та їх похідних розглядаються тільки у вузлах цієї сітки. З метою виконання цієї умови всі змінні задані сітковими функціями, а похідні будь-якого порядку − кінцевими різницями.

Нехай для деякої області Θ задана сітка

$$G = \{(x_i; y_j) | i = 0; 1; 2; \ldots ; n; j = 0; 1; 2; \ldots ; m; x_0 = 0; x_n = s_x; y_0 = 0; y_m = s_y\}.$$

Тоді функцію Φ = Φ (x_i; y_j); i = 0; 1; 2; . . . ; n; j = 0; 1; 2; . . . ;m дискретного аргументу (x_i; y_j) називають сітковою функцією, визначеною на сітці **G.** Будь-якій неперервній функції ***f(x; y)***, заданій в області Θ, можна поставити у відповідність сіткову функцію **Φ(x_i; y_j)** (для зручності будемо позначати **Φij**), задану на сітці

$$G = \{(x_i; y_j) | i = 0; 1; 2; \ldots ; n; j = 0; 1; 2; \ldots ; m; x_0 = 0; x_n = s_x; y_0 = 0; y_m = s_y\}$$

(спроектувати функцію ***f(x; y)*** на сітку **G**), приймаючи до уваги певне правило співвідношення. Наприклад:

$$\Phi_{ij} = f(x_i, y_j)$$

$$\Phi_{ij} = \frac{1}{(x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}})(y_{i+\frac{1}{2}} - y_{i-\frac{1}{2}})} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{i-\frac{1}{2}}}^{y_{i+\frac{1}{2}}} f(x, y)\, dx\, dy,$$

де $x_{i+1/2}$ $y_{i+1/2}$ − координати серединних точок на відповідних кроках координатної сітки, що визначаються виразами:

$$x_{i+\frac{1}{2}} = \frac{x_{i+1} + x_i}{2}$$
$$x_{i-\frac{1}{2}} = \frac{x_i + x_{i-1}}{2}$$
$$y_{i+\frac{1}{2}} = \frac{y_{i+1} + y_i}{2}$$
$$y_{i-\frac{1}{2}} = \frac{y_i + y_{i-1}}{2}$$

Слід мати на увазі, що одна й та ж сіткова функція, задана на двох різних сітках, що мають спільні вузли, не обов'язково буде мати в цих вузлах рівні значення.

За визначенням похідна функції неперервного аргументу x у точці x_0 є ліміт відношення приросту функції до приросту аргументу, коли приріст аргументу близиться до нуля:

$$\frac{\partial f(x)}{\partial x} = \lim_{(x-x_0)\to 0} \frac{f(x) - f(x_0)}{x - x_0}.$$

Знехтувавши границею, похідну функції неперервного аргументу можна приблизно замінити (апроксимувати) різницевим виразом, заданим на відповідній сітковій функції $\mathbf{\Phi(x_i; y_j)}$. Дана апроксимація може бути виконана декількома способами:

$$\frac{\partial f(x,y)}{\partial x} \approx \frac{\Phi_{i+1,j} - \Phi_{i,j}}{\Delta x_i},$$
$$\frac{\partial f(x,y)}{\partial x} \approx \frac{\Phi_{i,j} - \Phi_{i-1,j}}{\Delta x_i},$$
$$\frac{\partial f(x,y)}{\partial y} \approx \frac{\Phi_{i,j+1} - \Phi_{i,j}}{\Delta y_j},$$
$$\frac{\partial f(x,y)}{\partial y} \approx \frac{\Phi_{i,j} - \Phi_{i,j-1}}{\Delta y_j},$$

де Δx_i;Δy_j − кінцеві різниці координат, що визначаються виразами:

$$\boldsymbol{\Delta xi = x_{i+1} - x_i;}$$
$$\boldsymbol{\Delta yj = y_{j+1} - y_j :}$$

Для кожного з перетворень характерна така похибка апроксимації, що прямує до нуля коли крок сітки прямує до нуля.

Похідні другого порядку апроксимуються наступним чином:

$$\frac{\partial^2 f(x,y)}{\partial x^2} \approx \frac{\frac{\Phi_{i+1,j}-\Phi_{i,j}}{\Delta x_i} - \frac{\Phi_{i,j}-\Phi_{i-1,j}}{\Delta x_{i-1}}}{\frac{\Delta x_i+\Delta x_{i-1}}{2}}$$
$$\frac{\partial^2 f(x,y)}{\partial y^2} \approx \frac{\frac{\Phi_{i,j+1}-\Phi_{i,j}}{\Delta y_j} - \frac{\Phi_{i,j}-\Phi_{i,j-1}}{\Delta y_{j-1}}}{\frac{\Delta y_j+\Delta y_{j-1}}{2}}$$

У випадку коли сітка рівномірна вирази спрощуються:

$$\frac{\partial^2 f(x,y)}{\partial x^2} \approx \frac{\Phi_{i+1,j} - 2\Phi_{i,j} + \Phi_{i-1,j}}{\Delta x^2}$$
$$\frac{\partial^2 f(x,y)}{\partial y^2} \approx \frac{\Phi_{i,j+1} - 2\Phi_{i,j} + \Phi_{i,j-1}}{\Delta y^2}$$

5.2. Аналіз задачі

Застосувавши вище приведений метод до нашої задачі, ми можемо визначити структуру обчислювальної моделі та формули, за якими будемо вести розрахунок.

Структура обчислювальної моделі зображена на рис. 5.3. Сітка дискретизації квадратна та рівномірна.

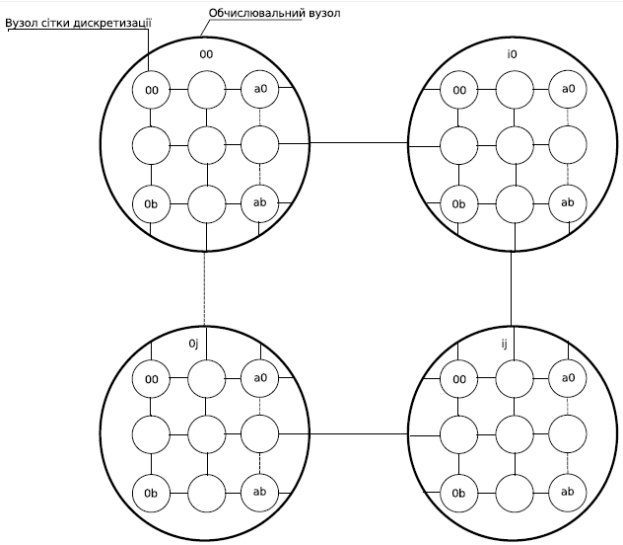


Рис. 5.3. Структура обчислювальної моделі

У кожному обчислювальному вузлі буде послідовно розраховуватися матриця точок сітки дискретизації. Реальні вузли будуть працювати паралельно між собою та обмінюватися даними.
Нехай початкові умови функції всередині області $U_0 = 0$, а на границі області

$$U_0 = (1 + 3\pi) \sin(3\pi(x + y)).$$

Тоді головна ітераційна формула матиме вигляд:

$$U_{nm}(z) = U_{nm}(z-1) - \omega\frac{h^2}{4}(LU_{nm} - f_{nm}),$$

де:

- ***n; m*** − координати вузла сітки дискретизації;
- ***z*** − номер ітерації;
- ***h*** − крок у просторі, що визначається за формулою **h = 1/N** ;
- ***N*** − кількість вузлів сітки дискретизації вздовж однієї з координат;
- f_{nm} − значення правої частини диференційного рівняння у вузлі сітки дискретизації;
- LU_{nm} − різницевий оператор, що визначається за формулою:

$$LU_{nm} = U_{nm} - \frac{U_{n+1,m} - 2U_{n,m} + U_{n-1,m}}{h^2} - \frac{U_{n,m+1} - 2U_{n,m} + U_{n,m-1}}{h^2}.$$

Локальним критерієм завершення алгоритму слугує досягнення заданої точності:

$$\frac{U_{nm}(z) - U_{nm}(z-1)}{U_{nm}(z)} \leq \varepsilon,$$

де ε задається до початку розрахунку.

5.3. Аналіз програмної реалізації

Спрощено алгоритм основного циклу програми виглядає наступним чином:

1. Зробити розрахунки для тих вузлів сітки дискретизації, що не потребують даних від сусідніх процесів.
2. Отримати колонку сітки дискретизації від лівого сусіднього процесу, якщо він є.
3. Передати крайню праву колонку сітки дискретизації до правого сусіднього процесу, якщо він є.
4. Отримати колонку сітки дискретизації від правого сусіднього процесу, якщо він є.
5. Передати крайню ліву колонку сітки дискретизації до лівого сусіднього процесу, якщо він є.

6. Отримати рядок сітки дискретизації від верхнього сусіднього процесу, якщо він є.
7. Передати нижній рядок сітки дискретизації до нижнього сусіднього процесу, якщо він є.
8. Отримати рядок сітки дискретизації від нижнього сусіднього процесу, якщо він є.
9. Передати верхній рядок сітки дискретизації до верхнього сусіднього процесу, якщо він є.
10. Завершити розрахунки для інших вузлів, які не були розраховані у пункті 1.
11. Перевірити локальні критерії завершення алгоритмів. Якщо усі вузли сітки у процесі завершили свою − завершити розрахунки у процесі.
12. Якщо всі процеси завершили розрахунки − завершити алгоритм.

Розглянемо деякі проблеми, зв'язані з реалізацією цього алгоритму:

- В мові С матриці зберігаються по рядкам (елементи одного рядку розташовані послідовно в пам'яті). Алгоритм розв'язання задачі передбачає передачу рядків та стовпців матриці. Елементи одного рядку можна передати за допомогою звичайного виклику **MPI_Send**, так як вони розташовані послідовно. Елементи одного стовпця таким чином неможливо передати, тому що вони розташовані в пам'яті з кроком, рівним ширині матриці. Для передачі стовпців матриці створимо тип «стовпець матриці» за допомогою функції конструктору типу **MPI_Type_vector** наступним чином:

```
MPI_Type_vector(<висота матриці>,
1, /* довжина одного блоку */
<ширина матриці>,
MPI_DOUBLE,
&column_type);
```

Для передачі стовпця матриці цей тип можна застосувати наступним чином:

```
MPI_Send(<Покажчик на перший елемент стовпця>,
1, /* кількість стовпців */
column_type,
<ранг задачі-отримувача>,
<тег>,
MPI_COMM_WORLD);
```

- Аналогічна проблема виникає при початковому розподілі координат вузлів сітки дискретизації по процесам. Фактично нам потрібно відправити квадратну підматрицю. Реалізація змінюється тільки у ширині блоку і їх кількості при конструюванні типу.

MPI_Type_vector(<висота блоку, що відправляємо>,
<ширина блоку, що відправляємо>,
<ширина матриці>,
MPI_DOUBLE,
&new_type);

- Попри те, що процес закінчив розрахунок, він усе одно повинен взаємодіяти з сусідніми процесами, щоб ті теж могли завершити розрахунки. Для цього треба реалізувати подвійну перевірку − умови зупинки розрахунку процесу та умови глобального кінця розрахунку. У програмі спочатку виконується перевірка усіх локальних умов закінчення алгоритму. Якщо усі вузли сітки дискретизації, що належать до процесу, закінчили роботу, то встановлюється ознака закінчення обчислень у процесі. Після цього усі ознаки закінчення обчислень у процесах збираються нульовим процесом и над ними виконується операція логічного «І» за допомогою **MPI_Reduce**.

MPI_Reduce(&node_stop,
&global_stop,
1,
MPI_SHORT,
MPI_LAND,
0,
MPI_COMM_WORLD);

Результат цієї операції є глобальною ознакою закінчення алгоритму, оскільки він матиме значення true тільки коли усі процеси закінчили обчислення. Після цього глобальна ознака закінчення обчислення розповсюджується між усіма процесами за допомогою MPI_Bcast.

5.3.1. Код програми

Увага! Перед запуском прикладів коду на кластері ознайомтесь з інструкціями в додатку Б.

Файл code/lab_diffeq/main.c

```
#define _GNU_SOURCE
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
/* Опис типів, структур та функцій */
```

```
struct my_matrix          /* Структура, що описує матриці */
{
   int rows;              /* Кількість строк у матриці */
   int cols;              /* Кількість стовпців у матриці */
   double *data;          /* Покажчик на матрицю, розвернуту у одномір-
ний масив*/
};
MPI_Datatype node_mat_t; /* Тип для розповсюдження частин сітки
дискретизації по матриці процесів*/
MPI_Datatype mat_col;      /* Тип для передачі стовпця матриці */
void inline evaluate(int);      /*Функція, що містить ітераційні формули */
double func(double X,double Y);/* Повертає значення правої частини
диференційного рівняння у точці X,Y*/
void init_grid();          /* Ініціалізує значення сітки дискретизації у про-
цесі */
void fatal_error(const char *message, int errorcode); /* Функція виведення
помилок вводу-виводу */
struct my_matrix *matrix_alloc(int rows, int cols, double initial);    /* Функ-
ція
для виділення пам'яті та ініціалізацію матриці*/
void matrix_print(const char *filename, struct my_matrix *mat);          /*
Функція
для виведення матриці у файл*/
struct my_matrix *read_matrix(const char *filename);                /* Функ-
ція
для зчитування матриці із файлу*/
/*
**********************************************************************
******** */
/* Опис змінних */
int np;                    /* Кількість процесів */
int rank;              /* Ранг процесу у MPI_COMM_WORLD */
int nodes_width;           /* Ширина матриці процесів */
int node_X;                /* Координата X у матриці процесів */
int node_Y;                /* Координата Y у матриці процесів */
int total_width;           /* Ширина матриці вузлів сітки дискретизації*/
int points_per_node;           /* Ширина частини матриці вузлів сітки дис-
кретизації,
що припадає на кожен процес*/
const char* inpfileX="inpX.csv"; /* Ім'я файлу з якого вводяться горизон-
тальні
```

```
координати сітки дискретизації */
const char* inpfileY=”inpY.csv”; /* Ім’я файлу з якого вводяться вертикальні
координати сітки дискретизації */
const char* inpfileInit=”inpInit.csv”;/*Ім’я файлу з якого вводяться початкові
та граничні умови для сітки дискретизації*/
const char* outpfile=”outp.csv”;/* Ім’я файлу куди виводиться результат */
my_matrix* allcoords_X;  /* Матриця реальних горизонтальних координат вузлів
сітки дискретизації */
my_matrix* allcoords_Y;  /* Матриця реальних вертикальних координат вузлів сітки
дискретизації */
my_matrix* all_grid_init;  /*Всі початкові та граничні умови функції*/
my_matrix* total_mat;  /* Повна сітка дискретизації, що виводиться у процесі 0*/
my_matrix* grid_init;   /*Частина початкових умов для процесу*/
my_matrix* coords_X;  /* Частина матриці реальних координат сітки дискретизації,
яка зберігається у кожному з процесів*/
my_matrix* coords_Y;  /* Частина матриці реальних координат сітки дискретизації,
яка зберігається у кожному з процесів*/
my_matrix* node_mat;  /* Частина сітки дискретизації, яка зберігається у кожному
з процесів*/
my_matrix* f_xy;   /* Масив значень функції в вузлах сітки дискретизації*/:
MPI_Status stat;    /* Змінна, у яку повертається статус прийому повідомлень */
double epsilon=0.01;  /* Точність рішення*/
double omega=0.4;        /* Коефіцієнт релаксації*/
int max_iter=1000000; /*Максимальна кількість ітерацій у випадку нев’язки функції */
double h;         /* Крок сітки дискретизації */
bool* local_stop;    /* Масив для запам’ятовування локальних умов зупинки */
int* node_iter;
short node_stop;    /* Ознака завершення обчислень у процесі*/
```

```
short global_stop;  /* Глобальна ознака завершення обчислень*/
double* left_col;   /* Стовпці, що процес буде приймати з лівого та правого
процесів під час ітерацій */
double* right_col;
double* top_row;  /* Рядки, що процес буде приймати з верхнього та нижнього
процесів під час ітерацій */
double* bot_row;
/*
*****************************************************************
******** */
/* Головна програма */
int main(int argc, char *argv[])
{
   MPI_Init(&argc,&argv);                /* Ініціалізуємо середовище */
   MPI_Comm_size(MPI_COMM_WORLD,&np);/* Отримуємо розмір
MPI_COMM_WORLD */
   MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* Отримуємо ранг
процесу у MPI_COMM_WORLD*/
   nodes_width=sqrt(np);       /* Отримуємо ширину матриці процесів*/
   node_X=rank%nodes_width; /* Отримуємо горизонтальну координату
процесу у матриці */
   node_Y=rank/nodes_width;      /* Отримуємо вертикальну координату
процесу у матриці */
   if (rank==0)
  {
          allcoords_X=read_matrix(inpfileX);            /* Вводимо координати
          */
          allcoords_Y=read_matrix(inpfileY);            /* Вводимо координати
          */
          all_grid_init=read_matrix(inpfileInit);    /* Вводимо координати */
          total_width=allcoords_X->rows;    /* Отримуємо ширину сітки
          дискретизації */
   }
   MPI_Bcast(&total_width,1,MPI_INT,0,MPI_COMM_WORLD); * Розповсюджуємо ширину
сітки дискретизації */
   h=1.0/total_width;            /* Визначаємо крок сітки дискретизації */
            points_per_node=total_width/nodes_width;
```

```
   /* Створюємо тип, що відповідає частині матриці, яка буде передаватися */

MPI_Type_vector(points_per_node,points_per_node,total_width,MPI_DOUBLE
,&node_mat_t);
   MPI_Type_commit(&node_mat_t);   /* Реєструємо тип */
   coords_X=matrix_alloc(points_per_node,points_per_node,0.0);
   coords_Y=matrix_alloc(points_per_node,points_per_node,0.0);
   grid_init=matrix_alloc(points_per_node,points_per_node,0.0);
   if(rank==0)
            {   /* Процес 0 розсилає реальні координати сітки дискретизації*/
         for(int i=0;i<nodes_width;i++)
         {
            for(int j=0;j<nodes_width;j++)
            {
               if(!(i==0&&j==0))
               {
                     MPI_Send(allcoords_X-
                     >data+i*total_width*points_per_node+j*points_per_node
                     ,1,node_mat_t,i*nodes_width+j,0,MPI_COMM_WORLD);
                     MPI_Send(allcoords_Y-
                     >data+i*total_width*points_per_node+j*points_per_node
                     ,1,node_mat_t,i*nodes_width+j,0,MPI_COMM_WORLD);
                     MPI_Send(all_grid_init-
                     >data+i*total_width*points_per_node+j*points_per_node
                     ,1,node_mat_t,i*nodes_width+j,0,MPI_COMM_WORLD);
               }
            }
         }
         for(int i=0;i<points_per_node;i++)
         {
            for(int j=0;j<points_per_node;j++)
            {
               coords_X->data[i*points_per_node+j]=allcoords_X-
         >data[i*total_width+j];
               coords_Y->data[i*points_per_node+j]=allcoords_Y-
         >data[i*total_width+j];
               grid_init->data[i*points_per_node+j]=all_grid_init-
         >data[i*total_width+j];
            }
```

```
        }
          }
else
{        /* Інші процеси їх приймають */
        my_matrix* temp_X=matrix_alloc(total_width,points_per_node,0);
        my_matrix* temp_Y=matrix_alloc(total_width,points_per_node,0);
        my_matrix* temp_init=matrix_alloc(total_width,points_per_node,0);
        MPI_Recv(temp_X-
        >data,1,node_mat_t,0,0,MPI_COMM_WORLD,&stat);
        MPI_Recv(temp_Y-
        >data,1,node_mat_t,0,0,MPI_COMM_WORLD,&stat);
        MPI_Recv(temp_init-
        >data,1,node_mat_t,0,0,MPI_COMM_WORLD,&stat);
        for(int i=0;i<points_per_node;i++)
        {
           for(int j=0;j<points_per_node;j++)
           {
              coords_X->data[i*points_per_node+j]=temp_X-
        >data[i*total_width+j];
              coords_Y->data[i*points_per_node+j]=temp_Y-
        >data[i*total_width+j];
              grid_init->data[i*points_per_node+j]=temp_init-
        >data[i*total_width+j];
           }
        }
}
/* Ініціалізуємо умови локальної зупинки алгоритму*/
local_stop=(bool*)malloc(points_per_node*points_per_node*sizeof(bool));
for(int i=0;i<points_per_node*points_per_node;i++)
{
        local_stop[i]=false;
}
           for(int i=0;i<points_per_node;i++)    /* Граничні точки залиша-
        ються постійними*/
{
        if(node_Y==0)
        {
           local_stop[i]=true;
        }
        if(node_X==0)
        {
```

```
            local_stop[i*points_per_node]=true;
        }
        if(node_X==nodes_width-1)
        {
            local_stop[(i+1)*points_per_node-1]=true;
        }
        if(node_Y==nodes_width-1)
        {
            local_stop[points_per_node*(points_per_node-1)+i]=true;
        }
            }
    /* Ініціалізуємо і встановлюємо початкові значення внутрішнього діа-
пазону */
    node_mat=matrix_alloc(points_per_node,points_per_node,0);
    /* Встановлюємо початкові значення граничних вузлів сітки дискрети-
зації */
    for(int i=0;i<points_per_node*points_per_node;i++)
    {
        node_mat->data[i]=grid_init->data[i];
    }
    /* Створимо масив значень функції в локальних вузлах сітки дискрети-
зації*/
    f_xy=matrix_alloc(points_per_node,points_per_node,0.0);
    for(int i=0;i<points_per_node*points_per_node;i++)
    {
        f_xy->data[i]=func(coords_X->data[i],coords_Y->data[i]);
    }
    /*Визначимо типи для передач під час ітерацій*/
    /* Тип, що визначає стовпець матриці*/

MPI_Type_vector(points_per_node,1,points_per_node,MPI_DOUBLE,&mat_
col);
    MPI_Type_commit(&mat_col);/* Реєструємо тип */

left_col=(double*)malloc(points_per_node*points_per_node*sizeof(double));

right_col=(double*)malloc(points_per_node*points_per_node*sizeof(double))
;

top_row=(double*)malloc(points_per_node*points_per_node*sizeof(double));
```

```
bot_row=(double*)malloc(points_per_node*points_per_node*sizeof(double));
   node_iter=(int*)malloc(points_per_node*points_per_node*sizeof(int));
   for(int i=0;i<points_per_node*points_per_node;i++)
   {
         node_iter[i]=0;
   }
   /* Початок ітерацій */
   do
   {
         /* Обмін між процесами у сітці*/
         if(node_X!=0)
         {
           /* Приймемо стовпець з лівого процесу */
           MPI_Recv(left_col,1,mat_col,rank-1,rank-
         1,MPI_COMM_WORLD,&stat);
         }
         if(node_X!=nodes_width-1)
         {
           /* Відішлемо стовпець до правого процесу */
          MPI_Send(node_mat->data+points_per_node-1,1,mat_col,
                  rank+1,rank,MPI_COMM_WORLD);
           /* Приймемо стовпець з правого процесу */

         MPI_Recv(right_col,1,mat_col,rank+1,rank+1,MPI_COMM_WORL
         D,&stat);
         }
         if(node_X!=0)
         {
                  /* Відішлемо стовпець до лівого процесу */
           MPI_Send(node_mat->data,1,mat_col,rank-
         1,rank,MPI_COMM_WORLD);
      }
      if(node_Y!=0)
         {
           /* Приймемо строку з верхнього процесу */
           MPI_Recv(top_row,points_per_node,MPI_DOUBLE,rank-
         nodes_width,
           rank-nodes_width,MPI_COMM_WORLD,&stat);
         }
         if(node_Y!=nodes_width-1)
```

```
{
   /* Відішлемо строку до нижнього процесу */
   MPI_Send(node_mat->data+(points_per_node-
1)*points_per_node,points_per_node
   ,MPI_DOUBLE,rank+nodes_width,rank,MPI_COMM_WORLD);
   /* Приймемо строку з нижнього процесу */

MPI_Recv(bot_row,points_per_node,MPI_DOUBLE,rank+nodes_wid
th,
   rank+nodes_width,MPI_COMM_WORLD,&stat);
}
if(node_Y!=0)
{
   /* Відішлемо строку до верхнього процесу */
   MPI_Send(node_mat->data,points_per_node,MPI_DOUBLE,
   rank-nodes_width,rank,MPI_COMM_WORLD);
}
/* Виклик функції, що проводить ітераційну обробку*/
for(int i=0;i<points_per_node;i++)
{
   for(int j=0;j<points_per_node;j++)
   {
      evaluate(i*points_per_node+j);
   }
}
/*Якщо усі вузли сітки дискретизації у процесі завершили
обчислення - встановлюємо ознаку завершення обчислень у проце-
сі*/
node_stop=1;
for(int i=0;i<points_per_node*points_per_node;i++)
{
   if (!local_stop[i])
   {
      node_stop=0;
      break;
   }
}
global_stop=1;
/*Перевіряємо чи всі процеси закінчили обчислення*/
MPI_Reduce(&node_stop,&global_stop,1,MPI_SHORT,MPI_BAND,0,MPI_
COMM_WORLD);
```

```
        /*Якщо всі - то посилаємо сигнал про завершення*/
        MPI_Bcast(&global_stop,1,MPI_SHORT,0,MPI_COMM_WORLD);
           }
           while(!global_stop);
/* Збираємо результат */
total_mat=matrix_alloc(total_width,total_width,0.0);
if(rank==0)
{
        for(int i=0;i<nodes_width;i++)
        {
           for(int j=0;j<nodes_width;j++)
           {
              if(!(i==0&&j==0))
                   {
                      my_matrix*
                   temp=matrix_alloc(points_per_node,points_per_node,0.0);
                      /* Приймаємо частини матриці сітки дискретизації з
                   кожного процесу */
                      MPI_Recv(temp->data,points_per_node*points_per_node,

                   MPI_DOUBLE,i*nodes_width+j,i*nodes_width+j,MPI_COMM_
                   WORLD,&stat);
                      /* Та розташовуємо їх у повній матриці */
               for(int k=0;k<points_per_node;k++)
                      {
                         for(int l=0;l<points_per_node;l++)
                              {
                                 total_mat->data[i*total_width*points_per_node+
                                 j*points_per_node+k*total_width+l]
                                 =temp->data[k*points_per_node+l];
                              }
                      }
                   }
           }
        }
        for(int i=0;i<points_per_node;i++)
        {
           for(int j=0;j<points_per_node;j++)
              {
                   total_mat->data[i*total_width+j]=node_mat-
                   >data[i*points_per_node+j];
```

```
                        }
                }
        }
        else
        {
                MPI_Send(node_mat->data,points_per_node*points_per_node
                ,MPI_DOUBLE,0,rank,MPI_COMM_WORLD);
        }
        if (rank==0)
        {
                matrix_print(outpfile,total_mat);;
        }
        MPI_Finalize();
}
/*
*****************************************************************
********* */
/* Допоміжні функції */
double func(double X,double Y)
{
        return X*Y;
}
/*Функція, що містить ітераційні формули */
void inline evaluate(int index)
{
        /*Якщо досягнута потрібна точність, то не потрібно проводити об-
числення */
        if (local_stop[index])
        {
                return;
        }
        /*Якщо вузол сітки знаходиться на границі між обчислювальними
         вузлами, то потрібно взяти значення з буферу передачі*/
        double left=index%points_per_node==0
        ?left_col[index]:node_mat->data[index - 1];
        double right=(index + 1)%points_per_node==0
        ?right_col[index+1-points_per_node]:node_mat->data[index + 1];
        double top=index/points_per_node == 0 ? top_row[index %
points_per_node]
        :node_mat->data[index-points_per_node];
        double bottom=index/points_per_node==points_per_node-1
```

```
    ?bot_row[index%points_per_node]
     :node_mat->data[index+points_per_node];
    /*Власне обчислення*/
    double old_node=node_mat->data[index];
    double LU=old_node-(left+right+top+bottom-4*old_node)/(h*h);
    node_mat->data[index]=old_node-omega*h*h/4*(LU-f_xy->data[index]);
    /*Якщо досягнута точність або перевищена максимальна кількість
ітерацій,
    то завершуємо обчислення у цьому вузлі сітки*/
    if (fabs(node_mat->data[index]-old_node) <=
   fa bs(node_mat->data[index]*epsilon)||++node_iter[index]>max_iter)
    {
            local_stop[index]=true;
    }
}
/* Ініціалізує значення сітки дискретизації у процесі */
void init_grid()
{
    for(int i=0;i<points_per_node*points_per_node;i++)
    {
            node_mat->data[i]=grid_init->data[i];
    }
}
void fatal_error(const char *message, int errorcode)
{
    printf("fatal error: code %d, %s\n", errorcode, message);
    fflush(stdout);
    MPI_Abort(MPI_COMM_WORLD, errorcode);
}
    struct my_matrix *matrix_alloc(int rows, int cols, double initial)
{
    struct my_matrix *result = (my_matrix*)malloc(sizeof(struct my_matrix));
    result->rows = rows;
    result->cols = cols;
    result->data = (double*)malloc(sizeof(double) * rows * cols);
    for(int i = 0; i < rows; i++)
            {
            for(int j = 0; j < cols; j++)
               {
                  result->data[i * cols + j] = initial;
               }
```

```
        }
        return result;
            }
    void matrix_print(const char *filename, struct my_matrix *mat)
    {
        FILE *f = fopen(filename, "w");
        if(f == NULL)
        {
            fatal_error("cant write to file", 2);
        }
        for(int i = 0; i < mat->rows; i++)
        {
            for(int j = 0; j < mat->cols; j++)
            {
                fprintf(f, "%lf ", mat->data[i * mat->cols + j]);
            }
            fprintf(f, "\n");
        }
        fclose(f);
    }
struct my_matrix *read_matrix(const char *filename)
{
    FILE *mat_file = fopen(filename, "r");
    if(mat_file == NULL)
    {
        fatal_error("can't open matrix file", 1);
    }
    int rows;
    int cols;
    fscanf(mat_file, "%d %d", &rows, &cols);
    struct my_matrix *result = matrix_alloc(rows, cols, 0.0);
    for(int i = 0; i < rows; i++)
    {
        for(int j = 0; j < cols; j++)
        {
            fscanf(mat_file, "%lf", &result->data[i * cols + j]);
        }
    }
    fclose(mat_file);
    return result;
}
```

5.4. Варіанти

Варіант визначається по табл. 5.1.

Табл. 5.1. Варіанти завдання

№	Тип пере-силок	Тип просто-ру	Тип сітки
1	Блокуючі	Двовимірний	Однорідна
2	Неблокуючі	Двовимірний	Однорідна
3	Блокуючі	Двовимірний	Неоднорідна
4	Неблокуючі	Двовимірний	Неоднорідна
5	Блокуючі	Тривимірний	Однорідна
6	Неблокуючі	Тривимірний	Однорідна
7	Блокуючі	Тривимірний	Неоднорідна
8	Неблокуючі	Тривимірний	Неоднорідна

ДОДАТОК А
Робота з кластером по протоколу SSH

Навчальна і основна кластерні обчислювальні системи Центру суперкомп'ютерних обчислень НТУУ «КПІ» працюють під управлінням операційної системи Linux. В Linux стандартним способом віддаленого доступу є протокол Secure Shell (скорочено SSH). Віддалений доступ можливий як через Інтернет, так і з мережі НТУУ «КПІ».
Найбільш популярні програми-клієнти, які дозволяють встановлювати зв'язок за протоколом SSH − це утиліти ssh в Linux та PuTTY для Windows. Розглянемо роботу з програмою PuTTY. Завантажити програму можна з сайту http://www.putty.org/.
Для запуску задач на кластерній системі необхідно мати акаунт на ній. Акаунт створюється адміністратором по заявці. Вам мають бути повідомлені наступні дані:

- логін вашого акаунта (далі user);
- пароль вашого акаунта (далі pass);
- IP-адресу кластеру або його символьне ім'я (далі host);
- порт, на який встановлюється SSH-з'єднання (далі port).

А.1. Віддалений доступ до командного рядку
А.1.1. Віконний варіант PuTTY

Після інсталяції програми PuTTY можна користуватись як віконним варіантом програми, так і консольним. Обидва варіанти представлені виконуваним файлом **putty.exe**. Запуск цього файлу без параметрів відкриває головне вікно програми (**Пуск → Програми → PuTTY → PuTTY**).
Віконна версія програми має перевагу в тому, що вона дозволяє зберігати «сесію» − адресу, порт серверу і тип підключення. Для створення сесії необхідно у головному вікні (рис. А.1) ввести адресу сервера host в поле «Host Name», порт port в поле «Port» і обрати тип з'єднання SSH. В поле «Saved Sessions» необхідно ввести назву сесії (довільне ім'я) і натиснути кнопку «Save». Введене символьне ім'я з'явиться в списку нижче і відтепер завжди можна завантажити збережені параметри обравши зі списку відповідне символьне ім'я і натиснувши кнопку «Load».
Після завантаження параметрів для встановлення з'єднання з сервером необхідно натиснути кнопку «Open». Після цього має відкритись вікно консолі (рис. А.2).

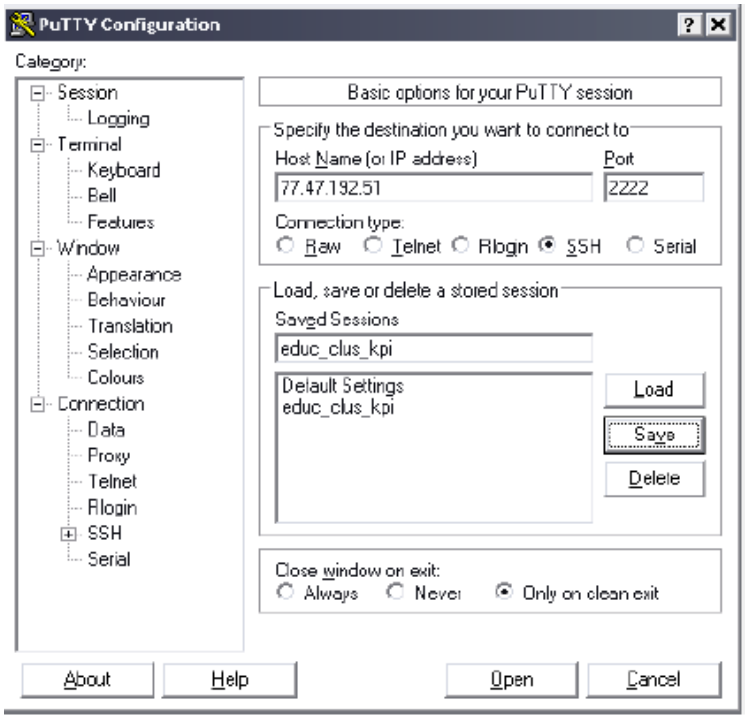


Рис. А.1. Головне вікно PuTTY

Рис. А.2. Вікно консолі

Якщо консоль з'явилась, а натомість програма повідомила про помилку, то це означає, що або не працює мережа, або не працює сервер, або ви неправильно ввели надані вам дані.
В даній консолі необхідно ввести ваш user, а потім ваш pass (введені символи паролю не відображається на екрані з метою забезпечення безпеки). Після цього ви отримаєте доступ до конмандного рядку, а поточним каталогом стане домашній каталог користувача.

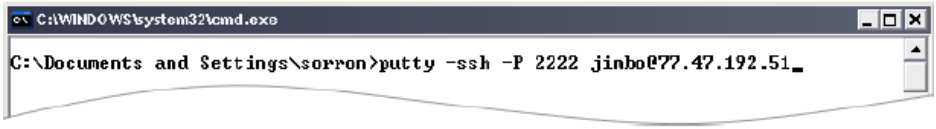


Рис. А.3. Запуск PuTTY з командного рядку Windows

А.1.2. Консольний варіант PuTTY

Розглянемо запуск консольного варіанту програми PuTTY. Вибираємо **Пуск → Виконати**, набираємо в полі «cmd» і натискаємо клавішу **Enter**. З'явиться системна консоль ***(рис. А.3),*** звідки можна запустити програму. Для встановлення з'єднання необхідно ввести командний рядок у наступному форматі:

C:\>putty -ssh -P port user@host

Після цього натискаємо Enter і з'являється вікно віддаленої консолі обчислювального вузла, де необхідно ввести пароль. Після вводу коректного паролю відбувається вхід у систему і перехід до домашньої директорії користувача:

login as: user
user@77.47.192.51's password:
Last login:
user@n001$ pwd
/home/user
user@n001$

Необхідно зауважити, що таким чином відкривається доступ до першого (головного) вузла кластеру. Щоб перейти на інший вузол необхідно дати команду:

user@n001$ ssh n<nodenumber>

де <nodenumber> − номер необхідного вузла.

Приклад переходу на вузол n002:

user@n001$ ssh n002
user@n002's password:
Last login:
user@n002$

А.2. Робота з файлами на кластері

Для повноцінної роботи з кластером необхідна також можливість завантажувати файли на кластер і з кластеру. Для цього разом з програмою PuTTY встановлюється утиліта **pscp.exe** (PuTTY secure copy client).

Зверніть увагу! Програма **pscp.exe** працює тільки в консолі Windows. Необхідно виконати команду **Пуск → Виконати**, набрати в полі «**cmd**» і

натиснути клавішу **Enter**. З'явиться системна консоль, звідки можна запускати програму **pscp.exe**.

Перегляд списку файлів. Для виведення списку файлів в заданій директорії на обчислювальному вузлі призначена команда консолі Windows:

C:\>pscp -P 2222 -ls user@host:directory

Після вводу кожної команди відбувається запит пароля користувача.

Приклад виконання команди:

```
C:\>pscp -P 2222 -ls user@77.47.192.51:/home/user
user@77.47.192.51's password:
Listing directory /home/user
drwx------ 4 user user 4096 Apr 21 20:14 .
drwxr-xr-x 83 root root 4096 Apr 26 13:22 ..
-rw------- 1 user user 432 Apr 27 06:46 .bash_history
-rw-r--r-- 1 user user 33 Apr 20 11:54 .bash_logout
-rw-r--r-- 1 user user 176 Apr 20 11:54 .bash_profile
-rw-r--r-- 1 user user 124 Apr 20 11:54 .bashrc
drwx------ 2 user user 4096 Apr 21 19:18 .ssh
-rw------- 1 user user 704 Apr 21 20:14 .viminfo
drwxr-xr-x 2 user user 4096 Apr 21 20:14 gmptest
C:\>
```

Копіювання файлів на кластер. Для копіювання одного файлу на обчислювальний вузол введіть команду в консолі Windows:

C:\>pscp -P port localfile user@host:file

де:

• localfile − ім'я файлу на локальному комп'ютері;

• file − ім'я файлу на обчислювальному вузлі.

Приклад виконання команди:

```
C:\>pscp -P 2222 C:\foo.txt user@77.47.192.51:/home/user/foo.txt
user@77.47.192.51's password:
foo.txt | 0 kB | 0.0 kB/s | ETA: 00:00:00 | 100%
C:\>
```

Перевіримо наявність файлу на обчислювальному вузлі командою **ls**:

```
user@n001$ ls
foo.txt gmptest
user@n001$
```

Також на кластер можна скопіювати каталог і всі його підкаталоги. Для цього слід додати параметр ***-r***:

C:\>pscp -P port -r localdirectory user@host:directory

де:

- **localdirectory** − ім'я директорії на локальному комп'ютері, яка буде скопійована;

- **directory** − ім’я директорії на сервері в яку буде скопійовано локальну директорію.

Приклад:

C:\dir>dir

Содержимое папки **C:\dir**

27.04.2010 07:23 <DIR> .
27.04.2010 07:23 <DIR> ..
27.04.2010 07:23 46 foo1.txt
27.04.2010 07:23 <DIR> innerdir
1 файлов 46 байт
3 папок 1 773 776 896 байт свободно
C:\dir>cd innerdir
C:\dir\innerdir>dir

Содержимое папки ***C:\dir\innerdir***

27.04.2010 07:23 <DIR> .
27.04.2010 07:23 <DIR> ..
27.04.2010 07:23 46 foo2.txt
1 файлов 46 байт
2 папок 1 773 776 896 байт свободно
C:\dir\innerdir>pscp -P 2222 -r C:\dir user@77.47.192.51:/home/user/
user@77.47.192.51’s password:
foo1.txt | 0 kB | 0.0 kB/s | ETA: 00:00:00 | 100%
foo2.txt | 0 kB | 0.0 kB/s | ETA: 00:00:00 | 100%
C:\dir\innerdir>

Перевіримо наявність каталогу на обчислювальному вузлі командою **ls**:

user@n001:~$ ls
dir foo.txt gmptest
user@n001:~/dir$ cd dir
user@n001$ ls
foo1.txt innerdir
user@n001:~/innerdir$ cd innerdir
user@n001:~/innerdir$ ls
foo2.txt
user@n001:~/innerdir$

Копіювання файлів з кластеру. Для копіювання файлів з обчислювального вузла на локальний комп’ютер призначена команда:

C:\>pscp -P port user@host:file localfile

Приклад роботи команди:

C:\dir>pscp -P 2222 user@77.47.192.51:/home/user/dir/innerdir/foo2.txt
C:\dir\foo3.txt
user@77.47.192.51’s password:

foo3.txt | 0 kB | 0.0 kB/s | ETA: 00:00:00 | 100%
C:\dir>dir
Содержимое папки C:*dir*
27.04.2010 07:23 <DIR> .
27.04.2010 07:23 <DIR> ..
27.04.2010 07:23 46 foo1.txt
27.04.2010 07:51 46 foo3.txt
27.04.2010 07:23 <DIR> innerdir
2 файлов 92 байт
3 папок 1 773 776 896 байт свободно
C:\dir>

ДОДАТОК Б
Компіляція та запуск паралельних програм на кластері

Б.1. Система управління чергою задач
Б.1.1. Призначення системи управління чергою задач

Під час запуску програм на своєму персональному комп'ютері ви точно знаєте, скільки програм вже запущено та скільки вони використовують ресурсів комп'ютера. Виходячи з цієї інформації ви вирішуєте, коли і скільки програм запускати.
Запуск програм на кластері відрізняється від запуску програм на персональному комп'ютері: ви не знаєте, скільки яких ресурсів використовується в даний час на кожному з вузлів кластеру. Тому ви не можете самостійно обрати, на яких вузлах кластеру запускати свої програми. Для вирішення цієї проблеми використовується система управління чергою задач. Ідея такої системи полягає в наступному: всі користувачі кластеру повідомляють системі управління чергою, які програми вони бажають запустити та скільки обчислювальних ресурсів цим програмам потрібно.
Заявки від користувачів кластеру кладуться в чергу. Система управління чергою самостійно запускає програми з черги тоді, коли може задовольнити потреби програми в ресурсах. Таким чином, система управління чергою має повну інформацію про те, які ресурси кластеру зайняті (та якими саме програмами яких користувачів), а які ресурси кластеру вільні.
Виходячи зі сказаного вище, можна зробити висновок: для продуктивної роботи всіх користувачів кластеру необхідно (та в інтересах користувачів) запускати обчислювальні програми на кластері тільки через систему управління чергою задач.
Після того, як ви отримали доступ до командного рядку кластеру за допомогою протоколу **SSH**, ви можете виконувати команди на вузлі **n001**. На цьому вузлі дозволяється виконувати переміщення та копіювання даних, компіляцію програм, запуск обчислювальних програм за допомогою системи управління чергою. На інших вузлах дозволяється виконувати лише інформаційні команди під час визначення причин проблем роботи програм.
На кластері Центру суперкомп'ютерних обчислень НТУУ «КПІ» встановлена реалізація системи управління чергою задач P**BS** (англ. ***Portable Batch System***) **Torque**. Нижче розглянуті основні команди для роботи з **Torque**.

Б.1.2. Команда qsub

Команда **qsub** призначена для додавання програми в чергу виконання. Після успішного виконання програма видає на консоль ідентифікатор задачі в системі управління чергами (не плутати з рангом задачі в MPI або ідентифікатором задачі в OpenMP). Цей ідентифікатор знадобиться для відстежування стану виконання програми або для видалення програми з черги.

qsub скрипт_старту_програми

скрипт_старту_програми − ім'я файлу, в якому вказується деяка інформація для системи управління чергою задач та команди для запуску програми. Цей файл має наступний синтаксис:

#!/bin/bash
#PBS -S /bin/bash
#PBS -N <назва задачі>
#PBS -l <специфікація ресурсів>
#PBS -o <ім'я файлу для виводу stdout>
#PBS -e <ім'я файлу для виводу stderr>
cd <каталог з програмою>
<команда запуску програми>

Призначення рядків цього файлу наступне:

-S /bin/bash Параметр вказує планувальнику, що команди запуску програми слід
виконувати за допомогою оболонки /bin/bash (краще не змінюйте цей параметр).

-N <назва задачі> Параметр задає назву задачі. Назва буде відображатись у списку задач. (Необов'язковий параметр.)

-l <специфікація ресурсів> Параметр задає скільки яких ресурсів необхідно виділити програмі. Різні види ресурсів розділяються комою. Можна задавати наступні види ресурсів:

- walltime=HH:MM:SS − необхідний час виконання програми (HH − години, MM − хвилини, SS − секунди). Це реальний час виконання програми, а не машинний. Якщо програма буде виконуватись довше заявленого часу, вона буде примусово завершена.
- nodes=N:ppn=P − необхідна кількість вузлів (N) та ядер на кожному вузлі (P). Всього програмі буде виділено N _ P ядер.
- Хоча в Torque можна задавати ресурси оперативної пам'яті, на кластері Центру суперкомп'ютерних обчислень НТУУ «КПІ» оперативна пам'ять розподіляється за принципом гарантований 1 Гб на кожне обчислювальне ядро.

-o <ім'я файлу для виводу stdout> Повний шлях до файлу, в який буде записано стандартний потік виводу програми. (При звичайному запуску вручну ця інформація виводиться на консоль; необов'язковий параметр.
-e <ім'я файлу для виводу stderr> Повний шлях до файлу, в який буде записано стандартний потік помилок програми. (При звичайному запуску вручну ця інформація виводиться на консоль; необов'язковий параметр.)
Якщо не вказати імена файлів для запису виводу програми, то ці файли створюються у поточному каталозі, а імена файлів починаються з назви задачі.

Б.1.3. Команда qstat

Команда ***qstat*** призначена для спостереження за станом виконання задач.
Команду ***qstat*** можна виконувати без параметрів:

```
user@n001$ qstat
Job id Name User Time Use S Queue
------------------------ ---------------- --------------- -------- - -----
1000.pbs example.sh user 01:11:12 R batch
1001.pbs example42.sh user 01:15:01 R batch
1002.pbs run.sh user 0 Q batch
```

В даному випадку видно, що користувач додав три задачі до черги. В стовпцях вказана наступна інформація:
• **Job id** − ідентифікатори задач.
• **Name** − ім'я скрипту старту програми або назва задачі (якщо задана).
• **Time Use** − кількість використаного процесорного часу.
• **S** − стан виконання задачі: «**Q**» − чекає в черзі, «**R**» − виконується.
Також команду ***qstat*** можна виконувати з параметром ***-a*** (виводить трохи більше інформації) чи з параметром ***-f*** (виводить дуже детальну інформацію).

Б.1.4. Команда qdel

Команда qdel призначена для видалення задач із черги. Видаляти можна як
задачу, яка очікує в черзі, так і задачу, яка вже виконується. В останньому випадку
програма буде аварійно завершена.

```
user@n001$ qdel ідентифікатор_задачі
```

Де ідентифікатор_задачі, який був виданий на консоль командою qsub. Якщо ви не пам'ятаєте цього ідентифікатору, дізнайтесь його командою qstat.

Б.2. Компіляція та запуск програм на MPI

Нехай у вас є вихідні коди програми, написаної з використанням MPI. Для запуску цієї програми на кластері необхідно виконати наступні кроки:
1. Створіть окремий каталог для цієї програми. Наприклад, якщо ваш логін user
та ви хочете створити каталог example, виконайте наступну команду:
user@n001$ mkdir /home/user/example
2. Завантажте у створений каталог вихідні коди програми.
3. Перейдіть у створений каталог командою cd (англ. **Change Directory**):
user@n001$ cd /home/user/example
4. Скомпілюйте програму. Якщо програма написана стороннім розробником, разом з програмою можуть поставлятись інструкції для компіляції.
Приклади програм на мові С з даного документу, що написані з використанням MPI, компілюються наступним чином:
***user@n001$ mpicc -W -Wall -O2 -std=c99 ***
ім'я_файлу_1.c ім'я_файлу_2.c ... -o program
Де ***ім'я_файлу_1.c***, ***ім'я_файлу_2.c*** − імена всіх файлів, що входять до складу програми; program − ім'я для виконуваного файлу програми, що створить компілятор.
Якщо програма написана на мові С++, необхідно використовувати компілятор **mpic++.**
5. Створіть в каталозі програми скрипт запуску для системи управління задачами (чи створіть цей файл на вашому персональному комп'ютері та завантажте його на кластер у каталог програми). Нехай ім'я цього файлу буде ***/home/user/example/runjob.sh***. Вміст файлу може виглядати наступним чином:
#!/bin/bash
#PBS -S /bin/bash
#PBS -l nodes=7:ppn=2,walltime=02:00:00
#PBS -o /home/user/example/stdout.txt
#PBS -e /home/user/example/stderr.txt
cd /home/user/example
mpiexec ./program
Цей скрипт повідомляє Torque про те, що програмі необхідно виділити по 2 ядра на 7 вузлах, програмі знадобиться якнайбільше 2 години часу для обчислень, вивід програми буде перенаправлений у файли ***/home/user/example/stdout.txt*** та ***/home/user/example/stderr.txt*.**
Зверніть увагу! Запускаючи MPI програми на кластері через систему управління чергою задач, не треба передавати команді mpiexec додаткових параметрів ***-n*** чи ***-np***, що задають кількість задач. Ця інформація буде передана до mpiexec системою управління чергою задач автоматично.
6. Створіть або завантажте на кластер вхідні дані для програми.

7. Додайте програму в чергу виконання командою ***qsub***:
user@n001$ qsub /home/user/example/runjob.sh
8. Стан виконання задачі можна дізнатись за допомогою команди qstat.

Б.3. Компіляція та запуск програм на OpenMP

Нехай у вас є вихідні коди програми, написаної з використанням OpenMP. Для запуску цієї програми на кластері необхідно виконати наступні кроки:
1. Створіть окремий каталог для цієї програми. Наприклад, якщо ваш логін ***user*** та ви хочете створити каталог ***example***, виконайте наступну команду:
user@n001$ mkdir /home/user/example
2. Завантажте у створений каталог вихідні коди програми.
3. Перейдіть у створений каталог командою ***cd*** (англ. Change Directory):
user@n001$ cd /home/user/example
4. Скомпілюйте програму. Якщо програма написана стороннім розробником, разом з програмою можуть поставлятись інструкції для компіляції.
Приклади програм на мові С з даного документу, що написані з використанням OpenMP, компілюються наступним чином:
***user@n001$ gcc -W -Wall -O2 -std=c99 -fopenmp ***
ім'я_файлу_1.c ім'я_файлу_2.c ... -o program
Де ***ім'я_файлу_1.c***, ***ім'я_файлу_2.c*** − імена всіх файлів, що входять до складу програми; program − ім'я для виконуваного файлу програми, що створить компілятор.
Якщо програма написана на мові С++, необхідно використовувати компілятор ***g++***.
5. Створіть в каталозі програми скрипт запуску для системи управління задачами (чи створіть цей файл на вашому персональному комп'ютері та завантажте його на кластер у каталог програми). Нехай ім'я цього файлу буде ***/home/user/example/runjob.sh***. Вміст файлу може виглядати наступним чином:

```
#!/bin/bash
#PBS -S /bin/bash
#PBS -l nodes=1:ppn=8,walltime=02:00:00
#PBS -o /home/user/example/stdout.txt
#PBS -e /home/user/example/stderr.txt
cd /home/user/example
export OMP_NUM_THREADS=8
./program
```

Цей скрипт повідомляє Torque про те, що програмі необхідно виділити 8 ядер на одному вузлі, програмі знадобиться якнайбільше 2 години часу для обчислень, вивід програми буде перенаправлений у файли ***/home/user/example/stdout.txt*** та ***/home/user/example/stderr.txt***. Значення змінної **OMP_NUM_THREADS** має бути рівне кількості ядер, вказаних в параметрі ***ppn=P***.

Зверніть увагу! Запускаючи OpenMP програми на кластері через систему управління чергою задач, не слід виділяти програмі більше одного вузла, так як програми на OpenMP працюють тільки на системах зі спільною пам'яттю.

6. Створіть або завантажте на кластер вхідні дані для програми.
7. Додайте програму в чергу виконання командою ***qsub***:

user@n001$ qsub /home/user/example/runjob.sh

8. Стан виконання програми можна дізнатись за допомогою команди ***qstat***.

Б.4. Компіляція програми, що використовує TBB

Для компіляції програми з використанням TBB необхідно явно вказати компілятору розташування заголовочних файлів бібліотеки та файлів, необхідних для лінкування. Нехай TBB встановлено в каталог ***/opt/intel/tbb/*** на 64-бітному комп'ютері, тоді команда для компіляції буде наступною:

user@n001$ g++ -I/opt/intel/tbb/include/ -o my_program my_program.cc -L/opt/intel/tbb/lib/intel64/ -ltbb

В дистрибутиві TBB також є спеціальний сценарій командного рядку, який встановлює необхідні компілятору змінні оточення і позбавляє необхідності явного вказання шляхів розташування бібліотеки. Для налаштування змінних необхідно один раз за термінальну сесію виконати команду:

user@n001$ source /opt/intel/tbb/bin/tbbvars.sh intel64

Після чого компіляція виконується наступним чином:

user@n001$ g++ -o my_program my_program.cc -ltbb

Б.5. Компіляція програми, що використовує ArBB

Для компіляції програми з використанням ArBB необхідно явно вказати компілятору розташування заголовочних файлів бібліотеки та файлів, необхідних для лінкування. Нехай ArBB встановлено в каталог ***/opt/intel/arbb/*** на 64-бітному комп'ютері, тоді команда для компіляції буде наступною:

user@n001$ g++ -I/opt/intel/arbb/include/ -o my_program my_program.cc -L/opt/intel/arbb/lib/intel64/ -larbb -ltbb

В дистрибутиві ***ArBB*** також є спеціальний сценарій командного рядку, який встановлює необхідні компілятору змінні оточення і позбавляє необ-

хідності явного вказання шляхів розташування бібліотеки. Для налаштування змінних необхідно один раз за термінальну сесію виконати команду:

user@n001$ source /opt/intel/arbb/tools/arbbvars.sh intel64

Після чого компіляція виконується наступним чином:

user@n001$ g++ -o my_program my_program.cc -larbb -ltbb

Б.6. Тестові запуски програм

Тестові (пробні, для налагодження) запуски обчислювальних програм дозволяється виконувати на інтерфейсному вузлі для уникнення очікування в черзі. Вимогою до тестових задач є використання не більше 4 ядер та час виконання, не більший 5 хвилин.

Тестовий запуск програм на MPI. Виконайте завантаження програми та її вхідних даних на кластер, а також компіляцію програми так само, як показано в розділі вище. Після цього виконайте команду:

user@n001$ mpiexec -np 4 ./program

Де 4 − кількість задач.

Тестовий запуск програм на OpenMP. Виконайте завантаження програми та її вхідних даних на кластер, а також компіляцію програми так само, як показано в розділі вище. Після цього виконайте команди:

user@n001$ export OMP_NUM_THREADS=4

user@n001$ ./program

Де 4 − кількість задач.