66.26 Arquitecturas paralelas

Trabajo Práctico Final

Integrantes:

Alumno	padron
Llauró, Manuel Luis	95736
Blanco, Sebastian Ezequiel	98539

GitHub:
https://github.com/BlancoSebastianEzequiel/66.26-TP-Final

Índice

1. Objetivo

Se propone la verificación empírica de la ley de amdahl (trabajo constante) versus la ley de Gustafson (tiempo constante) aplicada a un problema de paralelismo utilizando el modelo de programación MapReduce.
Haremos una multiplicación de matrices (ambas de `NxN`) y se realizarán las mediciones de tiempo variando la cantidad de threads involucrados en el procesamiento. Luego se realizarán las mismas mediciones manteniendo fija la cantidad de threads pero variando la dimensión de las matrices.
Finalmente se hara una multiplicacion de dos matrices diferentes de `NxN` usando la libreria CBLAS e instruccion de vectorizacion (MMX) para el compilador con solo un procesador. De esta manera la idea es comparar el tiempo que tarda el map-reduce en serie frente a cblas y la vectorizacion.

2. Desarrollo teorico

2.1. Speed up

Es la mejora en la velocidad de ejecución de una tarea ejecutada en dos arquitecturas similares con diferentes recursos.
La noción de speedup fue establecida por la ley de Amdahl, que estaba dirigida particularmente a la computación paralela. Sin embargo, la speedup se puede usar más generalmente para mostrar el efecto en el rendimiento después de cualquier mejora en los recursos.
De forma genérica se define como:

$$\textbf{speed\_up} = \frac{Rendimiento\_con\_mejora}{Rendimiento\_sin\_mejora} \tag{1}$$

En el caso de mejoras aplicadas a los tiempo de ejecución de una tarea:

$$\textbf{speed\_up} = \frac{T\_ejecucion\_sin\_mejora}{T\_ejecucion\_con\_mejora} \tag{2}$$

2.2. Ley de Amdahl

Utilizada para averiguar la mejora máxima de un sistema de información cuando solo una parte de éste es mejorado.
Establece que la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente.

Suponiendo que nuestro algoritmo se divide en una parte secuencial `s` u una parte paralelizable `p` y siendo `N` la cantidad de threads, entonces podemos decir que:

$$\textbf{speed\_up} = \frac{s+p}{s+\frac{p}{N}} \tag{3}$$

Amdahl establece un límite superior al speedup que puede obtenerse al introducir una mejora en un determinado algoritmo. Este límite superior está determinado por la porción de la tarea sobre la que se aplique la mejora. Entonces si tomamos la ecuacion anterior y calculamos el limite de la misma con `N` tendiendo a infinito tenemos:

$$\textbf{speed\_up\_max} = 1 + \frac{p}{s} \tag{4}$$

2.3. Ley de Gustafson

Establece que cualquier problema suficientemente grande puede ser eficientemente paralelizado. La ley de Gustafson está muy ligada a la ley de Amdahl, que pone límite a la mejora que se puede obtener gracias a la paralelización, dado un conjunto de datos de tamaño fijo, ofreciendo así una visión pesimista del procesamiento paralelo. Por el contrario la ley de Gustafson propone realizar mas trabajo con la misma cantidad de recursos, de esta manera aprovecho la paralelizacion para calcular mas cosas.

Entonces siendo `s` el tiempo de la ejecucucion de la seccion serie, siendo `p` el tiempo de la ejecucion de la seccion paralela y siendo `N` la cantidad de procesadores podemos calcular el speed up como:

$$\textbf{speed\_up} = \frac{s + p * N}{s + p} \tag{5}$$

2.4. Map-reduce

MapReduce es una técnica de procesamiento y un programa modelo de computación distribuida. El algoritmo MapReduce contiene dos tareas importantes.

`Map` toma un conjunto de datos y se convierte en otro conjunto de datos, en el que los elementos se dividen en tuplas `(pares: clave, valor)`.

En el medio ocurre la fase de agrupamiento la cual consiste de agrupar los valores con misma clave en un vector para entregarle a la fase de reduce un conjunto de tuplas `(clave, valores)` donde en este caso el valor son todos los valores en una lista.

`Reduce` recibe un conjunto de tuplas `(clave, valores)` donde el valor es una lista de todos los valores que tenian la misma clave. Entonces reduce aplica una funcion a todos estos valores para retornar un unico valor y asi devolver un conjunto de tuplas `(clave, valor)`

La principal ventaja de MapReduce es que es fácil de escalar procesamiento de datos en múltiples nodos.
De acuerdo a este modelo, basado en la programación funcional, la tarea del usuario consiste en la definición de una función map y una función reduce y definidas estas funciones, el procesamiento es fácilmente paralelizable, ya sea en una sola máquina o en un cluster.

2.5. High Performance Portable Libraries for Dense Linear Algebra

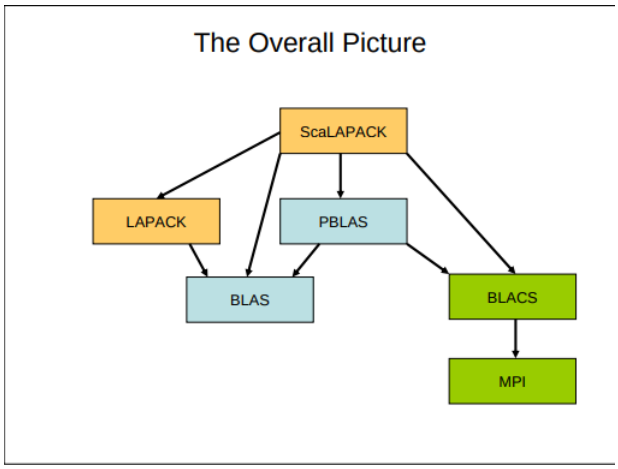

Figura 1: overall picture

2.5.1. LAPACK

LAPACK está escrito en Fortran 90 y proporciona rutinas para resolver sistemas de ecuaciones lineales simultáneas, soluciones de mínimos cuadrados de sistemas de ecuaciones lineales, problemas de valores propios y problemas de valores singulares. También se proporcionan las factorizaciones matriciales asociadas (LU, Cholesky, QR, SVD, Schur, Schur generalizado), al igual que los cálculos relacionados, tales como la reordenación de las factorizaciones de Schur y la estimación de los números de condición. Se manejan matrices densas y con bandas, pero no matrices dispersas generales. En todas las áreas, se proporciona una funcionalidad similar para matrices reales y complejas, con precisión simple y doble.

2.5.2. ScaLAPACK

Es una liberia de rutinas de álgebra lineal de alto rendimiento para máquinas de memoria distribuida en paralelo. ScaLAPACK resuelve sistemas lineales densos y en bandas, problemas de mínimos cuadrados, problemas de valores propios y problemas de valores singulares. Las ideas clave incorporadas en ScaLAPACK incluyen el uso de:

- Una distribución de datos de bloques cíclicos para matrices densas y una distribución de datos de bloques para matrices en bandas, parametrizable en tiempo de ejecución.
- Algoritmos de partición de bloque para asegurar altos niveles de reutilización de datos.

- Componentes modulares de bajo nivel bien diseñados que simplifican la tarea de paralelizar las rutinas de alto nivel haciendo que su código fuente sea el mismo que en el caso secuencial.

2.5.3. CBLAS

BLAS (Subprogramas de Álgebra Lineal Básica) son rutinas que proporcionan bloques de construcción estándar para realizar operaciones básicas de vectores y matrices. Las BLAS de nivel 1 realizan operaciones escalares, vectoriales y vectoriales, las BLAS de nivel 2 realizan operaciones de vectores matriciales y las BLAS de nivel 3 realizan operaciones de matriz-matriz. Debido a que los BLAS son eficientes, portátiles y ampliamente disponibles, se usan comúnmente en el desarrollo de software de álgebra lineal de alta calidad, LAPACK, por ejemplo.
CBLAS es una interfaz de lenguaje C para BLAS.
Nosotros estaremos usando Cblas para este tp.

3. Implementacion

3.1. Explicacion del modelo

La implementación del MapReduce para resolver el problema esta basado en el siguiente esquema:

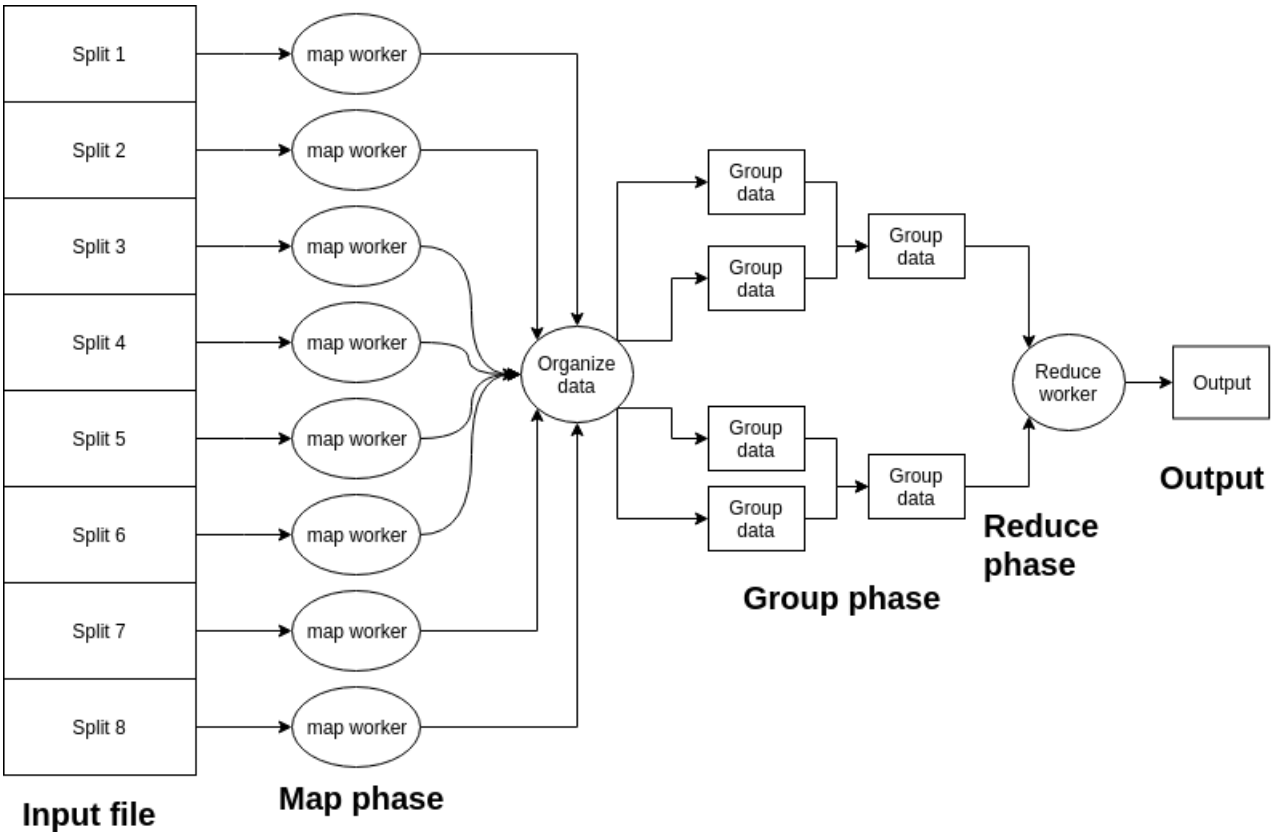

Figura 2: Esquema de un map reduce

En nuestro caso creamos una clase llamada `MapReduce` la cual usa una libreria de `python` llamada `multiprocessing` en donde usamos el modulo `pool` el cual ofrece un medio conveniente para paralelizar la ejecución de una función a través de múltiples valores de entrada, distribuyendo los datos de entrada a través de procesos (paralelismo de datos).

Entonces lo que hicimos fue instanciar dos `pool`, uno para hacer el map y el otro para el reduce de manera que el primero se le pasa como atributo la cantidad de worker en el cual se quiere paralelizar el problema y el segundo solo se usa uno de manera tal que la fase de reduce sea la serie.

3.2. Multiplicacion de matrices por bloques

3.2.0.1 Preprocesamiento

Sean dos matrices `A` de `NxN` y `B` de `NxN` las dividimos en `(N/2)x(N/2)` bloques cada una. Luego generamos una lista de tuplas donde cada una tiene la posicion `(r, c)` de un bloque de la matriz `A`, tiene el bloque en custion `a_block_rc`, y la fila numero `c` de bloques de la matriz `B`, quedando con este formato:
`(r, c, a_block_rc, b_block_c)`

3.2.0.2 Mapeo

Recibimos la posicion `r`, `c` del bloque `a`, el bloque `a` y una lista de bloques `b` que es la fila `c` de bloques en la matriz B.
Entonces multiplicamos el bloque `a` por cada bloque de la lista de bloques `b` y guardamos en un vector una tupla con una clave `r`, `c_b` donde `c_b` es el indice en la lista de bloques `b` y como valor guardamos la multiplicacion. Por cada multiplicacion, agregamos una de estas tuplas al vector de salida para luego devolver este.

3.2.0.3 Reduccion

Recibimos la posicion de un bloque de salida y una lista de multiplicaciones parciales de bloques. Se suman estas multiplicaciones parciales y se devuelve un vector con los valores resultantes del la multiplicacion. Pero por cada valor se calcula la posicion de salida del mismo en la matriz resultante y nos deshacemos de la posicion de los bloques

3.3. Multiplicacion de matrices de elemento por fila

3.3.0.1 Preprocesamiento

Sean dos matrices `A` de `NxN` y `B` de `NxN` generamos una lista de tuplas a partir de las dos matrices. Se itera por cada elemento (`a_ij`) de la matriz `A` y se guarda en cada tupla el numero de fila `i` del elemento `a_ij`, el elemento `a_ij` y la fila `j` de la matriz `B`. Quedando cada tupla de la siguiente manera:
`(i, a_ij, B[j])`

3.3.0.2 Mapeo

De esta manera, en la funcion map, obtenemos partes de esta lista de tuplas y devolvemos un par clave, valor donde la clave es la posicion de salida de la matriz

resultante `(i, j)` y el valor es la multiplicacion del elemento `a_ij` contra cada elemento de la fila `j` de la matriz B

3.3.0.3 Reduccion

Obtenemos una posicion de salida y una lista de valores que resultaron de la multiplicacion que se hizo en el map. Entonces se suman las multiplicaciones parciales y se obtiene el valor en la posicion de salida de la matriz resultante

3.4. Multiplicacion de matrices de columna por fila

3.4.0.1 Preprocesamiento

Sean dos matrices `A` de `NxN` y `B` de `NxN` generamos una lista de tuplas a partir de las dos matrices. Se guarda en cada tupla la columna `i` de la matriz `A` y la fila `i` de la matriz `B`. Quedando cada tupla de la siguiente manera:
`(A[:][i], B[i])`

3.4.0.2 Mapeo

Recibimos una columna de la matriz A y una fila de la matriz B y por cada elemento de la columna `elem_a` lo multiplicamos por cada elemento de la fila `elem_b` obteniendo una matriz parcial de la multiplicacion. Por cada multiplicacion guardamos en un vector una tupla con un par clave valor donde la clave es la posicion de salida de la matriz resultante y el valor es la multiplicacion anteriormente mencionada. Finalmente se devuelve el vector de tuplas.

3.4.0.3 Reduccion

Se recibe la posicion de salida de la matriz resultante y una lista de multiplicaciones parciales. Entonces se suman estas y se devuelve la posicion de salida y la suma.

3.5. Forma de ejecucion

Para el caso de Amdahl multiplicamos dos matrices de `100x100` y cada una de estas multiplicaciones la realizamos para `1`, `2`, `3`, `4`, `8`, `16`, `32`, `64` y `128` threads.

Para el caso de gustafson se usan siempre 4 threads multiplicando dos matrices de `2x2`, `4x4`, `16x16`, `64x64`, `100x100`, `200x200`, y `300x300`.

Luego para el caso de `cblas` y de instrucciones vectoriales (MMX) se usa un thread multiplicando dos matrices de `400x400`

Para poder probar este trabajo se debe clonar el repositorio (el link esta en la caratula) y abrir una terminal en el `root` del mismo.

Para compilar `cblas` y las instrucciones vectoriales que estan en lenguaje c se debe ejecutar:
`$ make.`

Para realizar el calculo de `cblas` y las instrucciones vectoriales que estan en lenguaje c se debe ejecutar:
`$ ./app.`

Para realizar el calculo de map-reduce se debe ejecutar:
`$ sh scripts/run.sh.`

Luego para generar los graficos que vemos en el informe se debe ejecutar:
`$ sh scripts/generate_output_data.sh`

Y finalmente para generar el informe debemos ejecutar:
`$ sh scripts/make_report.sh`

Tambien hay un script que corre estos ultimos tres comandos en un solo script:
`$ sh scripts/run_all.sh`

3.6. Datos sobre la computadora que se utilizó

El equipo sobre el que se realizarán las mediciones es una laptop con un procesador Intel core I7 que posee 4 nucleos a 2.7 Ghz, es decir, soporta hasta 4 threads en paralelo, con 16 Gb de memoria y corriendo sobre un sistema Linux.
Para averiguar estos datos en linux se ejecutaron los siguientes comandos:

- **Cantidad de cores:** `$ grep -c processor /proc/cpuinfo`
- **Velocidad de reloj:** `$ lscpu | grep GHz`
- **Memoria RAM:** `$ free -g`

4. Resultados

4.1. Multiplicacion por bloques

4.1.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	3178.862333	537.276983	100
1	2	2735.191107	537.839651	100
2	3	2217.736483	543.824673	100
3	4	1868.114948	537.915468	100
4	8	1906.499386	584.030390	100
5	16	1838.134766	536.170721	100
6	32	1919.524193	579.051971	100
7	64	1897.083282	517.779827	100
8	128	2035.792112	521.183252	100

Figura 3: Salida de los tiempos en serie y paralelo en milisegundos

De acuerdo a estos datos podemos calcular el speed up maximo, real y teórico.

	number_of_threads	theoretical_speed_up	real_speed_up	max_speed_up
0	1	1.000000	1.000000	6.916617
1	2	1.747367	1.717734	6.916617
2	3	2.327099	2.152307	6.916617
3	4	2.789910	2.394193	6.916617
4	8	3.976034	3.028579	6.916617
5	16	5.049405	3.646863	6.916617
6	32	5.837329	3.909908	6.916617
7	64	6.331306	4.411339	6.916617
8	128	6.611032	4.760814	6.916617

Figura 4: Speed up real, teorico y maximo segun la cantidad de threads

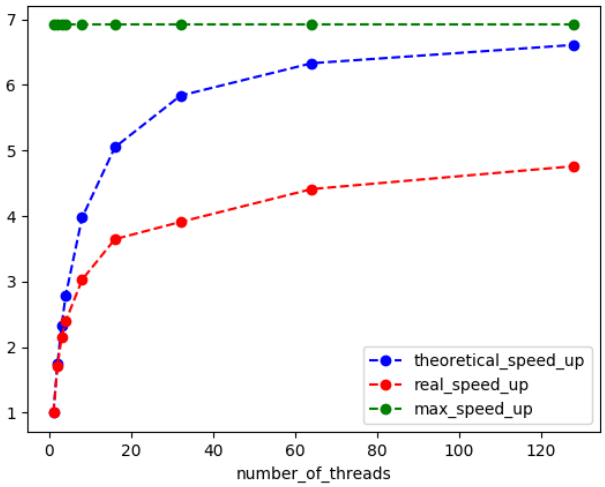

Figura 5: Grafico

Podemos observar que el speed up teorico tiende al maximo speed up mientras que el real encuentra un maximo luego de los 4 thread aproximadamente ya que la cantidad maxima de core de la computadora donde se corrio el programa es de 4

4.1.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	4	1.578808	2.407789	2
1	4	1.833916	1.959085	4
2	4	10.851145	11.962414	16
3	4	403.552532	135.166407	64
4	4	2299.946070	640.191793	100
5	4	16672.408342	4410.587788	200
6	4	56456.478119	14151.839733	300

Figura 6: Salida de los tiempos en serie y paralelo en milisegundos

Podemos ver que estos resultados demuestran que la sección serie del problema se mantiene casi constante respecto de la sección paralela que varía en forma ascendente con el tamaño de los datos de entrada. Así, la fracción secuencial representa menos al tiempo total en la medida que la carga de trabajo aumenta

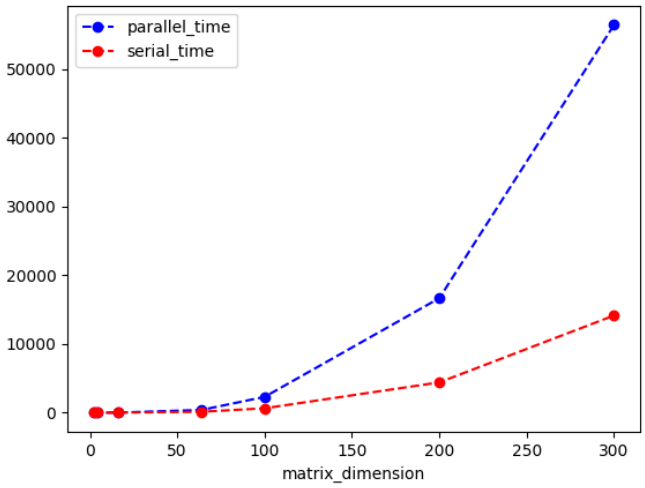

Figura 7: Tiempo paralelo y serie en funcion de la dimension de las matrices de entrada

Luego a partir de estos datos podemos calcular el speed up y obtuvimos lo siguiente:

	matrix_dimension	speed_up
0	2	2.188087
1	4	2.450500
2	16	2.426934
3	64	3.247290
4	100	3.346774
5	200	3.372396
6	300	3.398718

Figura 8: Tabla de valores del speed up

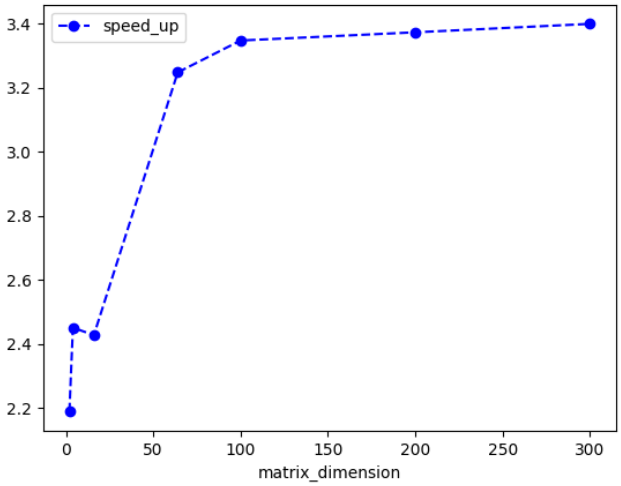

Figura 9: Grafico del speed up

4.2. Multiplicacion elemento por fila

4.2.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	3014.070272	565.473557	100
1	2	2426.795244	565.452337	100
2	3	1894.866705	563.212395	100
3	4	1854.820490	556.397915	100
4	8	1888.506889	557.536125	100
5	16	1852.516413	568.376541	100
6	32	1889.244318	566.127062	100
7	64	1901.825666	567.384481	100
8	128	1874.574900	557.483673	100

Figura 10: Salida de los tiempos en serie y paralelo en milisegundos

De acuerdo a estos datos podemos calcular el speed up maximo, real y teórico.

	number_of_threads	theoretical_speed_up	real_speed_up	max_speed_up
0	1	1.000000	1.000000	6.33017
1	2	1.727155	1.682125	6.33017
2	3	2.279727	2.057255	6.33017
3	4	2.713850	2.363701	6.33017
4	8	3.799003	3.082213	6.33017
5	16	4.748332	3.538496	6.33017
6	32	5.426320	3.927551	6.33017
7	64	5.843501	4.135334	6.33017
8	128	6.077108	4.250894	6.33017

Figura 11: Speed up real, teorico y maximo segun la cantidad de threads

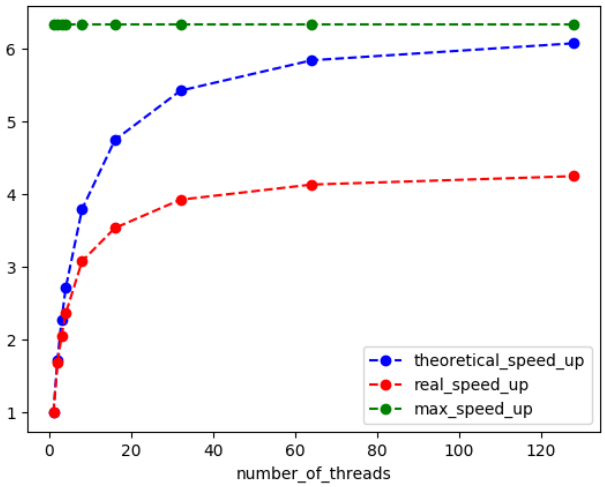

Figura 12: Grafico

Podemos observar que el speed up teorico tiende al maximo speed up mientras que el real encuentra un maximo luego de los 4 thread aproximadamente ya que la cantidad maxima de core de la computadora donde se corrio el programa es de 4

4.2.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	4	1.441479	2.287149	2
1	4	1.933575	1.022339	4
2	4	12.011766	4.426241	16
3	4	502.124786	178.322554	64
4	4	1892.707109	565.406799	100
5	4	16864.702940	5348.829508	200
6	4	66471.372128	18364.505768	300

Figura 13: Salida de los tiempos en serie y paralelo en milisegundos

Podemos ver que estos resultados demuestran que la sección serie del problema se mantiene casi constante respecto de la sección paralela que varía en forma ascendente con el tamaño de los datos de entrada. Así, la fracción secuencial representa menos al tiempo total en la medida que la carga de trabajo aumenta

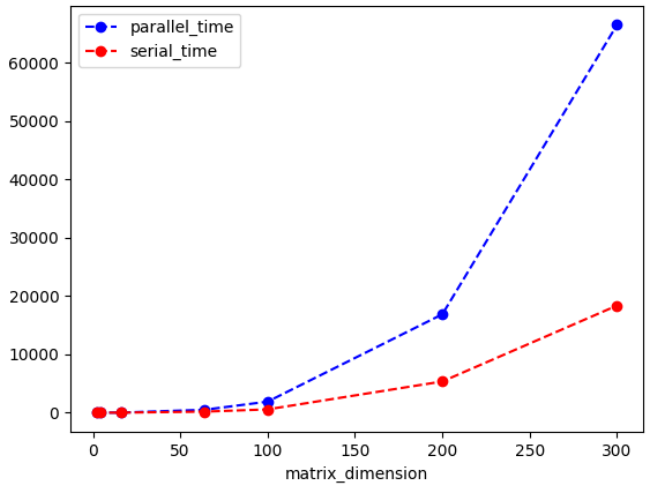

Figura 14: Tiempo paralelo y serie en funcion de la dimension de las matrices de entrada

Luego a partir de estos datos podemos calcular el speed up y obtuvimos lo siguiente:

	matrix_dimension	speed_up
0	2	2.159793
1	4	2.962413
2	16	3.192194
3	64	3.213800
4	100	3.309950
5	200	3.277626
6	300	3.350587

Figura 15: Tabla de valores del speed up

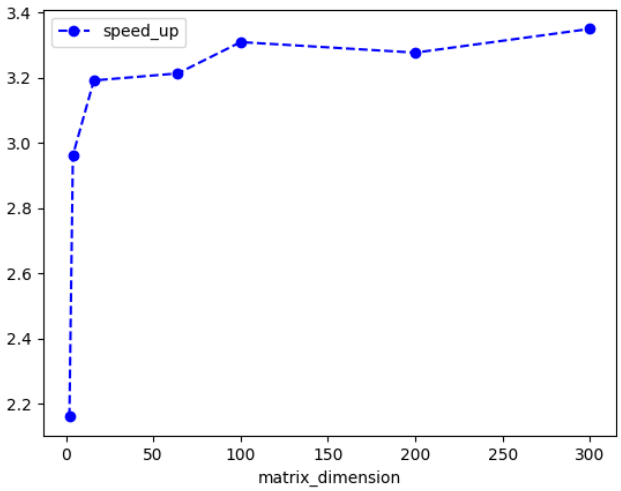

Figura 16: Grafico del speed up

4.3. Multiplicacion columna por fila

4.3.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	2716.657877	613.165617	100
1	2	2333.233118	605.530024	100
2	3	1750.370502	608.537197	100
3	4	1775.277376	637.945175	100
4	8	1777.448416	643.797874	100
5	16	1838.029385	613.466740	100
6	32	2003.676891	618.367672	100
7	64	1804.571152	641.252518	100
8	128	2157.028198	621.368408	100

Figura 17: Salida de los tiempos en serie y paralelo en milisegundos

De acuerdo a estos datos podemos calcular el speed up maximo, real y teórico.

	number_of_threads	theoretical_speed_up	real_speed_up	max_speed_up
0	1	1.000000	1.000000	5.430545
1	2	1.688984	1.658307	5.430545
2	3	2.192522	1.978959	5.430545
3	4	2.576604	2.230821	5.430545
4	8	3.494968	2.795964	5.430545
5	16	4.252883	3.365851	5.430545
6	32	4.770103	3.850384	5.430545
7	64	5.078944	3.653488	5.430545
8	128	5.248863	4.353351	5.430545

Figura 18: Speed up real, teorico y maximo segun la cantidad de threads

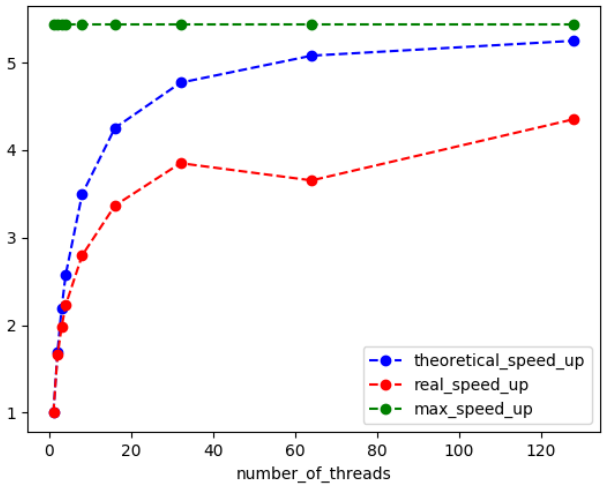

Figura 19: Grafico

Podemos observar que el speed up teorico tiende al maximo speed up mientras que el real empieza a tender a un maximo menor debido a que la cantidad maxima de core de la computadora donde se corrio el programa es de 4. Pero podemos ver tambien que hubo variaciones y que el speed up real tuvo anomalias y tendencias a crecer a pesar de el maximo de cores reales. Esto se debe a que al aumentar la cantidad de thread, la seccion serie del problema no se mantuvo constante si no que tuve ciertas variaciones equilibrando la caida de perfonmance de la seccion paralela. Esta variacion en el tiempo de la parte serie se debe a la forma en que quedan organizados los datos una vez procesados por los map workers

4.3.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	4	1.410007	2.422810	2
1	4	1.790047	3.273010	4
2	4	8.985043	7.999897	16
3	4	557.041168	226.171494	64
4	4	2036.477327	702.104807	100
5	4	17703.527451	5522.381067	200
6	4	67648.321867	19749.793291	300

Figura 20: Salida de los tiempos en serie y paralelo en milisegundos

Podemos ver que estos resultados demuestran que la sección serie del problema se mantiene casi constante respecto de la sección paralela que varía en forma ascendente con el tamaño de los datos de entrada. Así, la fracción secuencial representa menos al tiempo total en la medida que la carga de trabajo aumenta

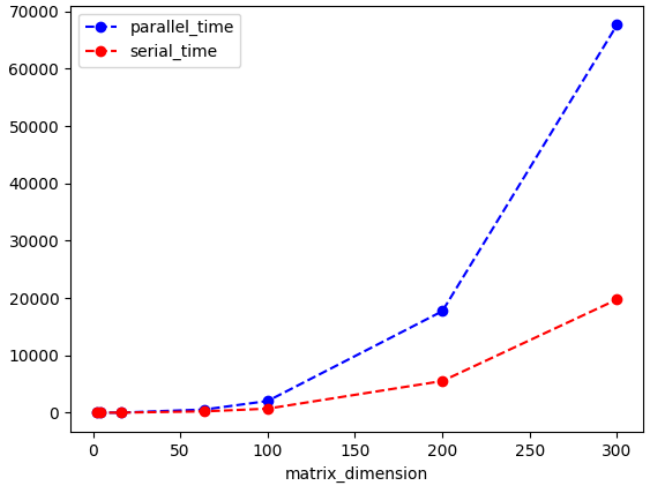

Figura 21: Tiempo paralelo y serie en funcion de la dimension de las matrices de entrada

Luego a partir de estos datos podemos calcular el speed up y obtuvimos lo siguiente:

	matrix_dimension	speed_up
0	2	2.103633
1	4	2.060652
2	16	2.587002
3	64	3.133678
4	100	3.230874
5	200	3.286696
6	300	3.322075

Figura 22: Tabla de valores del speed up

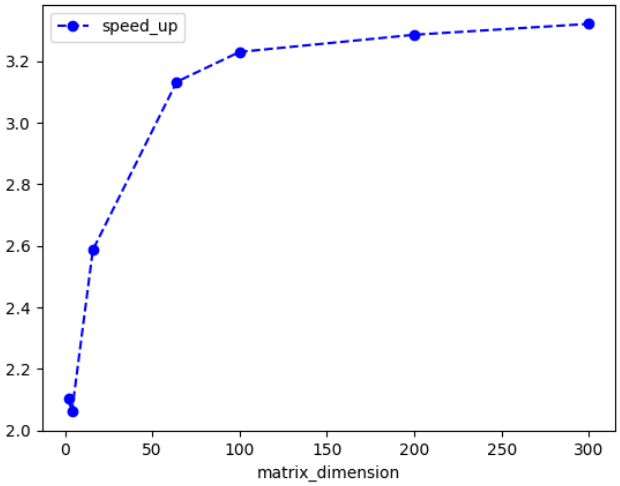

Figura 23: Grafico del speed up

4.4. Cblas e instrucciones vectorizadas

	program	time_elapsed	matrix_dim	number_of_threads
0	cblas_dgemm	0.032921	500	1
1	blocked_dgemm_sse	0.367660	500	1

Figura 24: Tiempo serie de multiplicacion en segundos

Podemos decir que en general, las corridas con mapReduce, en el caso donde se uso un solo thread, si sumamos el tiempo paralelo y serie (que es todo serie ya que se usa un solo thread) nos da en promedio de los tres tipos de multiplicacion tardaron alrededor de 3 segundos. En cambio podemos ver que usando `cblas` tenemos un tiempo de 0.03 segundos y usando las instrucciones vectorizadas 0.36 segundos.
De esta manera vemos que `cblas` y las instrucciones vectorizadas aprovechan mucho mejor el hardware para la realizacion de la misma operacion. Tambien es cierto que el map reduce hace uso de un modulo de `python` llamado `pool` que puede ser que sume latencia al momento de dividir el trabajo, pero cuando aumentamos mucho el volumen del trabajo se ve en los graficos que se aprovecha mejor la paralelizacion.

5. Conclusiones

Podemos decir que obtuvimos resultados esperados pero se tuvieron que hacer varias corridas y ajustar ciertos numeros para entender por que llegamos a estos resultados.
Primero para que las curvas de speed up de Amdahl nos den bien habia que tener en cuenta que la dimension de las matrices tenia que ser lo suficientemente grandes como para tener un becnhmark rasonable, pero tambien hay un limite superior para el cual no superamos la capacidad de los procesadores. Luego, una vez mapeados los datos, se aprovecho el uso de los multiprocesadores para reordenar los datos de manera tal que la parte de `reduce` pueda leerlos. Es decir, cuando los ordenamos lo hacemos dividiendo el trabajo en partes donde cada una la realiza cada procesador. De esta manera obtenemos una organizacion tipo arbol donde evitamos un cuello de botella.
Y segundo se coloco un `sleep` de medio segundo (que no afecto el calculo del tiempo paralelo-serie transcurrido) para evitar que cualquier trabajo que no sea puramente vinculado a la CPU afecte nuestro porgrama (como por ejemplo I/O)

Finalmente podemos decir que hay que tener en cuenta que hay otros programas corriendo en las cuatro CPU que tiene la computadora en la cual se probo este programa y que dependiendo del tamaño de informacion que manejamos, podemos tener un cuello de botella ya sea por intercambios de memoria o por exceso de memoria. Entonces se tuvieron que hacer varias corridas analizando el trafico de informacion mediante el comando `gnome-system-monitor` donde filtrando los procesos y solo viedno los de `python` pudimos ver el uso de cada CPU y graficos al respecto.

6. Anexo

6.1. src/app.py

```python
import time
from typing import Type
from math import ceil
# from src.controller.threaded import Threaded as MapReduce
from src.controller.pool import Pool as MapReduce
from src.controller.utils import get_random_matrix_of_dim_n
from src.model.element_by_row_block import ElementByRowBlock
from src.model.column_by_row import ColumnByRow
from src.model.by_blocks import ByBlocks
from src.model.multiply_matrices_interface import
    ↪ MultiplyMatricesInterface
from src.controller.generate_output_data import OutputData
from src.controller.utils import get_version_number


SAVE = True
VERSION = get_version_number()


def gustafson(model: Type[MultiplyMatricesInterface]):
    name = model.__name__
    print(f"--------------RUNNING GUSTAFSON--------------")
    output_data = OutputData(version=VERSION)
    num_workers = 4
    for matrix_dim in [2, 4, 16, 64, 100, 200, 300]:
        print(f"RUNNING WITH MATRIX DIMENSION: {matrix_dim}")
        serial, parallel = run(num_workers, matrix_dim, model)
        output_data.add_data(serial, parallel, num_workers,
    ↪ matrix_dim)
    if SAVE:
        output_data.save_data(name + '_gustafson_output.png')
        output_data.graph_gustafson_exec_time(name +
    ↪ '_gustafson_exec_time.png')
        output_data.graph_gustafson_speed_up(name +
    ↪ '_gustafson_speed_up.png')
        output_data.save_df_data_to_json()


def amdahl(model: Type[MultiplyMatricesInterface]):
    name = model.__name__
    print(f"--------------RUNNING AMDAHL--------------")
    output_data = OutputData(version=VERSION)
    matrix_dim = 100
    for num_workers in [1, 2, 3, 4, 8, 16, 32, 64, 128]:
        print(f"RUNNING WITH NUM WORKERS: {num_workers}")
        serial, parallel = run(num_workers, matrix_dim, model)
```

```
43          output_data.add_data(serial, parallel, num_workers,
    ↪ matrix_dim)
44      if SAVE:
45          output_data.save_data(name + '_amdahl_output.png')
46          output_data.graph_amdahl_speed_up(name +
    ↪ '_amdahl_speed_up.png')
47          output_data.save_df_data_to_json()
48
49
50  def run(num_workers, matrix_dim, model:
    ↪ Type[MultiplyMatricesInterface]):
51      map_worker = model.map_worker
52      reduce_worker = model.reduce_worker
53      mapper = MapReduce(map_worker, reduce_worker)
54
55      matrix_a = get_random_matrix_of_dim_n(matrix_dim)
56      matrix_b = get_random_matrix_of_dim_n(matrix_dim)
57
58      div = ceil(matrix_dim/2)
59
60      input_data = model.pre_processing(matrix_a, matrix_b, row_p=div,
    ↪ col_p=div)
61
62      partitioned_data = mapper.map(input_data,
    ↪ num_workers=num_workers)
63      mapper.reduce(partitioned_data)
64
65      statistics = mapper.get_statistics()
66      parallel_time = statistics.get_time_elapsed('parallel')
67      serial_time = statistics.get_time_elapsed('serial')
68      return serial_time, parallel_time
69
70
71  def run_model(model: Type[MultiplyMatricesInterface]):
72      print(f"************************************")
73      print(f"{model.__name__}")
74      print(f"************************************")
75      amdahl(model)
76      gustafson(model)
77
78
79  start = time.time()
80  run_model(ElementByRowBlock)
81  run_model(ColumnByRow)
82  run_model(ByBlocks)
83  end = time.time()
84  print(f"************************************")
85  print(f"The process lasted {end-start} seconds")
86  print(f"************************************")
```

Listing 1: app

6.2. src/graphs.py

```
1 from src.controller.generate_output_data import OutputData
2 from src.controller.utils import get_version_number
3
4 output = OutputData(get_version_number)
5 output.read_dfs_data_from_json()
6 output.graph_dfs()
7
8 dgemm_output_data = OutputData(get_version_number, avoid=True)
9 dgemm_df = dgemm_output_data.get_df_from_csv("src/data/dgemm.csv")
10 dgemm_output_data.save_df_in_image(dgemm_df, "dgemm.png")
```

Listing 2: graphs

6.3. src/main.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include "cblas/cblas_dgemm.h"
5 #include "controller/utils.h"
6 #include "vectorization/blocked_dgemm_sse.h"
7 #include "controller/file.h"
8
9 double run_cblas_dgemm(int N, double* A, double* B, double* C) {
10     clock_t start, stop;
11     double alpha = 1.0;
12     double beta = 0.0;
13     init_arr(N, N, 2, A);
14     init_arr(N, N, 1, B);
15     init_arr(N, N, 0, C);
16     start = clock();
17     mult(A, B, C, alpha, beta, N, N, N);
18     stop = clock();
19     double elapsed_seconds = ((double)(stop - start)) /
    ↪ CLOCKS_PER_SEC;
20     printf("Elapsed time = %f seconds\n", elapsed_seconds);
21     return elapsed_seconds;
22 }
23
24 double run_blocked_dgemm_sse(int N, double* A, double* B, double* C)
    ↪ {
25     clock_t start, stop;
26     init_arr(N, N, 2, A);
27     init_arr(N, N, 1, B);
28     init_arr(N, N, 0, C);
29     start = clock();
30     square_dgemm_blocked_sse(A, B, C, N, 2);
31     stop = clock();
32     double elapsed_seconds = ((double)(stop - start)) /
    ↪ CLOCKS_PER_SEC;
33     printf("Elapsed time = %f seconds\n", elapsed_seconds);
34     return elapsed_seconds;
35 }
36
37 int main() {
38     char* attributes[4] = {
39             "program",
40             "time_elapsed",
41             "matrix_dim",
42             "number_of_threads"
43     };
44     file_t* file = create_file("src/data/dgemm.csv", "w+",
    ↪ attributes, 4);
45     int N = 400;
46     double A[N*N];
```

```
47      double B[N*N];
48      double C[N*N];
49      double elapsed_seconds;
50
51
52      char elapsed_second_str[20];
53      char* values[4] = {"cblas_dgemm", elapsed_second_str, "500",
        ↪ "1"};
54
55      elapsed_seconds = run_cblas_dgemm(N, A, B, C);
56      double_to_string(elapsed_second_str, elapsed_seconds, 20);
57      add_row(file, values, 4);
58
59      elapsed_seconds = run_blocked_dgemm_sse(N, A, B, C);
60      double_to_string(elapsed_second_str, elapsed_seconds, 20);
61      values[0] = "blocked_dgemm_sse";
62      values[1] = elapsed_second_str;
63      add_row(file, values, 4);
64      delete(file);
65      return 0;
66  }
```

Listing 3: main

6.4. src/cblas/cblas_dgemm.h

```
1 #ifndef CBLAS_DGEMM
2 #define CBLAS_DGEMM
3
4 #include <stdio.h>
5 #include <cblas.h>
6
7 /*
8     The arguments provide options for how Intel MKL performs the
    ↪ operation.
9     In this case:
10
11    CblasRowMajor:
12    Indicates that the matrices are stored in row major order, with
    ↪ the elements
13    of each row of the matrix stored contiguously as shown in the
    ↪ figure above.
14
15    CblasNoTrans:
16    Enumeration type indicating that the matrices A and B should not
    ↪ be
17    transposed or conjugate transposed before multiplication.
18
19    m, n, k:
20    Integers indicating the size of the matrices:
21
22    A: m rows by k columns
23
24    B: k rows by n columns
25
26    C: m rows by n columns
27
28    alpha:
29    Real value used to scale the product of matrices A and B.
30
31    A:
32    Array used to store matrix A.
33
34    k:
35    Leading dimension of array A, or the number of elements between
    ↪ successive
36    rows (for row major storage) in memory. In the case of this
    ↪ exercise the
37    leading dimension is the same as the number of columns.
38
39    B:
40    Array used to store matrix B.
41
42    n:
43    Leading dimension of array B, or the number of elements between
    ↪ successive
```

```
44      rows (for row major storage) in memory. In the case of this
    ↪ exercise the
45      leading dimension is the same as the number of columns.
46
47      beta:
48      Real value used to scale matrix C.
49
50      C:
51      Array used to store matrix C.
52
53      n:
54      Leading dimension of array C, or the number of elements between
    ↪ successive
55      rows (for row major storage) in memory. In the case of this
    ↪ exercise the
56      leading dimension is the same as the number of columns.
57 */
58
59 int mult(double *A, double *B, double *C, double alpha, double beta,
    ↪ int m, int k, int n);
60
61 #endif  // CBLAS_DGEMM
```

Listing 4: cblas_dgemm

6.5. src/cblas/cblas_dgemm.c

```
1 #include "cblas_dgemm.h"
2
3 int mult(double *A, double *B, double *C, double alpha, double beta,
    ↪ int m, int k, int n) {
4     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k,
    ↪ alpha, A, k, B, n, beta, C, n);
5     return 0;
6 }
```

Listing 5: cblas_dgemm

6.6. src/vectorization/blocked_dgemm_sse.h

```
1 #ifndef BLOCKED_DGEMM_SSE_H
2 #define BLOCKED_DGEMM_SSE_H
3
4 void basic_dgemm_sse(const double *restrict A, const double
    ↪ *restrict B,
5         double *restrict C, int N, int block_size);
6 void do_block_sse(const double *A, const double *B, double *C, int
    ↪ i, int j,
7                   int k, int N, int block_size);
8 void square_dgemm_blocked_sse(const double *A, const double *B,
    ↪ double *C,
9                               int N, int block_size);
10
11 #endif  // BLOCKED_DGEMM_SSE_H
```

Listing 6: vectorization/blocked_dgemm

6.7. src/vectorization/blocked_dgemm_sse.c

```
1 #include "blocked_dgemm_sse.h"
2 #include "../controller/utils.h"
3 /*
4   In case you're wondering, dgemm stands for:
5   Double-precision, GEneral Matrix-Matrix multiplication.
6   A is M-by-K
7   B is K-by-N
8   C is M-by-N
9   lda is the leading dimension of the matrix (the M of square_dgemm).
10 */
11
12
13 void basic_dgemm_sse(const double *restrict A, const double
    ↪ *restrict B,
14         double *restrict C, int N, int block_size) {
15     unsigned i, j, k;
16     for (i = 0; i < block_size; ++i) {
17         for (j = 0; j < block_size; ++j) {
18             double cij = C[j*N + i];
19             #pragma vector always
20             for (k = 0; k < block_size; ++k) {
21                 cij += A[i + k * N] * B[k + j * N];
22             }
23             C[j*N + i] = cij;
24         }
25     }
26 }
27
28 void do_block_sse(const double *A, const double *B, double *C, int
    ↪ i, int j,
29                   int k, int N, int block_size) {
30     basic_dgemm_sse(A + i + k*N, B + k + j*N, C + i + j*N, N,
    ↪ block_size);
31 }
32
33 void square_dgemm_blocked_sse(const double *A, const double *B,
    ↪ double *C,
34                               int N, int block_size) {
35     unsigned bi, bj, bk;
36     for (bi = 0; bi < (N / block_size); ++bi) {
37         const unsigned i = bi * block_size;
38         for (bj = 0; bj < (N / block_size); ++bj) {
39             const unsigned j = bj * block_size;
40             for (bk = 0; bk < (N / block_size); ++bk) {
41                 const unsigned k = bk * block_size;
42                 do_block_sse(A, B, C, i, j, k, N, block_size);
43             }
44         }
45     }
46 }
```

Listing 7: vectorization/blocked_dgemm

6.8. src/controller/file.h

```
1 #ifndef FILE_H
2 #define FILE_H
3
4 #include<stdio.h>
5 #include<string.h>
6
7 typedef struct file file_t;
8
9 file_t* create_file(char* filename, char* mode, char** attributes,
    ↪ int cols);
10 int add_row(file_t *file, char **values, int size);
11 int build_row(char **values, char *row, int max_bytes, int size);
12 void delete(file_t* file);
13
14 #endif  // FILE_H
```

Listing 8: file

6.9. src/controller/file.c

```c
#include "file.h"
#include <stdlib.h>
#include "utils.h"

typedef struct file {
    FILE *fp;
    char *filename;
    char** attributes;
    int num_of_cols;
    int num_of_rows;
} file_t;

file_t* create_file(char* filename, char* mode, char** attributes,
    ↪ int cols) {
    file_t* file = (file_t*) malloc(sizeof(file_t));
    if (!file) {
        return NULL;
    }
    file->fp = fopen(filename, mode);
    file->filename = filename;
    file->num_of_cols = cols;

    char file_header[256];
    if (build_row(attributes, file_header, 256, cols) < 0) {
        delete(file);
        return NULL;
    }
    add_row(file, attributes, cols);
    file->attributes = attributes;
    file->num_of_rows = 0;
    return file;
}

int build_row(char **values, char *row, int max_bytes, int size) {
    int bytes = 0;
    int pos = 0;
    char* buff;
    for (int i = 0; i < size; i++) {
        buff = (row)+pos;
        if (i == size-1) {
            bytes = snprintf(buff, max_bytes, "%s", values[i]);
        } else {
            bytes = snprintf(buff, max_bytes, "%s, ", values[i]);
        }
        if (bytes < 0) {
            return bytes;
        }
        pos += bytes;
    }
    return bytes;
```

```
50 }
51
52 int add_row(file_t *file, char **values, int size) {
53     if (size != file->num_of_cols) {
54         return 1;
55     }
56     char file_header[256];
57     int bytes = build_row(values, file_header, 256,
    ↪ file->num_of_cols);
58     if (bytes < 0) {
59         delete(file);
60         return bytes;
61     }
62     fprintf(file->fp, "%s\n", file_header);
63     file->num_of_rows +=1;
64     return 0;
65 }
66
67 void delete(file_t* file) {
68     fclose(file->fp);
69     free(file);
70 }
```

Listing 9: file

6.10. src/controller/utils.h

```
1 #ifndef UTILS_H
2 #define UTILS_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7
8 void init_arr(double m, double n, double off, double* a);
9 void print_arr(char *name, int m, int n, double *array);
10 void multiply_matrices(const double *a, const double *b, double *c,
    ↪ int n);
11 bool matrix_compare(const double *a, const double *b, int n);
12 int double_to_string(char* buffer, double num, int max_bytes);
13
14 #endif  // UTILS_H
```

Listing 10: utils

6.11. src/controller/utils.c

```c
#include "utils.h"

void init_arr(double m, double n, double off, double* a) {
    int i;
    for (i = 0; i < (m*n); i++) {
        // a[i] = (i + 1) * off;
        a[i] = (i % 10) * off;
    }
}
void print_arr(char *name, int m, int n, double *array) {
    int i, j;
    printf("\n%s\n", name);
    for (i = 0; i < m; i++){
        for (j = 0; j < n; j++) {
            printf("%g\t", array[i+j*n]);
        }
        printf("\n");
    }
}

void multiply_matrices(const double *a, const double *b, double *c,
    ↪ int n) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            for (k = 0; k < n; k++) {
                c[i+k*n] = c[i+k*n] + a[i+j*n] * b[j+k*n];
            }
        }
    }
}

bool matrix_compare(const double *a, const double *b, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (a[i+j] != b[i+j]) {
                return false;
            }
        }
    }
    return true;
}

int double_to_string(char* buffer, double num, int max_bytes) {
    return snprintf(buffer, max_bytes, "%lf", num);
}
```

Listing 11: utils

6.12. src/controller/generate_output_data.py

```python
import json
import os
import pandas as pd
from subprocess import call
import matplotlib.pyplot as plt
from src.controller.utils import get_null_list_of_dim_n


class OutputData:
    def __init__(self, version=1, avoid=False):
        self.data = {
            'number_of_threads': [],
            'parallel_time': [],
            'serial_time': [],
            'matrix_dimension': []
        }
        self._colors = ['b-', 'g-', 'r-', 'c-', 'm-', 'y-', 'k-',
    ↪ 'w-']
        self.dfs_data = []
        pics_path = f"./docs/report/pics/graphs_v{version}/"
        if not os.path.isdir(pics_path) and avoid:
            os.system(f'mkdir {pics_path}')
        self.pics_path = pics_path
        files_path = f"./src/data/data_v{version}/"
        if not os.path.isdir(files_path) and avoid:
            os.system(f'mkdir {files_path}')
        self.files_path = files_path

    def add_data(self, serial, parallel, num_workers,
    ↪ matrix_dimension):
        self.data['number_of_threads'].append(num_workers)
        self.data['parallel_time'].append(parallel)
        self.data['serial_time'].append(serial)
        self.data['matrix_dimension'].append(matrix_dimension)

    @staticmethod
    def gustafson_speed_up(a, b, p):
        return (a + p * b) / (a + b)

    @staticmethod
    def amdahl_speed_up(s, p, n):
        return (s + p) / (s + p/n)

    @staticmethod
    def amdahl_max_speed_up(s, p):
        return 1 + p/s

    def save_data(self, df_name):
        df = pd.DataFrame(data=self.data)
        self.save_df_data(df, [], '', {}, df_name, False)
```

```
49
50      def save_df_in_image(self, df, df_name):
51          path = self.pics_path + df_name
52          df.to_html('table.html')
53          command = f'wkhtmltoimage -f png --width 0 table.html {path}'
54          call(command, shell=True)
55          call('rm table.html', shell=True)
56
57      def df_to_csv(self, df, df_name):
58          path = self.files_path + df_name
59          df.to_csv(path, index=False, sep=',', encoding='utf-8-sig')
60
61      def get_df_from_csv(self, filepath):
62          return pd.read_csv(filepath, low_memory=False, sep=',')
63
64      def graph_amdahl_speed_up(self, filename):
65          df = pd.DataFrame(data=self.data)
66          columns = [
67              'number_of_threads',
68              'parallel_time',
69              'serial_time'
70          ]
71          df = df.loc[:, columns]
72          df['theoretical_speed_up'] =
    ↪ get_null_list_of_dim_n(len(df.index))
73          df['theoretical_speed_up'] = df.apply(
74              lambda x: self.amdahl_speed_up(
75                  self.data['serial_time'][0],
76                  self.data['parallel_time'][0],
77                  x['number_of_threads']
78              ),
79              axis=1
80          )
81          df['real_speed_up'] = get_null_list_of_dim_n(len(df.index))
82          df['real_speed_up'] = df.apply(
83              lambda x: self.amdahl_speed_up(
84                  x['serial_time'],
85                  x['parallel_time'],
86                  x['number_of_threads']
87              ),
88              axis=1
89          )
90
91          df['max_speed_up'] = get_null_list_of_dim_n(len(df.index))
92          df['max_speed_up'] = df.apply(
93              lambda x: self.amdahl_max_speed_up(
94                  x['serial_time'],
95                  x['parallel_time']
96              ),
97              axis=1
98          )
99          max_speed_up = df['max_speed_up'][0]
```

```
100         df['max_speed_up'] = df['max_speed_up'].map(lambda x:
    ↪ max_speed_up)
101
102         columns = [
103             'number_of_threads',
104             'theoretical_speed_up',
105             'real_speed_up',
106             'max_speed_up'
107         ]
108         df = df.loc[:, columns]
109         self.save_df_data(
110             df,
111             ['theoretical_speed_up', 'real_speed_up',
    ↪ 'max_speed_up'],
112             'number_of_threads',
113             {
114                 'theoretical_speed_up': 'b-',
115                 'real_speed_up': 'r-',
116                 'max_speed_up': 'g-'
117             },
118             filename,
119             True
120         )
121
122     def graph(self, df, y_axis, x_axis, colors, graph_name):
123         for field in y_axis:
124             plt.plot(
125                 df[x_axis],
126                 df[field],
127                 colors[field],
128                 label=field,
129                 marker='o',
130                 linestyle='dashed'
131             )
132         plt.xlabel(x_axis)
133         plt.yscale('linear')
134         plt.legend(loc='best')
135         plt.savefig(self.pics_path + graph_name)
136         plt.clf()
137
138     @staticmethod
139     def file_exists(path):
140         return os.path.isfile(path) and os.access(path, os.R_OK)
141
142     @classmethod
143     def delete_all_data(cls):
144         os.system('rm  src/data/*')
145
146     def save_df_data_to_json(self):
147         path = f'{self.files_path}data.json'
148         data = []
149         if self.file_exists(path):
```

```
150             with open(path, encoding='utf-8-sig') as json_file:
151                 text = json_file.read()
152                 if text:
153                     data = json.loads(text)
154         with open(path, 'w') as f:
155             json.dump(self.dfs_data+data, f)
156 
157     def save_df_data(self, df, y_axis, x_axis, colors, graph_name,
    ↪ has_graph):
158         df_graph_name = graph_name.split('.png')[0] + '_table.csv'
159         self.df_to_csv(df, df_graph_name)
160         self.dfs_data.append({
161             'has_graph': has_graph,
162             'df_path_name': df_graph_name,
163             'y_axis': y_axis,
164             'x_axis': x_axis,
165             'colors': colors,
166             'graph_name': graph_name
167         })
168 
169     def read_dfs_data_from_json(self):
170         data_path = f"{self.files_path}data.json"
171         with open(data_path, encoding='utf-8-sig') as json_file:
172             text = json_file.read()
173             self.dfs_data = json.loads(text)
174 
175     def graph_dfs(self):
176         for df_data in self.dfs_data:
177             df_name = df_data['df_path_name']
178             table_graph_name = df_name.split('.csv')[0] + '.png'
179             df_path_name = self.files_path + df_name
180             df = pd.read_csv(df_path_name, low_memory=False, sep=',')
181             self.save_df_in_image(df, table_graph_name)
182             if df_data['has_graph']:
183                 y_axis = df_data['y_axis']
184                 x_axis = df_data['x_axis']
185                 colors = df_data['colors']
186                 graph_name = df_data['graph_name']
187                 self.graph(df, y_axis, x_axis, colors, graph_name)
188 
189     def graph_gustafson_exec_time(self, filename):
190         df = pd.DataFrame(data=self.data)
191         columns = [
192             'matrix_dimension',
193             'parallel_time',
194             'serial_time'
195         ]
196         df = df.loc[:, columns]
197         self.save_df_data(
198             df,
199             ['parallel_time', 'serial_time'],
200             'matrix_dimension',
```

```
            {
                'parallel_time': 'b-',
                'serial_time': 'r-'
            },
            filename,
            True
        )

    def graph_gustafson_speed_up(self, filename):
        df = pd.DataFrame(data=self.data)
        columns = [
            'matrix_dimension',
            'parallel_time',
            'serial_time'
        ]
        df = df.loc[:, columns]
        df['speed_up'] = [0] * len(df.index)
        df['speed_up'] = df.apply(
            lambda x: self.gustafson_speed_up(
                x['serial_time'],
                x['parallel_time'],
                self.data['number_of_threads'][0]
            ),
            axis=1
        )
        df = df.loc[:, ['matrix_dimension', 'speed_up']]
        self.save_df_data(
            df,
            ['speed_up'],
            'matrix_dimension',
            {'speed_up': 'b-'},
            filename,
            True
        )
```

Listing 12: generate_output_data

6.13. src/controller/map_reduce.py

```
1  import collections
2  import itertools
3  import multiprocessing as mp
4  from math import ceil
5  from src.controller.utils import chunks
6  from src.controller.statistics import Statistics
7
8
9  class MapReduce(object):
10
11     def __init__(self, map_func, reduce_func):
12         """
13         :param map_func: Function to map inputs to intermediate
       ↪ data. Takes as
14         argument one input value and returns a tuple with the key
       ↪ and a value
15         to be reduced.
16         :param reduce_func: Function to reduce partitioned version of
17         intermediate data to final output. Takes as argument a key
       ↪ as produced
18         by map_func and a sequence of the values associated with
       ↪ that key.
19         """
20         self.map_func = map_func
21         self.reduce_func = reduce_func
22         self.statistics = Statistics()
23
24     @staticmethod
25     def get_chunksize(inputs, num_workers):
26         chunksize = int(len(inputs) / num_workers)
27         if chunksize == 0:
28             return 1
29         return chunksize
30
31     def get_statistics(self):
32         return self.statistics
33
34     @staticmethod
35     def keys_repeated(map_responses):
36         map_responses = map_responses.copy()
37         map_responses =
       ↪ list(itertools.chain.from_iterable(map_responses))
38         keys = {}
39         for a_mapped_value in map_responses:
40             pos, values = a_mapped_value
41             if pos not in keys:
42                 keys[pos] = [False, values]
43             else:
44                 keys[pos][0] = True
45                 keys[pos][1] += values
```

```
46          keys = list(keys.items())
47          repeated = list(filter(lambda x: x[1][0], keys.copy()))
48          repeated = list(map(lambda x: (x[0], x[1][1]), repeated))
49          not_repeated = list(filter(lambda x: not x[1][0],
    ↪ keys.copy()))
50          not_repeated = list(map(lambda x: (x[0], x[1][1]),
    ↪ not_repeated))
51          return repeated, not_repeated
52
53      @staticmethod
54      def group_by_key(mapped_values):
55          """
56          Organize the mapped values by their key.
57          Returns an unsorted sequence of tuples with a key and a
    ↪ sequence of
58          values.
59          """
60          partitioned_data = collections.defaultdict(list)
61          for a_mapped_value in mapped_values:
62              key, value = a_mapped_value
63              partitioned_data[key].append(value)
64          return list(partitioned_data.items())
65
66      @staticmethod
67      def shuffle(map_responses, num_workers):
68          map_responses = list(filter(lambda x: len(x) != 0,
    ↪ map_responses))
69          map_responses =
    ↪ list(itertools.chain.from_iterable(map_responses))
70          map_responses.sort(key=lambda tup: tup[0])
71          map_responses = chunks(map_responses, num_workers)
72          map_responses = list(filter(lambda x: len(x) != 0,
    ↪ map_responses))
73          return map_responses
74
75      def group_by_key_mapped_values(self, map_responses, num_workers):
76          is_repeated = True
77          output = []
78          while is_repeated:
79              self.statistics.start('serial')
80              num_workers = ceil(num_workers/2)
81              map_responses = self.shuffle(map_responses, num_workers)
82              chunksize = self.get_chunksize(map_responses,
    ↪ num_workers)
83              self.statistics.stop('serial')
84              pool = mp.Pool(processes=num_workers)
85              self.statistics.start('parallel')
86              map_responses = pool.map(
87                  self.group_by_key,
88                  map_responses,
89                  chunksize=chunksize
90              )
```

```
91              self.statistics.stop('parallel')
92              pool.close()
93              self.statistics.start('serial')
94              repeated, not_repeated =
    ↪ self.keys_repeated(map_responses)
95              output += not_repeated
96              map_responses = repeated
97              is_repeated = len(repeated) != 0
98              self.statistics.stop('serial')
99          return output
100
101     def map(self, inputs, num_workers=None):
102         """
103         :param inputs: data to map-reduce
104         :param chunksize: The portion of the input data to hand to
    ↪ each worker.
105         This can be used to tune performance during the mapping
    ↪ phase.
106         :param num_workers: The number of workers to create.
107         :return: Process the inputs through the map and reduce
    ↪ functions given.
108         """
109
110     def reduce(self, partitioned_data, num_workers=1):
111         """
112         :param partitioned_data:
113         :param num_workers: The number of workers to create.
114         :return:
115         """
```

Listing 13: map_reduce

6.14. src/controller/pool.py

```
1  import time
2  import multiprocessing as mp
3  from src.controller.map_reduce import MapReduce
4
5
6  class Pool(MapReduce):
7
8      def __init__(self, map_func, reduce_fun):
9          super().__init__(map_func=map_func, reduce_func=reduce_fun)
10         self.sleep_sec = 0.5
11
12     def map(self, inputs, num_workers=1):
13         num_cpu = mp.cpu_count()
14         if num_workers > num_cpu:
15             num_workers = num_cpu
16         chunksize = self.get_chunksize(inputs, num_workers)
17         pool = mp.Pool(processes=num_workers)
18         self.statistics.start('parallel')
19         map_responses = pool.map(
20             self.map_func,
21             inputs,
22             chunksize=chunksize
23         )
24         self.statistics.stop('parallel')
25         data = self.group_by_key_mapped_values(map_responses,
       ↪ num_workers)
26         pool.close()
27         time.sleep(self.sleep_sec)
28         return data
29
30     def reduce(self, partitioned_data, num_workers=1):
31         pool = mp.Pool(processes=num_workers)
32         self.statistics.start('serial')
33         reduced_values = pool.map(self.reduce_func, partitioned_data)
34         self.statistics.stop('serial')
35         pool.close()
36         time.sleep(self.sleep_sec)
37         return reduced_values
```

Listing 14: pool

6.15. src/controller/process.py

```
1  import itertools
2  import multiprocessing as mp
3  from src.controller.map_reduce import MapReduce
4  from src.controller.utils import chunks
5  from src.controller.my_process import MyProcess
6
7
8  class Process(MapReduce):
9
10     def __init__(self, map_func, reduce_fun):
11         self.processes = []
12         super().__init__(map_func=map_func, reduce_func=reduce_fun)
13
14     def map(self, inputs, num_workers=1):
15         num_cpu = mp.cpu_count()
16         if num_workers > num_cpu:
17             num_workers = num_cpu
18         splitted_data = chunks(inputs, num_workers)
19         for i in range(0, num_workers):
20             arg = splitted_data[i]
21             self.processes.append(MyProcess(target=self.map_func,
   ↪ args=arg))
22         map_responses = []
23         self.statistics.start('parallel')
24         for process in self.processes:
25             process.start()
26         for process in self.processes:
27             process.join()
28             map_responses += process.get_output()
29         self.statistics.stop('parallel')
30         map_responses = list(filter(lambda x: len(x) != 0,
   ↪ map_responses))
31         map_responses =
   ↪ list(itertools.chain.from_iterable(map_responses))
32         return self.group_by_key(map_responses)
33
34     def reduce(self, partitioned_data, num_workers=1):
35         output = []
36         self.statistics.start('serial')
37         for item in partitioned_data:
38             output.append(self.reduce_func(item))
39         self.statistics.stop('serial')
40         return output
```

Listing 15: proccess

6.16. src/controller/my_process.py

```
1  from multiprocessing import Process, Queue
2
3
4  class MyProcess(Process):
5
6      def __init__(self, target, args):
7          self.target = target
8          self.args = args
9          self.output = Queue()
10         super().__init__(target=target, args=args)
11
12     def run(self):
13         for an_arg in self.args:
14             self.output.put(self.target(an_arg))
15
16     def get_output(self):
17         output_list = []
18         while self.output.qsize() != 0:
19             output_list.append(self.output.get())
20         return output_list
```

Listing 16: my_proccess

6.17. src/controller/statistics.py

```
1 from time import time
2
3
4 class Statistics:
5     def __init__(self):
6         self.timers = {
7             'serial': float(0),
8             'parallel': float(0),
9             'global': float(0),
10        }
11        self.time_elapsed = {
12            'serial': float(0),
13            'parallel': float(0),
14            'global': float(0),
15        }
16
17    def start(self, key):
18        self.timers[key] = time()
19
20    def stop(self, key):
21        stop_time = time()
22        self.time_elapsed[key] += (stop_time - self.timers[key])*1000
23
24    def get_time_elapsed(self, key):
25        return self.time_elapsed[key]
```

Listing 17: statistics

6.18. src/controller/utils.py

```
1  import numpy as np
2  import os
3  from math import ceil
4
5
6  def column(matrix, i):
7      return [row[i] for row in matrix]
8
9
10 def get_random_matrix_of_dim_n(N):
11     random_matrix = np.random.randint(low=1, high=255, size=(N, N))
12     for i in range(0, N):
13         random_matrix[i] = random_matrix[i].tolist()
14     return random_matrix.tolist()
15
16
17 def get_null_matrix_of_dim_n(N):
18     random_matrix = np.zeros((N, N))
19     for i in range(0, N):
20         random_matrix[i] = random_matrix[i].tolist()
21     return random_matrix.tolist()
22
23
24 def get_null_list_of_dim_n(N):
25     return np.zeros(N).tolist()
26
27
28 def get_partitions(matrix, row_p, col_p):
29     N = len(matrix)
30     col_size_p = ceil(N/col_p)
31     row_size_p = ceil(N/row_p)
32     blocks = array_to_list(np.zeros((row_p, col_p)))
33     for r in range(0, row_p):
34         for c in range(0, col_p):
35             left_side = c * col_size_p
36             right_side = left_side + col_size_p
37             up_side = r * row_size_p
38             down_side = up_side + row_size_p
39             rows = matrix[up_side:down_side]
40             block = []
41             for row in rows:
42                 block.append(row[left_side:right_side])
43             blocks[r][c] = block.copy()
44     return blocks
45
46
47 def multiply_two_matrices(matrix_a, matrix_b):
48     rows_a = len(matrix_a)
49     cols_b = len(matrix_b[0])
50     cols_a = len(matrix_a[0])
```

```
51      multiplication = array_to_list(np.zeros((rows_a, cols_b)))
52      for i in range(0, rows_a):
53          for j in range(0, cols_b):
54              partial_sum = 0
55              for k in range(0, cols_a):
56                  partial_sum += matrix_a[i][k] * matrix_b[k][j]
57              multiplication[i][j] = partial_sum
58      return multiplication
59
60
61  def sum_matrices(matrices):
62      rows = len(matrices[0])
63      cols = len(matrices[0][0])
64      result = array_to_list(np.zeros((rows, cols)))
65      for i in range(0, rows):
66          for j in range(0, cols):
67              for matrix in matrices:
68                  result[i][j] += matrix[i][j]
69      return result
70
71
72  def array_to_list(array):
73      rows = len(array)
74      for i in range(0, rows):
75          array[i] = array[i].tolist()
76      return array.tolist()
77
78
79  def print_matrix(matrix):
80      rows = len(matrix)
81      for i in range(0, rows):
82          print(f"{matrix[i]}\n")
83
84
85  def chunks(a_list, num):
86      """
87      :param a_list: a list to split in n chunks
88      :param num: number of chunks
89      :return: list splitted
90      """
91      avg = len(a_list) / float(num)
92      out = []
93      last = 0.0
94      while last < len(a_list):
95          out.append(a_list[int(last):int(last + avg)])
96          last += avg
97      return out
98
99
100 def get_version_number():
101     folders = 0
102     for _, dir_names, _ in os.walk('docs/report/pics/'):
```

```
103             folders += len(dir_names)
104         return folders
```

Listing 18: utils

6.19. src/model/multiply_matrices_interface.py

```
1  class MultiplyMatricesInterface:
2
3      @staticmethod
4      def pre_processing(matrix_a, matrix_b, **kwargs):
5          raise NotImplementedError
6
7      @staticmethod
8      def map_worker(chunk):
9          raise NotImplementedError
10
11     @staticmethod
12     def reduce_worker(item):
13         raise NotImplementedError
```

Listing 19: multiply_matrices_interface

6.20. src/model/element_by_row_block.py

```
1  from src.model.multiply_matrices_interface import
       ↪ MultiplyMatricesInterface
2
3
4  class ElementByRowBlock(MultiplyMatricesInterface):
5
6      @staticmethod
7      def pre_processing(matrix_a, matrix_b, **kwargs):
8          row_size = len(matrix_a)
9          col_size = len(matrix_a[0])
10         output = []
11         for i in range(0, row_size):
12             for j in range(0, col_size):
13                 element_by_row_block = [matrix_a[i][j]] + matrix_b[j]
14                 output.append((i, element_by_row_block))
15         return output
16
17     @staticmethod
18     def map_worker(chunk):
19         output = []
20         i, elements = chunk
21         elem_a = elements[0]
22         elements.pop(0)
23         col_size = len(elements)
24         for j in range(0, col_size):
25             output.append(((i, j), elem_a * elements[j]))
26         return output
27
28     @staticmethod
29     def reduce_worker(item):
30         output_pos, values = item
31         result = 0
32         for a_value in values:
33             result += a_value
34         return output_pos, result
```

Listing 20: element_by_row_block

6.21. src/model/column_by_row.py

```python
from src.model.multiply_matrices_interface import MultiplyMatricesInterface


class ColumnByRow(MultiplyMatricesInterface):

    @staticmethod
    def pre_processing(matrix_a, matrix_b, **kwargs):
        N = len(matrix_a)
        output = []
        for i in range(0, N):
            col_a = [row[i] for row in matrix_a]
            output.append((col_a, matrix_b[i]))
        return output

    @staticmethod
    def map_worker(chunk):
        col_a, row_b = chunk
        output = []
        for row, elem_a in enumerate(col_a):
            for col, elem_b in enumerate(row_b):
                key = (row, col)
                value = elem_a * elem_b
                output.append((key, value))
        return output

    @staticmethod
    def reduce_worker(item):
        output_pos, values = item
        result = 0
        for a_value in values:
            result += a_value
        return output_pos, result
```

Listing 21: column_by_row

6.22. src/model/by_blocks.py

```python
import numpy as np
from src.model.multiply_matrices_interface import
    ↪ MultiplyMatricesInterface
from src.controller.utils import get_partitions, sum_matrices


class ByBlocks(MultiplyMatricesInterface):

    @staticmethod
    def pre_processing(matrix_a, matrix_b, **kwargs):
        output = []
        row_p = kwargs.get('row_p', 2)
        col_p = kwargs.get('col_p', 2)
        blocks_a = get_partitions(matrix_a, row_p, col_p)
        blocks_b = get_partitions(matrix_b, row_p, col_p)
        for r_a in range(0, row_p):
            for c_a in range(0, col_p):
                a_block = blocks_a[r_a][c_a]
                output.append((r_a, a_block, blocks_b[c_a]))
        return output

    @staticmethod
    def map_worker(chunk):
        r_a, block_a, blocks_b = chunk
        output = []
        col_size = len(blocks_b)
        for c_b in range(0, col_size):
            result = np.matmul(block_a, blocks_b[c_b]).tolist()
            key = (r_a, c_b)
            output.append((key, result))
        return output

    @staticmethod
    def reduce_worker(item):
        output_pos, values = item
        result = sum_matrices(values)
        output = []
        row_size = len(result)
        block_pos_i, block_pos_j = output_pos
        for i in range(0, row_size):
            col_size = len(result[i])
            for j in range(0, col_size):
                pos = (block_pos_i*row_size+i,
    ↪ block_pos_j*col_size+j)
                output.append((pos, result[i][j]))
        return output
```

Listing 22: by_blocks