

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	1
2. Desarrollo teórico	2
2.1. Speed up	2
2.2. Ley de Amdahl	2
2.3. Ley de Gustafson	3
2.4. Map-reduce	3
2.5. High Performance Portable Libraries for Dense Linear Algebra	4
2.5.1. LAPACK	4
2.5.2. ScaLAPACK	4
2.5.3. CBLAS	5
2.6. Instrucciones vectoriales MMX	5
3. Implementación	6
3.1. Explicación del modelo	6
3.2. Multiplicación de matrices por bloques	7
3.2.0.1. Preprocesamiento	7
3.2.0.2. Mapeo	7
3.2.0.3. Reducción	7
3.3. Multiplicación de matrices de elemento por fila	7
3.3.0.1. Preprocesamiento	7
3.3.0.2. Mapeo	7
3.3.0.3. Reducción	8
3.4. Multiplicación de matrices de columna por fila	8
3.4.0.1. Preprocesamiento	8
3.4.0.2. Mapeo	8
3.4.0.3. Reducción	8
3.5. MMX	8
3.6. Cblas	9
3.7. Forma de ejecución	10
3.8. Datos sobre la computadora que se utilizó	11
4. Resultados	12
4.1. Multiplicación por bloques	12
4.1.0.1. Salida Amdahl	12
4.1.0.2. Salida Gustafson	14
4.2. Multiplicación elemento por fila	16
4.2.0.1. Salida Amdahl	16
4.2.0.2. Salida Gustafson	18
4.3. Multiplicación columna por fila	20
4.3.0.1. Salida Amdahl	20
4.3.0.2. Salida Gustafson	22

4.4. Cblas e instrucciones vectorizadas	24
5. Análisis de resultados	25
5.1. Pool	25
5.2. Prueba de latecia de pool, python y c	25
6. Conclusiones	26
7. Anexo	27
7.1. src/app.py	27
7.2. src/graphs.py	29
7.3. src/cblas.c	30
7.4. src/mmx.c	31
7.5. src/test_pool.c	32
7.6. src/test_pool.py	33
7.7. src/cblas/cblas_dgemm.h	35
7.8. src/cblas/cblas_dgemm.c	37
7.9. src/vectorization/blocked_dgemm_sse.h	38
7.10. src/vectorization/blocked_dgemm_sse.c	39
7.11. src/controller/file.h	41
7.12. src/controller/file.c	42
7.13. src/controller/utils.h	44
7.14. src/controller/utils.c	45
7.15. src/controller/generate_output_data.py	47
7.16. src/controller/map_reduce.py	53
7.17. src/controller/pool.py	56
7.18. src/controller/process.py	57
7.19. src/controller/my_process.py	58
7.20. src/controller/statistics.py	59
7.21. src/controller/utils.py	60
7.22. src/model/multiply_matrices_interface.py	62
7.23. src/model/element_by_row_block.py	63
7.24. src/model/column_by_row.py	64
7.25. src/model/by_blocks.py	65

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 haremos una multiplicación de matrices (ambas de $N \times N$) y se realizarán las mediciones de tiempo variando la cantidad de threads involucrados en el procesamiento.

Luego en Gustafson se realizarán 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 $N \times N$ usando la librería CBLAS e instrucción de vectorización (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 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:

$$\text{speed_up} = \frac{\text{Rendimiento_con_mejora}}{\text{Rendimiento_sin_mejora}} \quad (1)$$

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

$$\text{speed_up} = \frac{T_{\text{ejecucion_sin_mejora}}}{T_{\text{ejecucion_con_mejora}}} \quad (2)$$

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:

$$\text{speed_up} = \frac{s + p}{s + \frac{p}{N}} \quad (3)$$

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:

$$\text{speed_up_max} = 1 + \frac{p}{s} \quad (4)$$

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 paralelización 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:

$$\text{speed_up} = \frac{s + p * N}{s + p} \quad (5)$$

Definiendo:

$$\alpha = \frac{s}{s + p} \quad (6)$$

Podemos decir que:

$$\text{speed_up} = N - \alpha * (N - 1) \quad (7)$$

2.4. Map-reduce

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

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 tenían la misma clave. Entonces reduce aplica una función a todos estos valores para retornar un único valor y así 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

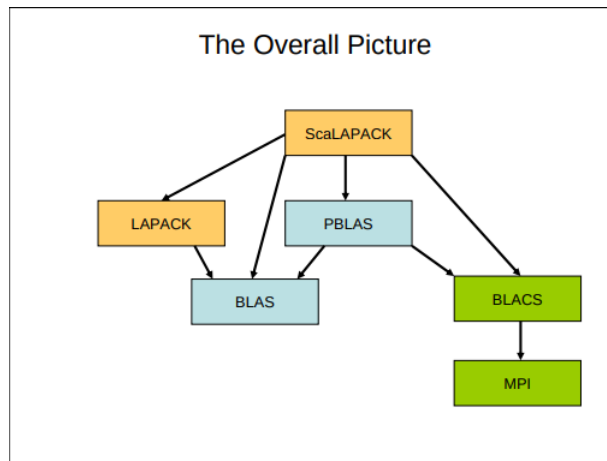


Figura 1: overall picture

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 librería 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 $N \times N$ y **B** de $N \times N$ las dividimos en $(N/2) \times (N/2)$ 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 cuestión **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 índice 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 $N \times N$ y **B** de $N \times N$ 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 multiplicaban dos matrices por bloques. Al momento de hacer la multiplicación parcial de elementos, donde cada elemento es un bloque de la matriz, se usa una instrucción llamada `#pragma vector always` para decirle al compilador que la siguiente fracción de código será vectorizada. Este código fue desarrollado 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,  
2 ↪ k, alpha , A, k, B, n, beta , C, n);
```

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 `140x140` con 2 threads y dos matrices de `200x200` 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` y las instrucciones vectoriales 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 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 Gustafson 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	15692.744017	857.866764	200
1	2	9746.788502	856.030464	200
2	4	9003.918648	853.797436	200
3	8	8714.613199	856.029510	200
4	16	8175.534725	857.297421	200
5	32	8740.187645	856.795311	200
6	64	8145.057917	858.325243	200
7	128	8780.988693	855.299711	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	19.292752
1	2	1.901443	1.560963	19.292752
2	4	3.461708	1.678950	19.292752
3	8	5.870135	1.729310	19.292752
4	16	9.001437	1.832273	19.292752
5	32	12.275488	1.724564	19.292752
6	64	15.004191	1.838266	19.292752
7	128	16.880346	1.717530	19.292752

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

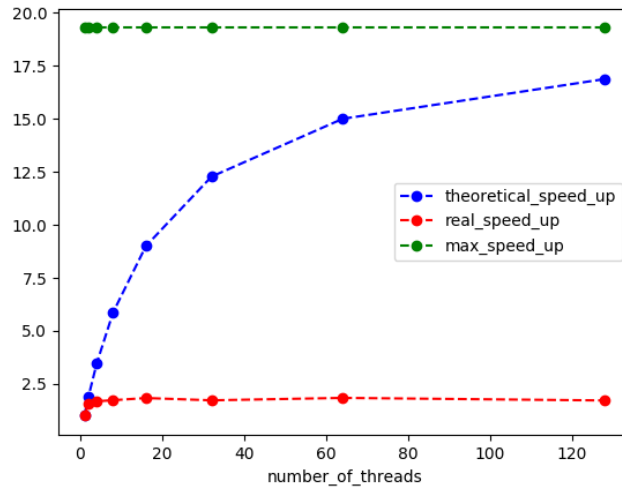


Figura 5: Gráfico

Podemos 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 tampoco. Se puede ver que tiene un speed-up de casi 2 lo cual quiere decir que hace uso de la mitad.

4.1.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	1648.200989	135.467768	100
1	2	3488.086462	311.276913	140
2	4	8769.488335	856.283665	200

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. Pero además podemos observar que hay mucha ineficiencia respecto del uso de la paralelización ya que al aumentar el trabajo en casi el doble usando 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. Mas adelante explicaremos que el módulo Pool es ineficiente respecto del uso de recursos

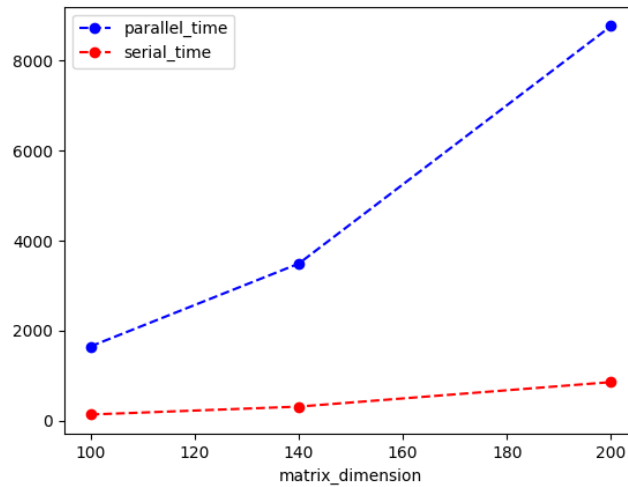


Figura 7: Tiempo paralelo y serie en función de la dimensión de las matrices de entrada

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

	alpha	speed_up
0	0.075949	3.772153
1	0.081929	3.754214
2	0.088957	3.733128

Figura 8: Tabla de valores del speed-up

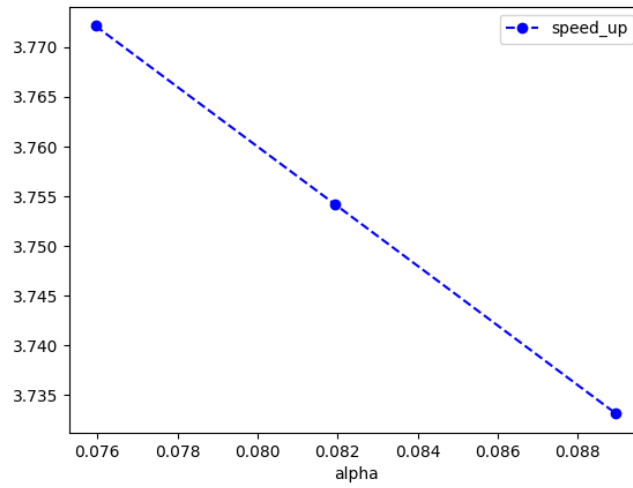


Figura 9: Gráfico del speed-up

4.2. Multiplicación elemento por fila

4.2.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	15708.036423	446.056843	200
1	2	11205.162048	454.832792	200
2	4	9440.731049	471.447229	200
3	8	9751.883030	448.982000	200
4	16	9765.306711	448.605299	200
5	32	9658.138990	449.154139	200
6	64	9812.398195	467.936754	200
7	128	10609.148502	500.906706	200

Figura 10: 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	36.215324
1	2	1.946259	1.385429	36.215324
2	4	3.693997	1.629722	36.215324
3	8	6.704163	1.583600	36.215324
4	16	11.313903	1.581577	36.215324
5	32	17.241461	1.598261	36.215324
6	64	23.361117	1.571359	36.215324
7	128	28.401509	1.454007	36.215324

Figura 11: Speed-up real, teórico y máximo según la cantidad de threads

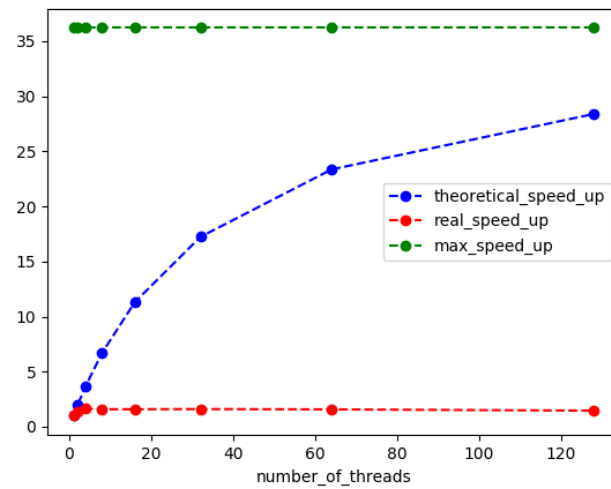


Figura 12: Gráfico

Podemos 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 tampoco. Se puede ver que tiene un speed-up de casi 2 lo cual quiere decir que hace uso de la mitad.

4.2.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	2096.164942	60.039043	100
1	2	3898.903131	162.311554	140
2	4	10634.383678	543.992043	200

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. Pero además podemos observar que hay mucha ineficiencia respecto del uso de la paralelización ya que al aumentar el trabajo en casi el doble usando 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. Mas adelante explicaremos que el módulo Pool es ineficiente respecto del uso de recursos

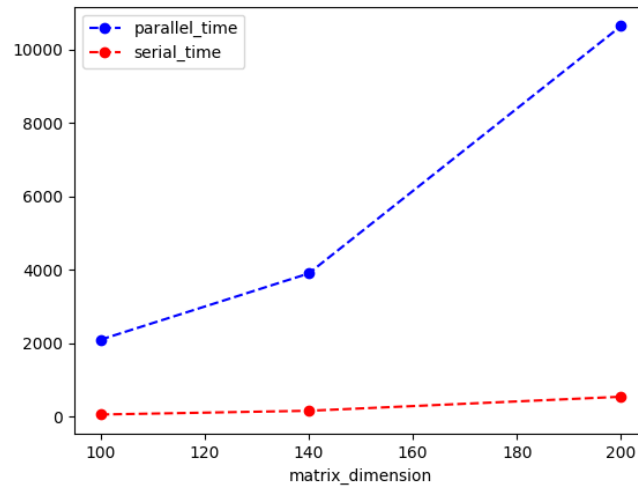


Figura 14: Tiempo paralelo y serie en función de la dimensión de las matrices de entrada

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

	alpha	speed_up
0	0.027845	3.916466
1	0.039966	3.880101
2	0.048665	3.854006

Figura 15: Tabla de valores del speed-up

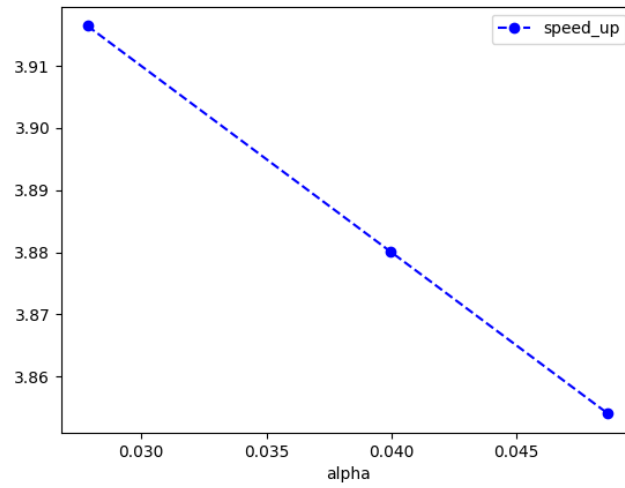


Figura 16: Gráfico del speed-up

4.3. Multiplicación columna por fila

4.3.0.1 Salida Amdahl

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	16468.362093	448.450327	200
1	2	10398.904562	452.495813	200
2	4	9070.483685	614.428759	200
3	8	8816.234589	449.283838	200
4	16	9031.642675	609.318018	200
5	32	8829.077721	442.583561	200
6	64	8517.833948	592.516422	200
7	128	8467.262030	437.215805	200

Figura 17: 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	37.722823
1	2	1.948351	1.558952	37.722823
2	4	3.705325	1.746718	37.722823
3	8	6.747843	1.825782	37.722823
4	16	11.447892	1.754681	37.722823
5	32	17.565203	1.824572	37.722823
6	64	23.969351	1.856878	37.722823
7	128	29.313008	1.899810	37.722823

Figura 18: Speed-up real, teórico y máximo según la cantidad de threads

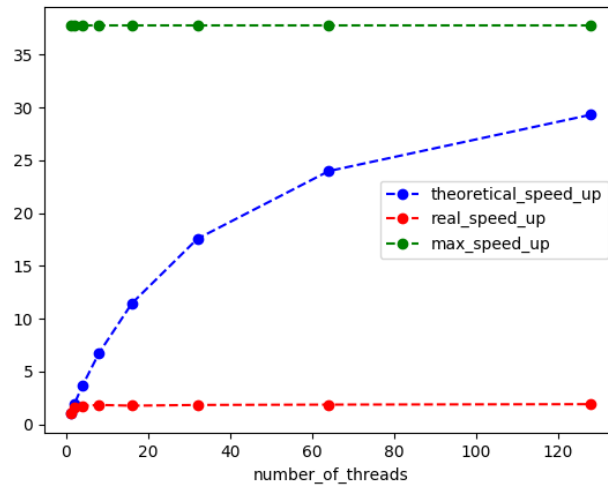


Figura 19: Gráfico

Podemos 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 tampoco. Se puede ver que tiene un speed-up de casi 2 lo cual quiere decir que hace uso de la mitad.

4.3.0.2 Salida Gustafson

	number_of_threads	parallel_time	serial_time	matrix_dimension
0	1	1642.557621	89.639187	100
1	2	3284.487009	153.816462	140
2	4	8694.979668	594.493628	200

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. Pero además podemos observar que hay mucha ineficiencia respecto del uso de la paralelización ya que al aumentar el trabajo en casi el doble usando 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. Mas adelante explicaremos que el módulo Pool es ineficiente respecto del uso de recursos

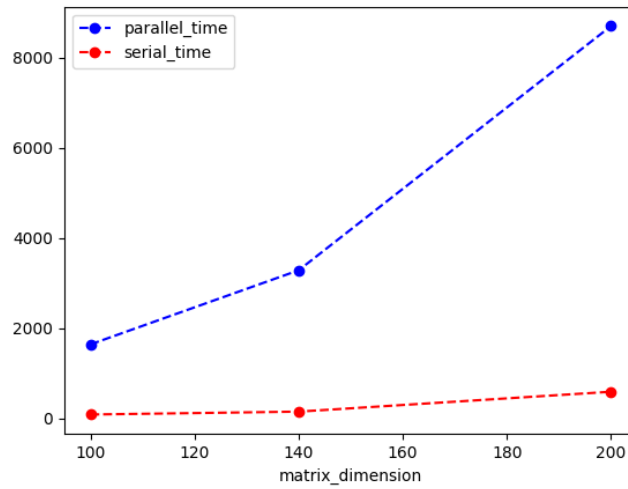


Figura 21: Tiempo paralelo y serie en función de la dimensión de las matrices de entrada

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

	alpha	speed_up
0	0.051749	3.844753
1	0.044736	3.865792
2	0.063996	3.808011

Figura 22: Tabla de valores del speed-up

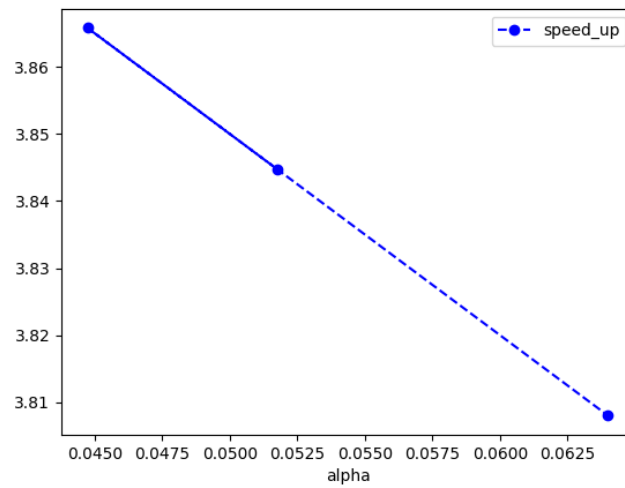


Figura 23: Gráfico del speed-up

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	0.109354	400	1
2	not vectorized blocked_dgemm_sse	0.366132	400	1

Figura 24: Tiempo serie de multiplicación en segundos

Podemos ver que `cblas` tiene una mejor performance que las instrucciones vectoriales. Por otro lado podemos ver que hay una mejora al usar las instrucciones vectoriales en el código `blocked_dgemm_sse` respecto del mismo código que no hace uso de las mismas. De esta manera vemos 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 25: 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 ineficiente como se explicó anteriormente debido al uso de recursos.

6. Conclusiones

Podemos decir que obtuvimos resultados inesperados pero se tuvieron que hacer varias corridas y ajustar ciertos números para entender por qué llegamos a éstos mismos.

Si bien, al paralelizar, hubo casos donde se obtuvo una mejora, no era la mejora esperada. Es decir, cuando se usaban 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 ineficiente 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 anteriormente en la sección `análisis de resultados`) y esto hace que el proceso sea muy lento y la performance baje considerablemente.

Finalmente podemos 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` pudimos ver el uso de cada CPU y gráficos al respecto.

7. Anexo

7.1. src/app.py

```
1 import time
2 from typing import Type
3 from math import ceil
4 from src.controller.pool import Pool as MapReduce
5 from src.controller.utils import get_random_matrix_of_dim_n
6 from src.model.element_by_row_block import ElementByRowBlock
7 from src.model.column_by_row import ColumnByRow
8 from src.model.by_blocks import ByBlocks
9 from src.model.multiply_matrices_interface import
    ↪ MultiplyMatricesInterface
10 from src.controller.generate_output_data import OutputData
11
12
13 SAVE = True
14
15
16 def gustafson(model: Type[ MultiplyMatricesInterface ]):
17     name = model.__name__
18     print(f"-----RUNNING GUSTAFSON-----")
19     output_data = OutputData()
20     for values in [(1, 100), (2, 140), (4, 200)]:
21         num_workers, matrix_dim = values
22         print(f"RUNNING WITH MATRIX DIMENSION: {matrix_dim}")
23         serial, parallel = run(num_workers, matrix_dim, model)
24         output_data.add_data(serial, parallel, num_workers,
    ↪ matrix_dim)
25         if SAVE:
26             output_data.save_data(name + '_gustafson_output.png')
27             output_data.graph_gustafson_exec_time(name +
    ↪ '_gustafson_exec_time.png')
28             output_data.graph_gustafson_speed_up(name +
    ↪ '_gustafson_speed_up.png')
29             output_data.save_df_data_to_json()
30
31
32 def amdahl(model: Type[ MultiplyMatricesInterface ]):
33     name = model.__name__
34     print(f"-----RUNNING AMDAHL-----")
35     output_data = OutputData()
36     matrix_dim = 200
37     for num_workers in [1, 2, 4, 8, 16, 32, 64, 128]:
38         print(f"RUNNING WITH NUM WORKERS: {num_workers}")
39         serial, parallel = run(num_workers, matrix_dim, model)
40         output_data.add_data(serial, parallel, num_workers,
    ↪ matrix_dim)
41         if SAVE:
42             output_data.save_data(name + '_amdahl_output.png')
```

```

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

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

Listing 3: main

7.4. src/mmx.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include "controller/utils.h"
5 #include "vectorization/blocked_dgemm_sse.h"
6 #include "controller/file.h"
7
8 double run_blocked_dgemm_sse(int N, double* A, double* B, double* C)
9     ↪ {
10     clock_t start, stop;
11     init_arr(N, N, 2, A);
12     init_arr(N, N, 1, B);
13     init_arr(N, N, 0, C);
14     start = clock();
15     square_dgemm_blocked_sse(A, B, C, N, 2);
16     stop = clock();
17     double elapsed_seconds = ((double)(stop - start)) /
18     ↪ CLOCKS_PER_SEC;
19     printf("Elapsed time = %f seconds\n", elapsed_seconds);
20     return elapsed_seconds;
21 }
22
23 int main(int argc, char **argv) {
24     if (argc != 2) {
25         return 1;
26     }
27     char* program_name = argv[1];
28     char* attributes[4] = {
29         "program",
30         "time_elapsed",
31         "matrix_dim",
32         "number_of_threads"
33     };
34     file_t* file = create_file("src/data/dgemm.csv", "a",
35     ↪ attributes, 4);
36     int N = 400;
37     double A[N*N];
38     double B[N*N];
39     double C[N*N];
40     double elapsed_seconds;
41     char elapsed_second_str[20];
42     char* values[4] = {program_name, elapsed_second_str, "400", "1"};
43     elapsed_seconds = run_blocked_dgemm_sse(N, A, B, C);
44     double_to_string(elapsed_second_str, elapsed_seconds, 20);
45     add_row(file, values, 4);
46     delete(file);
47     return 0;
48 }
```

Listing 4: main

7.5. src/test_pool.c

```
1 #include <time.h>
2 #include <stdio.h>
3 #include "controller/file.h"
4 #include "controller/utils.h"
5
6 #define SIZE 10000000
7
8 int main() {
9     clock_t start, stop;
10    start = clock();
11    unsigned long sum = 0;
12    for (unsigned long i = 0; i < SIZE; i++) {
13        sum += i;
14    }
15    stop = clock();
16    char* attributes[4] = {
17        "program",
18        "time_elapsed (sec)",
19        "iterations",
20        "number_of_threads"
21    };
22    file_t* file = create_file("src/data/test_pool.csv", "w+",
    ↪ attributes, 4);
23    double elapsed_seconds = ((double)(stop - start)) /
    ↪ CLOCKS_PER_SEC;
24    printf("Elapsed time = %f seconds\n", elapsed_seconds);
25
26    char elapsed_second_str[20];
27    char size_str[20];
28    double_to_string(elapsed_second_str, elapsed_seconds, 20);
29    int_to_string(size_str, SIZE, 20);
30    char* values[4] = {"c program", elapsed_second_str, size_str,
    ↪ "1"};
31    add_row(file, values, 4);
32    return 0;
33 }
```

Listing 5: main

7.6. 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 =
38     ↪ 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),
46         ↪ str(float(SIZE)), "4"],
47         index=test_pool_df.columns),
48     ignore_index=True)
48 test_pool_output_data.save_df_in_image(test_pool_df, "test_pool.png")
```

Listing 6: main

7.7. 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
9     ↪ operation.
10     In this case:
11
12     CblasRowMajor:
13     Indicates that the matrices are stored in row major order, with
14     ↪ the elements
15     of each row of the matrix stored contiguously as shown in the
16     ↪ figure above.
17
18     CblasNoTrans:
19     Enumeration type indicating that the matrices A and B should not
20     ↪ be
21     transposed or conjugate transposed before multiplication.
22
23     m, n, k:
24     Integers indicating the size of the matrices:
25
26     A: m rows by k columns
27
28     B: k rows by n columns
29
30     C: m rows by n columns
31
32     alpha:
33     Real value used to scale the product of matrices A and B.
34
35     A:
36     Array used to store matrix A.
37
38     k:
39     Leading dimension of array A, or the number of elements between
40     ↪ successive
41     rows (for row major storage) in memory. In the case of this
42     ↪ exercise the
43     leading dimension is the same as the number of columns.
44
45     B:
46     Array used to store matrix B.
47
48     n:
49     Leading dimension of array B, or the number of elements between
50     ↪ 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 7: cblas_dgemm

7.8. src/cblas/cblas_dgemm.c

```
1 #include "cblas_dgemm.h"
2
3 int mult(double *A, double *B, double *C, double alpha, double beta,
4     ↪ int m, int k, int n) {
5     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k,
6     ↪ alpha, A, k, B, n, beta, C, n);
7     return 0;
8 }
```

Listing 8: cblas_dgemm

7.9. 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 9: vectorization/blocked_dgemm

7.10. 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             for (k = 0; k < block_size; ++k) {
20                 cij += A[i + k * N] * B[k + j * N];
21             }
22             C[j*N + i] = cij;
23         }
24     }
25 }
26
27 void do_block_sse(const double *A, const double *B, double *C, int
    ↪ i, int j,
28     int k, int N, int block_size) {
29     basic_dgemm_sse(A + i + k*N, B + k + j*N, C + i + j*N, N,
    ↪ block_size);
30 }
31
32 void square_dgemm_blocked_sse(const double *A, const double *B,
    ↪ double *C,
33     int N, int block_size) {
34     unsigned bi, bj, bk;
35     for (bi = 0; bi < (N / block_size); ++bi) {
36         const unsigned i = bi * block_size;
37         for (bj = 0; bj < (N / block_size); ++bj) {
38             const unsigned j = bj * block_size;
39             for (bk = 0; bk < (N / block_size); ++bk) {
40                 const unsigned k = bk * block_size;
41                 do_block_sse(A, B, C, i, j, k, N, block_size);
42             }
43         }
44     }
45 }
```

Listing 10: vectorization/blocked_dgemm

7.11. 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,
16    ↪ int cols);
17 int add_row(file_t *file, char **values, int size);
18 int build_row(char **values, char *row, int max_bytes, int size);
19 void delete(file_t* file);
20 #endif // FILE_H
```

Listing 11: file

7.12. src/controller/file.c

```
1 #include "file.h"
2 #include <stdlib.h>
3 #include "utils.h"
4
5 file_t* create_file(char* filename, char* mode, char** attributes,
6   ↪ int cols) {
7     file_t* file = (file_t*) malloc(sizeof(file_t));
8     if (!file) {
9         return NULL;
10    }
11    file->fp = fopen(filename, mode);
12    file->filename = filename;
13    file->num_of_cols = cols;
14
15    if (strcmp("a", mode) != 0) {
16        char file_header[256];
17        if (build_row(attributes, file_header, 256, cols) < 0) {
18            delete(file);
19            return NULL;
20        }
21        add_row(file, attributes, cols);
22    }
23    file->attributes = attributes;
24    file->num_of_rows = 0;
25    return file;
26 }
27
28 int build_row(char **values, char *row, int max_bytes, int size) {
29     int bytes = 0;
30     int pos = 0;
31     char* buff;
32     for (int i = 0; i < size; i++) {
33         buff = (row)+pos;
34         if (i == size-1) {
35             bytes = snprintf(buff, max_bytes, "%s", values[i]);
36         } else {
37             bytes = snprintf(buff, max_bytes, "%s, ", values[i]);
38         }
39         if (bytes < 0) {
40             return bytes;
41         }
42         pos += bytes;
43     }
44     return bytes;
45 }
46
47 int add_row(file_t *file, char **values, int size) {
48     if (size != file->num_of_cols) {
49         return 1;
50     }
51 }
```

```

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 12: file

7.13. 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 int int_to_string(char* buffer, int num, int max_bytes);
14
15 #endif // UTILS_H
```

Listing 13: utils

7.14. src/controller/utils.c

```
1 #include "utils.h"
2
3 void init_arr(double m, double n, double off, double* a) {
4     int i;
5     for (i = 0; i < (m*n); i++) {
6         // a[i] = (i + 1) * off;
7         a[i] = (i % 10) * off;
8     }
9 }
10 void print_arr(char *name, int m, int n, double *array) {
11     int i, j;
12     printf("\n%s\n", name);
13     for (i = 0; i < m; i++){
14         for (j = 0; j < n; j++) {
15             printf("%g\t", array[i+j*n]);
16         }
17         printf("\n");
18     }
19 }
20
21 void multiply_matrices(const double *a, const double *b, double *c,
22     ↪ int n) {
23     int i, j, k;
24     for (i = 0; i < n; i++) {
25         for (j = 0; j < n; j++) {
26             for (k = 0; k < n; k++) {
27                 c[i+k*n] = c[i+k*n] + a[i+j*n] * b[j+k*n];
28             }
29         }
30     }
31 }
32 bool matrix_compare(const double *a, const double *b, int n) {
33     int i, j;
34     for (i = 0; i < n; i++) {
35         for (j = 0; j < n; j++) {
36             if (a[i+j] != b[i+j]) {
37                 return false;
38             }
39         }
40     }
41     return true;
42 }
43
44 int double_to_string(char* buffer, double num, int max_bytes) {
45     return snprintf(buffer, max_bytes, "%f", num);
46 }
47
48 int int_to_string(char* buffer, int num, int max_bytes) {
49     return snprintf(buffer, max_bytes, "%d", num);
50 }
```


50 }

Listing 14: utils

7.15. src/controller/generate_output_data.py

```
1 import json
2 import os
3 import pandas as pd
4 from subprocess import call
5 import matplotlib.pyplot as plt
6 from src.controller.utils import get_null_list_of_dim_n
7 import multiprocessing as mp
8
9
10 class OutputData:
11     pics_path = "./docs/report/pics/"
12     files_path = "./src/data/"
13
14     def __init__(self):
15         self.data = {
16             'number_of_threads': [],
17             'parallel_time': [],
18             'serial_time': [],
19             'matrix_dimension': []
20         }
21         self._colors = ['b-', 'g-', 'r-', 'c-', 'm-', 'y-', 'k-',
22             ↪ 'w-']
23         self.dfs_data = []
24         self.pics_path = "./docs/report/pics/"
25         self.files_path = "./src/data/"
26
27     def add_data(self, serial, parallel, num_workers,
28         ↪ matrix_dimension):
29         self.data['number_of_threads'].append(num_workers)
30         self.data['parallel_time'].append(parallel)
31         self.data['serial_time'].append(serial)
32         self.data['matrix_dimension'].append(matrix_dimension)
33
34     @staticmethod
35     def gustafson_speed_up(a, b, p):
36         """
37         :param a: serial section
38         :param b: parallel section
39         :param p: number of processors
40         :return: speed-up
41         """
42         alpha = a / (a + b)
43         return p - (alpha * (p-1))
44
45     @staticmethod
46     def amdahl_speed_up(s, p, n):
47         return (s + p) / (s + p/n)
48
49     @staticmethod
50     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:
111 ↪ max_speed_up)
112
113     columns = [
114         'number_of_threads',
115         'theoretical_speed_up',
116         'real_speed_up',
117         'max_speed_up'
118     ]
119     df = df.loc[:, columns]
120     self.save_df_data(
121         df,
122         [ 'theoretical_speed_up', 'real_speed_up',
123 ↪ 'max_speed_up' ],
124         'number_of_threads',
125         {
126             'theoretical_speed_up': 'b-',
127             'real_speed_up': 'r-',
128             'max_speed_up': 'g-'
129         },
130         filename,
131         True
132     )
133
134     def graph(self, df, y_axis, x_axis, colors, graph_name):
135         for field in y_axis:
136             plt.plot(
137                 df[x_axis],
138                 df[field],
139                 colors[field],
140                 label=field,
141                 marker='o',
142                 linestyle='dashed'
143             )
144             plt.xlabel(x_axis)
145             plt.yscale('linear')
146             plt.legend(loc='best')
147             plt.savefig(self.pics_path + graph_name)
148             plt.clf()
149
150     @staticmethod
151     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,
174 ↪ has_graph):
175         df_graph_name = graph_name.split('.')[0] + '_table.csv'
176         self.df_to_csv(df, df_graph_name)
177         self.dfs_data.append({
178             'has_graph': has_graph,
179             'df_path_name': df_graph_name,
180             'y_axis': y_axis,
181             'x_axis': x_axis,
182             'colors': colors,
183             'graph_name': graph_name
184         })
185
186     def read_dfs_data_from_json(self):
187         data_path = f"{self.files_path}data.json"
188         with open(data_path, encoding='utf-8-sig') as json_file:
189             text = json_file.read()
190             self.dfs_data = json.loads(text)
191
192     def graph_dfs(self):
193         for df_data in self.dfs_data:
194             df_name = df_data['df_path_name']
195             print(f"graph name: {df_data['graph_name']}")
196             print(f"df_name: {df_name}")
197             table_graph_name = df_name.split('.')[0] + '.png'
198             df_path_name = self.files_path + df_name
199             df = pd.read_csv(df_path_name, low_memory=False, sep=',')
200             print(f"df.columns: {list(df.columns)}")
201             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['speed_up'] = [0] * len(df.index)
238         df['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', 'speed_up']]
249         self.save_df_data(
250             df,
251             ['speed_up'],
252             'alpha',

```

```
253         { 'speed_up': 'b-' },  
254         filename ,  
255         True  
256     )
```

Listing 15: generate_output_data

7.16. 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
14         ↪ data. Takes as
15         ↪ argument one input value and returns a tuple with the key
16         ↪ and a value
17         ↪ to be reduced.
18         :param reduce_func: Function to reduce partitioned version of
19         ↪ intermediate data to final output. Takes as argument a key
20         ↪ as produced
21         ↪ by map_func and a sequence of the values associated with
22         ↪ that key.
23         """
24         self.map_func = map_func
25         self.reduce_func = reduce_func
26         self.statistics = Statistics()
27
28     @staticmethod
29     def get_chunksize(inputs, num_workers):
30         chunksize = int(len(inputs) / num_workers)
31         if chunksize == 0:
32             return 1
33         return chunksize
34
35     def get_statistics(self):
36         return self.statistics
37
38     @staticmethod
39     def keys_repeated(map_responses):
40         map_responses = map_responses.copy()
41         map_responses =
42         ↪ list(itertools.chain.from_iterable(map_responses))
43         keys = {}
44         for a_mapped_value in map_responses:
45             pos, values = a_mapped_value
46             if pos not in keys:
47                 keys[pos] = [False, values]
48             else:
49                 keys[pos][0] = True
50                 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 16: map_reduce

7.17. 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_func):
9         super().__init__(map_func=map_func, reduce_func=reduce_func)
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 17: pool

7.18. 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_func):
11         self.processes = []
12         super().__init__(map_func=map_func, reduce_func=reduce_func)
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,
22 ↪ args=arg))
23         map_responses = []
24         self.statistics.start('parallel')
25         for process in self.processes:
26             process.daemon = True
27             process.start()
28         for process in self.processes:
29             process.join()
30             map_responses += process.get_output()
31         self.statistics.stop('parallel')
32         map_responses = list(filter(lambda x: len(x) != 0,
33 ↪ map_responses))
34         return self.group_by_key(map_responses)
35
36     def reduce(self, partitioned_data, num_workers=1):
37         output = []
38         self.statistics.start('serial')
39         for item in partitioned_data:
40             output.append(self.reduce_func(item))
41         self.statistics.stop('serial')
42         return output
```

Listing 18: process

7.19. src/controller/my_process.py

[illegible]

Listing 19: my_process

7.20. 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 20: statistics

7.21. 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 21: utils

7.22. 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 22: multiply_matrices_interface

7.23. 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 23: element_by_row_block

7.24. src/model/column_by_row.py

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

Listing 24: column_by_row

7.25. src/model/by_blocks.py

```
1 import numpy as np
2 from src.model.multiply_matrices_interface import
  ↪ MultiplyMatricesInterface
3 from src.controller.utils import get_partitions, sum_matrices
4
5
6 class ByBlocks(MultiplyMatricesInterface):
7
8     @staticmethod
9     def pre_processing(matrix_a, matrix_b, **kwargs):
10         output = []
11         row_p = kwargs.get('row_p', 2)
12         col_p = kwargs.get('col_p', 2)
13         blocks_a = get_partitions(matrix_a, row_p, col_p)
14         blocks_b = get_partitions(matrix_b, row_p, col_p)
15         for r_a in range(0, row_p):
16             for c_a in range(0, col_p):
17                 a_block = blocks_a[r_a][c_a]
18                 output.append((r_a, a_block, blocks_b[c_a]))
19         return output
20
21     @staticmethod
22     def map_worker(chunk):
23         r_a, block_a, blocks_b = chunk
24         output = []
25         col_size = len(blocks_b)
26         for c_b in range(0, col_size):
27             result = np.matmul(block_a, blocks_b[c_b]).tolist()
28             key = (r_a, c_b)
29             output.append((key, result))
30         return output
31
32     @staticmethod
33     def reduce_worker(item):
34         output_pos, values = item
35         result = sum_matrices(values)
36         output = []
37         row_size = len(result)
38         block_pos_i, block_pos_j = output_pos
39         for i in range(0, row_size):
40             col_size = len(result[i])
41             for j in range(0, col_size):
42                 pos = (block_pos_i*row_size+i,
  ↪ block_pos_j*col_size+j)
43                 output.append((pos, result[i][j]))
44         return output
```

Listing 25: by_blocks