Міністерство освіти і науки України
Харківський національний університет імені В.Н. Каразіна

Факультет комп’ютерних наук
Кафедра теоретичної та прикладної системотехніки

лабораторну роботу № 1
«Фільтрування списку співробітників»

Виконав студент групи КУ-51
Шевченко Андрій

Перевірила викладач та лектор
Толстолузька О. ?.

2023 р.

Є файл, що містить записи кожного працівника. Кожен запис включає прізвище, ім’я, рік народження та рік прийому на роботу. Розробте алгоритм і напишіть MPI- програму, в якій один із процесів розподіляє всім іншим процесам приблизно однакові порції; інформації, а ці процеси формують список співробітників, стаж яких становить понад 5 років. Результати пересилаються головному процесу, що їх виводить на файл. До складу звіту мають бути включені:

а) Титульний лист.

б) Роздруківки відкомпільованих текстів MPI-програм. ((???))

в) Скріншоти результатів виконання програм.

г) Висновки щодо роботи.

Хід роботи

Вступ

В рамках цієї лабораторної роботи розглядається задача розподіленого обчислення за допомогою технології MPI (Message Passing Interface), яка є стандартом у сфері паралельних обчислень.

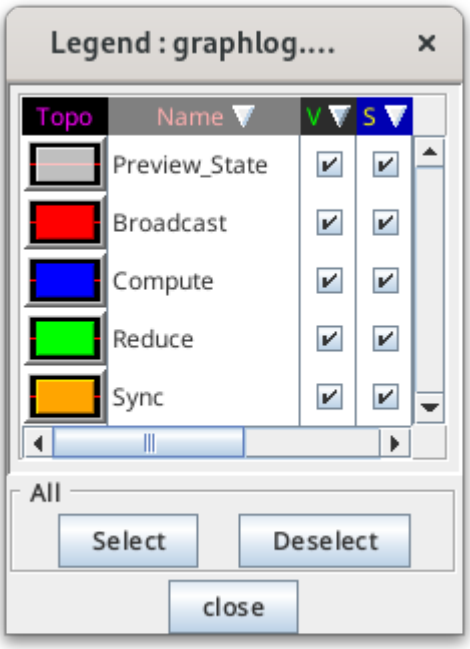
Центральною частиною дослідження є програма, написана на мові програмування `C` і скомпільована за допомогою компілятора mpicc, що базується на стандарті `OpenMPI`. Особливість даної реалізації полягає у використанні сокетів (`FI_PROVIDER` = `sockets`) для передачі даних між процесами, що надає додаткову гнучкість у конфігурації комунікацій.

Розроблений код організований у декілька файлів, що сприяє кращій структуруванню та зрозумілості проекту. Для кожної процедури було створено просту, але змістовну документацію, що полегшує орієнтацію в коді та його використання.

Для профілювання виконання програми використовувались інструменти MPE (MPI Parallel Environment) та mpiP. MPE надає інструментарій для детального аналізу виконання програми, дозволяючи візуалізувати різні фази обчислень. mpiP, в свою чергу, є легким врапером для MPI, що дозволяє здійснювати більш загальне профілювання із залученням функції `MPI_Wtime`().

Для дебагінгу програми використовувалася утиліта gdb, яка є могутнім інструментом для виявлення та усунення помилок в коді.

Параметри профілювання mpiP були налаштовані як `MPIP = -c -p -y -t 1.0 -k 2 -f ./logs`, що детально описано в документації `mpiP`. За результатами профілювання MPE генерує бінарний файл `clog2`, який може бути перетворений в `slog2` формат. Даний формат підтримується утилітою `Jumpshot`, що дозволяє графічно відобразити залежності між різними частинами програми.



ПЗ Jumpshot; Легенда графіків

mpiP, у свою чергу, надає текстовий лог файл, для аналізу якого було розроблено спеціальний парсер на мові `Ruby`. Цей парсер дозволяє збирати необхідні дані, переносити їх у електронні таблиці та створювати інформативні графіки залежностей.

Для генерації вхідних даних було використано скрипт на `Ruby` з використанням бібліотеки `Faker`, що забезпечує реалістичність тестових даних.

Весь код проекту розміщений у відповідному репозиторії. Більше технічної інформації та пояснень можна знайти у файлі `readme.md`, що супроводжує проект.

```
~/Documents/univer/mpi/lr1 master ?3
❯ make run
mpirun -np 3 --host fedora:3 ./employee.o
mpiP: Found MPIP environment variable [-c -p -y -t 1.0 -k 2 -f ./logs]
mpiP: Set the report print threshold to [1.00%].
mpiP: Set the callsite stack traceback depth to [2].
mpiP: Set the output directory to [./logs].
mpiP:
mpiP: mpiP V3.5.0 (Build Nov 13 2023/22:38:24)
mpiP:
Enabling the Default clock synchronization...
Input:  1000 records
Output: 374 records
mpiP:
mpiP: Storing mpiP output in [./logs/employee.o.3.424426.1.mpiP].
mpiP:
~/Documents/univer/mpi/lr1 master ?4
❯
```

Приклад роботи програми

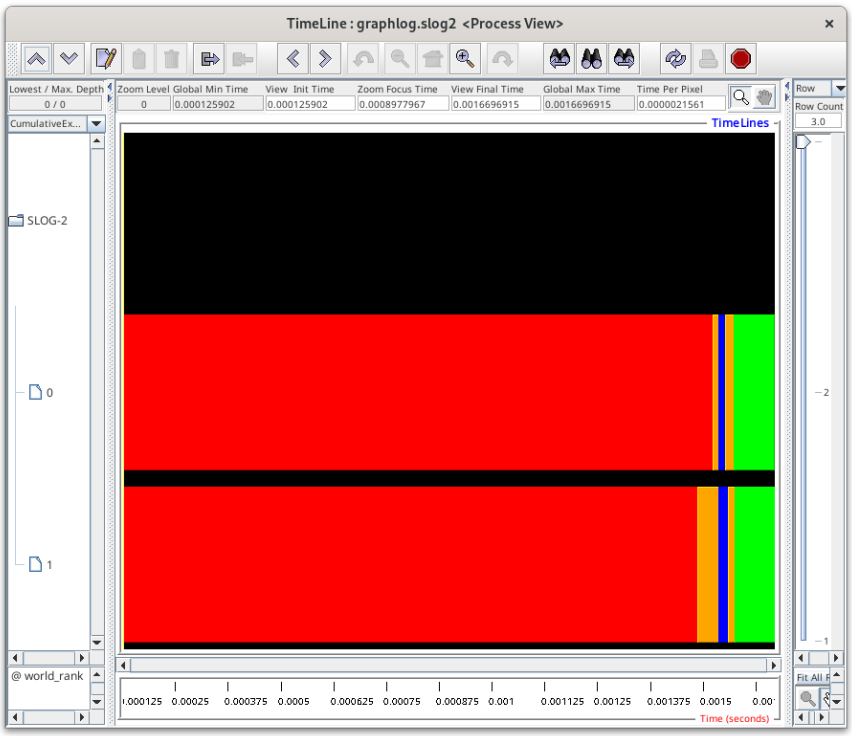
```
❯ tree .                                                                    [2/1884]
.
├── employee_filtering.c
├── employee_filtering.h
├── employee.o
├── employees.csv
├── generator.rb
├── localmpi.c
├── localmpi.h
├── logs
│   ├── 1k-records
│   │   ├── n2
│   │   │   ├── employee.o.2.412890.1.mpiP
│   │   │   ├── Screenshot from 2023-11-20 15-01-19.png
│   │   │   └── Screenshot from 2023-11-22 10-17-47.png
│   │   ├── n3
│   │   │   ├── employee.o.3.413093.1.mpiP
│   │   │   ├── Screenshot from 2023-11-20 15-02-11.png
│   │   │   └── Screenshot from 2023-11-22 10-18-56.png
│   │   ├── n4
│   │   │   ├── employee.o.4.413254.1.mpiP
│   │   │   ├── Screenshot from 2023-11-20 15-02-49.png
│   │   │   └── Screenshot from 2023-11-22 10-15-46.png
│   │   └── n6
│   │       ├── employee.o.6.413437.1.mpiP
│   │       ├── Screenshot from 2023-11-20 15-03-14.png
│   │       └── Screenshot from 2023-11-22 10-19-45.png
│   ├── output1.csv
│   ├── output1.ods
│   ├── Screenshot from 2023-11-20 15-04-22.png
│   └── Screenshot from 2023-11-20 15-05-34.png
├── main.c
├── Makefile
└── parser.rb

7 directories, 26 files
```

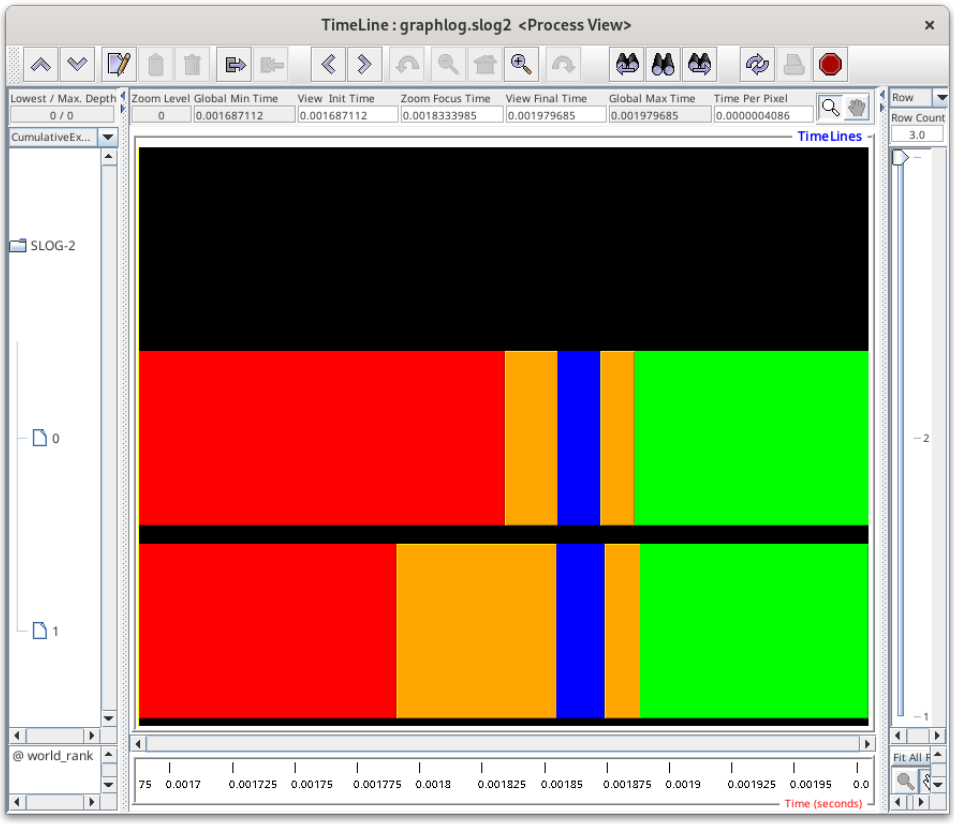
Структура лабораторної роботи

Виконання програми на різних наборах даних

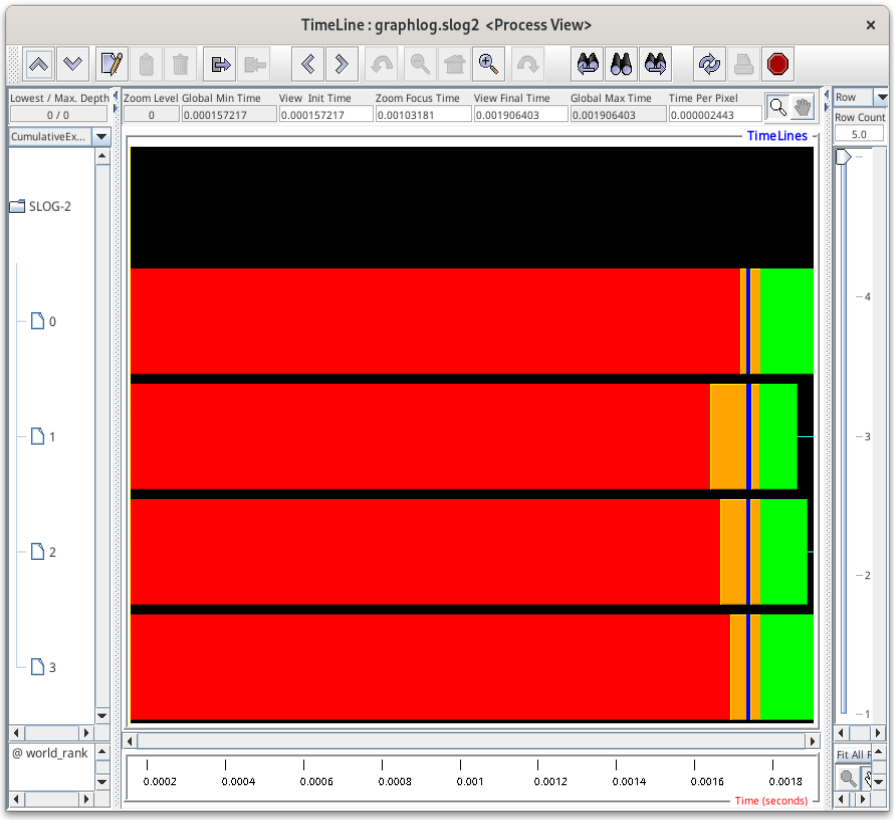
Об’єм даних 1000 записів. Перший графік — профілювання з урахунком читання файла головним процессором. Другий графік — без урахування часу на зчитування.

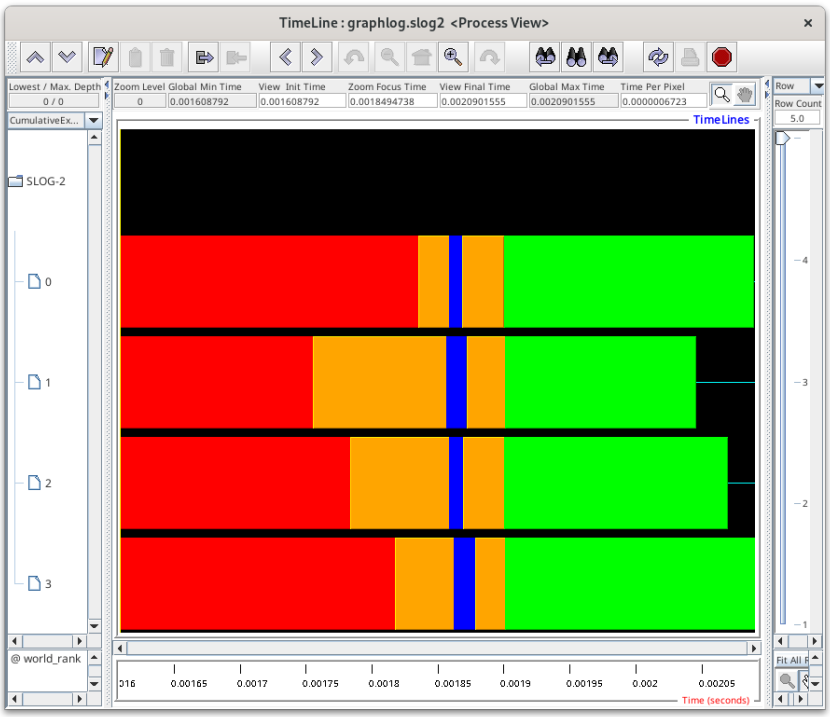
Розподіл між 2 процессами



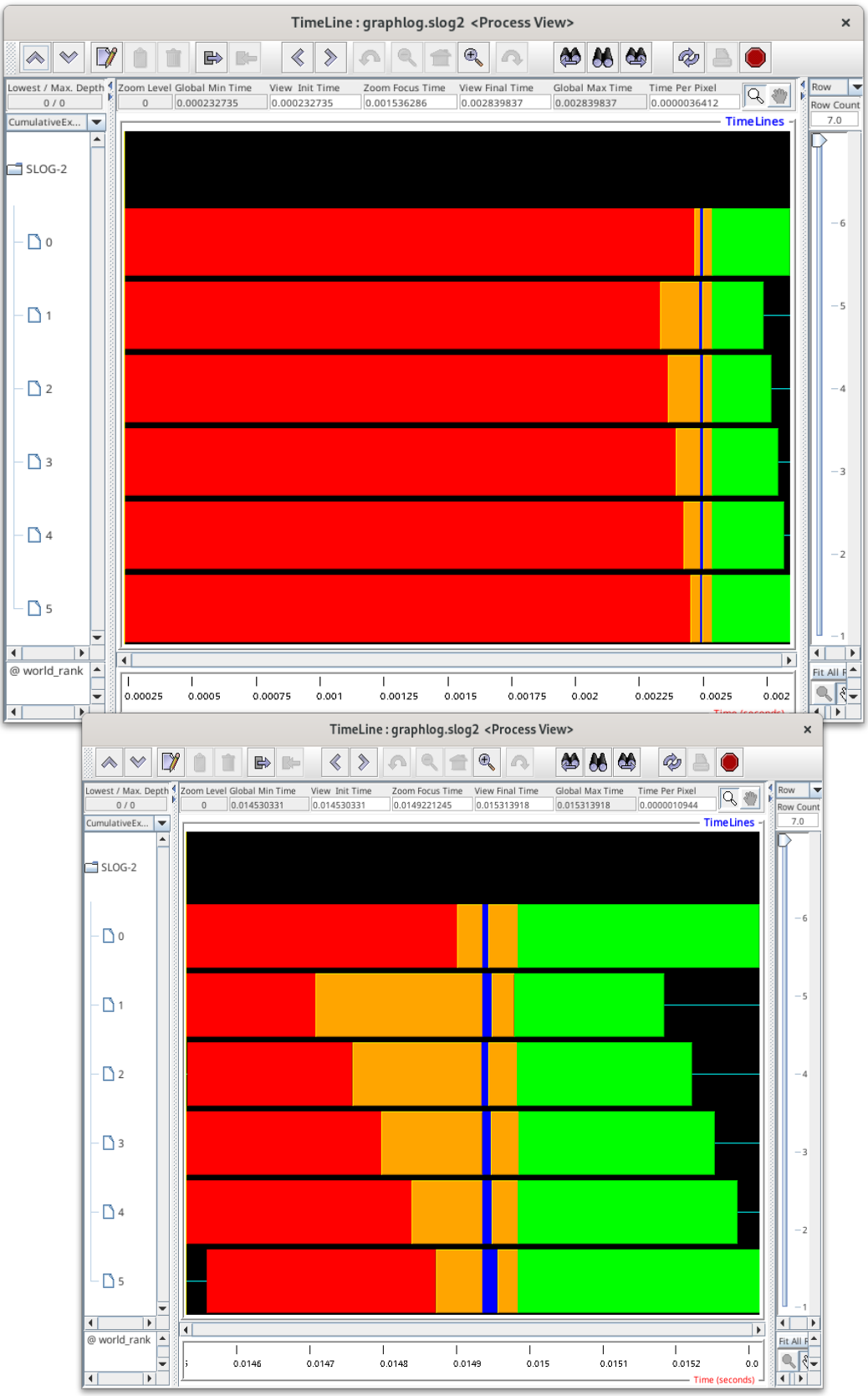


Розподіл між 4 процессами

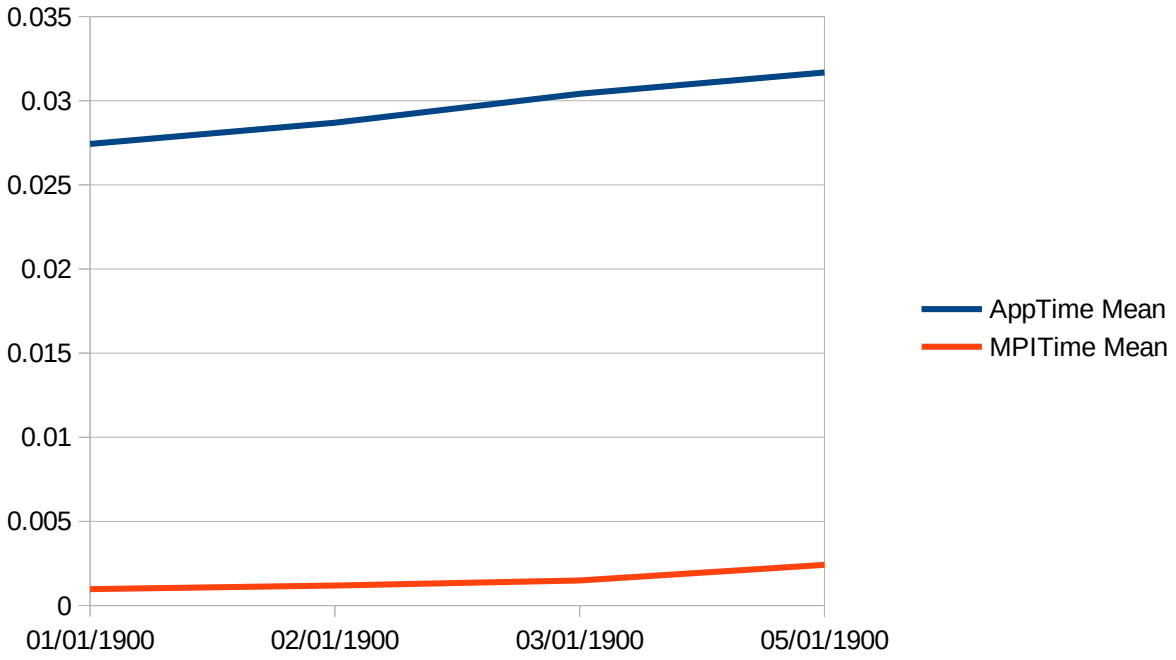




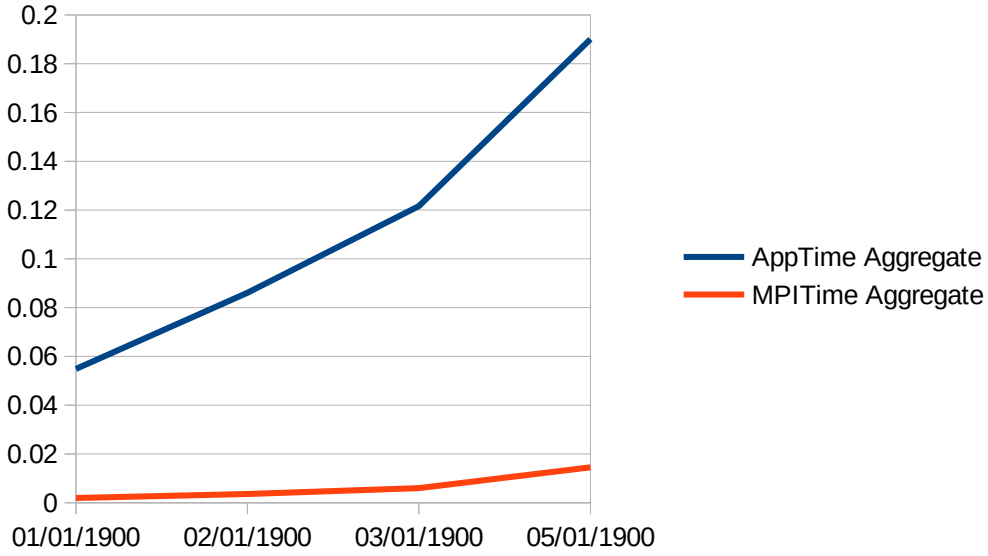
Розподіл між 6 процессами



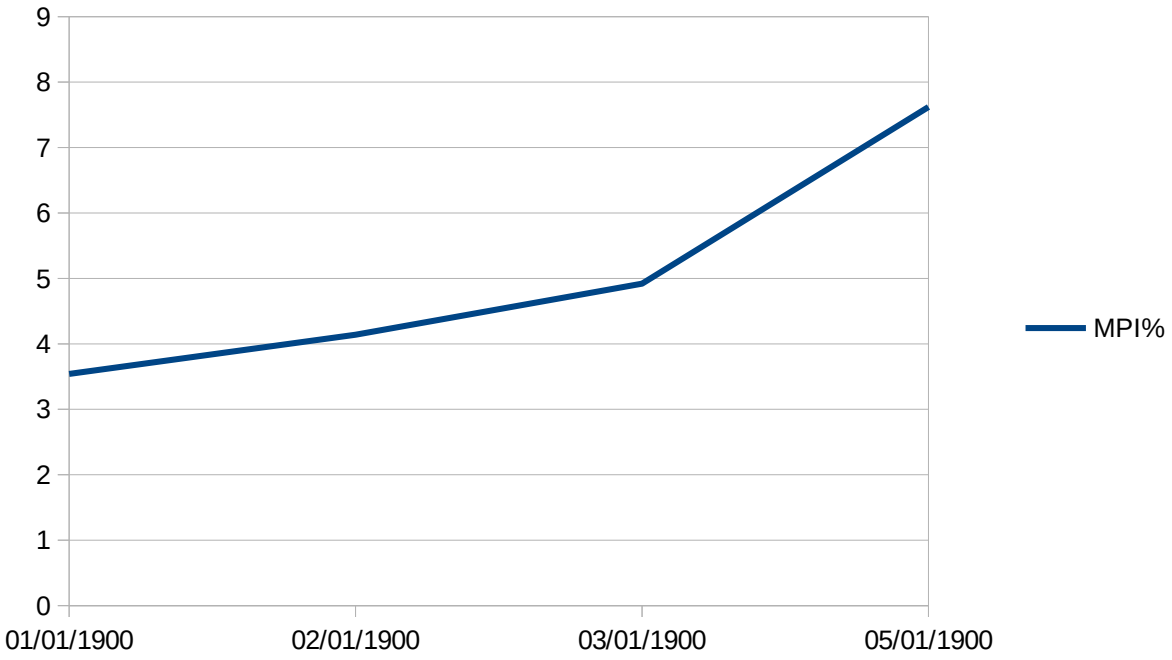
Залежність часу виконання програми та часу коммунікації між процессами від к-ть процессів



Середній час виконання кожної программи



Агрегований час виконання кожної програми



AppTimeAggregate/MPITimeaggregate залежність від к-ті потоків

Висновок

У ході виконання лабораторної роботи було реалізовано розподілене обчислення за допомогою технології MPI, що дало можливість глибше зрозуміти принципи паралельних обчислень та їх застосування. Під час реалізації програми, основна увага приділялася правильному розподілу даних між процесами та їх ефективній обробці.

Аналізуючи виконання програми, стає зрозуміло, що поточна реалізація не є оптимальною. Ключовою проблемою виявилася необхідність повного читання файлу головним процесом, після чого відбувається розподіл даних між іншими процесами. Такий підхід може стати значним ускладненням при збільшенні обсягу вхідних даних, адже він створює "вузьке горлечко" на етапі читання та розподілу даних.

Один із можливих шляхів оптимізації - це використання розподіленого читання файлу. Це передбачає визначення кількості ліній в файлі та надання кожному процесу прямого доступу до файлу зі зміщеним офсетом. Таким чином, кожен процес зможе самостійно зчитувати потрібний фрагмент даних, що може значно зменшити час, необхідний для ініціалізації та розподілу.

Під час тестування програми із збільшенням кількості вхідних даних з 1000 до 10000 записів не було виявлено суттєвих змін у графіках виконання, що вказує на те, що основна проблема лежить не в обсязі даних для фільтрації, а у їх первісній ініціалізації та розподілі.

На жаль, в рамках даної роботи не було здійснено порівняльного аналізу з послідовною імплементацією програми, що не дозволило отримати статистично значущі дані для повноцінного аналізу ефективності паралельної обробки. Прошу надати додатковий час для реалізації послідовної версії програми та подальшого аналізу отриманих результатів.

Додаток

Репозиторій: https://github.com/shevchenko-univer-subject/mpi-works/

Таблиці:

np	AppTime Mean	MPITime Mean
2	0.027433	0.000972
3	0.028695	0.001189
4	0.030408	0.001495
6	0.03168	0.002414

np	AppTime Aggregate	MPITime Aggregate
2	0.054867	0.001943
3	0.086086	0.003568
4	0.121632	0.005979
6	0.190082	0.014487

np	MPI%
2	3.54
3	4.14
4	4.92
6	7.62