Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

з лабораторної роботи № 7 з дисципліни
«Технології паралельних обчислень»

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

Виконав(ла) *ІП-14 Сергієнко Ю. В.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив *Дифучина О. Ю.* _____
(прізвище, ім'я, по батькові)

Київ 2024

Комп’ютерний практикум 7

Тема: Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багатодо-багатьох») та дослідження його ефективності.

Виконання:

1. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. **40 балів**.

Для реалізації паралельного алгоритму множення матриць на основі методу колективного обміну повідомлень необхідно використовувати наступні методи MPI: *MPI_Scatter()*, *MPI_Bcast()* та *MPI_Gather()*.

Спочатку будемо ініціалізувати матриці для обробки у процесі Мастера. Далі, використовуючи *MPI_Scatter()*, будемо передавати частини для обробки: у матриці A виберемо N * N / numtasks елементів на передачу (усі елементи у N / numtasks рядках), а для отримання будемо записувати стільки ж елементів у A_P матрицю. Вказуючи root = 0, ми зазначаємо, що лише Мастер буде надсилати дані, коли як отримувати їх будуть усі процеси.

```
MPI_Scatter(
    a[0], sendSize, MPI_DOUBLE,
    a_p[0], getSize, MPI_DOUBLE,
    0, MPI_COMM_WORLD
);
```

Для передачі матриці B будемо використовувати метод *MPI_Bcast()*, передаючи матрицю та весь її розмір. Таким чином ми будемо дублювати матрицю B для кожного процеса.

```
MPI_Bcast(
    b[0], N * N, MPI_DOUBLE,
    0, MPI_COMM_WORLD
);
```

Після успішного множення частини матриці A на матрицю B, будемо отримувати результат у Мастері за допомогою *MPI_Gather()*. Вкажемо матрицю, в яку ми записували результат множення, її розмір (к-сть усіх її

елементів) та матрицю на отримування результату. Також вкажемо root = 0, що означатиме можливість прийняття даних лише Мастером.

```
MPI_Gather(
    temp[0], sendSize, MPI_DOUBLE,
    c[0], getSize, MPI_DOUBLE,
    0, MPI_COMM_WORLD
);
```

У даній реалізації було застосовано основні методи «один-до-багатьох» та «багато-до-одного» типів. Результат роботи алгоритму відображено на рисунку 1.

```
PS C:\Users\Yurii\Documents\jaba\kpi6\tpo\lab7> mpiexec -n 5 ./cmake-build-debug/lab7
Main has started with 5 tasks.
Initial Matrix size: 100x100
Time Took: 2 ms

****
First 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
Last 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
********
Done.
PS C:\Users\Yurii\Documents\jaba\kpi6\tpo\lab7>
```

Рисунок 1 – Результат роботи першого алгоритму

Алгоритм виявився робочим, але не універсальним. Якщо задати кількість процесів, неподільних розміру матриці, то ми втратимо частину елементів. Щоб цього уникнути, необхідно використовувати методи *MPI_Scatterv()* та *MPI_Gatherv()*, вказуючи масив розмірів та масив offsets вхідної та вихідної матриці відповідно до процесу. Тобто, перед початком розподілу даних, необхідно обрахувати яку к-сть даних буде опрацьовувати процес. Таким чином можливо досягнути незалежності алгоритму від к-сті процесів (рисунок 2).

```
PS C:\Users\Yurii\Documents\jaba\kpi6\tpo\lab7> mpiexec -n 9 ./cmake-build-debug/lab7
Main has started with 9 tasks.
Initial Matrix size: 100x100
Time Took: 2 ms

****
First 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
Last 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
********
Done.
PS C:\Users\Yurii\Documents\jaba\kpi6\tpo\lab7>
```

Рисунок 2 – Результат роботи другого алгоритму

Для реалізації алгортиму способом «багато-до-багатьох» необхідно при зборі даних використати метод *MPI_Allgatherv()*, що в свою чергу розподіляє результат обчислень по всіх процесах. Головною відмінністю від звичайного MPI_Gather() методу є відсутність параметру root, адже результат буде записаний усім воркерам та Мастеру.

```
MPI_Allgatherv(
    temp[0], sizes[taskid], MPI_DOUBLE,
    c[0], sizes, offsets, MPI_DOUBLE,
    MPI_COMM_WORLD
);
```

Оскільки розрахунки кожної частини матриці можуть зайняти різні проміжки часу, необхідно дочекатись виконанння їх усіх. Для цього використаємо метод *MPI_Barrier()*, який дочікується виконання усіх процесів комунікатора.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Результат роботи алгортиму зображений на рисунку 3.

```
PS C:\Users\Yurii\Documents\jaba\kpi6\tpo\lab7> mpiexec -n 3 ./cmake-build-debug/lab7
Main has started with 3 tasks.
Initial Matrix size: 100x100
Time Took: 2 ms

****
First 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
Last 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
********
Done.

****
First 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
Last 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
********
Done.

****
First 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
Last 10:
10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00 10000.00
********
Done.
PS C:\Users\Yurii\Documents\jaba\kpi6\tpo\lab7>
```

Рисунок 3 – Результат роботи третього алгоритму

2. 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**.

Порівняємо три реалізації: метод «один-до-одного» (блокуючого обміну) із КП 6, «один-до-багатьох» та «багато-до-одного» з визначенням offsets та «багато-до-багатьох». Результати досліджень відображені у таблиці 1.

Таблиця 1 – Час роботи алгоритмів

Size	Threads	1-1, ms	1-M – M-1, ms	M-M, ms
500	2	496	266	284
	6	146	116	135
	12	88	94	97
1000	2	3992	2125	2163
	6	947	890	918
	12	703	701	722
1500	2	14218	7830	7958
	6	3497	3228	3325
	12	2677	2935	2988
3000	12	36565	37694	37993

Після спостережень можна вивести наступну статистику: при двох потоках краще спрацьовують методи 1-M та M-M. Це можна легко пояснити: для методу блокуючого обміну була використана реалізація, де Мастер не приймає участі в обрахунках, тоді як у методах колективного обміну це неможливо. На середній к-сті процесів (6 у даному випадку) 1-M та M-M є швидшими у 1.05 – 1.25 рази. При великій к-сті процесів в середньому виграє 1-1 метод. Залежності від розмірів як такої немає.

Порівнюючи 1-M та M-M методи можемо зазначити, що M-M стабільно відпрацьовує трохи довше і це правильно, адже ми дочікуємось закінчення усіх процесів у *MPI_Allgather()* методі.

Відобразимо результати обчислень на рисунках 4-6.

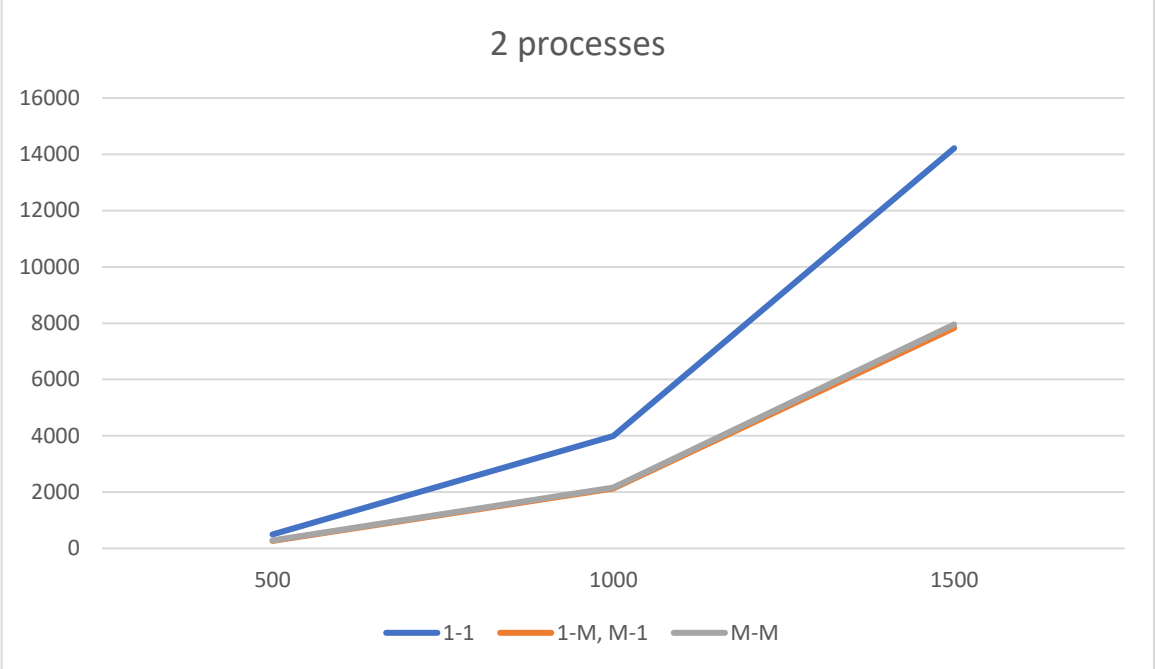

Рисунок 4 – Час роботи алгоритмів при 2 процесах

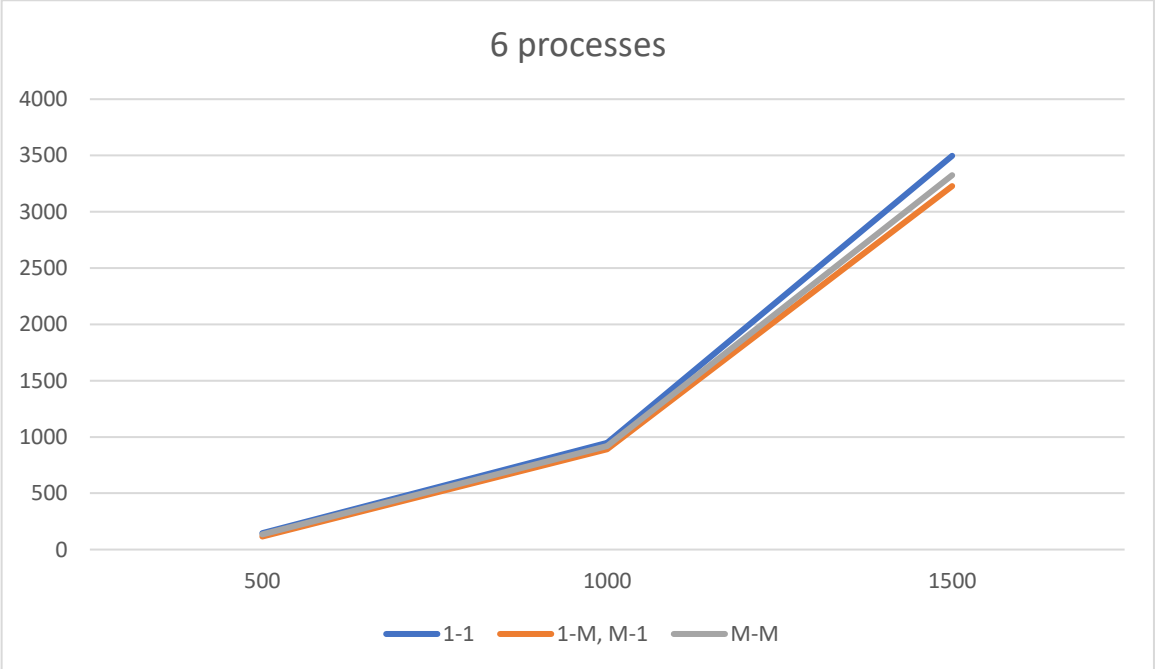

Рисунок 5 – Час роботи алгоритмів при 6 процесах

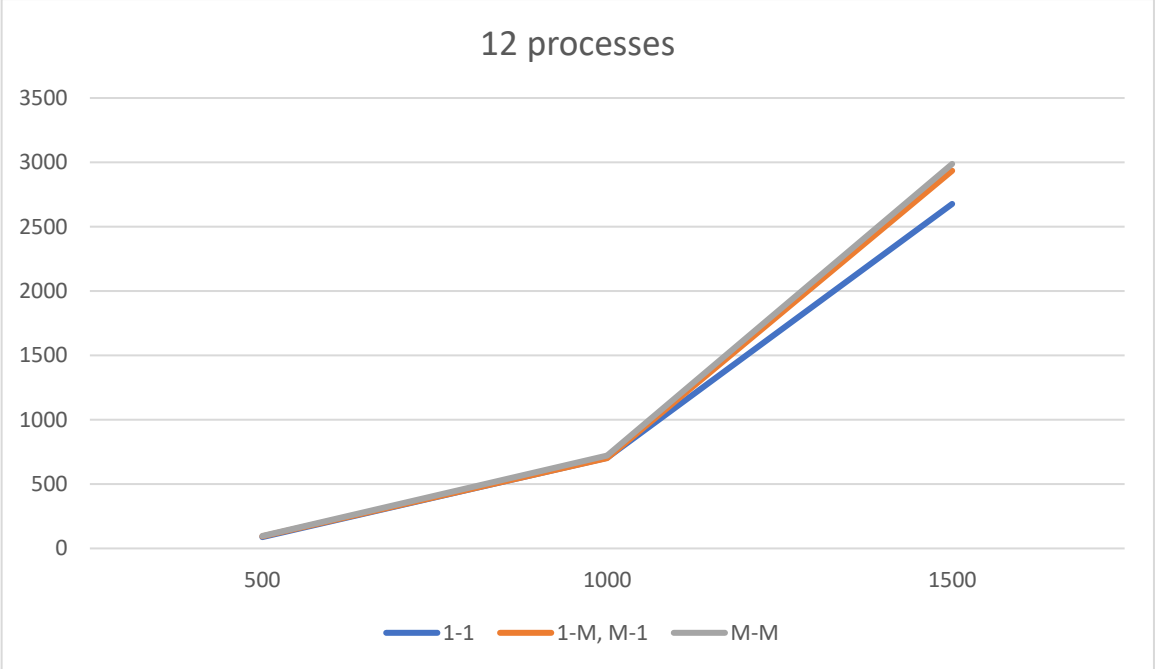

Рисунок 6 – Час роботи алгоритмів при 12 процесах

Висновок

Під час виконання даного комп’ютерного практикуму я здобув здання паралельного управління процесами комп’ютера за допомогою використання стандарту Message Passing Interface (MPI) мовою C++ (OpenMPI). Було паралельно помножено матриці методами один-до-багатьох, багато-до-одного та багато-до-багатьох. Після дослідження ефективності алгоритмів можна сказати, що усі методи мають право на існування. Було визначено, що при середній кількості процесів найшвидшим є метод один-до-багатьох.

Код програми доступний на Github.