МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
„ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

О. П. ЧЕРНИХ, В. В. ЧЕЛАК

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Навчально-методичний посібник
для студентів комп’ютерних спеціальностей

Харків 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

О. П. ЧЕРНИХ, В. В. ЧЕЛАК

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Навчально-методичний посібник
для студентів комп’ютерних спеціальностей

Затверджено
редакційно-видавничою
радою університету,
протокол №2 від 25.06.2022 р.

Харків 2022

УДК 004.4
Ч-49

Рецензенти:
І. О. Фурман, д-р. техн. наук, проф., ХНТУСГ (м. Харків)
Г. А. Кучук, д.-р. техн. наук, НТУ “ХПІ” (м. Харків)

Публікується за рішенням вченої ради університету,
Протокол №2 від 25.06.2022 р.

Ч-49 **Паралельні та розподілені обчислення**: Навчально-методичний посібник /О.П. Черних, В.В. Челак. – Харків: НТУ “ХПІ”, 2022. – 82 с.

Даний практикум містить 8 лабораторних робіт. У кожній роботі є теоретичні відомості, тексти програм з докладними коментарями та результат їх виконання, які необхідні для проведення лабораторних робіт.

Призначений студентам спеціальностей 123 «Комп’ютерна інженерія».

Іл. 30. Табл. 16. Бібліогр. 10 назв.

УДК 004.4

Вступ

Сучасна проблема ефективної комп'ютерної обробки даних об'єднує в єдине ціле два принципово різних інструментально-програмних напрямка: розробку структур високопродуктивних обчислювальних систем і моделей паралельного програмування для них.

Основна мета інженерів і програмістів полягає в тому, щоб зменшити час обчислень або час обробки даних, підвищити ефективність завантаження обчислювального устаткування і скоротити накладні витрати на синхронізацію процесів і обмін даними. Очевидний шлях прискорення будь-якого виду взаємопов'язаних робіт в рамках однієї складної задачі полягає в розподілі окремих частин цієї роботи по декількох взаємодіючим паралельним потокам обробки. Однак проблема паралельного рішення трудомістких обчислювальних завдань в системах, що включають безліч взаємодіючих процесорів, зіткнулася з крайньою неефективністю каналів обміну поточними розрахунковими даними. Для паралельно виконуваних на різних процесорах потоків команд чітко позначилися проблеми синхронізації процесів в моменти обміну проміжними даними. Для систем MPMD щодо розумним виходом стало використання для обміну даними технології обміну повідомленнями, тобто в рамках функціональних мов програмування (в основному C / C^{++}) були додані спеціальні функції (підпрограми), що забезпечують адресне пересилання даних. Набір безлічі таких підпрограм і спеціальних утиліт був оформлений у вигляді бібліотек до зазначених мов, а система їх підключення та використання для програмування і виконання паралельних програм отримала назву MPI – Message Passing Interface (інтерфейс передачі повідомлень).

Метою лабораторного практикуму є вивчення та практичне використання студентами методів програмування обчислювальних задач великої розмірності в багатопроцесорних обчислювальних системах, ознайомлення з програмним інструментарієм розробки та виконання

паралельних програм, отримання практичних навичок вирішення паралельних завдань з використанням інтерфейсу передачі повідомлень MPI і його безліччю підпрограм обміну даними через поділювану пам'ять.

В якості багатопроцесорного обчислювального середовища використовується навчальний клас програмування, локальна мережа якого містить 5-10 персональних комп'ютерів. Кожен комп'ютер мережі має своє ім'я і IP- адресу. Ім'я конкретного користувача, що працює в мережі зі своєю паралельним завданням, вноситься адміністратором комп'ютерного залу до групи користувачів з правами адміністратора.

Обрані для лабораторного виконання легко розпаралелені обчислювальні завдання, взяті з літературних джерел та Інтернету, студентами допрацьовуються з метою одержання та введення своїх вихідних даних і контрольних значень, необхідних для оцінки правильності рішення паралельної завдання.

Лабораторний практикум включає в себе 8 лабораторних робіт, серед яких перша знайомить студентів з процедурою установки середовища MS-MPI на машинах комп'ютерного залу із завданням всіх необхідних параметрів конфігурації. Роботи (з другої по дев’яту включно) після деяких доопрацювань текстів відомих паралельних обчислювальних завдань знайомлять студентів з методологією побудови паралельних програм, конфігуруванням обчислювального середовища, інтерпретацією і дослідженням результатів роботи паралельної програми на безлічі вихідних даних і оформленням звіту по виконаній лабораторній роботі.

Лабораторна робота №1
УСТАНОВКА ТА НАЛАШТУВАННЯ MPI

Мета роботи: Встановити та налаштувати MS MPI і Visual Studio

Теоретичні відомості

MPI (Message Passing Interface) – інтерфейс обміну повідомленнями (інформацією) між обчислювальними процесами, які працюють одночасно. Він використовується для створення паралельних програм для обчислювальних систем, які мають розподілену пам'ять.

Установка пакета MS-MPI

Мережеві комп'ютери лабораторного залу повинні відповідати наступним системним вимогам:

– для компіляції програм C/C^{++} MPI потрібне середовище розробки **Visual Studio**. Для встановлення на комп'ютерах менеджера процесу (програми керування процесу – **smpd.exe**) потрібні привілеї адміністратора;

– для користувача на обраній кількості мережевих комп'ютерів створити однакове ім'я користувача, яке має бути наділене правами адміністратора і мати однаковий пароль для входу в операційну систему комп'ютера;

– будь-який комп'ютер локальної мережі лабораторного залу, не зачіпаючи сервера, вважається головним, якщо на ньому створюється, виконується і відбувається з нього управління паралельною програмою. Це дає можливість на кожному комп'ютері мережі одночасно працювати зі своїми навчальними паралельними програмами кільком студентам;

– одним з основних модулів **MS-MPI** є програма управління процесами **SMPD** (система демонів менеджера процесів), яка включає безліч аргументів, що визначають виклик і виконання необхідних для виконання паралельної програми опцій;

– установка MPI на кожному комп'ютері починається з запуску інсталяційного пакетів **msmpisdk.msi** та **MSMpiSetup**, які обрано щодо виконання лабораторного практикуму з паралельного програмування. Цій пакет повільно розповсюджується через Інтернет та має підтримку **Windows**

Vista/7/8.1/10;

– пакет **MS-MPI** необхідно встановити на усіх комп’ютерах, які будуть одночасно виконувати паралельну програму MPI. Програма установки виконується індивідуально на кожній машині повністю, або тільки менеджера процесу – **smpd.exe**. На цих комп'ютерах у користувача мають бути однакові ім'я адміністратора та його пароль. Тобто користувача на кожному комп’ютері необхідно зареєструвати у групу адміністраторів.

Процес установки **msmpisdk.msi** супроводжується видачою на екран монітору послідовно декількох вікон (рис. 1.1-1.3).

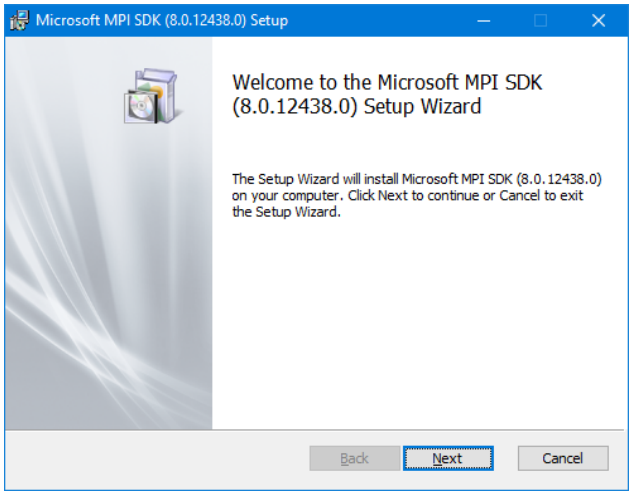

Рисунок 1.1 – Вікно привітання майстра установки msmpisdk

Для продовження установки необхідно натиснути кнопку «Next» і так далі, поки не отримаєте вікно рис.1.3.

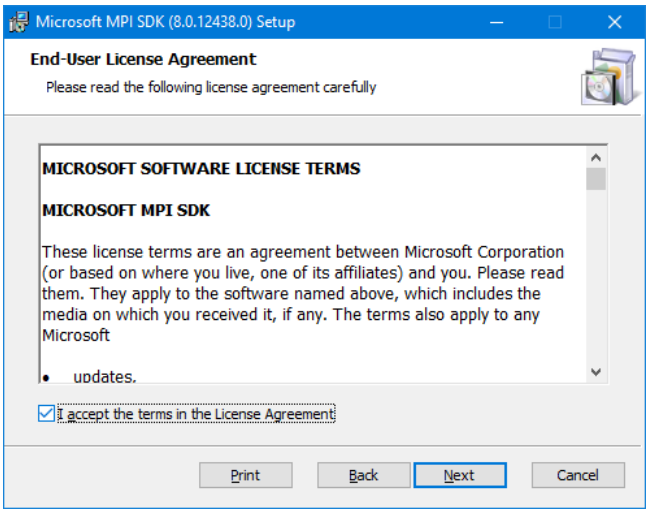

Рисунок 1.2 – Вікно ухвалення ліцензійної угоди

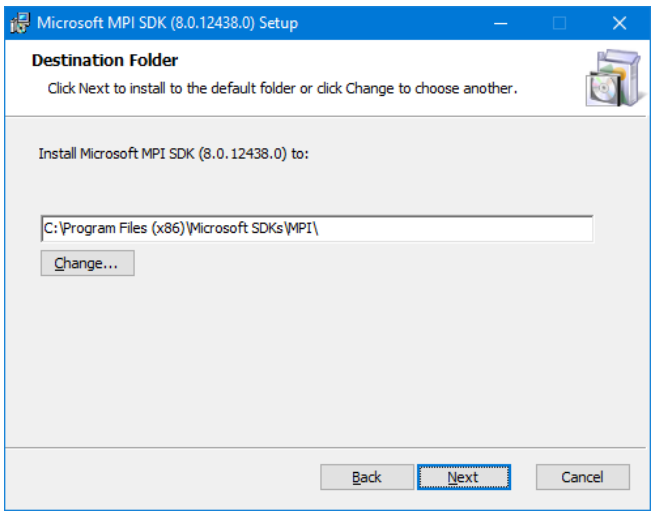

Рисунок 1.3 – Вікно шляху установки msmpisdk

Установка потрібної SDK MPI виконано.

Далі встановлюємо MSMpiSetup (див. рис. 1.4-1.7).

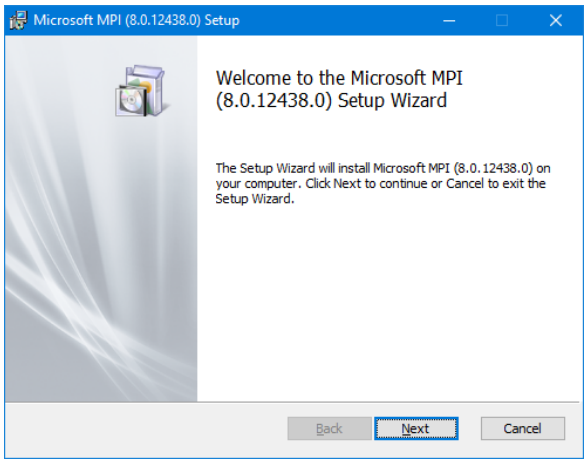

Рисунок 1.4 – Вікно привітання майстра установки MS-MPI

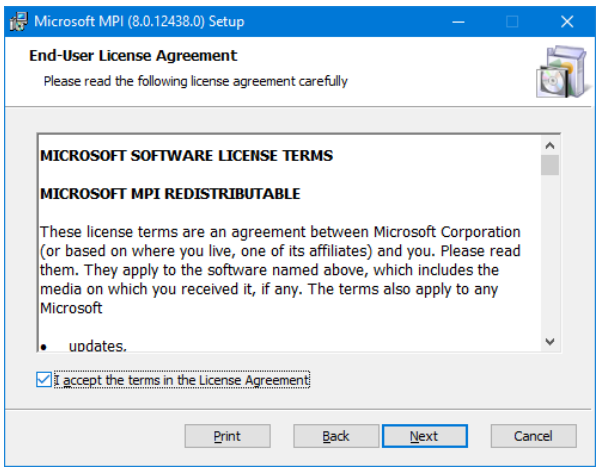

Рисунок 1.5 – Вікно ухвалення ліцензійної угоди

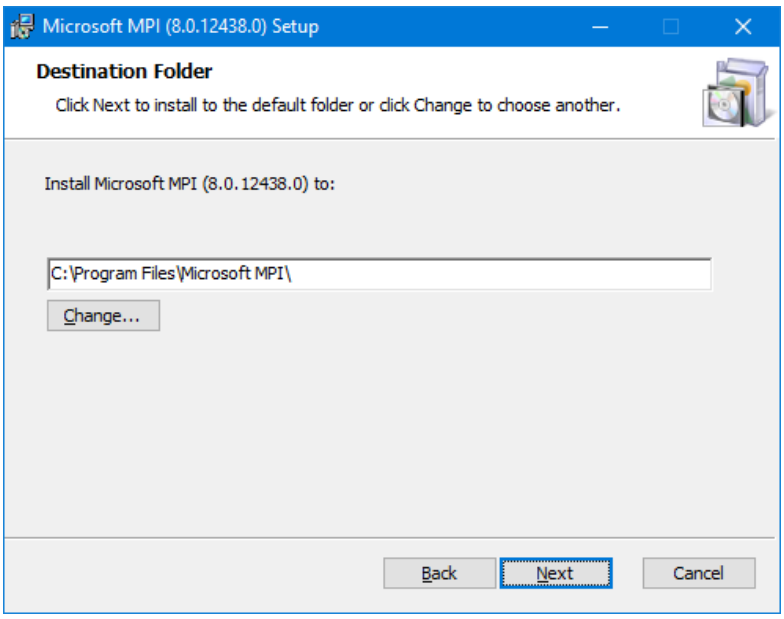

Рисунок 1.6 – Вікно шляху установки MS-MPI

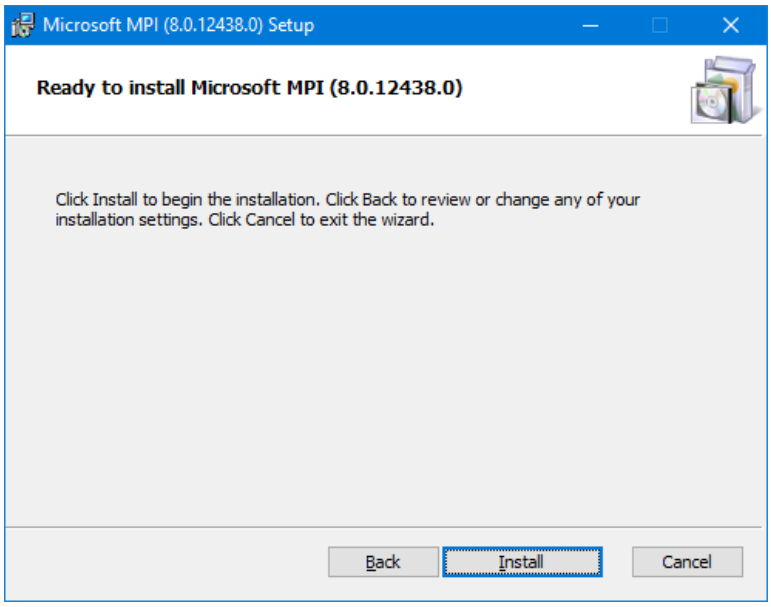

Рисунок 1.7 – Вікно інсталяції MS-MPI

Установка пакету MS-MPI виконано.

Робота у мережевій обчислювальній системі

Налаштування Visual Studio

Необхідно створити проект: New Project … -> Win32ConsoleApplication. (рис. 1.8)

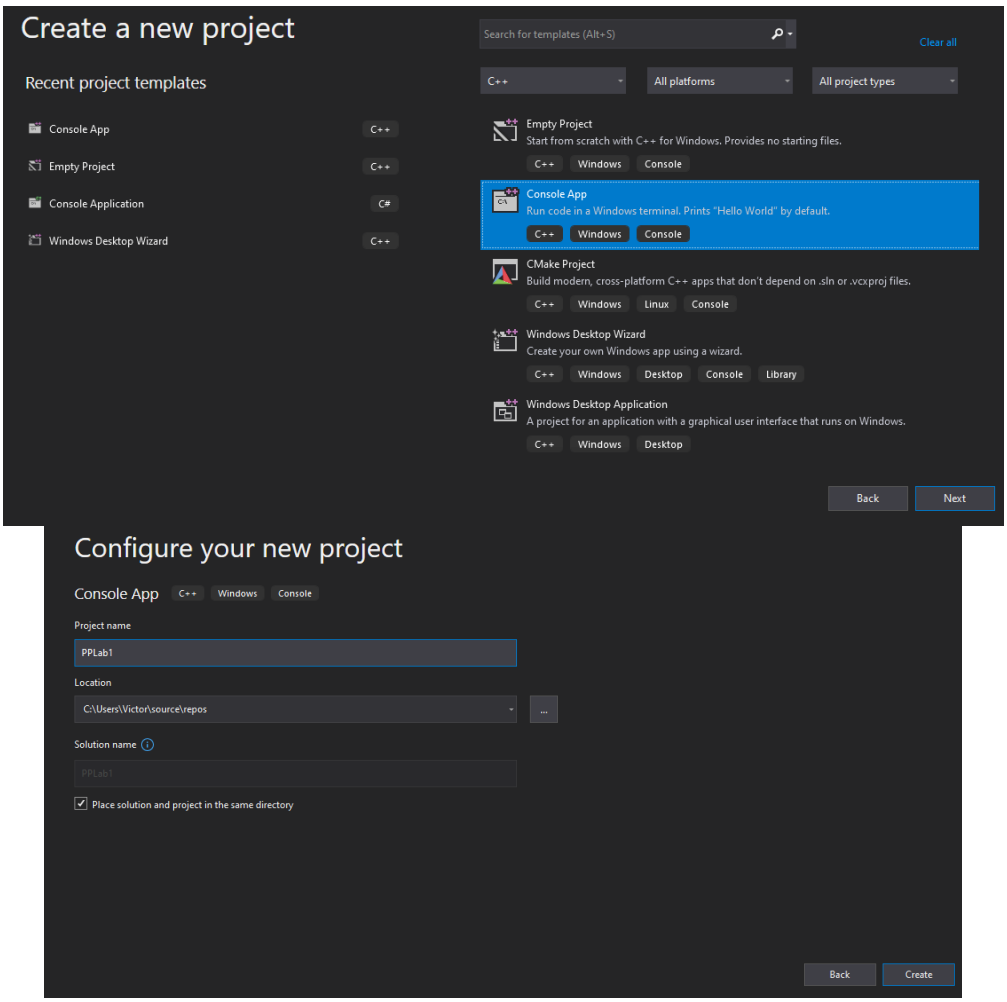

Рисунок 1.8 – Вікна створення проекту

Після створення проекту необхідно виконати підключення бібліотекі MPI з раніше встановленого msmpisdk.msi. Для цього потрібно відкрити «Propeties» проекту – правою кнопкою миші кликнути по назві проекту праворуч та обрати зі списку «Propeties».

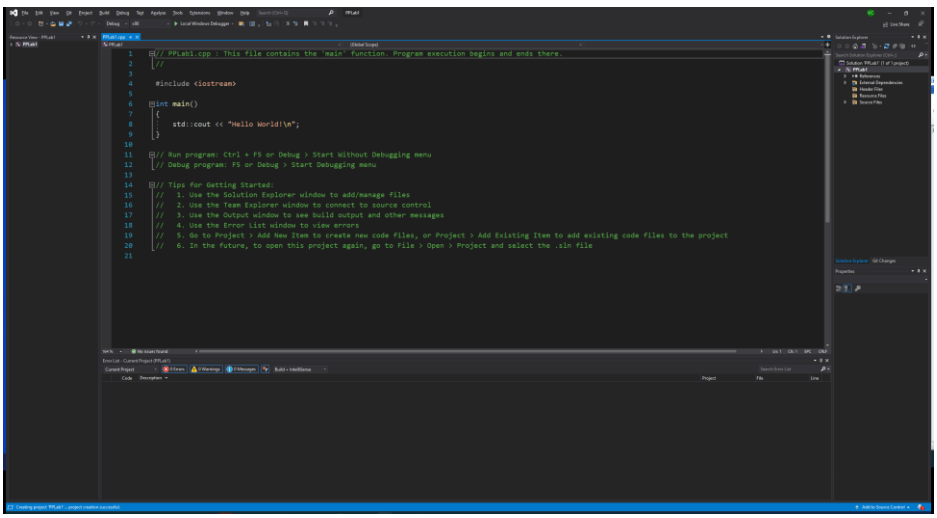

Рисунок 1.9 – Вікно зі створеним проектом

Вікно «Properties» проекту(див. рис. 1. 10):

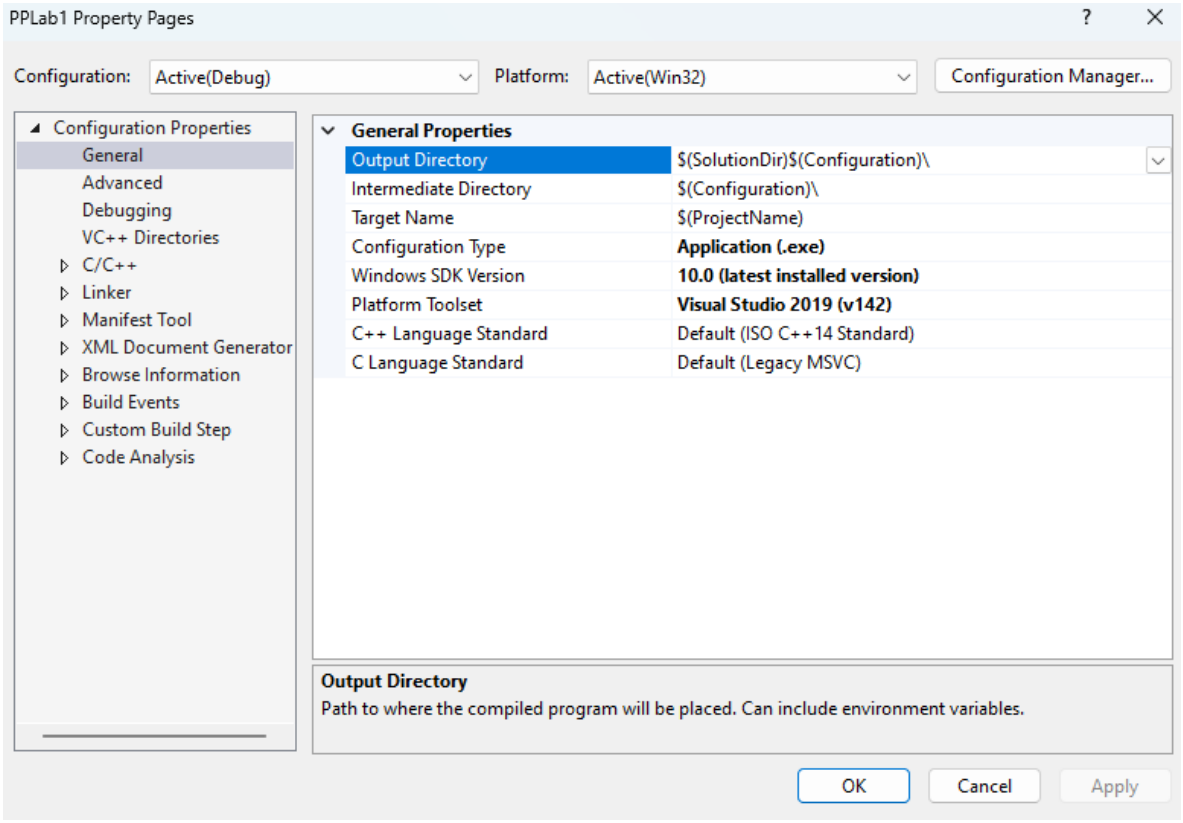

Рисунок 1.10 – Вікно «Properties»

Обрати конфігурацію «All Configurations» та платформу «All Platforms».

Відкрити C/C^{++} -> General. Обрати Additional Include Directories та вказати шлях до папки:

```
C:\Program Files (x86)\Microsoft SDKs\MPI\Include
```

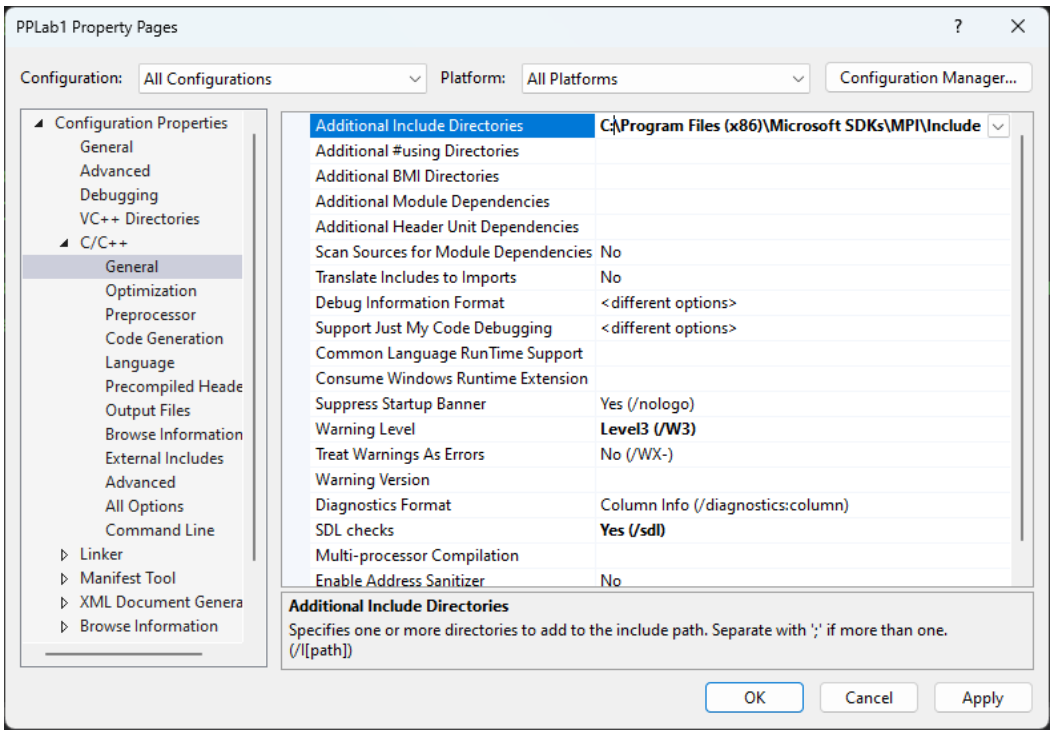

Рисунок 1.11 – Вікно обрання Additional Include Directories та шляху до папки: C:\Program Files (x86)\Microsoft SDKs\MPI\Include

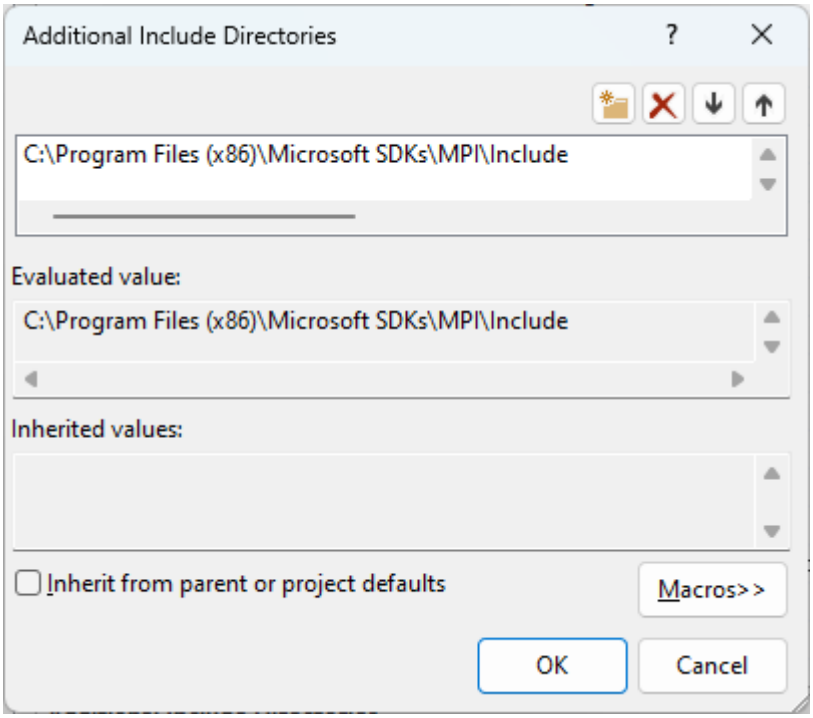

Рисунок 1.12 – Вікно додавання каталогів файлів, які підключаємо у загальних налаштуваннях проекту

Далі необхідно перейти до вкладки Linker/General та обрати Additional Library Directories. Далі для платформ:

– для платформи Win32 каталог:

```
C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86
```

– для платформи x64 каталог:

```
C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64
```

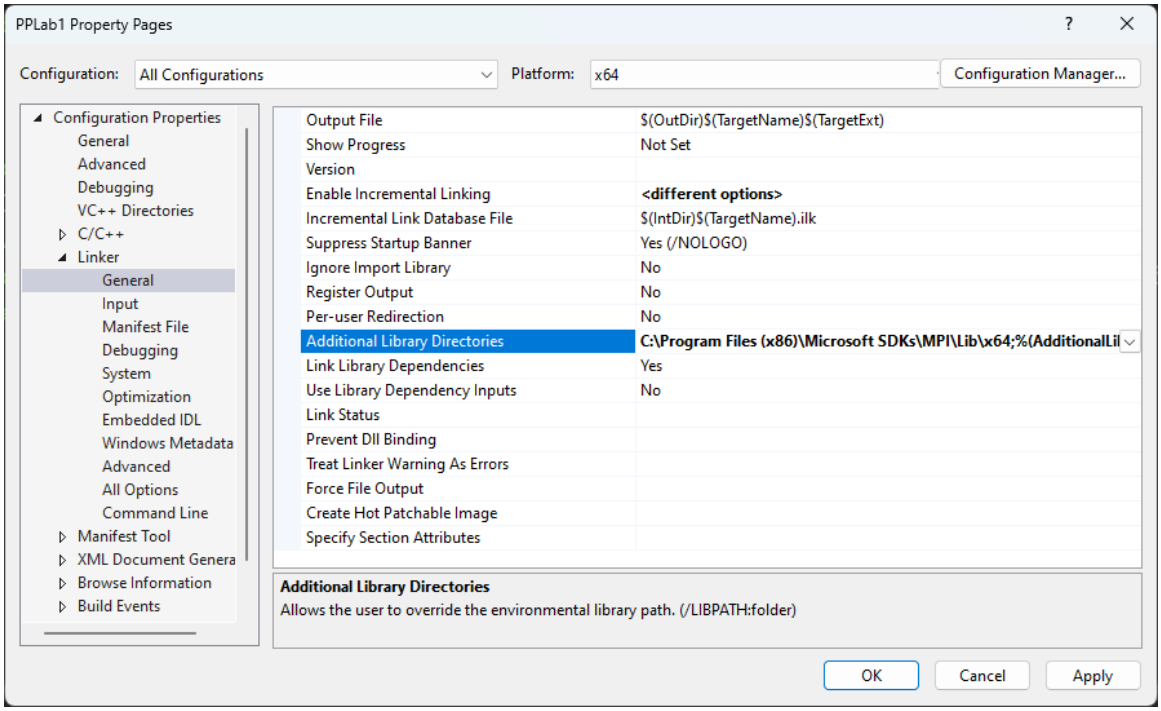

Рисунок 1.13 – Вікно налаштування Linker/General для x64 платформи

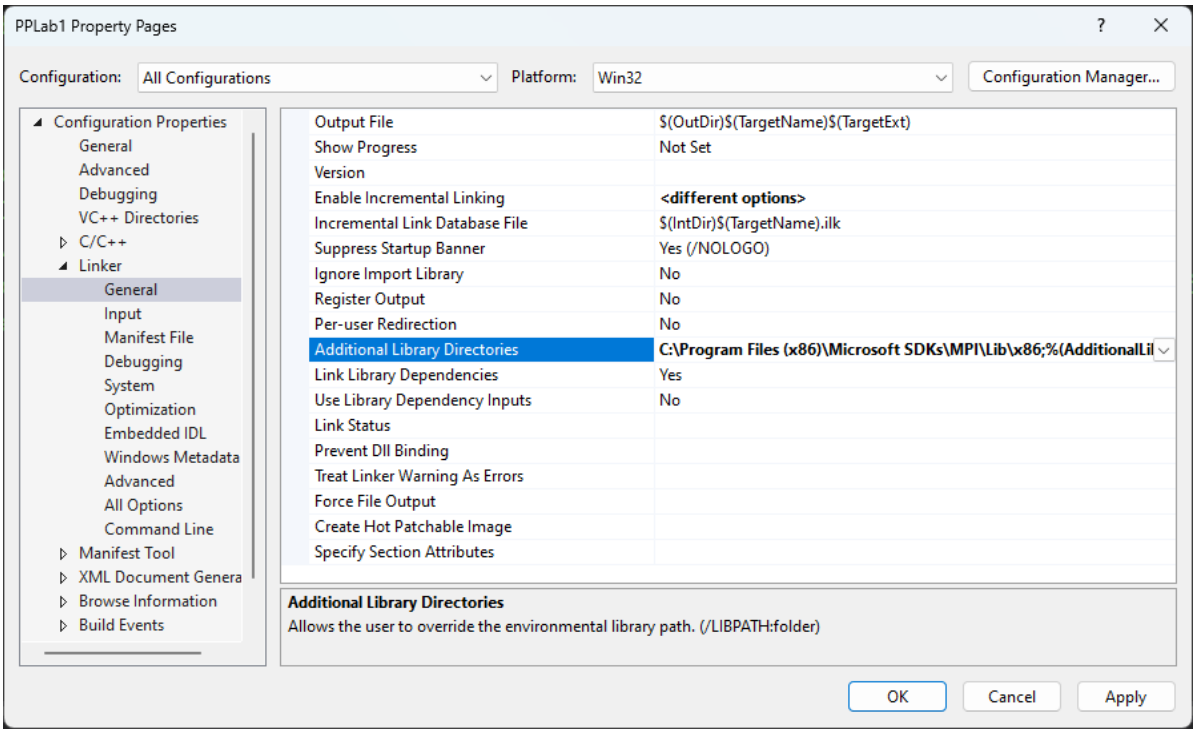

Рисунок 1.14 – Вікно налаштування Linker/General для x86 платформи

Далі необхідно налаштувати Input (для усіх платформ): обрати Additional Dependencies та додати бібліотеку msmpi.lib (рис. 1.15).

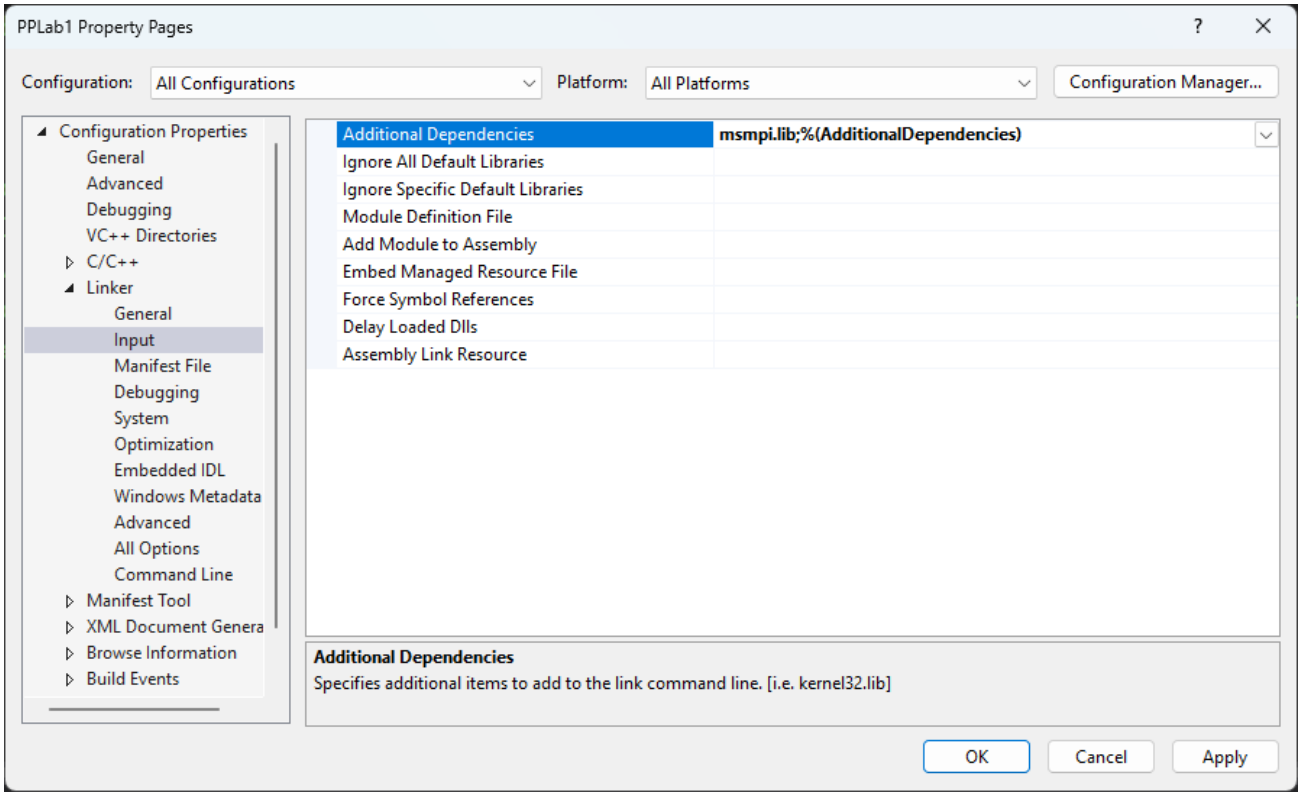

Рисунок 1.15 – Вікно налаштування Input

Запуск і виконання паралельної програми для перевірки працездатності встановленої MS-MPI

Текст програми:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]) {
	int ProcNum, ProcRank, RecvRank, namelen,recnamelen;
	char processor_name[MPI_MAX_PROCESSOR_NAME];
	MPI_Status Status;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
	MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
	MPI_Get_processor_name(processor_name, &namelen);
	if (ProcRank == 0) {
printf("\n Hello from %s process rank: %3d\n", processor_name ,ProcRank);
		for (int i = 1; i < ProcNum; i++) {
		MPI_Recv(&RecvRank, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &Status);
		MPI_Recv(&recnamelen, 1, MPI_INT, i, 2, MPI_COMM_WORLD,
&Status);
		char* recprocessor_name= new char [recnamelen];
MPI_Recv(recprocessor_name,recnamelen,MPI_CHAR,i,3,MPI_COMM_WORLD, &Status);
printf("\n Hello from %s process rank: %3d\n", recprocessor_name,RecvRank);
		}
	}
	else
	{
		MPI_Send(&ProcRank, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
```

```
            MPI_Send(&namelen, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
            MPI_Send(processor_name, namelen, MPI_CHAR, 0, 3,
MPI_COMM_WORLD);
      }
      MPI_Finalize();
      return 0;
}
```

Необхідно виконати компіляцію програми. Результат компіляції (exe-файл) вставити у каталог C:\Program Files\Microsoft MPI\Bin. Запустити консоль від адміністратору. Шлях у консолі: C:\Program Files\Microsoft MPI\Bin. У диспетчері задач служба MsMpiLaunchSvc повинна бути виключена (рис. 1. 16).

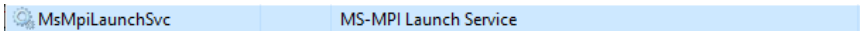

Рисунок 1.16 – Вікно сервісів в «Task Manager»

У консолі виконати наступний запит:

```
mpiexec.exe –n 4 –cores 4 PPlab1.exe
```

Якщо брандмауер запрошуватиме дозвіл, дозволити для приватних мереж.

Параметр –n відповідає за кількість процесів, які треба запустити для виконання обчислення. –cores кількість процесорів, які використовуємо для обчислень.

Створення кластера

1 Встановити на всі елементи (node) кластера (cluster) MS-MPI (середовище, з якого можна виконати запуск програми, що використовує протокол MPI) і MPI_SDK (бібліотека для написання програм з використанням MPI).

2 Для запуску завдань на MS-MPI на декількох машинах, необхідно створити облікові записи з однаковими логінами і паролями. Для повних привілеїв в паралельних та розподілених обчисленнях треба створити такі облікові записи з правами адміністратора:

2.1 Відкрити «Диспетчер завдань» під правами адміністратора.

2.2 Обрати у випадаючому списку «Файл» -> «Запустити нову задачу». Далі поставити галочку навпроти «Створити завдання з правами адміністратора» і написати назву менеджера локальних користувачів «lusrmgr.msc» (рис. 1.17).

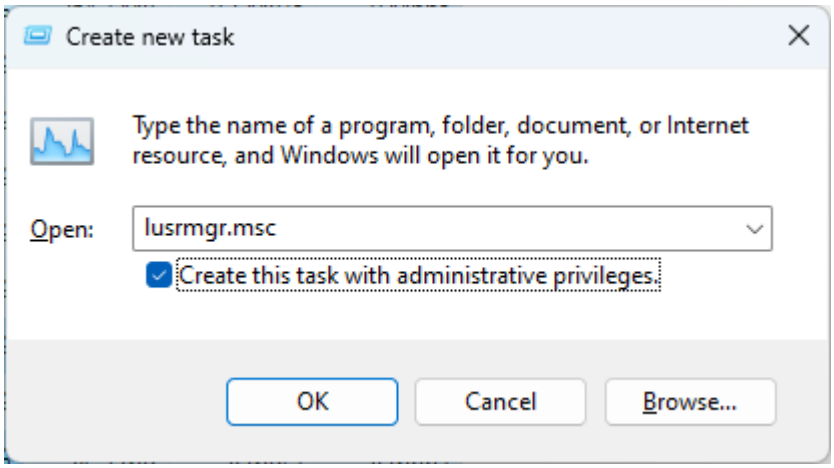

Рисунок 1.17 – Вікно створення задачі під правами адміністратора

2.3 Обрати «Новий користувач» кликанням за допомогою правої кнопки миші по папці «Користувачі» (рис.1.18).

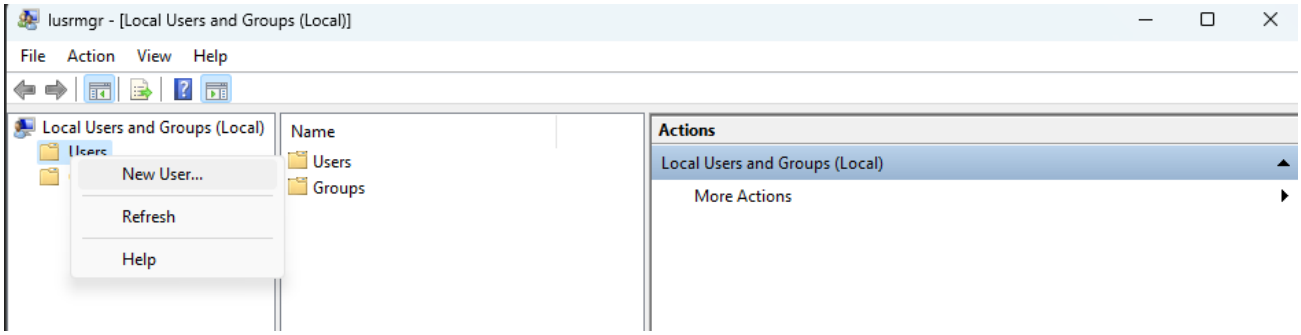

Рисунок 1.18 – Вікно створення «Новий користувач»

2.4 Заповнюємо поля вікна «Новий користувач» як наведено на рис. 1.19. У полях «Пароль» і «Підтвердження» вводити «clnpwd». Далі зняти галочку з «Вимагати зміни пароля при наступному вході в систему» і поставити на «Заборонити зміну пароля користувачем» та «Термін дії пароля необмежений» (див. рис.1.19).

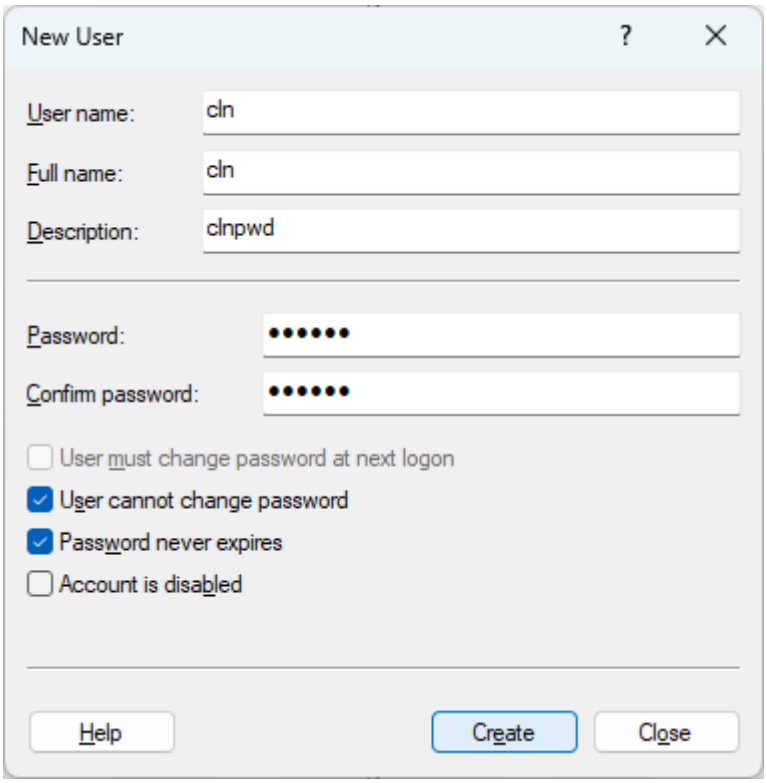

Рисунок 1.19 – Вікно заповнення полей «Новий користувач»

2.5 Натиснути на кнопку «Створити», після чого закрити вікно створення нового користувача і зайти в папку «Користувачі» (рис. 1.20). Правою кнопкою миші по створеному "cln" користувачеві, обираємо «Властивості».

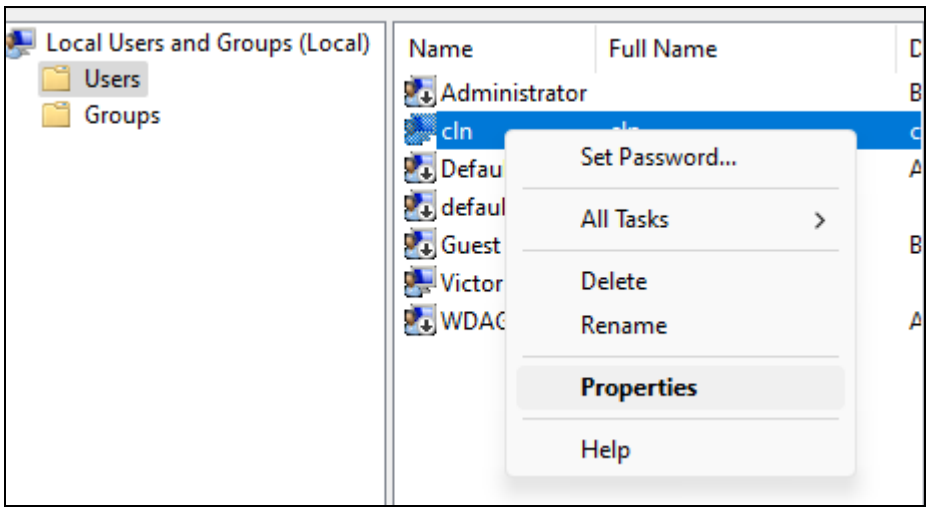

Рисунок 1.20 – Вікно обирання поля «Властивості» в каталозі «Користувачі»

2.7 Обираємо вкладку «Членство в групах» і натискаємо на кнопку «Додати ...» (див. рис. 1.21).

2.8 У вікні ввести ім'я «Адміністратори» та натиснути на кнопку «Перевірити імена», а потім на кнопку «Ок» (див. рис. 1.22).

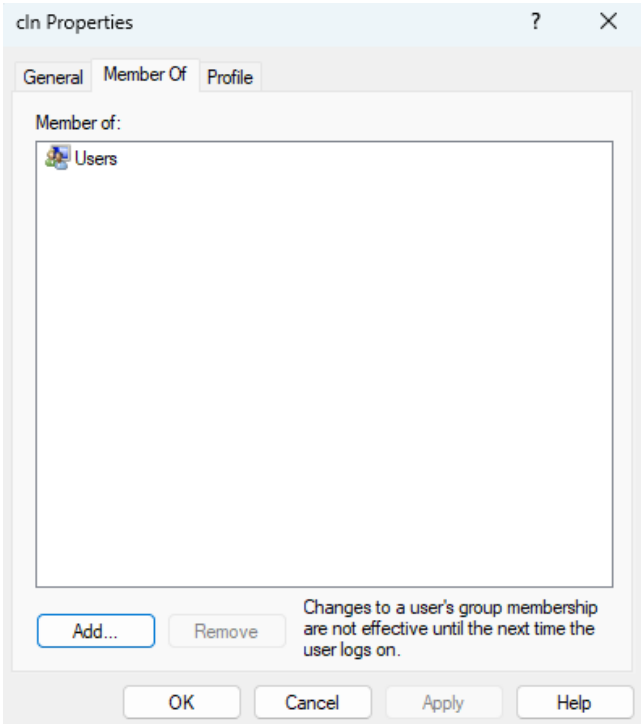

Рисунок 1.21 – Вікно обирання поля «Додати» у властивості «cln»

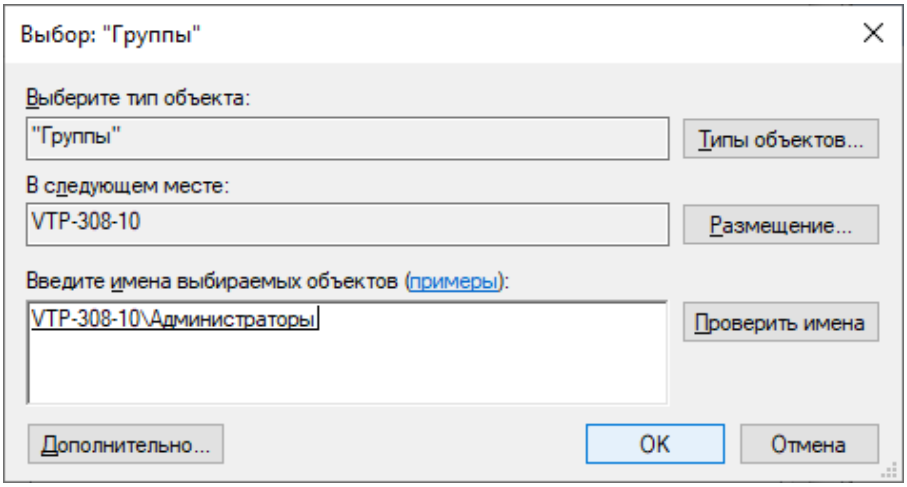

Рисунок 1.22 – Вікно додавання груп та їх перевірки

2.9 Після того, як пункти 2.1 по 2.8 будуть виконані на всіх елементах створюваного кластера, необхідно зайти під обліковим записом "cln" та вести пароль "clnpwd".

3 Необхідно переконатися, що MS-MPI Launch Service не працює. Для цього можна скористатися утилітою «Служби» – services.msc (за замовчуванням дана служба відключена).

4 На кожному елементі кластера необхідно запустити smpd-з'єднання. Для цього необхідно виконати:

4.1 Відкрити консоль від адміністратора (це можна зробити через диспетчер задач, подібно дії в пункті 2.1, але вказати назву процесу командного рядка "cmd").

4.2 Зайти в папку, де розташоване середовище MPI. Команда, якщо середовище MS_MPI було встановлено в папку за замовчуванням:

```
cd / d
C: \ Program Files \ Microsoft MPI \ Bin \
```

4.3 Запустити smpd на порт 8677. Команда:

```
smpd -p 8677
```

4.4 Якщо з'являється вікно брандмауера, треба дозволити його виконання (для будь-яких з'єднань).

4.5 Після того, як пункти 4.1 по 4.4 будуть виконані на всіх елементах кластера, треба обрати кластер готовий до використання. (Важливо: поки командний рядок з процесом smpd відкритий, то з'єднання елемента з іншими працює).

5 (необов'язковий) На кожному елементі кластера розташовані файли, що виконуються, в одній і тій же папці. Наприклад,

```
C: \ Program Files \ Microsoft MPI \ Bin \
```

6 Відкрити додатковий командний рядок на провідному елементі кластера. Далі зайти в папку з середовищем MPI:

```
cd / d C: \ Program Files \ Microsoft MPI \ Bin \
```

7 Запустити програму mpiexec. Формат використання:

```
mpiexec -c <кількість ядер> -hosts <кількість елементів> <ім'я ведучого комп'ютера> <ім'я другого комп'ютера> ... <ім'я останнього комп'ютера> <програма> .exe
```

або ж:

```
mpiexec -hosts <кількість елементів> <ім'я ведучого комп'ютера> <кількість ядер на провідному комп'ютері> <ім'я другого комп'ютера> <кількість ядер на другому комп'ютері> ... <ім'я останнього комп'ютера> <кількість ядер на останньому комп'ютері> <програма> .exe
```

8 При роботі з великим кластером, набагато ефективніше вказати всі елементи кластера через текстовий файл, перше ім'я в файлі - ім'я ведучого елемента. Кількість елементів кластера в файлі вказувати немає необхідності:

```
mpiexec -c <кількість ядер> -machinefile <шлях до файлу> <програма> .exe
```

Приклади команди:

```
mpiexec -c 4 -hosts 2 VTP-308-01 VTP-308-02 "MpiPrimeX64.exe"

mpiexec -hosts 2 VTP-308-01 4 VTP-308-02 1 "MpiPrimeX64.exe"

mpiexec -c 4 -machinefile m.txt "MpiPrimeX64X2 ^
16difficulty.exe"
```

Вміст файлу «m.txt» містить імена комп’ютерів створеного кластера для аудиторії 308 корпусу ВК:

```
VTP-308-01
VTP-308-02
VTP-308-03
VTP-308-04
VTP-308-05
VTP-308-06
VTP-308-07
VTP-308-08
VTP-308-09
VTP-308-10
```

Лабораторна робота №2

ОБЧИСЛЕННЯ ІНТЕГРАЛУ МЕТОДОМ ТРАПЕЦІЙ

Мета роботи: Розглянути можливості пакету MS-MPI щодо виконання обчислення інтегралів методом трапецій, тобто паралельного обчислення частин інтегралу. Дослідити час виконання програми на одній та декількох машинах, виявити оптимальну кількість паралельних процесів для виконання поставленої задачі. Проаналізувати залежність точності обчислень у залежності від кількості кроків інтегрування.

Теоретичні відомості

Під час виконання даної роботи пропонується розбивати обчислення інтегралу на декілька частин(кроків), що обумовлює реалізацію метода трапеції та створює посилання для паралельного обрахунку. Обмін даними між процесами результатами розрахунків своїх «порцій» пропонується вести за допомогою функцій ***MPI_Send()*** та ***MPI_Recv()*** пакету MPI.

2.1 Функція передачі MPI_Send

Функція передачі **MPI_Send** має наступний синтаксис:

```
MPI_SEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM)
```

Типи параметрів:

```
<type> BUF(*)
INT COUNT, DATATYPE, DEST, TAG, COMM
```

Функція ***MPI_Send*** виконує посилку повідомлення, що складається з «count» елементів типу «datatype», з ідентифікатором «tag» процесу «dest» у області зв’язку комунікатора «comm». Змінна «buf» – це, як правило, масив чи скалярна змінна. У останньому випадку значення count = 1.

Таблиця 2.1 – Параметри ***MPI_Send***

Тип	Параметри	Призначення
IN	buf	адреса початку розміщення даних для відправки
IN	count	кількість елементів, що відправляються
IN	datatype	тип елементів, що відправляються
IN	dest	номер процесу-отримувача у групі, пов'язаній з комунікатором comm
IN	tag	ідентифікатор повідомлення
IN	comm	комунікатор області зв'язку

2.2 Функція передачі MPI_Recv

Функція отримання даних ***MPI_Recv*** має наступний синтаксис:

```
MPI_RECV (BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS)
```

Типи параметрів:

```
<type> BUF(*)
INT COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE)
```

Функція ***MPI_Recv*** виконує приймання «count» елементів типу «datatype» повідомлення з ідентифікатором «tag» від процесу «source» у області зв'язку комунікатора «comm».

Таблиця 2.2 – Параметри MPI_Recv

Тип	Параметри	Призначення
OUT	buf	адреса початку розміщення даних для отримання
IN	count	максимальна кількість елементів, що отримуються
IN	datatype	тип елементів, що отримуються
IN	source	номер процесу-відправника
IN	tag	ідентифікатор повідомлення
IN	comm	комунікатор області зв'язку
OUT	status	атрибути прийняття повідомлення (можливі помилки)

2.3 Функція MPI_WTime()

Замір часу виконання програми можна зробити за допомогою функції ***MPI_WTime()***, що повертає кількість секунд, що пройшла від певного моменту у минулому. Гарантується, що цей момент незмінний для всіх комп'ютерів та систем. Фрагмент коду для замірювання часу можна представити наступним чином:

```
double  t1,time;
t1=MPI_Wtime();
/* Блок вимірювання часу */ time=MPI_Wtime()-t1;
```

Після цього змінна «time» зберігатиме час виконання блоку у секундах.

Завдання: Необхідно модернізувати паралельну програму, що методом трапецій обчислює визначений інтеграл: INT(*f(x), x, a, b*). Перелік функцій і параметрів їх інтегрування наведено у таблиці. Індивідуально кожний студент вибирає функцію за номером, який обчислюється за формулою:

$$№ = (5 + N_{ж.гр.}) * \text{mod}(9) + 1 \qquad (2.1)$$

Виконати обчислення, обравши кількість інтервалів в межах [a,b] равною {10, 20, …, 50}.

Знайти аналітичне значення заданого интегралу, втілив вираз його обчислення у вигляді процедури. Значення її використовувати для обчислення похибок паралельної програми, які з результатом інтегрувань виводити на екран монітору.

Таблиця 2.3 – Індивідуальні завдання

№	$f(x)$	a	b	$\int f(x)\,dx$
1	$\frac{1}{x}$	1	2	$\ln(x)$
2	$\frac{1}{x}$	2	3	$\ln(x)$
3	$\frac{1}{\sin(x)}$	1	2	$\ln\left(\frac{\sin(x)}{\cos(x)+1}\right)$
4	e^x	-1	1	e^x
5	$\frac{1}{\cos(x)}$	2	3	$\ln\left(\frac{\cos(x)}{1-\sin(x)}\right)$
6	$tg(x)$	0	1	$-\ln(\cos(x))$
7	$\mathrm{tg}(x)$	1	2	$-\ln(\cos(x))$
8	$\mathrm{sh}(x)$	1	3	$(e^x+e^{-x})/2$
9	$\mathrm{ch}(x)$	2	3	$(e^x-e^{-x})/2$

Приклад 2.1

Текст програми

```
/*Підключаємо бібліотеку обслуговування процедури ввода/вивода */
#include <stdio.h>
#include <windows.h>  /*Підключаємо бібліотеку під програму MPI */
#include <math.h>
#include "mpi.h"
void main(int argc, char** argv) {
      int   my_rank; /* Ранг процесу */
      int   p;       /* Число процесорів */
      float a;  /* Нижняя граница інтегрування */
      float b;  /* Верхня граница інтегрування */
      int   n;  /* Повне число інтервалів */
      float h;         /* Шаг інтегрування */
      float local_a;   /* Нижній предел інтервалу */
      float local_b;   /* Верхній предел інтервала */
      int   local_n;   /* Число інтервалів на */
      /* участке інтегрування */
      double time;
      float integral;  /* Інтеграл на інтервалі */
      float total;     /* Загальний інтеграл */
      int   source;    /* Джерело значення інтегралу */
      int   dest = 0;  /* Збір результатів в процесі 0 */
      int   tag = 0;
      MPI_Status  status; /* статус виконання передачі */
      /* Оголошення функції введення вихідних даних */
```

```
      void Get_data(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank, int
p);
      float Trap(float local_a, float local_b, int local_n, float h);
      MPI_Init(&argc, &argv);

      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

      MPI_Comm_size(MPI_COMM_WORLD, &p);
      Get_data(&a, &b, &n, my_rank, p);
      double temp = MPI_Wtime();
      h = (b - a) / n;
      local_n = n / p;
      local_a = a + my_rank*local_n*h;
      local_b = local_a + local_n*h;

      integral = Trap(local_a, local_b, local_n, h);
      if (my_rank == 0)
      {
            total = integral;
            for (source = 1; source < p; source++) {

                  MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &status);
                  total = total + integral;
            }
      }
      else {
            MPI_Send(&integral, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
      }
      if (my_rank == 0)
      {
            double time = MPI_Wtime() - temp;
            printf("Low limit %f\nHigh limit %f\nIntegral steps is %d\n", a,
b, n);
            printf("Function of integration is x/sqrt(1+x*x)\n\n");
            printf(" Integral valuation by %d steps\n", n);
            printf(" from %f to %f is %1.10f\n", a, b, total);
            printf(" time of calculating is %1.5f milisecond(s)\n", time *
1000);
      }
      MPI_Finalize();
} /*  main  */

/*********************************************************/
void Get_data(
      float*  a_ptr    /* out */,
      float*  b_ptr    /* out */,
      int*    n_ptr    /* out */,
      int     my_rank  /* in  */,
      int     p        /* in  */)
{
      int source = 0;
      int dest;
      int tag;
      MPI_Status status;
```

```
      if (my_rank == 0) {
            scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
            for (dest = 1; dest < p; dest++)
            { tag = 0;




            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
            tag = 1;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag, MPI_COMM_WORLD);
            tag = 2;
            MPI_Send(n_ptr, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
            }
      }
      else {
                  tag = 0;

MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
                  tag = 1;
MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &status);
                  tag = 2;
MPI_Recv(n_ptr, 1, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
            }
} /* Get_data */

/*****************************************************/
float Trap(
      float  local_a    /* in */,
      float  local_b    /* in */,
      int    local_n    /* in */,
      float  h          /* in */)
{
      float integral;
      float x;
      int i;
      float f(float x);
      integral = (f(local_a) + f(local_b)) / 2.0;
      x = local_a;

      for (i = 1; i <= local_n - 1; i++)
      {
            x = x + h;
            integral = integral + f(x);
      }
      integral = integral*h;
      return integral;
} /*  Trap  */

/*****************************************************/
float f(float x)
{                 /* f(x)=4/(1+x*x)   */
      float return_val;
      return_val = x / sqrt(1 + x*x);
      return return_val;
}
```

Алгоритм перевірки результатів програми

1) Розрахунок інтеграла аналітично.

Приклад:

$$\int_0^1 \frac{x}{\sqrt{1+x^2}} dx = \sqrt{x+1}\Big|_0^1 = \sqrt{2} - 1 = 0.41421356237$$

2) Дослідження швидкодії виконання обчислень при різної кількості процесів.

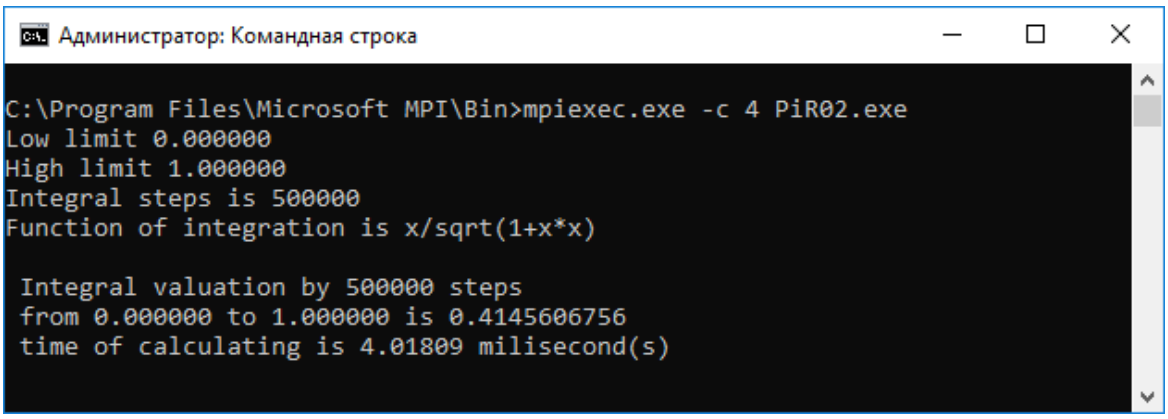

Рисунок 2.1 – Результати запуску програми для 4 процесорів

3) Висновок про похибку обчислень у декількох процесах. Наприклад, при запуску одного процесу результат обчислення дорівнює 0.4142135978. Тоді відносна похибка буде (за еталон беремо результат аналітичного обчислення):

$$\frac{0.4142135978 - 0.41421356237}{0.41421356237} * 100\% = 0.00000855\%$$

4) Порівняння відносних похибок для одного, двох та чотирьох процесів з урахуванням швидкодії

5) Аналіз у MATLAB 2017b. Створимо невеликий скрипт розрахунку методом трапецій прикладу при різної кількості інтервалів (від 1 до 20):

```
Result= [];
index = 1;
```

```
for intervals=1:20

% Шаг = Верхняя граница – Нижняя граница / Количество
интервалов
x= 0: 1/ intervals : 1;    % Шаг = 1/ intervals
y= x./sqrt(1+x.*x);
Z = trapz(x,y);
Result(index) =Z;
index=index+1;
end
```

6) Отримання результатів моделювання у MATLAB. З встановленою розрядністю MATLAB виконаємо розрахунок аналітичного виразу прикладу: sqrt(2)-1. Виконаємо побудову залежності відносної похибки у % від кількості інтервалів для методу трапецій:

```
plot(1:20,((sqrt(2)-1)-Result)./(sqrt(2)-1)*100)/
```

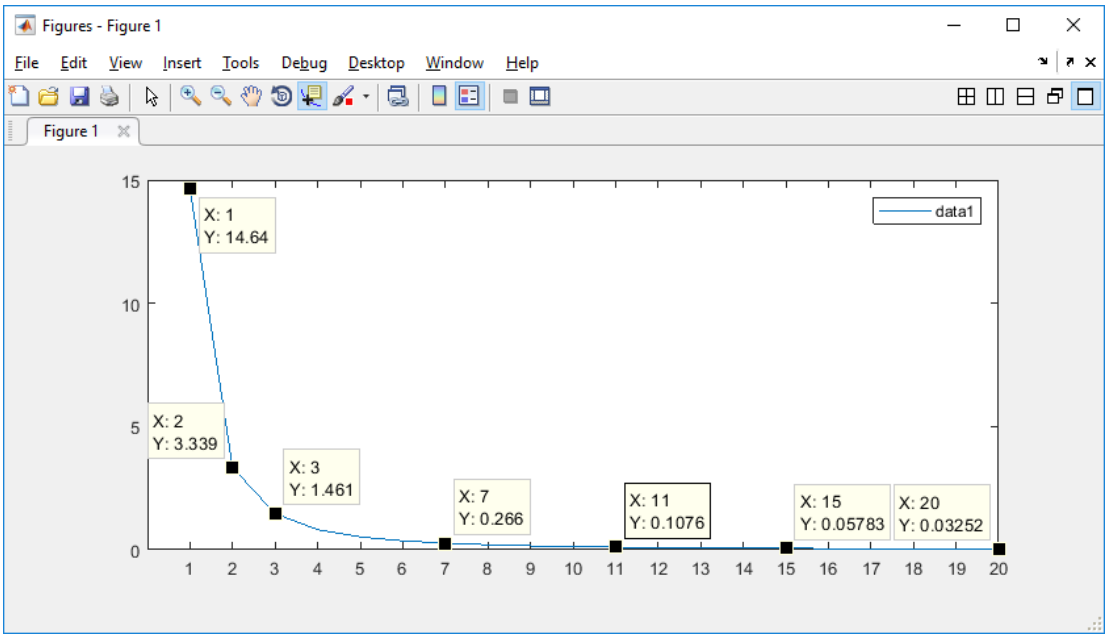

Рисунок 2.2 – Похибка від кількості інтервалів

Як бачимо з рис. 2.2, починаючи з 3 інтервалів похибка складає менш ніж 1.5%. Потрібно зазначити, що цей приклад має незначний інтервал [0;1],

а саме тому – не має велику ефективність при 500 тис. інтервалів. Однак – якщо брати умови більш ніж заданий інтервал наприклад від [0; 500 000] – похибка буде зменшуватись при збільшенні кількості інтервалів для методу трапецій.

У звіті привести схему паралельної обробки даних, описати фрагменти, які треба було додати у текст програми та оговорити результати роботи паралельної програми.

Лабораторна робота №3

ОБЧИСЛЕННЯ СКАЛЯРНОГО МНОЖЕННЯ ВЕКТОРІВ

Ціль роботи: Розглянути можливості пакету MS-MPI щодо виконання паралельних операцій між двома векторами. Дослідити час виконання програми на одній та декількох машинах, виявити оптимальну кількість паралельних процесів для виконання поставленої задачі**.**

Теоретичні відомості

Для виконання даної роботи пропонується, окрім процедур MPI_Send та MPI_Recv (див. лабораторну роботу 2), використати широкомовну передачу даних за допомогою MPI_Bcast та MPI_Reduce, яка «збиратиме» дані в один буфер та паралельно обчислюватиме необхідну операцію з ними.

3.1 Функція широкомовної передачі MPI_Bcast

Широкомовна розсилка MPI_Bcast має наступний синтаксис:

```
MPI_Bcast (BUFFER, COUNT, DATATYPE, ROOT, COMM)
```

Типи параметрів:

```
<type> BUFFER(*)
INT COUNT, DATATYPE, ROOT, COMM
```

Таблиця 3.1 – Параметри MPI_Bcast

Тип	Параметри	Призначення
INOUT	buffer	адреса початку розміщення даних для розсилки у пам’яті
IN	count	кількість елементів для посилки
IN	datatype	тип елементів посилки
IN	root	номер процесу-відправника
IN	comm	комунікатор

MPI_Bcast() виконує розсилку даних зі змінної «buffer» з кількістю елементів «count» типу «datatype» процесора «root» усім процесам комунікатора «comm».

2.2 Функція широкомовної передачі MPI_Reduce

Функція MPI_REDUCE поєднує елементи вхідного буферу кожного процесу у групі через операцію *OP* та повертає поєднане значення у вихідний буфер процесу с рангом root.

Колективна редукція MPI_Reduce має наступний синтаксис:

```
MPI_REDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,COMM)
```

Типи параметрів:

```
<type> SENDBUF(*), RECVBUF(*)
INT COUNT, DATATYPE, OP, ROOT, COMM
```

Таблиця 3.2 – Параметри MPI_Reduce

Тип	**Параметри**	**Призначення**
IN	sendbuf	адреса початку вхідного буфера
OUT	recvbuf	адреса початку буферу результатів (викор. тільки у процесі-отримувачі root)
IN	count	кількість елементів у вхідному буфері
IN	datatype	тип елементів вхідного буфера
IN	op	операція, по якій виконується редукція
IN	root	номер процесу-отримувача результату операції
IN	comm	комунікатор

Схема виконання MPI_Reduce для операції (op) “+” (див. рис. 3.1):

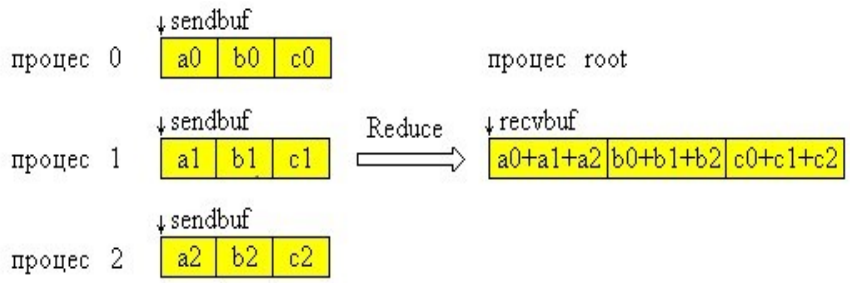

Рисунок 3.1 – Схема виконання додавання за допомогою MPI_reduce

Окрім зазначеної операції додавання, в пакет MPI вбудовані наступні операції для колективної редукції:

Таблиця 3.3 – Операції MPI_reduce

Назва в MPI	Функція
MPI_MAX	Максимум
MPI_MIN	Мінімум
MPI_SUM	Сума
MPI_PROD	Добуток
MPI_LAND	Логічне І
MPI_BAND	Побітне І
MPI_LOR	Логічне АБО
MPI_BOR	Побітне АБО
MPI_LXOR	Логічне виключне АБО
MPI_BXOR	Побітне виключне АБО

Завдання: Розробити програму у середовищі C++ з використанням функцій пакету MS-MPI, у якій:

1) Реалізувати випадкову генерацію двох векторів розмірності *N*.

2) Реалізувати над ними паралельну операцію *OP* з можливістю виконання на декількох процесорах паралельно.

3) Заміряти час виконання програми на одному процесорі та на декількох. Порівняти результати, зробити висновки щодо доцільності використання двох, трьох або більше процесорів.

Індивідуально кожний студент вибирає функцію за номером, який обчислюється за формулою:

$$№ = (5 + N_{ж.гр.}) * mod(9) + 1 \quad (3.1)$$

Таблиця 3.4 – Варіанти індивідуальних завдань

№	N	ОР
1	300	Сума
2	250	Різниця
3	180	Логічне І
4	200	Логічне АБО
5	250	Сума
6	175	Скалярний добуток
7	150	Сума
8	300	Логічне І
9	250	Різниця
10	200	Скалярний добуток
11	300	Сума
12	250	Різниця
13	180	Логічне І
14	200	Логічне АБО
15	250	Сума

Приклад 3.1

Текст програми

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

#define MAX_LOCAL_ORDER 300

void main(int argc, char* argv[]) {
    float  local_x[MAX_LOCAL_ORDER];
    float  local_y[MAX_LOCAL_ORDER];
    int    n;
    int    n_bar;  /* = n/p */
    float  dot;
    int    p;
    int    my_rank;

    void Read_vector(char* prompt, float local_v[], int n_bar, int p,
            int my_rank);
    float Parallel_dot(float local_x[], float local_y[], int n_bar);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
```

```
        printf("Enter the order of the vectors\n");
        scanf("%d", &n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    n_bar = n/p;

    Read_vector("the first vector", local_x, n_bar, p, my_rank);
    Read_vector("the second vector", local_y, n_bar, p, my_rank);
    double temp = MPI_Wtime(); // Початок заміру часу
    dot = Parallel_dot(local_x, local_y, n_bar);

    if (my_rank == 0)
      {

           double time = MPI_Wtime() - temp;
       printf("The dot product is %f\n", dot);
       printf(" time of calculating is %1.5f milisecond(s)\n", time * 1000);

      }
    MPI_Finalize();
}  /* main */


/*********************************************************************/
void Read_vector(
         char*  prompt     /* in  */,
         float  local_v[]  /* out */,
         int    n_bar      /* in  */,
         int    p          /* in  */,
         int    my_rank    /* in  */) {
    int i, q;
    float temp[MAX_LOCAL_ORDER];
    MPI_Status status;

    if (my_rank == 0) {
        printf("Enter %s\n", prompt);
        for (i = 0; i < n_bar; i++)
            scanf("%f", &local_v[i]);
        for (q = 1; q < p; q++) {
            for (i = 0; i < n_bar; i++)
                scanf("%f", &temp[i]);
            MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
            &status);
    }
}  /* Read_vector */


/*********************************************************************/
float Serial_dot(
          float  x[]  /* in */,
          float  y[]  /* in */,
          int    n    /* in */) {

    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
        sum = sum + x[i]-y[i];
    return sum;
} /* Serial_dot */
```

```
/*********************************************************************/
float Parallel_dot(
          float  local_x[]  /* in */,
          float  local_y[]  /* in */,
          int  n_bar        /* in */) {

    float  local_dot;
    float  dot = 0.0;
    float  Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
         } /* Parallel_dot */
```

Алгоритм перевірки результатів програми

1) Виконання програми (див. рис. 3.2).

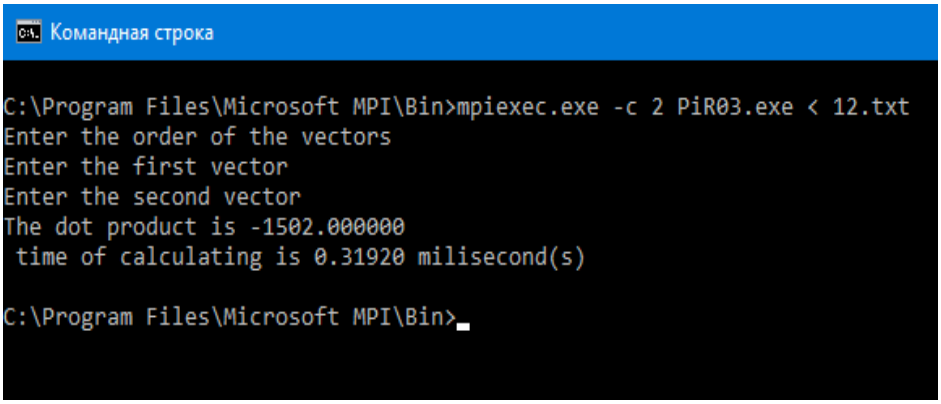

Рисунок 3.2 – Результати запуску програми (2 процесора)

2) Перевірка обчислень у MATLAB 2017 (див. рис. 3.3).

```
>> X1 = [242 330 -448 -80 -462 -22 -47 -37 407 75 65 -64 -464 -434 -179 -56 -30 -15
>> X2 = [431 101 9 325 109 239 328 -104 89 -425 -410 -486 -467 371 412 -495 172 -49
>> plot(1:250, X1,1:250, X2,1:250, X1-X2)
>> Res = sum(X1-X2)

Res =

       -1502
```

Рисунок 3.3 – Перевірка у MATLAB

Порівняючи результат на рис. 3.2 та рис. 3.3, можна зробити висновок, що результати повністю збігаються.

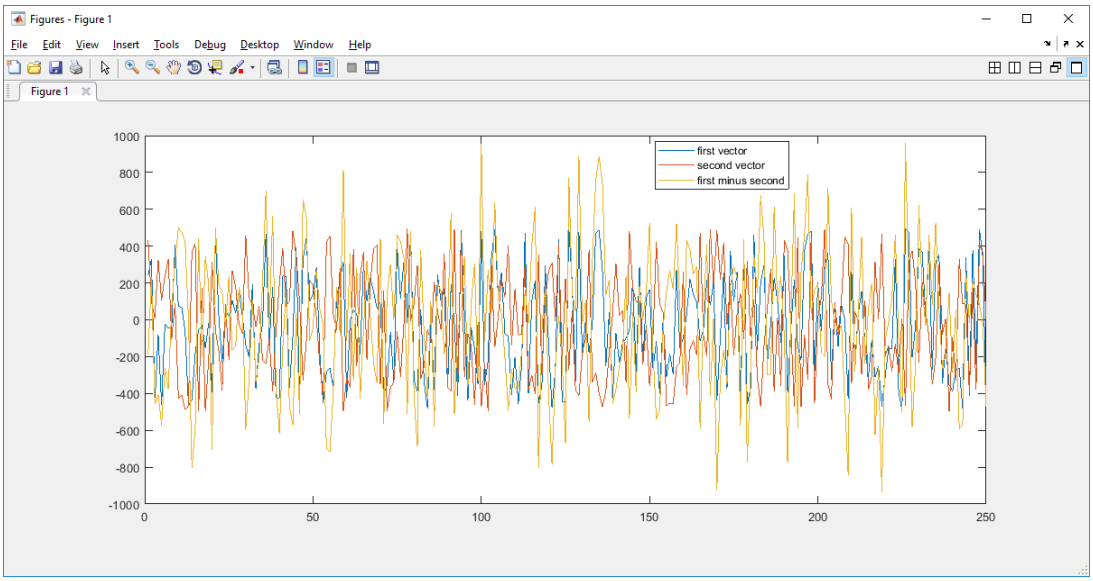

Рисунок 3.4 – Перший, другий вектор та їх скалярна операція (X1-X2)

У звіті привести схему паралельної обробки даних, описати фрагменти, які треба було додати у текст програми й оговорити результати роботи паралельної програми.

Лабораторна робота №4

ОБЧИСЛЕННЯ МНОЖЕННЯ МАТРИЦІ НА ВЕКТОР

Мета роботи: Розглянути можливості пакету MPI щодо виконання паралельного скалярного добутку між матрицею та вектором великої розмірності. Дослідити час виконання програми на одній та декількох машинах, виявити оптимальну кількість паралельних процесів для виконання поставленої задачі.

Теоретичні відомості

Для виконання даної роботи, окрім MPI_Bcast (див. лабораторну роботу №3), пропонується використати розбиття даних на рівні посилки за допомогою MPI_Scatter та MPI_Gather та MPI_Allgather, які «збиратимуть» дані в один буфер (масив).

4.1 Функція MPI_Scatter

MPI_Scatter має наступний синтаксис:

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

Параметри функції наведені у табл. 4.1, типи аргументів вказані безпосередньо у тілі функції.

Функція MPI_Scatter розбиває повідомлення з буфера відправки процесу «root» на рівні частини розміром «sendcount» та відправляє «i»-у частину в буфер прийому процесу з номером «i» (у тому числі й самому собі). Процес «root» використовує обидва буфери (відправки та прийому), тому у підпрограмі, що викликається, усі параметри є важливими. Інші процеси групи з комунікатором «comm» є лише отримувачами, тому для них параметри, що описують буфер відправки, не є суттєвими.

Таблиця 4.1 – Параметри MPI_Scatter

Тип	Параметри	Призначення
IN	sendbuf	адреса початку розміщення даних для розподілення (використовується тільки у процесі-відправнику root)
IN	sendcount	кількість елементів, що відправляються кожному процесу
IN	sendtype	тип елементів відправки
OUT	recvbuf	адреса початку буфера прийому
IN	recvcount	кількість елементів, що отримуються
IN	recvtype	тип елементів, що отримуються
IN	root	номер процесу-відправника
IN	comm	комунікатор

4.2 Функція MPI_Gather

MPI_Gather має наступний синтаксис:

```
MPI_GATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
            RECVCOUNT, RECVTYPE, ROOT, COMM)
```

Типи параметрів:

```
<type> SENDBUF(*), RECVBUF(*)
INT SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM
```

Схема роботи MPI_Gather:

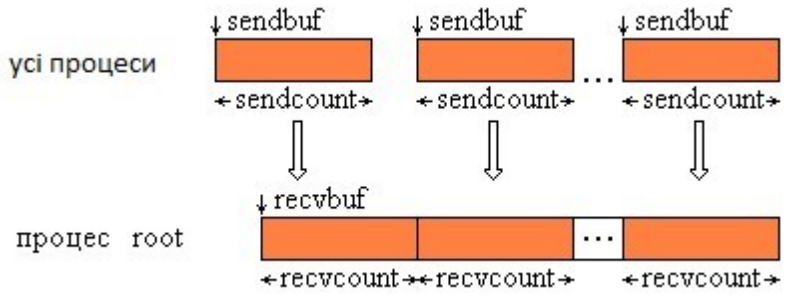

Рисунок 4.1 – Схема виконання MPI_Gather

Таблиця 4.2 – Параметри MPI_Gather

Тип	Параметри	Призначення
IN	sendbuf	адреса початку розміщення даних для відправки
IN	sendcount	кількість елементів, що відправляються
IN	sendtype	тип елементів відправки
OUT	recvbuf	адреса початку буфера прийому (використовується тільки у процесіотримувачі root)
IN	recvcount	кількість елементів, що отримуються від кожного процесу (використовується тільки у процесі-отримувачі root)
IN	recvtype	тип елементів, що отримуються
IN	root	номер процесу-отримувача
IN	comm	комунікатор

Функція MPI_Gather виконує збирання блоків даних, що відправляються усім процесам групи, в один масив процесу з номером «root». Довжина блоків має бути однакова. Поєднання виконується в порядок збільшення номерів процесів-відправників. Тобто дані, що відправлені процессом «i» зі свого буферу «sendbuf», розміщуються у «i»-у порцію буферу «recvbuf» процесу «root».

4.3 Функція MPI_Allgather

Функція MPI_Allgather виконується аналогічно MPI_Gather, але отримувачами є усі процеси групи. Дані, що відправлені процесом «i» зі свого буферу «sendbuf», розміщуються у «i»-у порцію буферу «recvbuf» кожного процесу. Після завершення операції вміст буферів прийому «recvbuf» у всіх процесів однаковий.

MPI_Allgather має наступний синтаксис:

```
MPI_ALLGATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
```

```
RECVCOUNT, RECVTYPE, COMM )
```

Типи параметрів:

```
<type> SENDBUF(*), RECVBUF(*)
INT SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM
```

Таблиця 4.3 – Параметри MPI_Allgather

Тип	Параметри	Призначення
IN	sendbuf	адреса початку розміщення даних для відправки
IN	sendcount	кількість елементів, що відправляються
IN	sendtype	тип елементів відправки
OUT	recvbuf	адреса початку буфера прийому
IN	recvcount	кількість елементів, що отримуються від кожного процесу
IN	recvtype	тип елементів, що отримуються
IN	comm	комунікатор

Схема виконання MPI_Allgather:

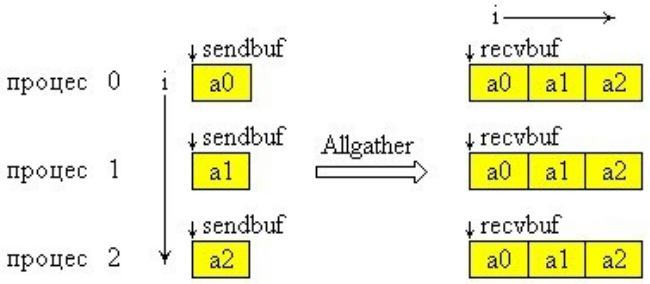

Рисунок 4.2 – Схема виконання MPI_Allgather

4.4 Матриці і матричні операції

Матриці і матричні операції широко використовуються при математичному моделюванні найрізноманітніших процесів, явищ і систем. Матричні обчислення складають основу наукових і інженерних розрахунків. Серед областей додатків можуть бути обчислювальна математика, фізика, економіка та ін.

Будучи обчислювально-трудомісткими, матричні обчислення представляють собою класичну область застосування паралельних обчислень. З одного боку, використання високопродуктивних багатопроцесорних систем дозволяє істотно підвищити складність вирішуваних завдань. З іншого боку, в силу своєї простий формулювання матричні операції дають хорошу можливість для демонстрації багатьох прийомів і методів паралельного програмування.

Принципи розпаралелення:

– Стрічкове розбиття матриці. При стрічковому (block-striped) розбитті кожному потоку виділяється та чи інша підмножина рядків (rowwise – горизонтальне розбиття) або стовпців (columnwise – вертикальне розбиття) матриці. Поділ рядків і стовпців на смуги в більшості випадків відбувається на безперервній (послідовної) основі. При такому підході для горизонтального розбиття по рядках, наприклад, матриця A представляється у вигляді:

$$A=(A_0,A_1,...,A_{p-1})^T,\ A_i=(a_{i_0},a_{i_1},...,a_{i_{k-1}}),\ i_j=ik+j,\ 0\le j<k,\ k=m/p,$$

де ai = (ai1, ai2, ... ain), 0≤ i <m, є i-й рядок матриці A (передбачається, що кількість рядків m кратно числу обчислювальних елементів (процесорів і / або ядер) p, тобто m=k*p). У всіх алгоритмах множення матриці на вектор, будемо використовувати поділ даних на безперервній основі.

Інший можливий підхід до формування смуг полягає в застосуванні тієї чи іншої схеми чергування (циклічності) рядків або стовпців. Як правило, для чергування використовується число потоків p – в цьому випадку при горизонтальному розбитті матриця A набирає вигляду:

$$A=(A_0,A_1,...,A_{p-1})^T,\ A_i=(a_{i_0},a_{i_1},...,a_{i_{k-1}}),\ i_j=i+jp,\ 0\le j<k,\ k=m/p.$$

– Блочне розбиття матриці. При блоковому (chessboard block) поділі матриця ділиться на прямокутні набори елементів – при цьому, як правило,

використовується поділ на безперервній основі. Нехай кількість потоків становить p = s * q, кількість рядків матриці є кратним s, а кількість стовпців – кратним q, тобто m = k * s і n = l * q.

Представимо вихідну матрицю A у вигляді набору прямокутних блоків таким чином:

$$\begin{pmatrix} A_{00} & A_{02} & \dots A_{0q} \\ A_{s1} & A_{s2} & \dots A_{sq} \end{pmatrix}$$

де Aij – блок матриці, що складається з елементів:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \dots a_{i_0 j_{l-1}} \\ & \dots & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix},$$

$$= ik + v, 0 \le v < k, k = m/s, \; j_u = jl + u, 0 \le u \le l, l =$$

Множення матриці на вектор при поділі даних по рядках

При такому способі поділу даних в якості базової підзадачі може бути обрана операція скалярного множення одного рядка матриці на вектор. У загальному вигляді схема інформаційної взаємодії підзадач в ході виконуваних обчислень показана на рис.4.3.

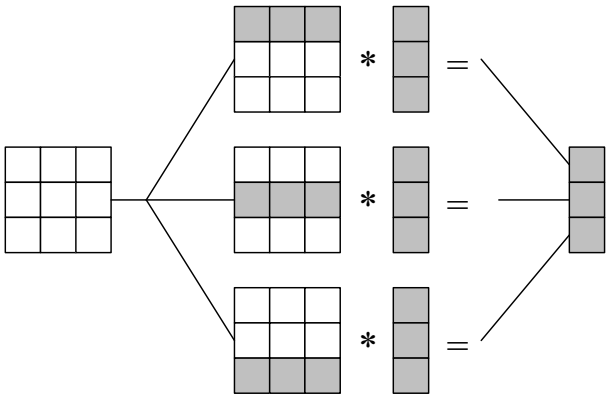

Рисунок 4.3 – Організація обчислень при виконанні паралельного алгоритму множення матриці на вектор, заснованого на поділі матриці по рядках

Множення матриці на вектор при поділі даних по стовпцях

При такому способі поділу даних в якості базової підзадачі може бути обрана операція множення стовпця матриці A на один з елементів вектору b. Схема інформаційної взаємодії підзадач в ході виконуваних обчислень показана на рис. 4.4.

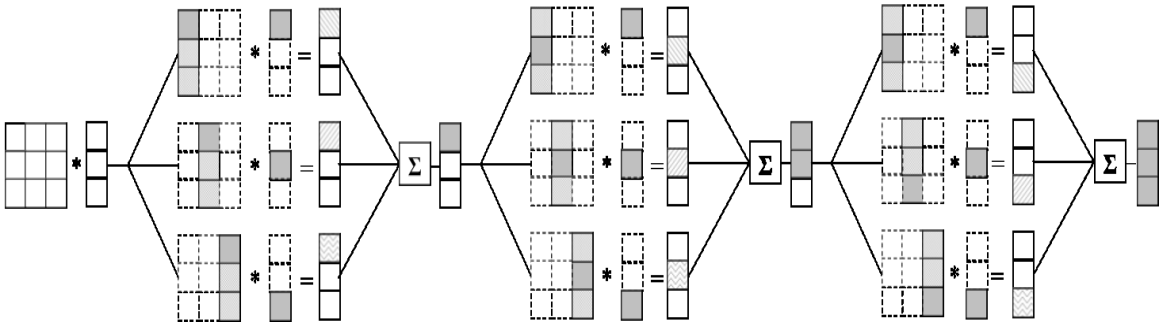

Рисунок 4.4 – Організація обчислень при виконанні паралельного алгоритму множення матриці на вектор з використанням розбиття матриці по стовпцях

В результаті такого підходу паралельні області будуть створюватися для обчислення кожного окремого елемента результуючого вектору. Програма буде представлятися в вигляді набору послідовних (однопотокових) і паралельних (багатопотокових) ділянок. Подібний принцип організації паралелізму отримав назву «вилочного» (fork-join) або пульсуючого паралелізму.

Множення матриці на вектор при блочному поділі даних

Розглянемо тепер паралельний алгоритм множення матриці на вектор, який заснований на іншому способі поділу даних - на розбитті матриці на прямокутні фрагменти (блоки). При такому способі поділу даних вихідна матриця A представляється у вигляді набору прямокутних блоків:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots A_{0q-1} \\ & \dots & \\ A_{s-11} & A_{s-12} & \dots A_{s-1q-1} \end{pmatrix},$$

де $A_{ij.}\ 0 \le i < s, 0 \le j < q,$ є блоком матриці:

$$A_{ij} = \begin{pmatrix} a_{i_0 j_0} & a_{i_0 j_1} & \dots a_{i_0 j_{l-1}} \\ & \dots & \\ a_{i_{k-1} j_0} & a_{i_{k-1} j_1} & a_{i_{k-1} j_{l-1}} \end{pmatrix}, \; i_v = ik + v, 0 \le v < k, k = m / s,$$

$$j_u = jl + u, 0 \le u \le l, l = n / q$$

Загальна схема виконуваних обчислень для множення матриці на вектор при блочному поділі даних показана на рис. 4.5.

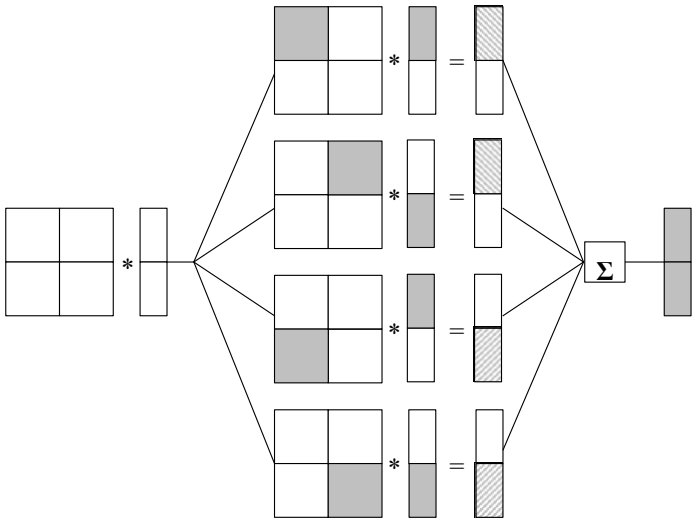

Рисунок 4.5 – Загальна схема паралельного алгоритму множення матриці на вектор при блочному поділі даних

Розглянувши представлену схему паралельних обчислень, можна зробити висновок про те, що інформаційна залежність базових підзадач проявляється тільки на етапі підсумовування результатів перемноження блоків матриці A і блоків вектору b.

Завдання: Розробити програму у середовищі C++ з використанням функцій пакету MS-MPI, у якій дослідити оптимальну кількість процесорів для виконання поставленої задачі та проаналізувати прискорення роботи програми.

Таблиця 4.4 – Варіанти індивідуальних завдань

Варіант	Kstb	Kstr	Osn
1	150	20	8
2	200	15	4
3	100	40	10
4	200	10	6
5	50	50	21
6	80	40	4
7	100	10	10
8	130	25	13
9	180	15	18
10	200	15	16
11	150	20	8
12	200	15	4
13	100	40	10
14	150	20	8
15	200	15	4

Приклад 4.1

Текст програми

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include "mpi.h"
#include <locale>
```

```
#define MAX_ORDER 200

typedef float LOCAL_MATRIX_T[MAX_ORDER][MAX_ORDER];
const int Kstb=200,
Kstr=15,
Osn=4;
void main(int argc, char* argv[])  {
      setlocale(LC_ALL,"Russian");
      int my_rank;
      int p;
      LOCAL_MATRIX_T  local_A;
      float global_x[MAX_ORDER];
      float local_x[MAX_ORDER];

    float local_y[MAX_ORDER];
      int m, n;
      int local_m, local_n;
      double t1,tim;
      FILE *f1=fopen("C:\\4.txt","wb");
      fprintf(f1,"%d\r\n",Kstb);
      fprintf(f1,"%d\r\n",Kstr);
      for (int i=0;i<(Kstb*Kstr+Kstr);i++){
            m=int(rand()%Osn);
            fprintf(f1,"%d ",m);
      }
      fclose(f1);
            void Read_matrix(char* prompt, LOCAL_MATRIX_T local_A, int
local_m, int n, int my_rank, int p,FILE *f);
      void Read_vector(char* prompt, float local_x[], int local_n, int
my_rank, int p,FILE *f);

      void Parallel_matrix_vector_prod( LOCAL_MATRIX_T local_A, int m, int
n, float local_x[], float global_x[], float local_y[], int local_m, int
local_n);

      void Print_matrix(char* title, LOCAL_MATRIX_T local_A, int local_m,
int n, int my_rank, int p);
      void Print_vector(char* title, float local_y[], int local_m, int
my_rank, int p);
      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &p);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      FILE *f=fopen("C:\\4.txt","rb");
      if (my_rank == 0)     {
            fscanf(f,"%d %d", &m, &n);
      }
      MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    local_m = m/p;
      local_n = n/p;

    Read_matrix("", local_A, local_m, n, my_rank, p,f);
      Print_matrix("Введена матрица", local_A, local_m, n, my_rank, p);
     Read_vector("", local_x, local_n, my_rank, p,f);
       Print_vector("Введен вектор", local_x, local_n, my_rank, p);
       t1=MPI_Wtime();
       Parallel_matrix_vector_prod(local_A, m, n, local_x, global_x,
local_y, local_m, local_n);
       tim=MPI_Wtime()-t1;
       Print_vector("Произведение равно ", local_y, local_m, my_rank, p);
       printf("Время выполнения %f микросекунд(ы)\n",tim*100000);
```

```
 fclose(f);
 MPI_Finalize();

}  /* main */

/********************************************************** ************/
void Read_matrix(
                         char* prompt   /* in  */,
                         LOCAL_MATRIX_T  local_A  /* out */,
                         int             local_m  /* in  */,
                         int             n        /* in  */,
                         int             my_rank  /* in  */,
                         int             p        /* in  */,
                         FILE *f)
{

    int i, j;
      LOCAL_MATRIX_T  temp;


      for (i = 0; i < p*local_m; i++)
            for (j = n; j < MAX_ORDER; j++)
                  temp[i][j] = 0.0;

    if (my_rank == 0){
            printf("%s\n", prompt);
            for (i = 0; i < p*local_m; i++)
                  for (j = 0; j < n; j++)
                        fscanf(f,"%f",&temp[i][j]);
    }
      MPI_Scatter(temp, local_m*MAX_ORDER, MPI_FLOAT, local_A,
local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);
}  /* Read_matrix */

void Read_vector(
                         char*  prompt     /* in  */,
                         float  local_x[]  /* out */,
                         int    local_n    /* in  */,
                         int    my_rank    /* in  */,
                         int    p          /* in  */,
                         FILE *f)
{

    int   i;
      float temp[MAX_ORDER];
     if (my_rank == 0){
             printf("%s\n", prompt);
             for (i = 0; i < p*local_n; i++)
                   fscanf(f,"%f",&temp[i]);
       }
       MPI_Scatter(temp, local_n, MPI_FLOAT, local_x, local_n, MPI_FLOAT, 0,
MPI_COMM_WORLD);
}  /* Read_vector */


void Parallel_matrix_vector_prod(
                                    LOCAL_MATRIX_T  local_A     /* in  */,
                                    int             m           /* in  */,
                                    int             n           /* in  */,
                                    float           local_x[]   /* in  */,
```

```
                                    float           global_x[]  /* in  */,
                                    float           local_y[]   /* out */,
                                    int             local_m     /* in  */,
                                    int             local_n     /* in  */)
{
     /* local_m = m/p, local_n = n/p */

    int i, j;

    MPI_Allgather(local_x, local_n, MPI_FLOAT, global_x, local_n, MPI_FLOAT,
MPI_COMM_WORLD);
      for (i = 0; i < local_m; i++)    {
            local_y[i] = 0.0;
            for (j = 0; j < n; j++)
                  local_y[i] = local_y[i] + local_A[i][j]*global_x[j];
      }
}
/* Parallel_matrix_vector_prod */

void Print_matrix(
                        char*           title      /* in */,
                        LOCAL_MATRIX_T  local_A    /* in */,
                  int             local_m    /* in */,
                        int             n          /* in */,
                        int             my_rank    /* in */,
                        int             p          /* in */)
{

    int   i, j;
      float temp[MAX_ORDER][MAX_ORDER];

    MPI_Gather(local_A, local_m*MAX_ORDER, MPI_FLOAT, temp,
local_m*MAX_ORDER, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (my_rank == 0){
            printf("%s\n", title);
            for (i = 0; i < p*local_m; i++) {
                  for (j = 0; j < n; j++)
                        printf("%4.1f ", temp[i][j]);
                  printf("\n");
            }
      }
}  /* Print_matrix */

void Print_vector(
                        char*  title      /* in */,
                        float  local_y[]  /* in */,
                        int    local_m    /* in */,
                        int    my_rank    /* in */,
                        int    p          /* in */)
{     int   i;
    float temp[MAX_ORDER];
      MPI_Gather(local_y, local_m, MPI_FLOAT, temp, local_m, MPI_FLOAT, 0,
MPI_COMM_WORLD);
      if (my_rank == 0){
            printf("%s\n", title);
            for (i = 0; i < p*local_m; i++)
                  printf("%4.1f ", temp[i]);
            printf("\n");
      }
}  /* Print_vector */
```

Алгоритм перевірки результатів програми

1) Запуск програми на одному та декількох процесорах. Порівняння часу виконання.

```
Введен вектор
 0,0  0,0  2,0  2,0  2,0  2,0  3,0  1,0  3,0  2,0  1,0  3,0  0,0  3,0  2,0
Произведение равно
35,0 38,0 54,0 40,0 26,0 36,0 45,0 26,0 46,0 44,0 39,0 19,0 49,0 44,0 48,0 27,0 48,0 39,0 31,0 26,0 28,0 34,0 44,0 27,0
33,0 42,0 37,0 47,0 26,0 40,0 40,0 42,0 38,0 42,0 32,0 29,0 31,0 31,0 44,0 63,0 53,0 28,0 36,0 33,0 41,0 31,0 29,0 51,0
50,0 38,0 34,0 50,0 25,0 31,0 37,0 27,0 31,0 47,0 35,0 45,0 56,0 28,0 44,0 36,0 28,0 34,0 25,0 26,0 42,0 24,0 49,0 40,0
36,0 37,0 46,0 48,0 37,0 23,0 24,0 35,0 40,0 27,0 43,0 48,0 40,0 29,0 34,0 29,0 36,0 51,0 43,0 44,0 35,0 32,0 30,0 40,0
36,0 38,0 42,0 55,0 33,0 39,0 46,0 38,0 33,0 38,0 47,0 52,0 45,0 35,0 43,0 39,0 38,0 32,0 33,0 40,0 44,0 47,0 37,0 26,0
37,0 36,0 36,0 20,0 28,0 39,0 44,0 34,0 47,0 59,0 40,0 26,0 50,0 48,0 41,0 31,0 41,0 21,0 37,0 53,0 50,0 61,0 44,0 50,0
51,0 45,0 42,0 58,0 44,0 30,0 34,0 58,0 19,0 38,0 48,0 48,0 45,0 47,0 66,0 34,0 38,0 36,0 55,0 46,0 42,0 51,0 35,0 29,0
33,0 32,0 43,0 34,0 41,0 26,0 41,0 32,0 34,0 26,0 35,0 43,0 52,0 25,0 60,0 32,0 36,0 40,0 58,0 55,0 34,0 34,0 34,0 33,0
33,0 55,0 36,0 45,0 37,0 43,0 33,0 31,0
Время выполнения 30,100011 микросекунд(ы)
```

Рисунок 4.6 – Запуск на 2 машинах

Отриманий час – на машині з ранком 0 – 11,20 мікросекунди. 65.98 – усі машини разом (загальний час). (Середній час – 9.8251 та 96.3247 відповідно). Другий - запуск на PC01 та вивід результату у PC01 – за 4 мікросекунди. (Середній час 15.2546 мікросекунд)

Отримаємо вектор-стовпець (розміром 200 елементів) за схемою скалярного добутку матриці на вектор-стовпець (див. рис. 4.5).

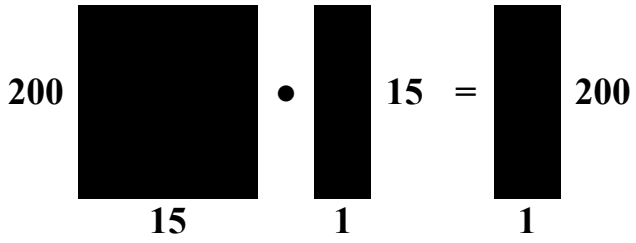

Рисунок 4.7 – Схема добутку матриці на вектор

2) Перевірка у MATLAB 2017b.

Опишемо змінну матриці та вектору:

```
Matrix = [ 200  рядків по 15 елементів в кожному ]
Vector = [ 15 елементів]
```

Щоб виконати скалярний добуток необхідно виконати транспонування вектор-рядку у вектор-стовпець. Для більш легкої праці виконаємо транспонування результату (щоб отримати вектор-рядок).

```
Result = (Matrix*Vector')'
```

Result =

35	38	54	40	26	36	45	26	46	44	39	19	49	44	48	27
48	39	31	26	28	34	44	27	33	42	37	47	26	40	40	42
38	42	32	29	31	31	44	63	53	28	36	33	41	31	29	51
50	38	34	50	25	31	37	27	31	47	35	45	56	28	44	36
28	34	25	26	42	24	49	40	36	37	46	48	37	23	24	35
40	27	43	48	40	29	34	29	36	51	43	44	35	32	30	40
36	38	42	55	33	39	46	38	33	38	47	52	45	35	43	39
38	32	33	40	44	47	37	26	37	36	36	20	28	39	44	34
47	59	40	26	50	48	41	31	41	21	37	53	50	61	44	50
51	45	42	58	44	30	34	58	19	38	48	48	45	47	66	34
38	36	55	46	42	51	35	29	33	32	43	34	41	26	41	32
34	26	35	43	52	25	60	32	36	40	58	55	34	34	34	33
33	55	36	45	37	43	33	31								

Виконуючи віднімання результату, який отримали у MATLAB від результату, який отримали у MPI, отримаємо вектор-строку нулів. Тобто всі значення збігаються та паралельне обчислення за допомогою пакету MPI виконано правильно.

У звіті привести схему паралельної обробки даних, описати фрагменти, які треба було додати у текст програми й оговорити результати роботи паралельної програми.

Лабораторна робота №5

ОБЧИСЛЕННЯ МНОЖЕННЯ МАТРИЦІ НА МАТРИЦЮ

Мета роботи: отримати паралельний скалярний добутку матриць великої розмірності. Дослідити час виконання програми на одній та декількох машинах, виявити оптимальну кількість паралельних процесів для виконання поставленої задачі.

Теоретичні відомості

Високопродуктивні матричні обчислення можна реалізувати із застосуванням як багатопоточного, так і розподіленого програмування. У даній темі як приклад подібних обчислень розглядається одна з найпростіших матричних задач – множення матриць, для якої наводяться різні алгоритми, засновані на пересилання повідомлень.

5.1 Стрічкові алгоритми множення матриць

У *стрічкових алгоритмах* матриці розбиваються на безперервні послідовності рядків або стовпців (стрічки, або смуги). Зокрема, смугою може служити окремий рядок або стовпець.

Стрічковий алгоритм

Спочатку проводиться розсилка в процес рангу K елементів K-го рядка матриці A і елементів K-го стовпця матриці B.

Потім запускається цикл (число ітерацій одно N), в ході якого виконуються дві дії:

1) виконується множення рядка матриці A і стовпці матриці B, що містяться в даному процесі, і результат записується у відповідний елемент рядка c;

2) виконується циклічна пересилання стовпців матриці B в сусідні процеси (напрямок пересилки може бути довільним: як по зростанню рангів процесів, так і по їх зменшенням).

Після завершення циклу в кожному процесі буде міститися рядок c, рівна одній з рядків твору AB (номер рядка відповідає номеру рядка матриці A, переслати цього процесу на початку виконання алгоритму). Чи залишиться переслати рядки c головному процесу (процесу рангу 0).

У таблиці 1 наведена схема алгоритму для матриць A і B близько 4 за умови, що циклічна пересилання стовпців матриці B виконується в напрямку зменшення рангів процесів. Рядки і стовпці, а також номери ітерацій тут і далі нумеруються від 0.

Таблиця 5.1 − Схема алгоритма для матриць A і B

Номер ітерації	Процес	Рядок матриці A	Стовпець матриці B	Результат (елемент матриці C)
0	0	A_0	B^0	$c00$
	1	A_1	B^1	$c11$
	2	A_2	B^2	$c22$
	3	A_3	B^3	$c33$
1	0	A_0	B^3	$c03$
	1	A_1	B^0	$c10$
	2	A_2	B^1	$c21$
	3	A_3	B^2	$c32$
2	0	A_0	B^2	$c02$
	1	A_1	B^3	$c13$
	2	A_2	B^0	$c20$
	3	A_3	B^1	$c31$
3	0	A_0	B^1	$c01$
	1	A_1	B^2	$c12$
	2	A_2	B^3	$c23$
	3	A_3	B^0	$c30$

Блокові алгоритми множення матриць

В даних алгоритмах матриці розбиваються на блоки, що представляють собою подматріці вихідних матриць. Для простоти будемо припускати, що всі матриці є квадратними розміру [redacted], а кількість блоків

по горизонталі і по вертикалі є однаковим і рівним ■ (при цьому розмір кожного блоку дорівнює [redacted], де [redacted]). У цьому випадку операція матричного множення може бути представлена в блоковому вигляді:

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,q-1} \\ & & \dots & \\ A_{q-1,0} & A_{q-1,1} & \dots & A_{q-1,q-1} \end{pmatrix} \times \begin{pmatrix} B_{0,0} & B_{0,1} & \dots & B_{0,q-1} \\ & & \dots & \\ B_{q-1,0} & B_{q-1,1} & \dots & B_{q-1,q-1} \end{pmatrix} =$$

$$= \begin{pmatrix} C_{0,0} & C_{0,1} & \dots & C_{0,q-1} \\ & & \dots & \\ C_{q-1,0} & C_{q-1,1} & \dots & C_{q-1,q-1} \end{pmatrix}$$

При цьому кожен блок ■ матриці ■ визначається як добуток відповідних блоків матриць ■ і ■:

[redacted]

Блоковий алгоритм (алгоритм Кеннона)

Алгоритм Кеннона відрізняється від алгоритму Фокса двома аспектами. По-перше, початкова пересилка блоків матриць A і B в процеси виконується таким чином, щоб одержувані блоки відразу могли бути перемножити без будь-яких пересилань даних. По-друге, при організації циклу виконується циклічна пересилання не тільки блоків матриці B (по стовпцях), але і блоків матриці A (по рядках).

Дії по початковій пересилання складаються з наступних кроків:

1) в кожен процес (i, j) пересилаються блоки Aij і Bij, матриця Cij обнуляється;

2) для кожного рядка i (i = 0, ..., q - 1) декартовой решітки процесів виконується циклічний зсув блоків матриці A на i позицій вліво (т. Е. В напрямку зменшення номерів стовпців);

3) для кожного стовпця j (j = 0, ..., q - 1) декартовой решітки процесів виконується циклічний зсув блоків матриці B на j позицій вгору (т. Е. В напрямку зменшення номерів рядків).

Результат подібного перерозподілу блоків в разі q = 3 наведено в таблиці 4.

Потім запускається цикл з q ітерацій, в ході якого виконуються три дії:

1) що містяться в процесі (i, j) блоки матриць A і B перемножуються, і результат додається до матриці Cij;

2) для кожного рядка i (i = 0, ..., q - 1) виконується циклічна пересилання блоків матриці A, що містяться в кожному процесі (i, j) цього рядка, в напрямку зменшення номерів стовпців;

3) для кожного стовпця j (j = 0, ..., q - 1) виконується циклічна пересилання блоків матриці B, що містяться в кожному процесі (i, j) цього стовпця, в напрямку зменшення номерів рядків.

На останній ітерації дії (2) і (3) можна не виконувати.

Після завершення циклу в кожному процесі буде міститися матриця Cij, рівна відповідному блоку твори AB. Чи залишиться переслати ці блоки головному процесу.

Таблиця 5.2 – Схема алгоритма Кеннона

Про-цес	Вихідний блок A і напрямок його зсуву	Вихідний блок B і напрямок його зсуву	Отриманий блок A	Отриманий блок B
(0,0)	A_{00}, зсув вліво на 0	B_{00}, зсув вгору на 0	A_{00}	B_{00}
(0,1)	A_{01}, зсув вліво на 0	B_{01}, зсув вгору на 1	A_{01}	B_{11}
(0,2)	A_{02}, зсув вліво на 0	B_{02}, зсув вгору на 2	A_{02}	B_{22}
(1,0)	A_{10}, зсув вліво на 1	B_{10}, зсув вгору на 0	A_{11}	B_{10}
(1,1)	A_{11}, зсув вліво на 1	B_{11}, зсув вгору на 1	A_{12}	B_{21}
(1,2)	A_{12}, зсув вліво на 1	B_{12} зсув вгору на 2	A_{10}	B_{02}
(2,0)	A_{20}, зсув вліво на 2	B_{20} зсув вгору на 0	A_{22}	B_{20}
(2,1)	A_{21}, зсув вліво на 2	B_{21}, зсув вгору на 1	A_{20}	B_{01}
(2,2)	A_{22}, зсув вліво на 2	B_{22}, зсув вгору на 2	A_{21}	B_{12}

$$T_p = q[(n^2 / p) \cdot (2n / q - 1) + (n^2 / p)] \cdot \tau + (2q + 2)(\alpha + w(n^2 / p) / \beta)$$

В виразах параметр q = √p визначає розмір процесорної решітки.

Завдання: Розробити програму у середовищі C++ з використанням функцій пакету MS-MPI, у якій виконати скалярний добуток матриць розмірності MxN. Дослідити час виконання. Перевірити коректність виконання розрахунків програми на матрицях невеликої розмірності.

Таблиця 5.3 – Варіанти індивідуальних завдань

Варіант	M	N
1	10	10
2	15	15
3	11	11
4	16	16
5	12	12
6	17	17
7	13	13
8	20	20
9	19	19
10	25	25
11	10	10
12	15	15
13	11	11
14	16	16
15	12	12

Приклад 5.1

Текст програми

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

#define MATRIX_ORDER 15
#define ARRAY_ORDER 300
#define Entry(mat,i,j) (mat[MATRIX_ORDER*(i)+(j)])
```

```
void Initialize(float my_matrix[], int my_rank);
void Mult(float product[],float factor[]);
void Print_matrix(char* title, float matrix[]);

int main(int argc, char** argv) {
      int m;
      FILE *f=fopen("C:\\4.txt","wb");
      for (int i=0;i<ARRAY_ORDER*2;i++){
            m=int(rand()%10);
            fprintf(f,"%d ",m);}
      fclose(f);
      float my_matrix[ARRAY_ORDER] ;
      float temp[ARRAY_ORDER] ;
      float product[ARRAY_ORDER];
      FILE *f1=fopen("C:\\4.txt","rb");
      for (int i=0;i<ARRAY_ORDER;i++)
            fscanf(f,"%f",&my_matrix[i]);
      for (int i=0;i<ARRAY_ORDER;i++)
            fscanf(f,"%f",&temp[i]);
      fclose(f1);
      int size;
      int p;
      int my_rank;
      MPI_Status status;
      int i;
     MPI_Init(&argc,&argv);
       MPI_Comm_size(MPI_COMM_WORLD,&size);
       if(size!=2){
             printf("Should be run in 2 processes\n");
             MPI_Finalize();
             return 0;
       }
       else
       {
             MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
           if(my_rank==0)
               {

MPI_Send(my_matrix,ARRAY_ORDER,MPI_FLOAT,1,0,MPI_COMM_WORLD );
               }
               else
               {

MPI_Recv(product,ARRAY_ORDER,MPI_FLOAT,0,0,MPI_COMM_WORLD,& status);
                     Print_matrix("The 1ST is", product);
                     double t=MPI_Wtime();
                     Mult(product, temp);
                     double t1=MPI_Wtime()-t;
                     Print_matrix("The 2ND matrix",temp);
                     Print_matrix("The product is", product);
                     printf("Время выполнения %f микросекунд(-
ы)\n",t1*1000000);
               }
               MPI_Finalize();
               return 0;
       }
}
void Initialize(
```

```
                        float  my_matrix[]  /* out */,
                        int    my_rank      /* in  */)
{
      my_matrix[0] = my_matrix[3] = (float) my_rank;
      my_matrix[1] = (float) (my_rank + 1);
      my_matrix[2] = (float) (my_rank + 2);
}  /* Initialize */

void Mult(
              float  product[]  /* in/out */,
              float  factor[]   /* in     */)
{
      int    i, j, k;
      float  temp[ARRAY_ORDER];

    for (i = 0; i < MATRIX_ORDER; i++)
            for (j = 0; j < MATRIX_ORDER; j++) {
                  Entry(temp,i,j) = 0.0;
                  for (k = 0; k < MATRIX_ORDER; k++)
                        Entry(temp,i,j) = Entry(temp,i,j) +
Entry(product,i,k)*Entry(factor,k,j);
            }

    for (i = 0; i < ARRAY_ORDER; i++)
            product[i] = temp[i];
}  /* Mult */

void Print_matrix(
                          char*  title     /* in */,
                          float  matrix[]  /* in */)
{

    int i, j;
      printf("%s\n", title);
      for (i = 0; i < MATRIX_ORDER; i++) {
            for (j = 0; j < MATRIX_ORDER; j++)
                  printf("%.f ", Entry(matrix,i,j));
            printf("\n");     }
     }  /* Print_matrix */
```

Алгоритм перевірки результатів програми

1) Виконання програми (див. рис. 5.1).

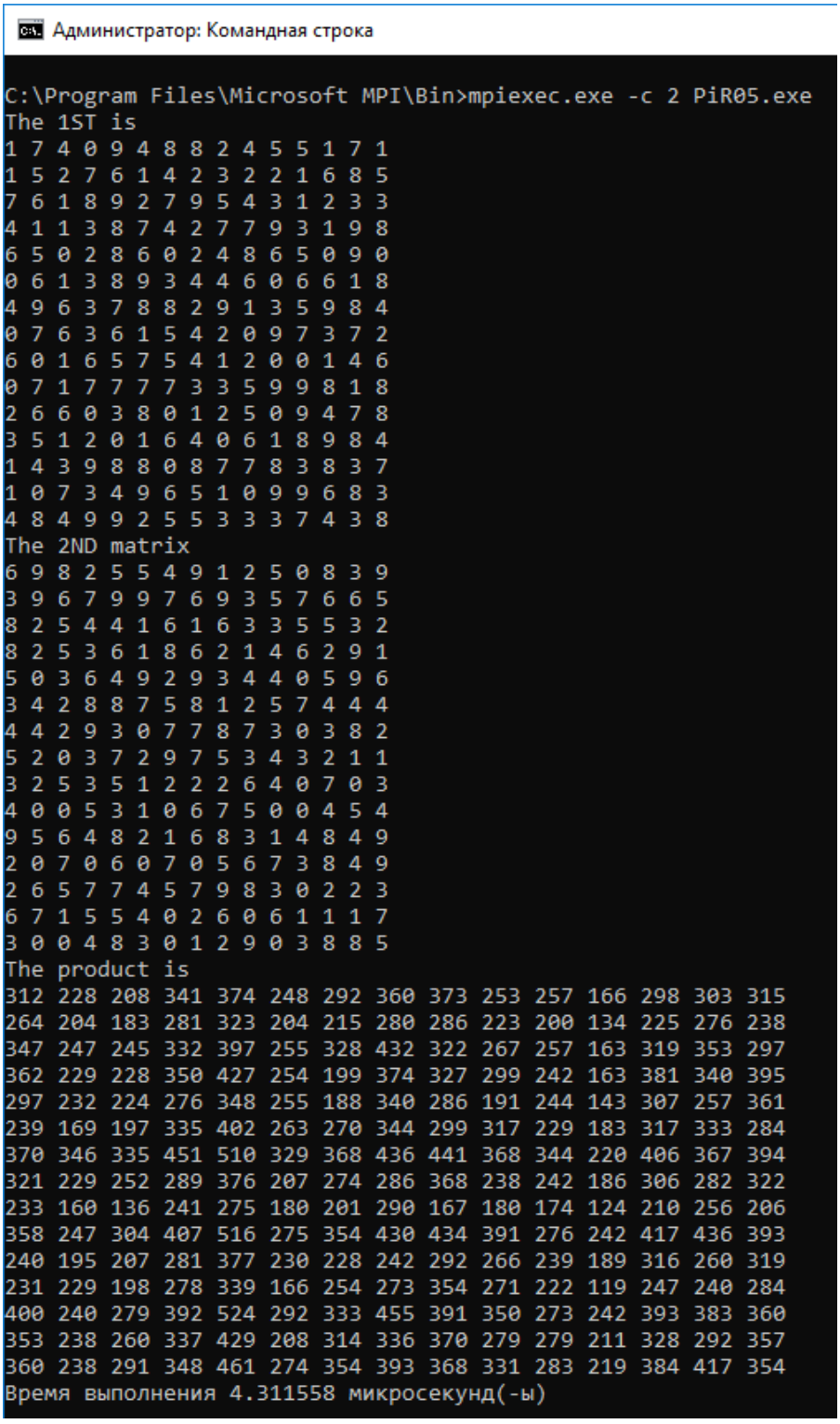

Рисунок 5.1 – Повний вигляд рішення

2) Перевіримо правильність розрахунків – скористаємося пакетом MatLab:

A = [ ];

Тут знаходяться данні із рис.5.2 – перша матриця

B = [ ];

Тут знаходяться данні із рис.5.2 – друга матриця

C = A*B

Отримаємо вивід результату добутку матриць

D = [ ];

Тут знаходяться данні із рис.5.1 матриця результату

Якщо рішення співпадає, то необхідно отримати діагональну матрицю (де елементи на головній діагоналі будуть рівними 1, а інші елементи – 0).

Результат тестування матлабу (див. рис. 5.2-5.4).

```
A =

     1     7     4     0     9     4     8     8     2     4     5     5     1     7     1
     1     5     2     7     6     1     4     2     3     2     2     1     6     8     5
     7     6     1     8     9     2     7     9     5     4     3     1     2     3     3
     4     1     1     3     8     7     4     2     7     7     9     3     1     9     8
     6     5     0     2     8     6     0     2     4     8     6     5     0     9     0
     0     6     1     3     8     9     3     4     4     6     0     6     6     1     8
     4     9     6     3     7     8     8     2     9     1     3     5     9     8     4
     0     7     6     3     6     1     5     4     2     0     9     7     3     7     2
     6     0     1     6     5     7     5     4     1     2     0     0     1     4     6
     0     7     1     7     7     7     7     3     3     5     9     9     8     1     8
     2     6     6     0     3     8     0     1     2     5     0     9     4     7     8
     3     5     1     2     0     1     6     4     0     6     1     8     9     8     4
     1     4     3     9     8     8     0     8     7     7     8     3     8     3     7
     1     0     7     3     4     9     6     5     1     0     9     9     6     8     3
     4     8     4     9     9     2     5     5     3     3     3     7     4     3     8

B =

     6     9     8     2     5     5     4     9     1     2     5     0     8     3     9
     3     9     6     7     9     9     7     6     9     3     5     7     6     6     5
     8     2     5     4     4     1     6     1     6     3     3     5     5     3     2
     8     2     5     3     6     1     8     6     2     1     4     6     2     9     1
     5     0     3     6     4     9     2     9     3     4     4     0     5     9     6
     3     4     2     8     8     7     5     8     1     2     5     7     4     4     4
     4     4     2     9     3     0     7     7     8     7     3     0     3     8     2
     5     2     0     3     7     2     9     7     5     3     4     3     2     1     1
     3     2     5     3     5     1     2     2     2     6     4     0     7     0     3
     4     0     0     5     3     1     0     6     7     5     0     0     4     5     4
     9     5     6     4     8     2     1     6     8     3     1     4     8     4     9
     2     0     7     0     6     0     7     0     5     6     7     3     8     4     9
     2     6     5     7     7     4     5     7     9     8     3     0     2     2     3
     6     7     1     5     5     4     0     2     6     0     6     1     1     1     7
     3     0     0     4     8     3     0     1     2     9     0     3     8     8     5
```

Рисунок 5.2 – Matlab перша частина

```
C =

   312   228   208   341   374   248   292   360   373   253   257   166   298   303   315
   264   204   183   281   323   204   215   280   286   223   200   134   225   276   238
   347   247   245   332   397   255   328   432   322   267   257   163   319   353   297
   362   229   228   350   427   254   199   374   327   299   242   163   381   340   395
   297   232   224   276   348   255   188   340   286   191   244   143   307   257   361
   239   169   197   335   402   263   270   344   299   317   229   183   317   333   284
   370   346   335   451   510   329   368   436   441   368   344   220   406   367   394
   321   229   252   289   376   207   274   286   368   238   242   186   306   282   322
   233   160   136   241   275   180   201   290   167   180   174   124   210   256   206
   358   247   304   407   516   275   354   430   434   391   276   242   417   436   393
   240   195   207   281   377   230   228   242   292   266   239   189   316   260   319
   231   229   198   278   339   166   254   273   354   271   222   119   247   240   284
   400   240   279   392   524   292   333   455   391   350   273   242   393   383   360
   353   238   260   337   429   208   314   336   370   279   279   211   328   292   357
   360   238   291   348   461   274   354   393   368   331   283   219   384   417   354


D =

   312   228   208   341   374   248   292   360   373   253   257   166   298   303   315
   264   204   183   281   323   204   215   280   286   223   200   134   225   276   238
   347   247   245   332   397   255   328   432   322   267   257   163   319   353   297
   362   229   228   350   427   254   199   374   327   299   242   163   381   340   395
   297   232   224   276   348   255   188   340   286   191   244   143   307   257   361
   239   169   197   335   402   263   270   344   299   317   229   183   317   333   284
   370   346   335   451   510   329   368   436   441   368   344   220   406   367   394
   321   229   252   289   376   207   274   286   368   238   242   186   306   282   322
   233   160   136   241   275   180   201   290   167   180   174   124   210   256   206
   358   247   304   407   516   275   354   430   434   391   276   242   417   436   393
   240   195   207   281   377   230   228   242   292   266   239   189   316   260   319
   231   229   198   278   339   166   254   273   354   271   222   119   247   240   284
   400   240   279   392   524   292   333   455   391   350   273   242   393   383   360
   353   238   260   337   429   208   314   336   370   279   279   211   328   292   357
   360   238   291   348   461   274   354   393   368   331   283   219   384   417   354
```

Рисунок 5.3 – Matlab друга частина

```
ans =

    1.0000   -0.0000   -0.0000   -0.0000    0.0000    0.0000    0.0000    0.0000    0.0000   -0.0000   -0.0000    0.0000    0.0000   -0.0000    0.0000
         0    1.0000         0         0         0         0         0         0         0         0         0         0         0         0         0
   -0.0000    0.0000    1.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000    0.0000   -0.0000   -0.0000    0.0000   -0.0000
   -0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000    0.0000   -0.0000   -0.0000    0.0000   -0.0000
    0.0000    0.0000    0.0000   -0.0000    1.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000    0.0000   -0.0000    0.0000    0.0000    0.0000
    0.0000   -0.0000   -0.0000   -0.0000    0.0000    1.0000    0.0000   -0.0000    0.0000    0.0000    0.0000    0.0000    0.0000   -0.0000    0.0000
    0.0000   -0.0000   -0.0000    0.0000    0.0000    0.0000    1.0000    0.0000    0.0000   -0.0000   -0.0000    0.0000    0.0000   -0.0000    0.0000
   -0.0000   -0.0000    0.0000    0.0000   -0.0000    0.0000   -0.0000    1.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000   -0.0000
   -0.0000   -0.0000   -0.0000    0.0000    0.0000    0.0000    0.0000    0.0000    1.0000   -0.0000   -0.0000    0.0000   -0.0000   -0.0000    0.0000
    0.0000   -0.0000   -0.0000   -0.0000    0.0000   -0.0000    0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000    0.0000   -0.0000    0.0000
   -0.0000    0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000    1.0000   -0.0000   -0.0000    0.0000   -0.0000
   -0.0000    0.0000    0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000   -0.0000
         0         0         0         0         0         0         0         0         0         0         0         0    1.0000         0         0
   -0.0000    0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000   -0.0000    0.0000    0.0000   -0.0000   -0.0000    1.0000   -0.0000
         0         0         0         0         0         0         0         0         0         0         0         0         0         0    1.0000
```

Рисунок 5.4 – Matlab третя частина

У звіті привести схему паралельної обробки даних, описати фрагменти, які треба було додати у текст програми й оговорити результати роботи паралельної програми.

Лабораторна робота №6

ЗНАЙОМСТВО ІЗ СТРУКТУРОЮ MPI-ПРОГРАМИ ТА ПРОЦЕДУРАМИ БЛОКУЮЧОГО ДВОТОЧКОВОГО ОБМІНУ MPI

Мета роботи: ознайомитися із структурою MPI-програми та процедурами блокуючого двохкрапкового обміну MPI.

Теоретичні відомості

Учасниками двокрапкового обміну є два процеси: процес-відправник і процес-одержувач. Блокують операції двокрапкового обміну припиняють виконання викликає процесу. Він переходить в стан очікування завершення передачі даних. Блокування гарантує виконання дій в заданому порядку, забезпечуючи передбачуваність поведінки програми. З іншого боку, вона створює умови для виникнення тупикових ситуацій, коли обидва процеси-учасника обміну блокуються одночасно.

Основними підпрограмами двокрапкового обміну повідомленнями є **MPI_Send** і **MPI_Recv**.

MPI_Send(buf, count, datatype, dest, tag, comm) – відправка повідомлення. Всі параметри є вхідними:

– **buf** - адреса буфера відправки;

– **count** – кількість елементів, що пересилаються даних (невід'ємне ціле значення);

– **datatype** – тип даних, що пересилаються;

– **tag** – тег повідомлення (ціле значення);

– **comm** – комунікатор.

MPI_Recv(buf, count, datatype, source, tag, comm, status) – прийом повідомлення.

Вихідні параметри:

– **buf, count, type –** буфер пам'яті для прийому повідомлення,

призначення кожного окремого параметра відповідає опису в **MPI_Send;**

– **source –** ранг процесу, від якого повинен бути виконаний прийом повідомлення**;**

– **tag –** тег повідомлення, яке повинно бути прийнято для процесу**;**

– **comm –** комунікатор, в рамках якого відбуватиметься передача даних**;**

– **status –** покажчик на структуру даних з інформацією про результат виконання операції прийому даних**.**

MPI_Finalize() – завершення роботи з MPI.

MPI_Finalize повинні викликати всі процеси перед завершеним своєї роботи.

Мінлива status має тип MPI_Status і є структурою з полями **status.MPI_SOURCE** та **status.MPI_TAG.**

MPI_Init(int *argc, char **argv) – підключення до MPI. Аргументи **argc** і **argv** потрібні тільки в програмах на C, де вони при запуску програми задають кількість аргументів у командному рядку і вектор цих аргументів. Даний виклик передує всім іншим викликам підпрограм MPI.

MPI_Comm_rank(comm, pid) – визначення номера (рангу) процесу. Тут **pid** – ідентифікатор процесу, що пов'язується з областю взаємодії, що належить комунікатора **comm**.

MPI_Comm_size(comm, size) – визначення розміру області взаємодії. Тут **comm** – вхідний параметр, що задає ім'я комунікатора, а вихідним є параметр цілого типу **size**, що визначає кількість процесів в області взаємодії.

Завдання:

Завдання 1.

У вихідному тексті програми пропущені виклики процедур підключення до MPI, визначення кількості процесів та рангу процесу. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
int myid, numprocs;
// todo: add your code here....
fprintf(stdout,"Process %d of %d\n", myid, numprocs);
MPI_Finalize();
return 0;
}
```

Завдання 2.

У вихідному тексті програми пропущені виклики процедур стандартного блокуючого двохкрапкового обміну. Мається на увазі, що при запуску двох процесів один з них відправляє повідомлення іншому. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
int myid, numprocs;
char message[20];
int myrank;
MPI_Status status;
int TAG = 0;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
strcpy(message, "Hi, Second Processor!");
MPI_Send(/*todo: add your params here*/);
}
else
{
MPI_Recv(/*todo: add your params here*/);
printf("received: %s\n", message);
}
MPI_Finalize();
return 0;
}
```

Завдання 3.

У вихідному тексті програми пропущені виклики процедур стандартного блокуючого двохкрапкового обміну. Мається на увазі, що при запуску парного числа процесів, ті з них, які мають парний ранг,

відправляють повідомлення наступним по величині процесам. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
int myrank, size, message;
int TAG = 0;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
message = myrank;
if((myrank % 2) == 0)
{
if((myrank + 1) != size)
MPI_Send(/*todo: add your params here*/);
}
else
{
if(myrank != 0)
MPI_Recv(/*todo: add your params here*/);
printf("received :%i\n", message);
}
MPI_Finalize();
return 0;
}
```

Таблиця 6.1 – Індивідуальне завдання

№ варіанту	**№ завдання**
1	1,2
2	1,3
3	2,3
4	1,2
5	1,3
6	2,3
7	1,2
8	1,3
9	2,3
10	1,2
11	1,3
12	2,3
13	1,2
14	1,3

Лабораторна робота №7

ЗНАЙОМСТВО З ПРОЦЕДУРАМИ БУФЕРІЗОВАНОГО ТА НЕБЛОКУЮЧОГО ДВОХКРАПКОВОГО ОБМІНУ MPI

Мета роботи: ознайомитися із процедурами буферизованого та неблокуючого двохкрапкового обміну MPI.

Теоретичні відомості

Двохкрапковий буферізований обмін

Надсилаючи звіт про проблеми в буферізованому режимі джерело копіює повідомлення в буфер, а потім передає його в неблокуючому режимі. Виділення буфера і його розмір контролюються програмістом, який повинен заздалегідь створити буфер достатнього розміру. Буферізована передача завершується відразу, оскільки повідомлення негайно копіюється в буфер для подальшої передачі.

Після завершення роботи з буфером його необхідно відключити. Після відключення буфера можна знову використовувати займану ним пам'ять, проте слід пам'ятати, що в мові С даний виклик не звільняє автоматично пам'ять, відведену для буфера. Буферізований обмін рекомендується використовувати в тих ситуаціях, коли програмісту потрібно більший контроль над розподілом пам'яті.

При виконанні буферизованного обміну програміст повинен заздалегідь створити буфер достатнього розміру за допомогою виклику підпрограми:

*int MPI_Buffer_attach(void *buf, size)*

В результаті виклику створюється буфер з ім'ям buf і розміром size в байтах, який можна використовувати тільки один раз, після чого його потрібно відключити шляхом виклику підпрограми відключення:

*int MPI_Buffer_detach(void *buf, int *size)*

При виконанні відключення повертається адреса **(buf)** і розмір який відключається буфера **(size)**. Ця операція блокує роботу процесу до тих пір, поки всі повідомлення, що знаходяться в буфері, що не будуть оброблені. Завдяки цьому, виклик даної підпрограми можна використовувати для форсованої передачі повідомлень. Після завершення виклику можна знову використовувати пам'ять, яку займав буфер, проте слід пам'ятати, що в мові С даний виклик не звільняє автоматично пам'ять, відведену для буфера. Розмір буфера повинен перевищувати обсяг повідомлення на величину **MPI_BSEND_OVERHEAD**.

Двохкрапкові неблокуючі обміни

Виклик підпрограми неблокуючих передачі ініціює, але не завершує її. Передача даних з буфера або їх зчитування відбувається під час виконання інших операцій. Завершується обмін викликом додаткової процедури, яка перевіряє, скопійовані дані в буфер передачі. До завершення обміну запис в буфер або зчитування з нього виробляти не можна, так як повідомлення може бути ще не

відправлено або ніхто не почув. Неблокуюча передача може бути прийнята підпрограмою блокуючого прийому і навпаки.

Неблокуючий обмін виконується в два етапи:

1) Ініціалізація обміну.

2) Перевірка завершення обміну.

Для маркування неблокуючих операцій обміну використовуються ідентифікатори операцій обміну.

Перевірка фактичного виконання передачі або прийому в неблокуючому режимі здійснюється за допомогою виклику підпрограм очікування, які блокують роботу процесу до завершення операції або неблокуючих підпрограм перевірки, які повертають логічне значення

"істина", якщо операція виконана. Підпрограма MPI_Wait блокує роботу процесу до завершення прийому або передачі повідомлення. Функції MPI_Wait і MPI_Test можна використовувати для завершення операцій прийому і передачі.

Якщо при виконанні неблокуючих операцій виявиться, що продовження обчислень неможливо без отримання даних, що передаються, то може бути використана блокуюча операція очікування завершення операції:

*int MPI_Wait (MPI_Request * request, MPI_status * status),*

де

– request – дескриптор операції, певний при виклику неблокірующіх функції;

– status – результат виконання операції обміну (тільки для завершеною операції).

Перевірка стану виконуваної неблокуючої операції передачі даних проводиться за допомогою функції:

*int MPI_Test (MPI_Request * request, int * flag, MPI_status * status),*

де

– request – дескриптор операції, певний при виклику неблокірующіх функції;

– flag – результат перевірки (true, якщо операція завершена);

– status – результат виконання операції обміну (тільки для завершеною операції).

Підпрограми-пробники

Отримати інформацію про повідомленні до його приміщення в буфер прийому можна за допомогою підпрограм-пробників MPI_Probe і MPI_IProbe. На підставі отриманої інформації приймається рішення про

подальші дії. За допомогою виклику підпрограми MPI_Probe фіксується надходження (але не прийом!) Повідомлення. Потім визначається джерело повідомлення, його довжина, виділяється буфер відповідного розміру і стає можливим прийом повідомлення.

Підпрограма MPI_Probe виконує блокуючу перевірку доставки повідомлення:

*int MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status * status)*

де

– source – ранг джерела або джокер;
– tag – значення тега або джокер;
– comm – ім'я комунікатора.
– status – вихідний параметр, що містить потрібну інформацію.

Неблокуюча перевірка повідомлення виконується підпрограмою:

*Iint MPI_Iprobe (int source, int tag, MPI_Comm comm, int * flag, MPI_Status * status)*

Вхідні параметри ті ж, що і у підпрограми MPI_Probe, а вихідні flag – ім'я прапора і status – статус. Якщо повідомлення вже надійшло і може бути прийнято, змінна flag отримує значення true. Перевірка прийому може виконуватися за допомогою MPI_Iprobe неодноразово. Немає необхідності виконувати прийом повідомлення відразу ж після його надходження.

Завдання

Завдання 1.

У вихідному тексті програми пропущені виклики процедур буферизованого обміну. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
 #include <stdio.h>
 int main(int argc,char *argv[])
 {
 int *buffer;
 int myrank;
 MPI_Status status;
 int buffsize = 1;
 int TAG = 0;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if (myrank == 0)
 {
 buffer = (int *) malloc(buffsize + MPI_BSEND_OVERHEAD);
 /*...*/
 buffer = (int *) 10;
 /*...*/
 }
 else
 {
 MPI_Recv(&buffer, buffsize, MPI_INT, 0, TAG,
MPI_COMM_WORLD,
 &status);
 printf("received: %i\n", buffer);
 }
 MPI_Finalize();
 return 0;
 }
```

Завдання 2.

У вихідному тексті програми пропущені виклики підпрограм-пробників. Додати ці виклики, скомпілювати та запустити програму.

```
 #include "mpi.h"
 #include <stdio.h>
 int main(int argc,char *argv[])
 {
 int myid, numprocs, **buf, source, i;
 int message[3] = {0, 1, 2};
 int myrank, data = 2002, count, TAG = 0;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if(myrank == 0)
 {
 MPI_Send(&data, 1, MPI_INT, 2, TAG, MPI_COMM_WORLD);
 }
 else if (myrank == 1) {
 MPI_Send(&message, 3, MPI_INT, 2, TAG, MPI_COMM_WORLD);}
 else{
```

```
     /*...*/
     source = status.MPI_SOURCE;
     MPI_Get_count(/*...*/);
     for (i = 0; i < count; i++){
     buf[i] = (int *)malloc(count*sizeof(int));
     }
     MPI_Recv(&buf[0], count, MPI_INT, source, TAG,
MPI_COMM_WORLD,&status);
     for (i = 0; i < count; i++){
     printf("received: %d\n", buf[i]);}}
     MPI_Finalize();
     return 0;}
```

Лабораторна робота №8

Знайомство з процедурами колективного обміну MPI

Мета роботи: ознайомитися з процедурами колективного обміну MPI.

Теоретичні відомості

Колективний обмін даними може бути реалізовано у вигляді:

– широкомовлення (broadcast);

– збір даних (gather);

– розподіл даних (scatter);

– операції приведення (reduce) і сканування (scan).

Широкомовлення

При широкомовної передачі один і той же набір даних передається кожному процесу в межах загального для них комунікатора. Для виконання широкомовного обміну використовується підпрограма MPI_Bcast, яка викликається в усіх процесах з області взаємодії. При цьому повинні обов'язково використовуватися однакові значення параметрів за кількістю (count) і типом даних (datatype). Одна підпрограма викликається і для передачі і для отримання даних.

Сигнатура функції:

*int MPI_Bcast (void * buf, int count, MPI_Datatype type, int root, MPI_Comm comm),*

де

– buf, count, type – буфер пам'яті з якого відправляється повідомленням (для процесу з рангом 0) і для прийому повідомлень (для всіх інших процесів);

– count – кількість елементів в буфері +1;

– type – тип даних, що передються (приймаються).

– root – ранг процесу, що виконує розсилку даних;

– comm – комунікатор, в рамках якого відбуватиметься передача даних.

Функція MPI_Bcast здійснює розсилку даних з буфера buf, що містить count елементів типу type, з процесу, що має номер root, всім процесам, що входять в комунікатор comm.

Збір та розподіл даних

Розподіл і збір даних виконуються за допомогою підпрограм MPI_Scatter і MPI_Gather відповідно. Список аргументів у обох підпрограм однаковий, але діють вони по-різному.

За широкомовної розсилки всім процесам передається один і той самий набір даних, а при розподілі даних передаються його частини. Тобто різним поцесам передаються різні дані.

Сигнатура функції:

*int MPI_Scatter(void *sbuf, int scount, MPI_Datatype stype,*

*void *rbuf, int rcount, MPI_Datatype rtype, int root,*

MPI_Comm comm),

де

– sbuf, scount, stype – параметри переданого повідомлення (scount визначає кількість елементів, переданих на кожен процес);

– rbuf, rcount, rtype – параметри повідомлення, прийнятого в процесах;

– root – ранг процесу, що виконує розсилку даних;

– comm – комунікатор, в рамках якого відбуватиметься передача даних.

Під час виклику цієї функції процес з рангом root зробить передачу даних всім іншим процесам в комунікаторі. Кожному процесу буде відправлено scount елементів. Процес з рангом 0 отримає блок даних з sbuf елементів з індексами від 0 до scount-1, процесу з рангом 1 буде відправлений блок з sbuf елементів з індексами від scount до 2 * scount-1 і т.д. Таким чином, загальний розмір повідомлення, що відправляється має

дорівнювати scount * p елементів, де p є кількість процесів в комунікаторі comm.

Слід зазначити, що оскільки функція MPI_Scatter визначає колективну операцію, виклик цієї функції при виконанні розсилки даних повинен бути забезпечений на кожному процесі комунікатора.

Відзначимо також, що функція MPI_Scatter передає всім процесам повідомлення однакового розміру. Виконання більш загального варіанта операції розподілу даних, коли розміри повідомлень для процесів можуть бути різні, забезпечується за допомогою функції MPI_Scatterv.

На противагу попередній підпрограма MPI_Gather виконує збір повідомлень від інших процесів в буфер головного процесу.

Сигнатура функції:

*int MPI_Gather(void *sbuf, int scount, MPI_Datatype stype,*

*void *rbuf, int rcount, MPI_Datatype rtype,*

int root, MPI_Comm comm),

де,

– sbuf, scount, stype – параметри переданого повідомлення;

– rbuf, rcount, rtype – параметри прийнятого повідомлення;

– root – ранг процесу, що виконує збір даних;

– comm - комунікатор, в рамках якого відбуватиметься передача даних.

В обмінах, виконуваних подпрограммами MPI_Allgather і MPI_Alltoall, немає головного процесу. Деталі відправки і прийому важливі для всіх процесів, що беруть участь в обміні. MPI_Allgather збирає дані від всіх процесів і розподіляє їх усім процесам. Дія цієї підпрограми рівносильна дії послідовності викликів підпрограми MPI_Gather, в кожному з яких в якості головного використовуються різні процеси. Прийом виконується в усіх процесах. Буфер прийому послідовно заповнюється повідомленнями від всіх процесів-відправників.

Операції приведення і сканування

Операції приведення і сканування відносяться до категорії глобальних обчислень. У глобальній операції приведення до даних від всіх процесів із заданого комунікатора застосовується операція MPI_Reduce.

Аргументом операції приведення є масив даних – по одному елементу від кожного процесу. Результат такої операції – єдине значення.

Сигнатура функції:

*int MPI_Reduce(void *sendbuf, void *recvbuf, int count,*

MPI_Datatype type, MPI_Op op, int root, MPI_Comm comm),

де

– sendbuf – буфер пам'яті з повідомленням;

– recvbuf – буфер пам'яті для результуючого повідомлення (тільки для процесу з рангом root);

– count – кількість елементів в повідомленнях;

– type – тип елементів повідомлень;

– op – операція, яка повинна бути виконана над даними;

– root – ранг процесу, на якому повинен бути отриманий результат;

– comm – комунікатор, в рамках якого виконується операція.

У якості операції можуть бути використані визначені у MPI (див. табл. 8.1).

І ще один варіант операції збору і обробки даних, при якому забезпечується отримання всіх часткових результатів редукування, може бути реалізований за допомогою функції:

*int MPI_Scan(void *sendbuf, void *recvbuf, int count,*

MPI_Datatype type, MPI_Op op, MPI_Comm comm),

де

– sendbuf – буфер пам'яті з повідомленням;

– recvbuf – буфер пам'яті для результуючого повідомлення;

– count – кількість елементів в повідомленнях;

– type – тип елементів повідомлень;

– op – операція, яка повинна бути виконана над даними;

– comm - комунікатор, в рамках якого виконується операція.

Таблиця 8.1 – Операції MPI

Операции	Описание
MPI_MAX	Визначення максимального значення
MPI_MIN	Визначення мінімального значення
MPI_SUM	Визначення суми значень
MPI_PROD	Визначення добутку значень
MPI_LAND	Виконання логічної операції "І" над значеннями повідомлень
MPI_BAND	Виконання бітової операції "І" над значеннями повідомлень
MPI_LOR	Виконання логічної операції "АБО" над значеннями повідомлень
MPI_BOR	Виконання бітової операції "АБО" над значеннями повідомлень
MPI_LXOR	Виконання логічної операції виключає "АБО" над значеннями повідомлень
MPI_BXOR	Виконання бітової операції виключає "АБО" над значеннями повідомлень
MPI_MAXLOC	Визначення максимальних значень і їх індексів
MPI_MINLOC	Визначення мінімальних значень і їх індексів

3 Завдання:

Завдання 1.

У вихідному тексті програми пропущені виклики процедур широкомовної розсилки. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
char data[24];
int myrank, count = 25;
MPI_Status status;
```

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
strcpy(data, "Hi, Parallel Programmer!");
//...
printf("send: %s\n", data);
}
else
{
//...
printf("received: %s\n", data);
}
MPI_Finalize();
return 0;
}
```

Завдання 2.

Мається на увазі, що три чисельних значення, введених з клавіатури, пересилаються широкомовною розсилкою усім іншим процесам. Виклики підпрограм широкомовної розсилки пропущені. Додайте ці виклики, скомпілювати та запустити програму.

Текст програми із виправленнями:

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
int myrank;
int root = 0;
int count = 1;
float a, b;
int n;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0)
{
printf("Enter a, b, n\n");
scanf("%f %f %i", &a, &b, &n);
//...
}
else
{
//...
printf("%i Process got %f %f %i\n", myrank, a, b, n);
}
```

```
MPI_Finalize();
return 0;
}
```

Завдання 3.

Створюється новий комунікатор, а потім повідомлення між процесами, що входять до нього, пересилаються широкомовною розсилкою. Виклики підпрограм створення нової групи процесів (на 1 менше ніж повна кількість запущених на виконання процесів) та нового комунікатора пропущені. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
char message[24];
MPI_Group MPI_GROUP_WORLD;
MPI_Group group;
MPI_Comm fcomm;
int size, q, proc;
int* process_ranks;
int rank, rank_in_group;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
printf("New group contains processes:");
q = size - 1;
process_ranks = (int*) malloc(q*sizeof(int));
for (proc = 0; proc < q; proc++)
{
process_ranks[proc] = proc;
printf("%i ", process_ranks[proc]);
}
printf("\n");
//...
if (fcomm != MPI_COMM_NULL) {
MPI_Comm_group(group, &fcomm);
MPI_Comm_rank(fcomm, &rank_in_group);
if (rank_in_group == 0) {
strcpy(message, "Hi, Parallel Programmer!");
MPI_Bcast(&message, 25, MPI_BYTE, 0, fcomm);
printf("0 send: %s\n", message);
}
else
{
MPI_Bcast(&message, 25, MPI_BYTE, 0, fcomm);
printf("%i received: %s\n", rank_in_group, message);
```

```
}
MPI_Comm_free(&fcomm);
MPI_Group_free(&group);
}
MPI_Finalize();
return 0;}
```

Завдання 4.

Спочатку створюється підгрупа, що складається за процесів із рангами 1, 3, 5 і 7, та відповідний до неї комунікатор. Потім виконується редукція(сумування) за процесами, що входять у нову групу. Виклик підпрограми редукції та деякі інші важливі фрагменти пропущені. Додати ці виклики, скомпілювати та запустити програму.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[])
{
int myrank, i;
int count = 5, root = 1;
MPI_Group MPI_GROUP_WORLD, subgroup;
int ranks[4] = {1, 3, 5, 7};
MPI_Comm subcomm;
int sendbuf[5] = {1, 2, 3, 4, 5};
int recvbuf[5];
MPI_Init(&argc, &argv);
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup);
MPI_Group_rank(subgroup, &myrank);
\\...
if(myrank != MPI_UNDEFINED)
{
MPI_Reduce(&sendbuf,  &recvbuf,  count,  MPI_INT,  MPI_SUM,
root,
subcomm);
if(myrank == root) {
printf("Reduced values");
for(i = 0; i < count; i++){
printf(" %i ", recvbuf[i]);}
}
printf("\n");
MPI_Comm_free(&subcomm);
MPI_Group_free(&MPI_GROUP_WORLD);
\\...
}
MPI_Finalize();
return 0;
}
```

Таблиця 8.2 – Індивідуальне завдання

№ варіанту	**№ завдання**
1	1,2
2	1,3
3	1,4
4	2,2
5	2,3
6	2,4
7	3,4
8	1,2
9	1,3
10	1,4
11	2,2
12	2,3
13	2,4
14	3,4

Список використаної літератури

1. Gropp, William; Lusk, Ewing; Skjellum, Anthony (2014). Using MPI, 3rd edition: Portable Parallel Programming with the Message-Passing Interface. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series. ISBN 978-0-262-52739-2.

2. Firuziaan, Mohammad; Nommensen, O. (2002) Parallel Processing via MPI & OpenMP, Linux Enterprise, 10/2002

3. A. Eleliemy and F. M. Ciorba, "Hierarchical Dynamic Loop Self-Scheduling on Distributed-Memory Systems Using an MPI+MPI Approach," 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 2019, pp. 689-697, doi: 10.1109/IPDPSW.2019.00117.

4. F. Gomez, G. Indalecio, J. I. Zablah, N. Seoane, A. Garcia and T. F. Pena, "A study of the influence of VM allocation policies on MPI Bcast and MPI Exchange latency in cloud," in IEEE Latin America Transactions, vol. 15, no. 8, pp. 1490-1496, 2017, doi: 10.1109/TLA.2017.7994797.

5. B. Ramesh et al., "Scalable MPI Collectives using SHARP: Large Scale Performance Evaluation on the TACC Frontera System," 2020 Workshop on Exascale MPI (ExaMPI), Atlanta, GA, USA, 2020, pp. 11-20, doi: 10.1109/ExaMPI52011.2020.00007.

6. Y. Xiong, "Some New Approaches to MPI Implementations and a Possible Path to MPI Evolution," 2018 11th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), Beijing, China, 2018, pp. 1-4, doi: 10.1109/CISP-BMEI.2018.8633229.

7. T. Patinyasakdikul, D. Eberius, G. Bosilca and N. Hjelm, "Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs," 2019 IEEE International Conference on Cluster Computing (CLUSTER), Albuquerque, NM, USA, 2019, pp. 1-11, doi: 10.1109/CLUSTER.2019.8891015.

8. P. Vogel, M. A. Rückert, P. M. Jakob and V. C. Behr, "μ MPI—Initial Experiments With an Ultrahigh Resolution MPI," in IEEE Transactions on

Magnetics, vol. 51, no. 2, pp. 1-4, Feb. 2015, Art no. 6502104, doi: 10.1109/TMAG.2014.2329135.

9. Z. Chen, H. Yu, X. Fu and J. Wang, "MPI-SV: A Symbolic Verifier for MPI Programs," 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Seoul, Korea (South), 2020, pp. 93-96.

10. N. Hjelm, H. Pritchard, S. K. Gutiérrez, D. J. Holmes, R. Castain and A. Skjellum, "MPI Sessions: Evaluation of an Implementation in Open MPI," 2019 IEEE International Conference on Cluster Computing (CLUSTER), Albuquerque, NM, USA, 2019, pp. 1-11, doi: 10.1109/CLUSTER.2019.8891002.

Навчальне видання

ЧЕРНИХ Олена Петрівна
ЧЕЛАК Віктор Володимирович

ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ

Навчально-методичний посібник
для студентів комп'ютерних спеціальностей

Роботу до видання рекомендував В.Д. Дмитрієнко

План 2022 р., поз. 76

Підп. до друку 29.01.2021. Формат 60 х 84 1/16. Папір друк. №2.
Riso-друк. Гарнітура Таймс. Ум. друк. арк. 5,0. Наклад. 100 прим.
Зам №_____. Ціна договірна.

_

Видавничий центр НТУ „ХПІ".
Свідоцтво про державну реєстрацію ДК № 116 від 10.07.2004 р.
61002, Харків, вул. Фрунзе, 21

Друкарня НТУ "ХПІ". 61002, Харків, вул. Фрунзе, 21