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.
En Amdahl se hará una multiplicación de matrices (ambas de `NxN`) y se realizarán las mediciones de tiempo, variando la cantidad de threads involucrados en el procesamiento.
Luego, en Gustafson se realizará las mismas mediciones aumentando la dimensión de las matrices (aumentando el trabajo) y aumentando la cantidad de threads cada vez que aumentamos el trabajo.
Finalmente, se hará una multiplicación de dos matrices diferentes de `NxN` usando la librería CBLAS e instrucciones de vectoriales (MMX) para el compilador con sólo un procesador. De esta manera, la idea es comparar el tiempo que tarda el map-reduce en serie frente a CBLAS y la vectorización.

2. Desarrollo teórico

2.1. Speed up

Es la mejora en la velocidad de ejecución de una tarea ejecutada en dos arquitecturas similares con diferentes recursos.
El speed-up 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` y 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 speed-up 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 ecuación anterior y calculamos el límite 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 aprovecha la paralelizacion para calcular mas cosas.

Entonces siendo `s` el tiempo de la ejecución de la sección serie, siendo `p` el tiempo de la ejecución de la sección paralela y siendo `N` la cantidad de procesadores podemos calcular el speed-up como:

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

Definiendo:

$$\boldsymbol{\alpha} = \frac{s}{s + p} \tag{6}$$

Podemos decir que:

$$\textbf{speed\_up} = N - \alpha * (N - 1) \tag{7}$$

2.4. Map-reduce

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

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

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

Por otro lado, `Reduce` recibe un conjunto de tuplas `(clave, valores)`, y aplica una función a todos estos valores, para poder retornar un único valor y así devolver un conjunto de tuplas `(clave, valor)`

La principal ventaja de MapReduce es su facilidad 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

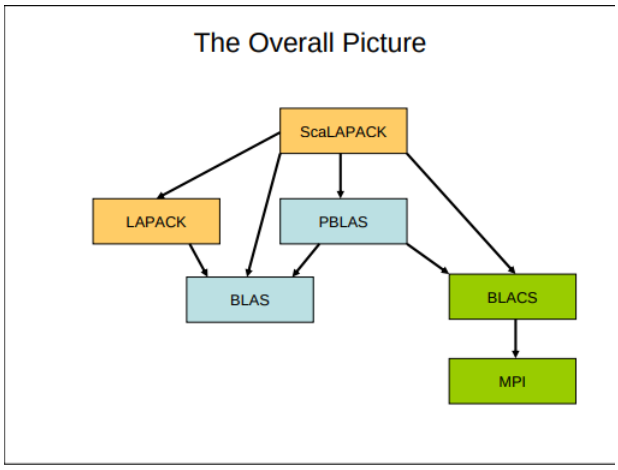

Figura 1: overall picture

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 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.

2.6. Instrucciones vectoriales MMX

MMX es un Conjunto de instrucciones SIMD diseñado por Intel e introducido en 1997 en sus microprocesadores Pentium MMX. Fue desarrollado a partir de un set introducido en el Intel i860. Ha sido soportado por la mayoría de fabricantes de microprocesadores x86 desde entonces.

Fue presentado como un acrónimo de MultiMedia eXtension o Multiple Math o Matrix Math eXtension, pero oficialmente sólo es un juego de consonantes sin significado, usado con la única intención de poder poner cortapisas legales de marca registrada a los desarrollos de terceros que trataran de usarlo.

3. Implementación

3.1. Explicación del modelo

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

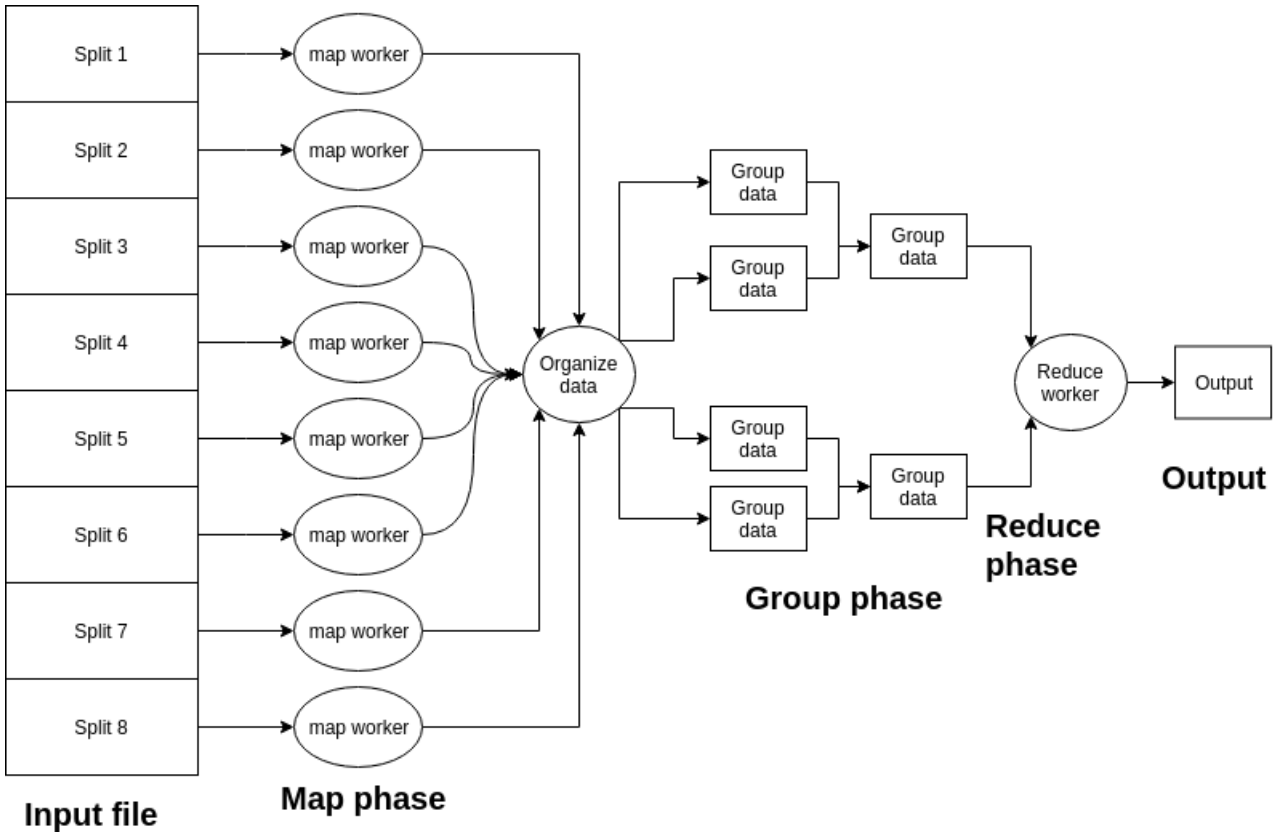

Figura 2: Esquema de un map reduce

En nuestro caso creamos una clase llamada `MapReduce` la cual usa una librería de `python` llamada `multiprocessing` en donde usamos el módulo `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 un `pool` para hacer el map de manera que se le pasa como atributo la cantidad de workers en el cual se quiere paralelizar el problema.

3.2. Multiplicación 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 posición `(r, c)` de un bloque de la matriz `A`, tiene el bloque en custión `a_block_rc`, y la fila número `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 posición `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 multiplicación. Por cada multiplicación, agregamos una de estas tuplas al vector de salida para luego devolver éste.

3.2.0.3 Reducción

Recibimos la posición 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 de la multiplicación. Pero por cada valor se calcula la posición de salida del mismo en la matriz resultante y nos deshacemos de la posición de los bloques

3.3. Multiplicación 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 número 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 función map, obtenemos partes de esta lista de tuplas y devolvemos un par clave, valor donde la clave es la posición de salida de la matriz

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

3.3.0.3 Reducción

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

3.4. Multiplicación 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 multiplicación. Por cada multiplicación guardamos en un vector una tupla con un par clave valor donde la clave es la posición de salida de la matriz resultante y el valor es la multiplicación anteriormente mencionada. Finalmente se devuelve el vector de tuplas.

3.4.0.3 Reducción

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

3.5. MMX

Se escribió un código en `c`, donde se multiplican dos matrices por bloques. Al momento de hacer la multiplicación parcial de elementos, donde cada uno es un bloque de la matriz, se usa una intrucción llamada `#pragma vector always` para decirle al compilador que la siguiente fracción de código será vectorizada.
Este código fue dessarrollado en el archivo `blocked_dgemm_sse.c` que se encuentra en el anexo.

3.6. Cblas

Se uso una función de la librería `cblas.h` de `c`. Esta función cuya firma es:

```
1   void cblas_dgemm(
2           CBLAS_LAYOUT layout,
3           CBLAS_TRANSPOSE TransA,
4           CBLAS_TRANSPOSE TransB,
5           const int M, const int N,
6           const int K,
7           const double alpha,
8           const double *A,
9           const int lda,
10          const double *B,
11          const int ldb,
12          const double beta,
13          double *C,
14          const int ldc);
15
```

Ésta llamada a la rutina `cblas_dgemm` multiplica dos matrices:

```
1       cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n,
    ↪ k, alpha, A, k, B, n, beta, C, n);
2
```

Los argumentos proporcionan opciones sobre cómo Intel MKL realiza la operación. En este caso:

- **CblasRowMajor:** Indica que las matrices se almacenan en el orden mayor de la fila, con los elementos de cada fila de la matriz almacenados de forma contigua.
- **CblasNoTrans:** Tipo de enumeración que indica que las matrices A y B no deben ser transpuestas o conjugadas antes de la multiplicación.
- **m, n, k:** Enteros que indican el tamaño de las matrices:
  - **A:** m filas por k columnas
  - **B:** k filas por n columnas
  - **C:** m filas por n columnas
- **alpha:** Valor real utilizado para escalar el producto de las matrices A y B.
- **A:** Arreglo utilizado para almacenar la matriz A.

- **k:** Dimensión inicial de la matriz A, o el número de elementos entre filas sucesivas (para el almacenamiento principal de fila) en la memoria. En el caso de este ejercicio, la dimensión principal es la misma que la cantidad de columnas.
- **B:** Arreglo utilizado para almacenar la matriz B.
- **n:** Dimensión inicial de la matriz B, o el número de elementos entre filas sucesivas (para el almacenamiento principal de fila) en la memoria. En el caso de este ejercicio, la dimensión principal es la misma que la cantidad de columnas.
- **beta:** Valor real utilizado para escalar la matriz C.
- **C:** Arreglo utilizado para almacenar la matriz C.
- **n:** Dimensión inicial de la matriz C, o el número de elementos entre filas sucesivas (para el almacenamiento principal de fila) en la memoria. En el caso de este ejercicio, la dimensión principal es la misma que la cantidad de columnas.

3.7. Forma de ejecución

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 se multiplican dos matrices de `100x100` con 1 thread, dos matrices de `126x126` con 2 threads y dos matrices de `158x158` con 4 threads.

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 carátula) y abrir una terminal en el `root` del mismo.

Para compilar `cblas`, las instrucciones vectoriales, y las pruebas del módulo `pool` que están en lenguaje c se debe ejecutar:
`$ make.`

Para realizar el cálculo de `cblas` y las instrucciones vectoriales que están en lenguaje c se debe ejecutar:
`$ make run_code.`

Para realizar las pruebas del módulo `pool` frente a `c` y `python` se debe ejecutar:
`$ make run_test_pool.`

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

Luego para generar los gráficos 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`

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

3.8. 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 núcleos 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

Para calcular el speed-up real de Amdahl se utilizó la ecuación `(2)`, para el speed-up teórico de Amdahl se utilizó la ecuación `(3)` y para el speed-up real Gustafson se utilizó la ecuación `(2)` y para el speed-up escalable de tiempo fijo se utilizó la ecuación `(7)`

4.1. Multiplicación por bloques

4.1.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	22587.220430	1151.420116	200
1	2	13474.523783	1010.428429	200
2	4	14435.565948	1187.568188	200
3	8	13161.564112	1047.145844	200
4	16	13951.312542	1120.860338	200
5	32	13464.200258	1077.635288	200
6	64	14718.558550	1027.906656	200
7	128	14212.234497	1077.681541	200

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

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

	number_of_threads	theoretical_speed_up	real_speed_up	max_speed_up
0	1	1.000000	1.000000	20.616837
1	2	1.907480	1.638848	20.616837
2	4	3.491888	1.519454	20.616837
3	8	5.972251	1.670710	20.616837
4	16	9.261614	1.574998	20.616837
5	32	12.781465	1.632438	20.616837
6	64	15.780046	1.507554	20.616837
7	128	17.877060	1.552568	20.616837

Figura 4: Speed-up real, teórico y máximo según la cantidad de threads

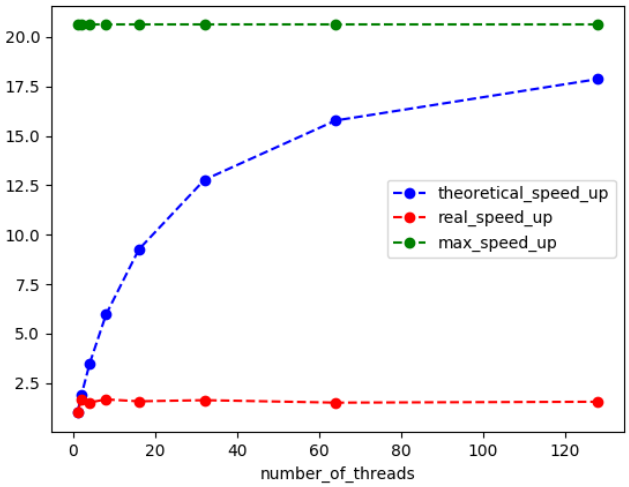

Figura 5: Gráfico

Se puede observar que el speed up teórico tiende al máximo speed-up, mientras que el real nos muestra que no usa toda la paralelizacion, ya que al pasar de 1 a 2 threads el tiempo no cae a la mitad, y al pasar de 1 a 4 threads no cae a la cuarta parte. Se puede ver que tiene un speed-up de casi 2 lo cual quiere decir que hace uso de la mitad de la paralelización.

4.1.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	2195.271015	244.652510	100
1	2	3420.389414	364.738464	126
2	4	6878.543139	583.956242	158

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

Se puede 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. Pero además se puede observar que hay mucha ineficiencia respecto del uso de la paralelización, ya que al aumentar el trabajo en casi el doble y usar dos procesadores debería tardar aproximadamente lo mismo. Sin embargo, vemos que el tiempo paralelo se duplicó, lo cual muestra que hay un problema de comunicación al aumentar el paralelismo. Más adelante se explicará que el módulo Pool es ineficiente respecto del uso de recursos.

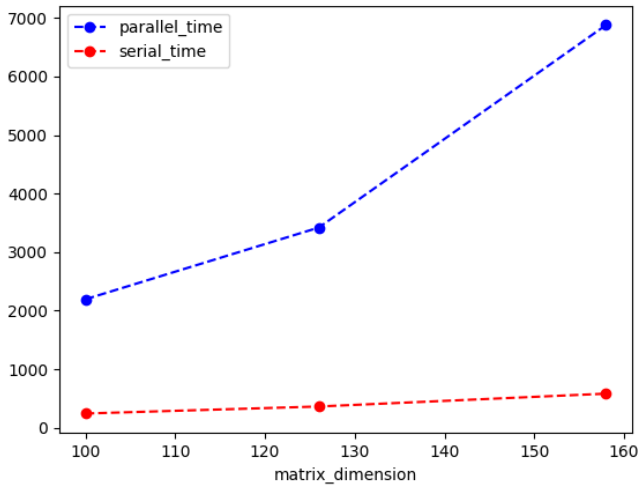

Figura 7: Tiempo paralelo y serie en función de la dimensión de las matrices de entrada

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

	alpha	fixed_time_speed_up
0	0.100271	3.699188
1	0.096361	3.710917
2	0.078252	3.765244

Figura 8: Tabla de valores de la fracción de la parte secuencial y del speed-up escalable

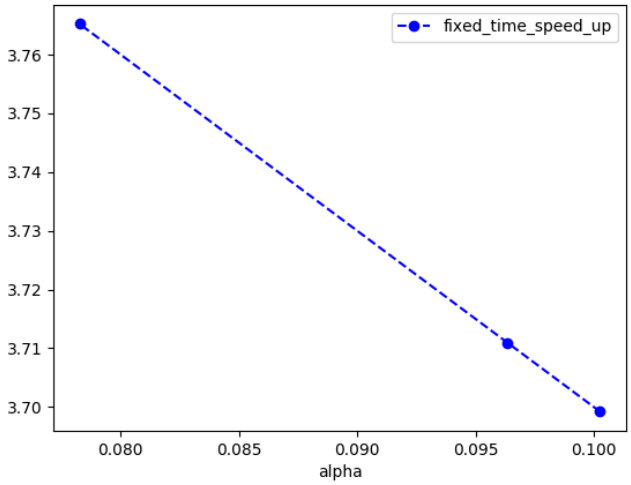

Figura 9: Gráfico del speed-up en función de la parte secuencial. Se utilizó la ecuación 7

	number_of_threads	real_speed_up
0	1	1.000000
1	2	0.644608
2	4	0.326958

Figura 10: Tabla de valores del speed-up

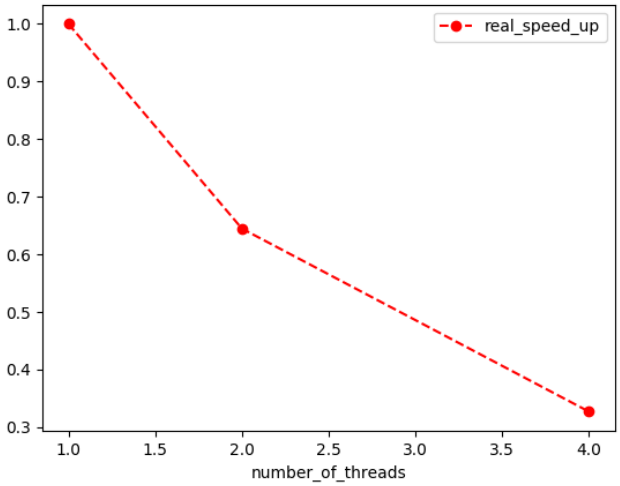

Figura 11: Gráfico del speed-up. Se utilizó la ecuación 2

4.2. Multiplicación elemento por fila

4.2.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	20479.001999	535.544634	200
1	2	15959.017277	547.099113	200
2	4	13265.343189	553.031445	200
3	8	13883.560896	570.143938	200
4	16	16056.910992	654.646397	200
5	32	15586.015940	540.157318	200
6	64	13835.798740	603.547573	200
7	128	15849.511385	571.648836	200

Figura 12: Salida de los tiempos en serie y paralelo en milisegundos

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

	number_of_threads	theoretical_speed_up	real_speed_up	max_speed_up
0	1	1.000000	1.000000	39.23958
1	2	1.950298	1.273137	39.23958
2	4	3.715906	1.520768	39.23958
3	8	6.788916	1.453921	39.23958
4	16	11.575187	1.257486	39.23958
5	32	17.876909	1.303133	39.23958
6	64	24.563218	1.455367	39.23958
7	128	30.213420	1.279724	39.23958

Figura 13: Speed-up real, teórico y máximo según la cantidad de threads

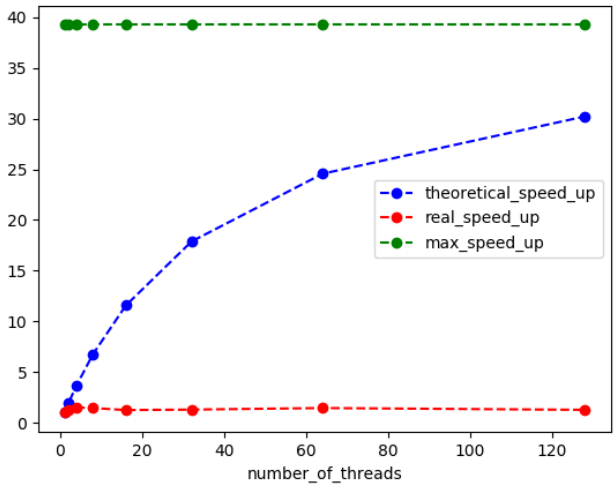

Figura 14: Gráfico

Se puede observar que el speed up teórico tiende al máximo speed-up, mientras que el real nos muestra que no usa toda la paralelizacion, ya que al pasar de 1 a 2 threads el tiempo no cae a la mitad, y al pasar de 1 a 4 threads no cae a la cuarta parte. Se puede ver que tiene un speed-up de casi 2 lo cual quiere decir que hace uso de la mitad de la paralelización.

4.2.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	2643.973112	117.126942	100
1	2	3899.180651	198.422194	126
2	4	9325.890303	308.493376	158

Figura 15: Salida de los tiempos en serie y paralelo en milisegundos

Se puede 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. Pero además se puede observar que hay mucha ineficiencia respecto del uso de la paralelización, ya que al aumentar el trabajo en casi el doble y usar dos procesadores debería tardar aproximadamente lo mismo. Sin embargo, vemos que el tiempo paralelo se duplicó, lo cual muestra que hay un problema de comunicación al aumentar el paralelismo. Más adelante se explicará que el módulo Pool es ineficiente respecto del uso de recursos.

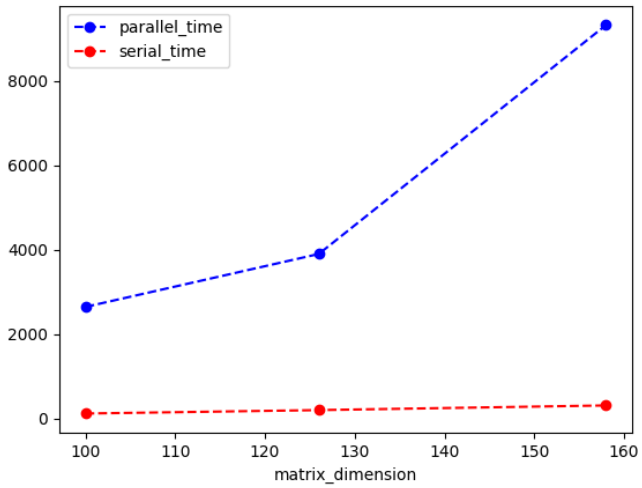

Figura 16: Tiempo paralelo y serie en función de la dimensión de las matrices de entrada

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

	alpha	fixed_time_speed_up
0	0.042420	3.872739
1	0.048424	3.854728
2	0.032020	3.903940

Figura 17: Tabla de valores de la fracción de la parte secuencial y del speed-up escalable

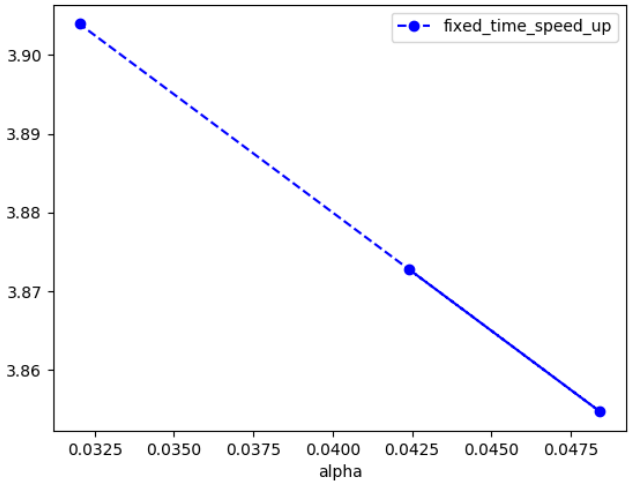

Figura 18: Gráfico del speed-up en función de la parte secuencial. Se utilizó la ecuación 7

	number_of_threads	real_speed_up
0	1	1.000000
1	2	0.673833
2	4	0.286588

Figura 19: Tabla de valores del speed-up

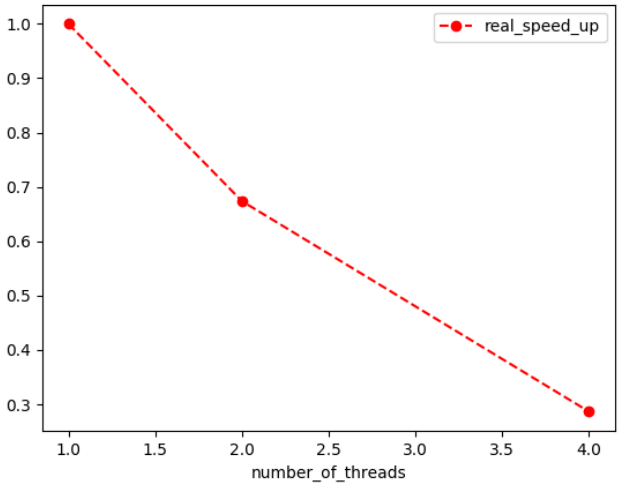

Figura 20: Gráfico del speed-up. Se utilizó la ecuación 2

4.3. Multiplicación columna por fila

4.3.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	21153.194666	772.554159	200
1	2	17277.929544	578.266859	200
2	4	13838.816881	783.934832	200
3	8	13627.315760	614.409447	200
4	16	13333.874702	575.849295	200
5	32	12298.032761	825.387955	200
6	64	12657.370806	563.635111	200
7	128	15142.727137	835.159779	200

Figura 21: Salida de los tiempos en serie y paralelo en milisegundos

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

	number_of_threads	theoretical_speed_up	real_speed_up	max_speed_up
0	1	1.000000	1.000000	28.380857
1	2	1.931928	1.227907	28.380857
2	4	3.617601	1.499427	28.380857
3	8	6.417223	1.539543	28.380857
4	16	10.467606	1.576289	28.380857
5	32	15.294279	1.670734	28.380857
6	64	19.876973	1.658402	28.380857
7	128	23.379647	1.372256	28.380857

Figura 22: Speed-up real, teórico y máximo según la cantidad de threads

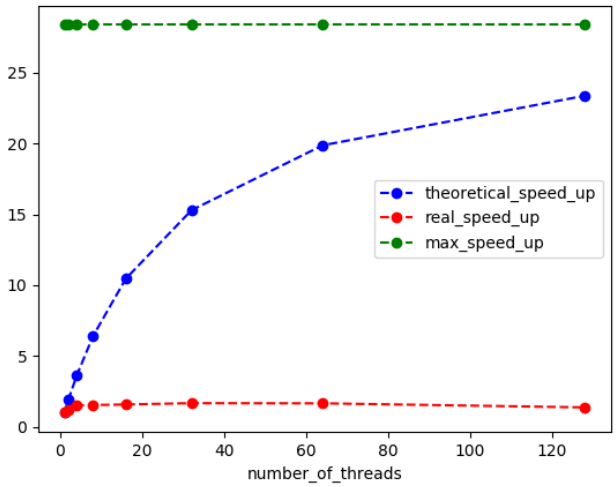

Figura 23: Gráfico

Se puede observar que el speed up teórico tiende al máximo speed-up, mientras que el real nos muestra que no usa toda la paralelizacion, ya que al pasar de 1 a 2 threads el tiempo no cae a la mitad, y al pasar de 1 a 4 threads no cae a la cuarta parte. Se puede ver que tiene un speed-up de casi 2 lo cual quiere decir que hace uso de la mitad de la paralelización.

4.3.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	2627.908230	77.727079	100
1	2	3932.931662	195.525408	126
2	4	6798.559427	337.243795	158

Figura 24: Salida de los tiempos en serie y paralelo en milisegundos

Se puede 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. Pero además se puede observar que hay mucha ineficiencia respecto del uso de la paralelización, ya que al aumentar el trabajo en casi el doble y usar dos procesadores debería tardar aproximadamente lo mismo. Sin embargo, vemos que el tiempo paralelo se duplicó, lo cual muestra que hay un problema de comunicación al aumentar el paralelismo. Más adelante se explicará que el módulo Pool es ineficiente respecto del uso de recursos.

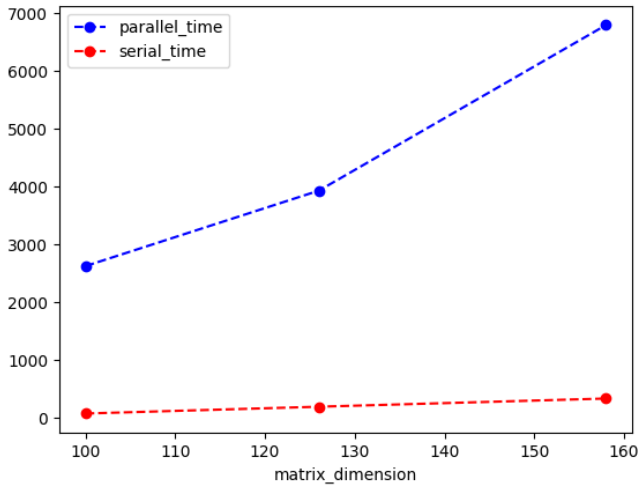

Figura 25: Tiempo paralelo y serie en función de la dimensión de las matrices de entrada

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

	alpha	fixed_time_speed_up
0	0.028728	3.913816
1	0.047360	3.857919
2	0.047261	3.858218

Figura 26: Tabla de valores de la fracción de la parte secuencial y del speed-up escalable

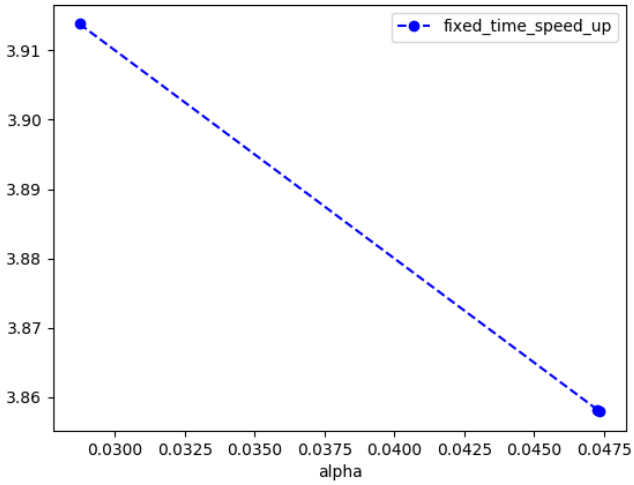

Figura 27: Gráfico del speed-up en función de la parte secuencial. Se utilizó la ecuación 7

	number_of_threads	real_speed_up
0	1	1.000000
1	2	0.655362
2	4	0.379163

Figura 28: Tabla de valores del speed-up

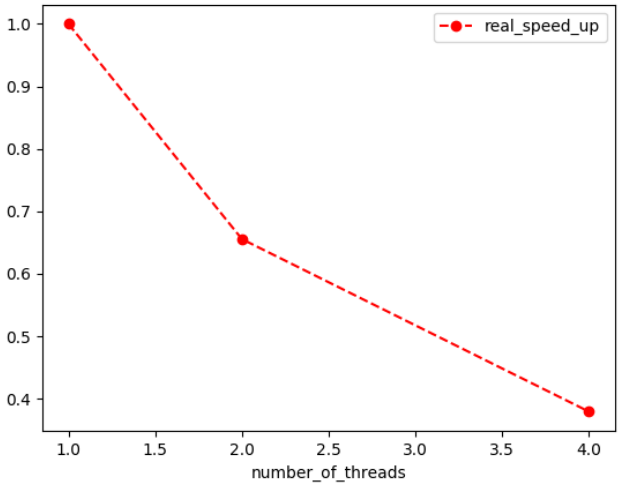

Figura 29: Gráfico del speed-up. Se utilizó la ecuación 2

4.4. Cblas e instrucciones vectorizadas

	program	time_elapsed	matrix_dim	number_of_threads
0	cblas_dgemm	0.053858	400	1
1	vectorized blocked_dgemm_sse	0.010935	400	1
2	not vectorized blocked_dgemm_sse	0.366132	400	1

Figura 30: Tiempo serie de multiplicación en segundos

Se puede ver que las instrucciones vectoriales, en el código `blocked_dgemm_sse`, tienen una mejora considerable respecto del mismo código sin vectorizar y de `cblas`. Por otro lado se puede ver que `cblas` tiene una considerable mejor performance que el código `blocked_dgemm_sse` sin vectorizar. De esta manera se puede ver que `cblas` y las instrucciones vectorizadas aprovechan mucho mejor el hardware para la realización de la misma operación. También es cierto que el map-reduce hace uso de un módulo de `python` llamado `pool` que suma latencia al momento de dividir el trabajo, ya que sufre mucho el costo de la comunicación entre procesadores. En el análisis de los resultados se explica con detalle el problema del módulo `pool` y se muestra una prueba de la diferencia considerable de performance entre `c`, `python` y `python usando pool`.

5. Análisis de resultados

5.1. Pool

El módulo de `multiprocessing` puede usar múltiples procesos, pero aún tiene que trabajar con el bloqueo global del intérprete de Python, lo que significa que no puede compartir memoria entre sus procesos. Por lo tanto, cuando intenta iniciar un Pool, necesita copiar variables útiles, procesar su cálculo y recuperar el resultado. Esto le cuesta un poco de tiempo para cada proceso y lo hace menos efectivo. Pero esto sucede porque se hace un cálculo muy pequeño: el `multiprocessing` solo es útil para cálculos más grandes, cuando la copia de la memoria y la recuperación de resultados es más barata (en el tiempo) que el cálculo.

Un cálculo costoso es más eficiente con el `multiprocessing`, incluso si no siempre se tiene lo que podría esperar (podría tener una aceleración x4, pero solo se obtuvo x2). Hay que tener en cuenta que `Pool` tiene que duplicar cada bit de memoria utilizada en el cálculo, por lo que puede ser costoso.
También hay que saber que `numpy` tiene una gran cantidad de funciones caras escritas en C / Fortran y que ya están en paralelo, por lo que no se puede hacer mucho para acelerarlas.

5.2. Prueba de latecia de pool, python y c

A continuación se hizo un calculo el cual requiere iterar unas 10000000 veces y hacer una acumulacion para una suma. Por lo tanto se hizo la misma operación en c, en python y en python pero usando el modulo `pool` donde dividimos el trabajo en 4 procesos.

	program	time_elapsed (sec)	iterations	number_of_threads
0	c program	0.028173	10000000	1
1	python	0.9374971389770508	10000000.0	1
2	python using pool	2.0666675567626953	10000000.0	4

Figura 31: tiempos de la pruebas en segundos

Podemos ver que `c` es mucho mas eficiente siendo 33 veces mas rápido que `python`. También podemos ver que usando el módulo `pool`, dividiendo el trabajo en 4 cores tarda más que el mismo trabajo sin usar `pool` en python, lo cual demuestra que este módulo es infeficiente como se explicó anteriormente debido al uso de recursos.

5.3. Prueba de multiplicación de matrices con C++

A modo de entender lo pagado en eficiencia al utilizar el lenguaje Python, procedimos a realizar la multiplicación de matrices por bloques en C++, un lenguaje muy similar a C, el cual se suele utilizar muy frecuentemente a la hora de buscar eficiencia. Los resultados obtenidos para la multiplicación de matrices por bloques, para el caso de matrices de 400x400 fueron de **0.178759 seg**. Como era de esperarse, podemos comprobar, que el uso de un lenguaje de alto nivel como Python, resultó muy costoso a la hora de compararlo con lenguajes como C o C++ para el cálculo de eficiencia. Es por eso, que debemos entender que los resultados obtenidos en este trabajo estan muy ligados al lenguaje utilizado y por eso, los resultados obtenidos en C y C++ resultaron de mucho mas eficientes.

6. Conclusiones

Se puede decir que se obtuvieron resultados inesperados pero se tuvieron que hacer varias corridas y ajustar ciertos números para entender por qué llegamos a éstos.
Si bien, al paralelizar, hubo casos donde se obtuvo una mejora, no era la mejora esperada. Es decir, cuando se usaron cuatro threads se supone que el tiempo paralelo caiga a la cuarta parte, y sin embargo se pudo observar que cayó a la mitad. Entonces, se hicieron pruebas simples comparando la performance de un código `c`, un código `python` y `python usando pool`. Se pudo observar que `c` es mucho mas rápido que `python`, y que el módulo `pool` es muy inefieciente como se explicó anteriormente, respecto del uso de recursos. Se puede obtener una mejora con el módulo `pool` si aumentamos considerablemente el trabajo pero, como se pudo observar en los resultados de Amdahl, no será una mejora esperada. También, respecto del módulo `pool`, al aumentar el trabajo para poder observar una mejora se tiene la desventaja que la PC se cuelga debido a que `pool` está haciendo copias útiles de variables (como se explicó anteirormente en la sección `análisis de resultados`), y ésto hace que el proceso sea muy lento y la performance baje considerablemente.

Finalmente, se puede decir que hay que tener en cuenta que hay otros programas corriendo en las cuatro CPU, y que dependiendo del tamaño de información 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 tráfico de información mediante el comando `gnome-system-monitor`, donde filtrando los procesos y solo viendo los de `python` se pudo ver el uso de cada CPU y gráficos al respecto.

7. Anexo

7.1. src/app.py

```python
import time
from typing import Type
from math import ceil
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


SAVE = True


def gustafson(model: Type[MultiplyMatricesInterface]):
    name = model.__name__
    print(f"----------RUNNING GUSTAFSON----------")
    output_data = OutputData()
    for values in [(1, 100), (2, 126), (4, 158)]:
        num_workers, matrix_dim = values
        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_fixed_time_speed_up.png')
        output_data.graph_gustafson_real_speed_up(
            name + '_gustafson_real_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()
    matrix_dim = 200
    for num_workers in [1, 2, 4, 8, 16, 32, 64, 128]:
        print(f"RUNNING WITH NUM WORKERS: {num_workers}")
        serial, parallel = run(num_workers, matrix_dim, model)
        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  OutputData.delete_json_data()
80  start = time.time()
81  run_model(ElementByRowBlock)
82  run_model(ColumnByRow)
83  run_model(ByBlocks)
84  end = time.time()
85  print(f"************************************")
86  print(f"The process lasted {end-start} seconds")
87  print(f"************************************")
```

Listing 1: app

7.2. src/graphs.py

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

Listing 2: graphs

7.3. src/cblas.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "cblas/cblas_dgemm.h"
#include "controller/utils.h"
#include "controller/file.h"

double run_cblas_dgemm(int N, double* A, double* B, double* C) {
    clock_t start, stop;
    double alpha = 1.0;
    double beta = 0.0;
    init_arr(N, N, 2, A);
    init_arr(N, N, 1, B);
    init_arr(N, N, 0, C);
    start = clock();
    mult(A, B, C, alpha, beta, N, N, N);
    stop = clock();
    double elapsed_seconds = ((double)(stop - start)) /
    ↪ CLOCKS_PER_SEC;
    printf("Elapsed time = %f seconds\n", elapsed_seconds);
    return elapsed_seconds;
}

int main() {
    char* attributes[4] = {
            "program",
            "time_elapsed",
            "matrix_dim",
            "number_of_threads"
    };
    file_t* file = create_file("src/data/dgemm.csv", "w+",
    ↪ attributes, 4);
    int N = 400;
    double A[N*N];
    double B[N*N];
    double C[N*N];
    double elapsed_seconds;
    char elapsed_second_str[20];
    char* values[4] = {"cblas_dgemm", elapsed_second_str, "400",
    ↪ "1"};
    elapsed_seconds = run_cblas_dgemm(N, A, B, C);
    double_to_string(elapsed_second_str, elapsed_seconds, 20);
    add_row(file, values, 4);
    delete(file);
    return 0;
}
```

Listing 3: main

7.4. src/mmx.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "controller/utils.h"
#include "vectorization/blocked_dgemm_sse.h"
#include "controller/file.h"

double run_blocked_dgemm_sse(int N, double* A, double* B, double* C) {
    clock_t start, stop;
    init_arr(N, N, 2, A);
    init_arr(N, N, 1, B);
    init_arr(N, N, 0, C);
    start = clock();
    square_dgemm_blocked_sse(A, B, C, N, 2);
    stop = clock();
    double elapsed_seconds = ((double)(stop - start)) / CLOCKS_PER_SEC;
    printf("Elapsed time = %lf seconds\n", elapsed_seconds);
    return elapsed_seconds;
}

int main(int argc, char **argv) {
    if (argc != 2) {
        return 1;
    }
    char* program_name = argv[1];
    char* attributes[4] = {
            "program",
            "time_elapsed",
            "matrix_dim",
            "number_of_threads"
    };
    file_t* file = create_file("src/data/dgemm.csv", "a", attributes, 4);
    int N = 400;
    double A[N*N];
    double B[N*N];
    double C[N*N];
    double elapsed_seconds;
    char elapsed_second_str[20];
    char* values[4] = {program_name, elapsed_second_str, "400", "1"};
    elapsed_seconds = run_blocked_dgemm_sse(N, A, B, C);
    double_to_string(elapsed_second_str, elapsed_seconds, 20);
    add_row(file, values, 4);
    delete(file);
    return 0;
}
```

Listing 4: main

7.5. src/test_pool.c

```c
#include <time.h>
#include <stdio.h>
#include "controller/file.h"
#include "controller/utils.h"

#define SIZE 10000000

int main() {
    clock_t start, stop;
    start = clock();
    unsigned long sum = 0;
    for (unsigned long i = 0; i < SIZE; i++) {
        sum += i;
    }
    stop = clock();
    char* attributes[4] = {
            "program",
            "time_elapsed (sec)",
            "iterations",
            "number_of_threads"
    };
    file_t* file = create_file("src/data/test_pool.csv", "w+",
    ↪ attributes, 4);
    double elapsed_seconds = ((double)(stop - start)) /
    ↪ CLOCKS_PER_SEC;
    printf("Elapsed time = %f seconds\n", elapsed_seconds);

    char elapsed_second_str[20];
    char size_str[20];
    double_to_string(elapsed_second_str, elapsed_seconds, 20);
    int_to_string(size_str, SIZE, 20);
    char* values[4] = {"c program", elapsed_second_str, size_str,
    ↪ "1"};
    add_row(file, values, 4);
    return 0;
}
```

Listing 5: main

7.6. src/fastTest/main.cpp

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <mutex>
5 #include <thread>
6 #include <chrono>
7
8 //NOTE: THIS CODE NEEDS C++11 COMPILER TO COMPILE.
9
10 using namespace std;
11 int const size = 400;
12 int const blocks = 4;
13
14 thread myThread[blocks];
15
16 void MatrixMultiplication(int const * const * matrixA, int const *
    ↪ const * matrixB, int** matrixC, int from, int block){
17       for(int row = from; row < block; row++){
18         for(int col = 0; col < size; col++){
19           for(int i=0; i < size ; i++){
20             matrixC[row][col] += matrixA[row][i]*matrixB[i][col];
21           }
22         }
23     }
24 }
25
26 int main()
27 {
28     int** matrixA;
29     int** matrixB;
30     int** matrixC;
31
32     matrixA = new int*[size];
33     matrixB = new int*[size];
34     matrixC = new int*[size];
35
36     for(int i = 0; i<size; i++){
37         matrixA[i] = new int[size];
38         matrixB[i] = new int[size];
39         matrixC[i] = new int[size];
40     }
41
42     //WE ARE INITIALIZING MATRICES.
43     for(int i = 0; i<size; i++){
44         for(int j = 0; j<size; j++){
45             matrixA[i][j] = rand() % 100;;
46             matrixB[i][j] = rand() % 100;;
47             matrixC[i][j] = 0;
48         }
49     }
```

```
50
51      auto start = std::chrono::high_resolution_clock::now();
52      for(int i=0; i < blocks ; i++){
53        int from = (size/blocks)*(i);
54        int to = (size/blocks)*(i+1);
55        myThread[i] = thread(MatrixMultiplication, matrixA, matrixB,
    ↪ matrixC, from, to);
56      }
57
58      for(int i=0; i < blocks ; i++){
59        myThread[i].join();
60      }
61      auto finish = std::chrono::high_resolution_clock::now();
62
63      std::chrono::duration<double> elapsed = finish - start;
64
65      std::cout << "Elapsed time: " << elapsed.count() << " s\n";
66
67      return 0;
68  }
```

Listing 6: main

7.7. src/test_pool.py

```
1 from multiprocessing import Pool
2 from time import time
3 from math import ceil
4 from src.controller.utils import chunks
5 from src.controller.generate_output_data import OutputData
6 import pandas as pd
7
8 SIZE = 10000000
9
10
11 def loop(s):
12     acum = 0
13     size = len(s)
14     for i in range(0, size):
15         acum += s[i]
16     return acum
17
18
19 a_list = list(range(SIZE))
20 pool = Pool(processes=4)
21 start = time()
22 pool_sum = pool.map(loop, chunks(a_list, 4), chunksize=ceil(SIZE/4))
23 pool.close()
24 pool.join()
25 end = time()
26 pool_elapsed_time = end - start
27 start = time()
28 serial_sum = loop(a_list)
29 end = time()
30 elapsed_time = end - start
31 ratio = pool_elapsed_time/elapsed_time
32 print(f"pool_elapsed_time: {pool_elapsed_time}")
33 print(f"elapsed_time: {elapsed_time}")
34 print(f"ratio: {ratio}")
35
36 test_pool_output_data = OutputData()
37 test_pool_df =
    ↪ test_pool_output_data.get_df_from_csv("src/data/test_pool.csv")
38 test_pool_df = test_pool_df.append(
39     pd.Series(
40         ["python", str(elapsed_time), str(float(SIZE)), "1"],
41         index=test_pool_df.columns),
42     ignore_index=True)
43 test_pool_df = test_pool_df.append(
44     pd.Series(
45         ["python using pool", str(pool_elapsed_time),
    ↪ str(float(SIZE)), "4"],
46         index=test_pool_df.columns),
47     ignore_index=True)
48 test_pool_output_data.save_df_in_image(test_pool_df, "test_pool.png")
```

Listing 7: main

7.8. 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 8: cblas_dgemm

7.9. 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 9: cblas_dgemm

7.10. 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 10: vectorization/blocked_dgemm

7.11. 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 always vector
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 11: vectorization/blocked_dgemm

7.12. 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 {
8     FILE *fp;
9     char *filename;
10    char** attributes;
11    int num_of_cols;
12    int num_of_rows;
13 } file_t;
14
15 file_t* create_file(char* filename, char* mode, char** attributes,
   ↪ int cols);
16 int add_row(file_t *file, char **values, int size);
17 int build_row(char **values, char *row, int max_bytes, int size);
18 void delete(file_t* file);
19
20 #endif  // FILE_H
```

Listing 12: file

7.13. src/controller/file.c

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

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;

    if (strcmp("a", mode) != 0) {
        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;
}

int add_row(file_t *file, char **values, int size) {
    if (size != file->num_of_cols) {
        return 1;
    }
```

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

Listing 13: file

7.14. src/controller/utils.h

```h
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

void init_arr(double m, double n, double off, double* a);
void print_arr(char *name, int m, int n, double *array);
void multiply_matrices(const double *a, const double *b, double *c,
    ↪ int n);
bool matrix_compare(const double *a, const double *b, int n);
int double_to_string(char* buffer, double num, int max_bytes);
int int_to_string(char* buffer, int num, int max_bytes);

#endif  // UTILS_H
```

Listing 14: utils

7.15. 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);
}

int int_to_string(char* buffer, int num, int max_bytes) {
    return snprintf(buffer, max_bytes, "%d", num);
```

```
50 }
```

Listing 15: utils

7.16. src/controller/generate_output_data.py

```
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
import multiprocessing as mp


class OutputData:
    pics_path = "./docs/report/pics/"
    files_path = "./src/data/"

    def __init__(self):
        self.data = {
            'number_of_threads': [],
            'parallel_time': [],
            'serial_time': [],
            'matrix_dimension': []
        }
        self._colors = ['b-', 'g-', 'r-', 'c-', 'm-', 'y-', 'k-',
    ↪ 'w-']
        self.dfs_data = []
        self.pics_path = "./docs/report/pics/"
        self.files_path = "./src/data/"

    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):
        """
        :param a: serial section
        :param b: parallel section
        :param p: number of processors
        :return: speed-up
        """
        alpha = a / (a + b)
        return p - (alpha * (p-1))

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

    @staticmethod
    def speed_up(s_n, p_n, s_1, p_1):
```

```
49          return (s_1 + p_1) / (s_n + p_n)
50
51      @staticmethod
52      def amdahl_max_speed_up(s, p):
53          return 1 + p/s
54
55      def save_data(self, df_name):
56          df = pd.DataFrame(data=self.data)
57          self.save_df_data(df, [], '', {}, df_name, False)
58
59      def save_df_in_image(self, df, df_name):
60          path = self.pics_path + df_name
61          df.to_html('table.html')
62          command = f'wkhtmltoimage -f png --width 0 table.html {path}'
63          call(command, shell=True)
64          call('rm table.html', shell=True)
65
66      def df_to_csv(self, df, df_name):
67          path = self.files_path + df_name
68          df.to_csv(path, index=False, sep=',', encoding='utf-8-sig')
69
70      def get_df_from_csv(self, filepath):
71          return pd.read_csv(filepath, low_memory=False, sep=',')
72
73      def graph_amdahl_speed_up(self, filename):
74          df = pd.DataFrame(data=self.data)
75          columns = [
76              'number_of_threads',
77              'parallel_time',
78              'serial_time'
79          ]
80          df = df.loc[:, columns]
81          df['theoretical_speed_up'] =
    ↪ get_null_list_of_dim_n(len(df.index))
82          df['theoretical_speed_up'] = df.apply(
83              lambda x: self.amdahl_speed_up(
84                  self.data['serial_time'][0],
85                  self.data['parallel_time'][0],
86                  x['number_of_threads']
87              ),
88              axis=1
89          )
90          df['real_speed_up'] = get_null_list_of_dim_n(len(df.index))
91          df['real_speed_up'] = df.apply(
92              lambda x: self.speed_up(
93                  x['serial_time'],
94                  x['parallel_time'],
95                  self.data['serial_time'][0],
96                  self.data['parallel_time'][0]
97              ),
98              axis=1
99          )
```

```
100
101         df['max_speed_up'] = get_null_list_of_dim_n(len(df.index))
102         df['max_speed_up'] = df.apply(
103             lambda x: self.amdahl_max_speed_up(
104                 x['serial_time'],
105                 x['parallel_time']
106             ),
107             axis=1
108         )
109         max_speed_up = df['max_speed_up'][0]
110         df['max_speed_up'] = df['max_speed_up'].map(lambda x:
    ↪ max_speed_up)
111
112         columns = [
113             'number_of_threads',
114             'theoretical_speed_up',
115             'real_speed_up',
116             'max_speed_up'
117         ]
118         df = df.loc[:, columns]
119         self.save_df_data(
120             df,
121             ['theoretical_speed_up', 'real_speed_up',
    ↪ 'max_speed_up'],
122             'number_of_threads',
123             {
124                 'theoretical_speed_up': 'b-',
125                 'real_speed_up': 'r-',
126                 'max_speed_up': 'g-'
127             },
128             filename,
129             True
130         )
131
132     def graph(self, df, y_axis, x_axis, colors, graph_name):
133         for field in y_axis:
134             plt.plot(
135                 df[x_axis],
136                 df[field],
137                 colors[field],
138                 label=field,
139                 marker='o',
140                 linestyle='dashed'
141             )
142         plt.xlabel(x_axis)
143         plt.yscale('linear')
144         plt.legend(loc='best')
145         plt.savefig(self.pics_path + graph_name)
146         plt.clf()
147
148     @staticmethod
149     def file_exists(path):
```

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

```
201             if df_data['has_graph']:
202                 y_axis = df_data['y_axis']
203                 x_axis = df_data['x_axis']
204                 colors = df_data['colors']
205                 graph_name = df_data['graph_name']
206                 self.graph(df, y_axis, x_axis, colors, graph_name)
207 
208     def graph_gustafson_exec_time(self, filename):
209         df = pd.DataFrame(data=self.data)
210         columns = [
211             'matrix_dimension',
212             'parallel_time',
213             'serial_time'
214         ]
215         df = df.loc[:, columns]
216         self.save_df_data(
217             df,
218             ['parallel_time', 'serial_time'],
219             'matrix_dimension',
220             {
221                 'parallel_time': 'b-',
222                 'serial_time': 'r-'
223             },
224             filename,
225             True
226         )
227 
228     def graph_gustafson_speed_up(self, filename):
229         df = pd.DataFrame(data=self.data)
230         columns = [
231             'matrix_dimension',
232             'parallel_time',
233             'serial_time',
234             'number_of_threads'
235         ]
236         df = df.loc[:, columns]
237         df['fixed_time_speed_up'] = [0] * len(df.index)
238         df['fixed_time_speed_up'] = df.apply(
239             lambda x: self.gustafson_speed_up(
240                 x['serial_time'],
241                 x['parallel_time'],
242                 mp.cpu_count()
243             ),
244             axis=1
245         )
246         df['alpha'] = df['serial_time'] / \
247             (df['serial_time'] + df['parallel_time'])
248         df = df.loc[:, ['alpha', 'fixed_time_speed_up']]
249         self.save_df_data(
250             df,
251             ['fixed_time_speed_up',],
252             'alpha',
```

```
253             {'fixed_time_speed_up': 'b-'},
254             filename,
255             True
256         )
257
258     def graph_gustafson_real_speed_up(self, filename):
259         df = pd.DataFrame(data=self.data)
260         columns = [
261             'matrix_dimension',
262             'parallel_time',
263             'serial_time',
264             'number_of_threads'
265         ]
266         df = df.loc[:, columns]
267         df['real_speed_up'] = [0] * len(df.index)
268         df['real_speed_up'] = df.apply(
269             lambda x: self.speed_up(
270                 x['serial_time'],
271                 x['parallel_time'],
272                 self.data['serial_time'][0],
273                 self.data['parallel_time'][0]
274             ),
275             axis=1
276         )
277         df = df.loc[:, ['number_of_threads', 'real_speed_up']]
278         self.save_df_data(
279             df,
280             ['real_speed_up'],
281             'number_of_threads',
282             {'real_speed_up': 'r-'},
283             filename,
284             True
285         )
```

Listing 16: generate_output_data

7.17. 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          mapped_values =
    ↪ list(itertools.chain.from_iterable(mapped_values))
61          partitioned_data = collections.defaultdict(list)
62          for a_mapped_value in mapped_values:
63              key, value = a_mapped_value
64              partitioned_data[key].append(value)
65          return list(partitioned_data.items())
66
67      @staticmethod
68      def shuffle(map_responses, num_workers):
69          map_responses = list(filter(lambda x: len(x) != 0,
    ↪ map_responses))
70          map_responses =
    ↪ list(itertools.chain.from_iterable(map_responses))
71          map_responses.sort(key=lambda tup: tup[0])
72          map_responses = chunks(map_responses, num_workers)
73          map_responses = list(filter(lambda x: len(x) != 0,
    ↪ map_responses))
74          return map_responses
75
76      def group_by_key_mapped_values(self, map_responses, num_workers):
77          is_repeated = True
78          output = []
79          while is_repeated:
80              # self.statistics.start('serial')
81              num_workers = ceil(num_workers/2)
82              map_responses = self.shuffle(map_responses, num_workers)
83              chunksize = self.get_chunksize(map_responses,
    ↪ num_workers)
84              # self.statistics.stop('serial')
85              pool = mp.Pool(processes=num_workers)
86              # self.statistics.start('parallel')
87              map_responses = pool.map(
88                  self.group_by_key,
89                  map_responses,
```

```
90                  chunksize=chunksize
91              )
92              # self.statistics.stop('parallel')
93              pool.close()
94              # self.statistics.start('serial')
95              repeated, not_repeated =
    ↪ self.keys_repeated(map_responses)
96              output += not_repeated
97              map_responses = repeated
98              is_repeated = len(repeated) != 0
99              # self.statistics.stop('serial')
100         return output
101
102     def map(self, inputs, num_workers=None):
103         """
104         :param inputs: data to map-reduce
105         :param chunksize: The portion of the input data to hand to
    ↪ each worker.
106         This can be used to tune performance during the mapping
    ↪ phase.
107         :param num_workers: The number of workers to create.
108         :return: Process the inputs through the map and reduce
    ↪ functions given.
109         """
110
111     def reduce(self, partitioned_data, num_workers=1):
112         """
113         :param partitioned_data:
114         :param num_workers: The number of workers to create.
115         :return:
116         """
```

Listing 17: map_reduce

7.18. 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         # data = self.group_by_key_mapped_values(map_responses,
   ↪ num_workers)
25         data = self.group_by_key(map_responses)
26         pool.close()
27         pool.join()
28         self.statistics.stop('parallel')
29         # time.sleep(self.sleep_sec)
30         return data
31
32     def reduce(self, partitioned_data, num_workers=1):
33         # pool = mp.Pool(processes=num_workers)
34         self.statistics.start('serial')
35         # reduced_values = pool.map(self.reduce_func,
   ↪ partitioned_data)
36         reduced_values = []
37         for data in partitioned_data:
38             reduced_values.append(self.reduce_func(data))
39         # pool.close()
40         # pool.join()
41         self.statistics.stop('serial')
42         # time.sleep(self.sleep_sec)
43         return reduced_values
```

Listing 18: pool

7.19. 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.daemon = True
26             process.start()
27         for process in self.processes:
28             process.join()
29             map_responses += process.get_output()
30         self.statistics.stop('parallel')
31         map_responses = list(filter(lambda x: len(x) != 0,
   ↪ 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 19: proccess

7.20. src/controller/my_process.py

```
1  from multiprocessing import Process, Queue
2  import time
3
4
5  class MyProcess(Process):
6
7      def __init__(self, target, args):
8          self.target = target
9          self.args = args
10         self.output = Queue()
11         self.finish = Queue()
12         self.output_list = []
13         super().__init__(target=target, args=args)
14
15     def run(self):
16         for an_arg in self.args:
17             self.output.put(self.target(an_arg))
18         self.finish.put(True)
19
20     def get_queue_data(self):
21         output_list = []
22         while self.output.qsize() != 0:
23             output_list.append(self.output.get())
24         return output_list
25
26     def get_output(self):
27         return self.output_list
28
29     def has_finished(self):
30         return self.finish.qsize() != 0
31
32     def join(self, **kwargs):
33         while self.finish.empty():
34             time.sleep(0.000000000000000001)
35         self.finish.get()
36         self.finish.close()
37         self.finish.join_thread()
38         self.output_list = self.get_queue_data()
39         self.output.close()
40         self.output.join_thread()
41         super().join(**kwargs)
```

Listing 20: my_proccess

7.21. 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 21: statistics

7.22. 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
```

Listing 22: utils

7.23. 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 23: multiply_matrices_interface

7.24. src/model/element_by_row_block.py

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


class ElementByRowBlock(MultiplyMatricesInterface):

    @staticmethod
    def pre_processing(matrix_a, matrix_b, **kwargs):
        row_size = len(matrix_a)
        col_size = len(matrix_a[0])
        output = []
        for i in range(0, row_size):
            for j in range(0, col_size):
                element_by_row_block = [matrix_a[i][j]] + matrix_b[j]
                output.append((i, element_by_row_block))
        return output

    @staticmethod
    def map_worker(chunk):
        output = []
        i, elements = chunk
        elem_a = elements[0]
        elements.pop(0)
        col_size = len(elements)
        for j in range(0, col_size):
            output.append(((i, j), elem_a * elements[j]))
        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 24: element_by_row_block

7.25. 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 25: column_by_row

7.26. 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 26: by_blocks