Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики і обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«ПАРАЛЕЛЬНІ ТА РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ»

Практична робота №7

## Тема:

## «Розробка паралельного алгоритму множення матриць з використанням МРІ-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності»

**Виконанав:**

Студент гр. ІТ-83

Заярнюк М.О

**Перевірила:**

Дифучина О. Ю.

Київ – 2021

**Завдання до комп’ютерного практикуму 7 «Розробка паралельного алгоритму множення матриць з використанням МРІ-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності»**

1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. **40 балів.**
3. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми.Порівняйте ефективність алгоритму при використанні методів обміну повідомленнями «один-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох».**60 балів.**
4. Теоретичні відомості

**MPI** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *Message Passing Interface*, **Інтерфейс передачі повідомлень**) — це специфікація, що була розроблена в 1993–1994 роках групою MPI Forum,і забезпечує реалізацію моделі [обміну повідомленнями між процесами](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D1%96%D0%BD_%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D0%BE%D0%BC%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F%D0%BC%D0%B8_%D0%BC%D1%96%D0%B6_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%B0%D0%BC%D0%B8). Остання версія цієї специфікації: MPI-2. У моделі програмування MPI програма породжує кілька процесів, що взаємодіють між собою за допомогою виклику підпрограм прийому й передачі повідомлень.

Зазвичай, при ініціалізації MPI-програми створюється фіксований набір процесів, причому (що, утім, необов'язково) кожний з них виконується на своєму процесорі. У цих процесах можуть виконуватися різні програми, тому MPI-модель іноді називають [MIMD](https://uk.wikipedia.org/wiki/MIMD)-моделлю (Multiple Instruction, Multiple Data), на відміну від [SIMD](https://uk.wikipedia.org/wiki/SIMD) моделі, де на кожному процесорі виконуються тільки однакові задачі. MPI підтримує двохточкові й глобальні, синхронні й асинхронні, блокуючі й неблокуючі типи комунікацій. Спеціальний механізм — комунікатор — ховає від програміста внутрішні комунікаційні структури. Структура комунікацій може змінюватися протягом часу життя процесу, але кількість задач має залишатися постійною (MPI-2 підтримує динамічну зміну кількості задач).

Специфікація MPI забезпечує переносимість програм на рівні вихідних кодів. Підтримується робота на гетерогенних кластерах і [симетричних мультипроцесорних системах](https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%BD%D0%B0_%D0%BC%D1%83%D0%BB%D1%8C%D1%82%D0%B8%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80%D0%BD%D1%96%D1%81%D1%82%D1%8C). Не підтримується запуск процесів під час виконання MPI-програми. У специфікації відсутні опис паралельного [вводу-виводу](https://uk.wikipedia.org/wiki/%D0%9A%D0%B0%D0%BD%D0%B0%D0%BB_%D0%B2%D0%B2%D0%BE%D0%B4%D1%83-%D0%B2%D0%B8%D0%B2%D0%BE%D0%B4%D1%83) й [налагодження програм](https://uk.wikipedia.org/wiki/%D0%9D%D0%B0%D0%BB%D0%B0%D0%B3%D0%BE%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC) — ці можливості можуть бути включені до складу конкретної реалізації MPI у вигляді додаткових пакетів чи утиліт. Сумісність різних реалізацій не гарантується.

Важливою властивістю паралельної програми є [детермінізм](https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D1%82%D0%B5%D1%80%D0%BC%D1%96%D0%BD%D1%96%D0%B7%D0%BC) — програма має завжди давати однаковий результат для однакових наборів вхідних даних. Модель передачі повідомлень загалом такої властивості не має, оскільки не визначено порядок одержання повідомлень від двох процесів третім. Якщо ж один процес послідовно надсилає кілька повідомлень іншому процесу, MPI гарантує, що одержувач отримає їх саме в тому порядку, у якому вони були надіслані. Відповідальність за забезпечення детермінованого виконання програми покладається на програміста.

Основні особливості та відмінності від комунікацій типу "точка-точка ":

* на прийом та/або передачу працюють одночасно ВСІ завдання-абоненти вказаного комунікатора;
* колективна функція виконує одночасно прийом, і передачу; вона має велику кількість параметрів, частина яких необхідна прийому, а частина передачі; у різних завданнях та чи інша частина ігнорується; як правило, значення ВСІХ параметрів (за винятком адрес буферів) повинні бути ідентичними у всіх завданнях;
* MPI призначає ідентифікатор повідомлень автоматично;

крім того, повідомлення передаються не за вказаним комунікатором, а по тимчасовому комунікатору-дуплікату; цим потоки даних колективних функцій надійно ізолюються друг від друга і від потоків, створених функціями "точка-точка".

MPI\_Bcast рассылает содержимое буфера из задачи, имеющей в указанной области связи номер root, во все остальные:

MPI\_Bcast( buf, count, dataType, rootRank, communicator );

Она эквивалентна по результату (но не по внутреннему устройству) следующему фрагменту:

MPI\_Comm\_size( communicator, &commSize );

MPI\_Comm\_rank( communicator, &myRank );

if( myRank == rootRank )

for( i=0; i<commSize; i++ )

MPI\_Send( buf, count, dataType, i,

tempMsgTag, communicator );

MPI\_Recv( buf, count, dataType, rootRank, tempMsgTag,

communicator, &status );

MPI\_Gather ("совок") собирает в приемный буфер задачи root передающие буфера остальных задач. Ее аналог:

MPI\_Send( sendBuf, sendCount, sendType, rootRank, ... );

if( myRank == rootRank ) {

MPI\_Type\_extent( recvType, &elemSize );

for( i=0; i<commSize; i++ )

MPI\_Recv( ((char\*))recvBuf) + (i \* recvCount \* elemSize),

recvCount, recvType, i, ... );

}

Заметьте, что а) recvType и sendType могут быть разные и, таким образом, будут задавать разную интерпретацию данных на приемной и передающей стороне; б) задача-приемник также отправляет данные в свой приемный буфер.

Векторный вариант "совка" - MPI\_Gatherv - позволяет задавать РАЗНОЕ количество отправляемых данных в разных задачах-отправителях. Соответственно, на приемной стороне задается массив позиций в приемном буфере, по которым следует размещать поступающие данные, и максимальные длины порций данных от всех задач. Оба массива содержат позиции/длины НЕ в байтах, а в количестве ячеек типа recvCount. Ее аналог:

MPI\_Send( sendBuf, sendCount, sendType, rootRank, ... );

if( myRank == rootRank ) {

MPI\_Type\_extent( recvType, &elemSize );

for( i=0; i<commSize; i++ )

MPI\_Recv( ((char\*))recvBuf) + displs[i] \* recvCounts[i]

\* elemSize, recvCounts[i], recvType, i, ... );

}

MPI\_Scatter ("разбрызгиватель") : выполняет обратную "совку" операцию - части передающего буфера из задачи root распределяются по приемным буферам всех задач. Ее аналог:

if( myRank == rootRank ) {

MPI\_Type\_extent( recvType, &elemSize );

for( i=0; i<commSize; i++ )

MPI\_Send( ((char\*)sendBuf) + i\*sendCount\*elemSize,

sendCount, sendType, i, ... );

}

MPI\_Recv( recvBuf, recvCount, recvType, rootRank, ... );

И ее векторный вариант - MPI\_Scatterv, рассылающая части неодинаковой длины в приемные буфера неодинаковой длины.

MPI\_Allgather аналогична MPI\_Gather, но прием осуществляется не в одной задаче, а во ВСЕХ: каждая имеет специфическое содержимое в передающем буфере, и все получают одинаковое содержимое в буфере приемном. Как и в MPI\_Gather, приемный буфер последовательно заполняется данными изо всех передающих. Вариант с неодинаковым количеством данных называется MPI\_Allgatherv.

MPI\_Alltoall : каждый процесс нарезает передающий буфер на куски и рассылает куски остальным процессам; каждый процесс получает куски от всех остальных и поочередно размещает их приемном буфере. Это "совок" и "разбрызгиватель" в одном флаконе. Векторный вариант называется MPI\_Alltoallv.

Пример использования коллективных функций передачи данных [здесь](https://www.opennet.ru/docs/RUS/MPI_intro/mpi_ex4.c.txr).

В учебнике, изданном MIT Press, есть хорошая [СХЕМА](https://www.opennet.ru/docs/RUS/MPI_intro/collmove.gif) для всех перечисленных в этом разделе функций. Понять, что как работает, по ней нелегко, зато вспоминать, если однажды уже разобрался, удобно.

Помните, что коллективные функции несовместимы с "точка-точка": недопустимым, например, является вызов в одной из принимающих широковещательное сообщение задач MPI\_Recv вместо MPI\_Bcast.

1. Лістнинг коду

ManyToMany.py

import sys  
from mpi4py import MPI  
from numpy import array, zeros\_like, arange  
from numpy import empty  
  
from MatrixGenerator import get\_matrix  
from MatrixMultiplication import matrix\_multiplication  
from SplitMatrix import split\_matrix  
  
comm = MPI.COMM\_WORLD  
my\_rank = comm.Get\_rank()  
num\_of\_processes = comm.Get\_size()  
MatrixDimension = int(sys.argv[1])  
time\_start = 0  
second\_matrix = second\_matrix = get\_matrix(MatrixDimension)  
  
if my\_rank == 0:  
 first\_matrix = second\_matrix = get\_matrix(MatrixDimension)  
 first\_matrix\_row = split\_matrix(first\_matrix, num\_of\_processes)  
  
else:  
 first\_matrix\_row = None  
  
if my\_rank == 0:  
 result\_matrix = [[[] for j in range(0, num\_of\_processes)] for i in range(0, num\_of\_processes)]  
 time\_start = MPI.Wtime()  
  
first\_matrix\_row = comm.scatter(first\_matrix\_row, root=0)  
  
\_sendData = array(matrix\_multiplication(first\_matrix\_row, second\_matrix))  
\_recvData = empty(MatrixDimension\*500, dtype=float)  
comm.Alltoall(\_sendData, \_recvData)  
  
data = comm.gather(\_recvData, root=0)  
  
# if my\_rank == 0:  
# for i, row in enumerate(data):  
# for j in range(len(row)):  
# result\_matrix[j][i] = row[j]  
# #print(result\_matrix)  
  
if my\_rank == 0:  
 spent\_time = MPI.Wtime() - time\_start  
 print("Finished in: ", spent\_time)  
  
MPI.Finalize()

OneToOne.py

import sys  
from mpi4py import MPI  
from numpy import array, zeros\_like, arange  
from numpy import empty  
  
from MatrixGenerator import get\_matrix  
from MatrixMultiplication import matrix\_multiplication  
from SplitMatrix import split\_matrix  
  
comm = MPI.COMM\_WORLD  
my\_rank = comm.Get\_rank()  
num\_of\_processes = comm.Get\_size()  
MatrixDimension = int(sys.argv[1])  
time\_start = 0  
second\_matrix = second\_matrix = get\_matrix(MatrixDimension)  
  
if my\_rank == 0:  
 first\_matrix = second\_matrix = get\_matrix(MatrixDimension)  
 first\_matrix\_row = split\_matrix(first\_matrix, num\_of\_processes)  
  
else:  
 first\_matrix\_row = None  
  
if my\_rank == 0:  
 result\_matrix = [[[] for j in range(0, num\_of\_processes)] for i in range(0, num\_of\_processes)]  
 time\_start = MPI.Wtime()  
  
first\_matrix\_row = comm.scatter(first\_matrix\_row, root=0)  
  
\_sendData = array(matrix\_multiplication(first\_matrix\_row, second\_matrix))  
\_recvData = empty(MatrixDimension\*500, dtype=float)  
comm.Alltoall(\_sendData, \_recvData)  
  
data = comm.gather(\_recvData, root=0)  
  
# if my\_rank == 0:  
# for i, row in enumerate(data):  
# for j in range(len(row)):  
# result\_matrix[j][i] = row[j]  
# #print(result\_matrix)  
  
if my\_rank == 0:  
 spent\_time = MPI.Wtime() - time\_start  
 print("Finished in: ", spent\_time)  
  
MPI.Finalize()

OneToMany.py

import sys  
from MatrixMultiplication import matrix\_multiplication  
from mpi4py import MPI  
from MatrixGenerator import get\_matrix  
from SplitMatrix import split\_matrix  
  
comm = MPI.COMM\_WORLD  
my\_rank = comm.Get\_rank()  
num\_of\_processes = comm.Get\_size()  
MatrixDimension = int(sys.argv[1])  
  
time\_start = 0  
second\_matrix = get\_matrix(MatrixDimension)  
  
if my\_rank == 0:  
 first\_matrix = get\_matrix(MatrixDimension)  
 first\_matrix\_row = split\_matrix(first\_matrix, num\_of\_processes)  
 time\_start = MPI.Wtime()  
else:  
 first\_matrix\_row = None  
  
first\_matrix\_row = comm.scatter(first\_matrix\_row, root=0)  
  
line = matrix\_multiplication(first\_matrix\_row, second\_matrix)  
data = comm.gather(line, root=0)  
  
if my\_rank == 0:  
 spent\_time = MPI.Wtime() - time\_start  
 print("Finished in: ", spent\_time)  
  
MPI.Finalize()

MatrixGenerator.py

from random import randint  
  
  
def get\_matrix(size):  
 return [[randint(0, 9) for i in range(size)] for j in range(size)]

MatrixMultiplication.py

def matrix\_multiplication(x, y):  
 return [[sum(a \* b for a, b in zip(X\_row, Y\_col)) for Y\_col in zip(\*y)] for X\_row in x]

Multiply.py

from Task1.MatrixMultiplication import matrix\_multiplication  
  
  
def line\_and\_second\_matrix\_multiplication(line, second\_matrix):  
 matrix\_size = len(line)  
 result\_line = []  
 for element in line:  
 line\_sum = matrix\_multiplication(element, second\_matrix)  
 result\_line.append(line\_sum)  
 return result\_line

SplitMatrix.py

def split\_matrix(input\_matrix, amount\_of\_workers):  
 rows\_split = []  
 division\_value = len(input\_matrix) // amount\_of\_workers  
 leftover = len(input\_matrix) % amount\_of\_workers  
 matrix\_division\_start = 0  
 matrix\_division\_end = division\_value + min(1, leftover)  
 for i in range(amount\_of\_workers):  
 rows\_split.append(input\_matrix[matrix\_division\_start:matrix\_division\_end])  
 leftover = max(0, leftover - 1)  
 matrix\_division\_start = matrix\_division\_end  
 matrix\_division\_end += division\_value + min(1, leftover)  
 return rows\_split

1. Дослідження ефективності

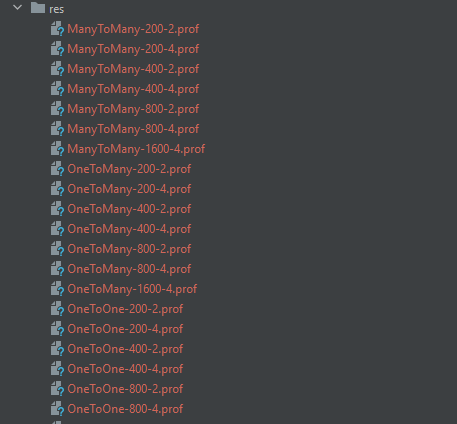
Було проведено 48 тестів, для кожного з типу передачі повідомлень було обрано наступні вхідні параметри:

2,4,8,10 – вузлів

200,400,800,1600 – розмірність матриці.

Кожен з типів передачі повідомлень було запущено з використанням цих даних, від чого і вийшло 48 тестів.

Результати були записані у папку «res», що знаходиться в проекті

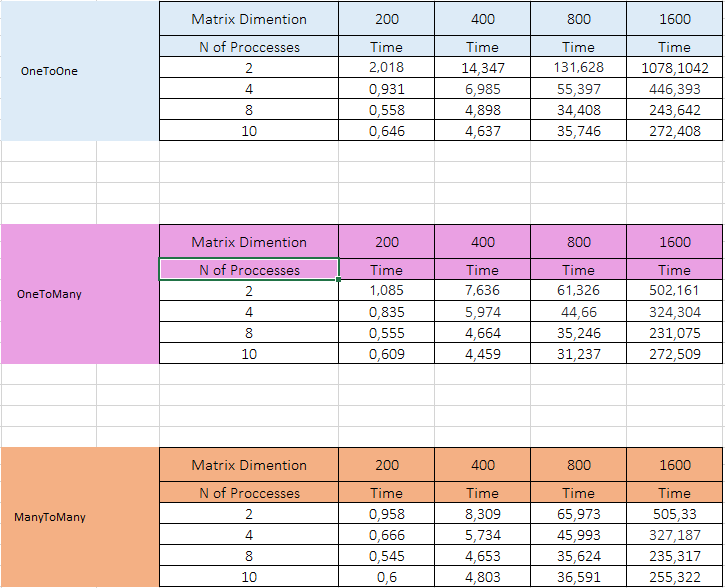


Неймінг файлів складається з:  
*НазваТипуПовідомлень-РозмірністьМатриці-КількістьВузлів.prof*

Для того, щоб відкрити дані файли треба виконати команду:

tuna res/ManyToMany-$MATRIX\_DIMENSION-$COUNT\_PROCESSES.prof

В ході проведення експерименту були отримані такі значення:



Та побудовані наступні графіки:

Для розмірності 200:

Для розмірності 400:

Для розмірності 800:

Для розмірності 1600:

Висновок:

В ході даної лабораторної роботи було розроблено та досліджено алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями.

Робота була виконана мовою програмування Python та бібліотекою “mpi4py” та іншими.

Усі компоненти та модулі програми були розбиті на різні файли та методи, для повторного використання.

Також, було проведено 48 тестів для дослідження ефективності розроблених алгоритмів, вони відрізнялися вхідними параметрами:

2,4,8,10 – вузлів

200,400,800,1600 – розмірність матриці.

Результати були записані у папку «res», що знаходиться в проекті.

А також, були побудовані графіки, на основі цих результатів.

*Після дослідження можна зробити наступний висновок:*

***Розроблені алгоритми втрачають різницю в ефективності при значенні в 8 вузлів та розмірності 800. В даній ситуації це ліміт їх продуктивності.***

***В при меншій кількості вузлів та розмірності – найменш ефективним є алгоритм «один-до-одного», а найбільш ефективним є алгоритм «багато-до-багатьох», хоч і різниця з «один-до-багатьох» є не такою великою, навіть в рамках погрішності.***

Дана лабораторна викладена за посиланням у GitHub:

https://github.com/mynameischeezee/MPI\_collective\_broadcast