Maestría en Computo Estadístico
Programación y análisis de algoritmos
Reporte Proyecto Final
13 de diciembre de 2020
Enrique Santibáñez Cortés
Repositorio de Git: Proyecto Final.

Introducción

El objetivo de este documento es presentar las comparaciones de los tiempos de ejecución de las operaciones matriciales (suma, resta, multiplicación e inversa de matrices) programada de forma secuencial y de forma paralela considerando distitos tamaños de las matrices y números de hilos programadas en C++ (g++ (Ubuntu 5.4.0 − 6ubuntu ∼ 16.04.12) 5.4.0 20160609). Los tiempos de ejecución se consideraron apartir que se ejecuta la función a operar, no se contempla los tiempos para generar las matrices ni el tiempo del menu. Las comparaciones se realizaron en un dispositivo con sistema operativo ubuntu 16.04 LTS, 16.GB RAM, procesador AMD A8$-$6410 APU with AMD Radeon R5 Graphics ×4.

Los códigos de las operaciones de forma secuencial se pueden ver con más de detalle en https://github.com/Enriquesec/Programcion_y_analisis_de_algoritmos/tree/master/Tareas/proyecto_intermedio_enrique_santibanez_cortes. y el código paralelizado en https://github.com/Enriquesec/Programcion_y_analisis_de_algoritmos/tree/master/Tareas/proyecto_final (se mencionara las ideas básicas de como se paralelizo en cada operación). Un ejemplo de como se midieron los tiempos de ejecución de las simulaciones se puede observar en el archivo *simulaciones.cpp* el cual compara los tiempo de ejecución para la suma de dos matrices de 40000×40000 de manera secuencia y de forma paralela considerando 4 hilos.

Metodología y Resultados.

- **Comparación de la Suma y Resta.**

Para este caso tenemos que la función que suma dos matrices de forma secuencia y paralelo es:

```
// Función suma.
void suma_matriz(matriz mat_1, matriz mat_2, matriz resultado, int inf, int sup){  // sumamos dos matrices.
    for (int i=0; i< resultado.m_filas; i++){
        for (int j = inf; j<sup; j++) {
            resultado.df[i][j] = mat_1.df[i][j]+mat_2.df[i][j];
        }
    }
}

// Función suma para asignarla a un hilo.
void* suma_matriz_hilo(void *args) {
    Argumentos *_args = (Argumentos*) args;
    suma_matriz(_args->mat_1, _args->mat_2, _args->resultado, _args->inf, _args->sup);
    pthread_exit(NULL);
    return NULL;
}
```

Figura 1: Suma matricial de forma secuencia y de forma paralelo.

Si observamos podemos ocupar la función \textit{suma_matriz} que se observa en la Figura 1 se ocupa para calcular la suma matricial secuencial y de forma paralela, la diferencia es que para ejecutar rádica en los límites que se introducen en la función. Cuando se ejecuta de forma paralela $inf = 0$ y $sup = n$(número de columnas de la matriz) y de forma paralelo estos parámetros toman los siguientes valores:

```
for(int i=0; i<NTHREADS; i++){
    if(i==NTHREADS-1){       // limites para cada hilo
        args[i].sup = nA;
        args[i].inf = subint*i;
        args[i].mat_1 = A;
        args[i].mat_2 = B;
        args[i].resultado = C;
    } else{
        args[i].inf = subint*i;
        args[i].sup = subint*(i+1);
        args[i].mat_1 = A;
        args[i].mat_2 = B;
        args[i].resultado = C;
    }
```

Figura 2: Suma matricial de forma secuencia y de forma paralelo.

La idea del algoritmo de forma paralelo es dividir el número de columnas entre el número de hilos con los que se esta trabajando, es como tuvieramos n submatrices por bloques de las matrices originales y cada hilo suma cada submatriz de cada matriz.

Una observación importante es que todas las operaciones que se presentan consideran una paralelización con el número de columnas, es decir, que si el número de columnas es menor al número de hilos entonces el tiempo de ejecución sería el mismo o muy parecido al secuencia. Por falta de tiempo todos los resultados que se aquí consideran matrices cuadradas con distintos tamaños.

Para las sumas consideramos 3 y 4 hilos en matrices cuadradas con tamaños: 100, 1000, 5000, 10000, 20000. Cada combinación de parámetros anteriores se simulación 4 veces y se obtuvo un tiempo promedio de ejecución. Los resultados son:

```
library(tidyverse)

datos <- read.csv("tiempos_simulacion.csv")
datos <- datos %>% select(operacion, num_hilos, tamano_matriz, tiempo,tamano)
datos <- datos[!is.na(datos$tiempo),]
datos$num_hilos[is.na(datos$num_hilos)] <- 1

datos_suma <- datos %>%
  group_by(operacion, num_hilos, tamano_matriz, tamano) %>%
  summarise(tiempo_mean=mean(tiempo)) %>%
  filter(operacion %in% c("suma","suma_p")) %>%
  mutate(cat=paste0(operacion, num_hilos))

datos_suma <- as.data.frame(datos_suma)

datos_suma %>%
  select(operacion, num_hilos ,tamano_matriz, tiempo_mean) %>%
  spread(key = "tamano_matriz", value = "tiempo_mean")
```

```
##   operacion num_hilos 10000x10000   1000x1000      100x100 15000x15000
## 1      suma         1    1.301270 0.010700375 0.0001239058    2.998980
## 2    suma_p         3    0.394985 0.006599613 0.0002071295    0.840663
```

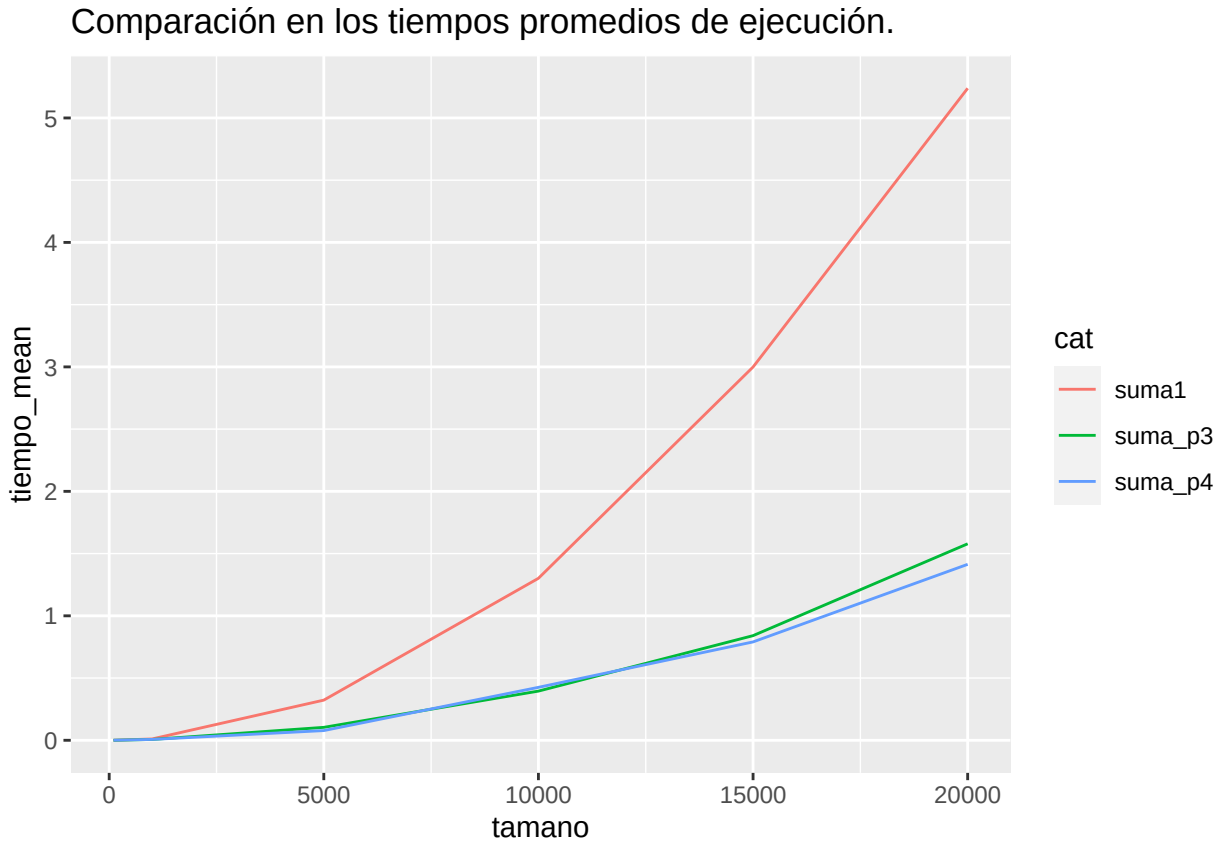
```
## 3    suma_p         4    0.425210 0.006894680 0.0003830400    0.790380
##   20000x20000 5000x5000
## 1     5.23824 0.3226485
## 2     1.57852 0.1038922
## 3     1.41359 0.0780401
```

Es decir,

Operación	Número de hilos	100	1000	5000	10000	15000	20000
secuencia		0.0001239058	0.010700375	0.3226485	1.301270	2.998980	5.23824
paralelo	3	0.0002071295	0.006599613	0.1038922	0.394985	0.840663	1.57852
paralelo	4	0.0003830400	0.006894680	0.0780401	0.425210	0.790380	1.41359

Observamos claramente que cuando el tamaño de la matriz es pequeño no existe diferencia entre los tiempos secuenciales y paralelos, incluso podemos decir que el secuencial es más rápido pero conforme el tamaño de la matriz se vuelve más grande cuando ocupamos 4 hilos es más rápido que ocupando 3 y ocupando de forma secuencial. Esto se explica muy sencillo, como tenemos n hilos realizando la suma de dos matrices, si partimos cada matriz de n formas para que cada hilo tengo dos submatrices con las cuales trabajar.

```
ggplot(datos_suma, aes(tamano,tiempo_mean, group=cat, color=cat))+
  geom_line()+
  labs(title="Comparación en los tiempos promedios de ejecución.")
```



No hay mucha diferencia con 3 y 4 hilos para los tamaños de muestras mostrados, sería interesante probar el tiempo de ejecución con un tamaño de las matrices mayor. Mi explicación sería que como no existen muchas operaciones, básicamente son 1 a 1 esto no hace que las operaciones crezcan exponencialmente.

Algo que note cuando ejecutaba la suma de dos matrices, es que la función paralelizada ocupaba mucha memoria RAM. No estoy muy seguro del por que pasa esto, ya que en con las otras operaciones no ocurría este aumento y en teoría trabajan de la misma forma.

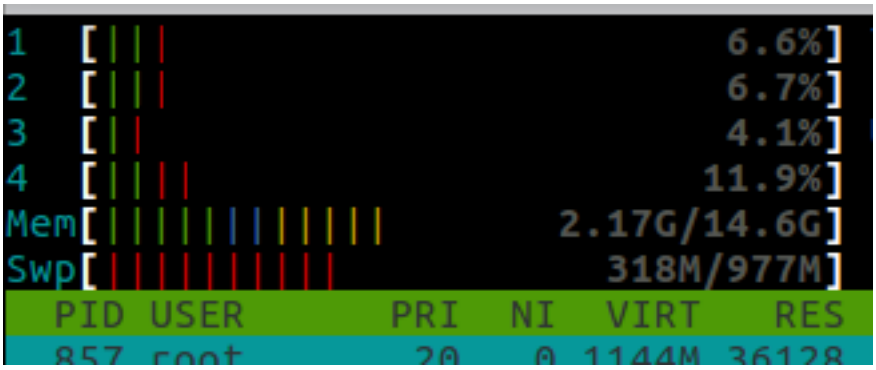


Figura 3: Memoria RAM antes de ejecutar las funciones.

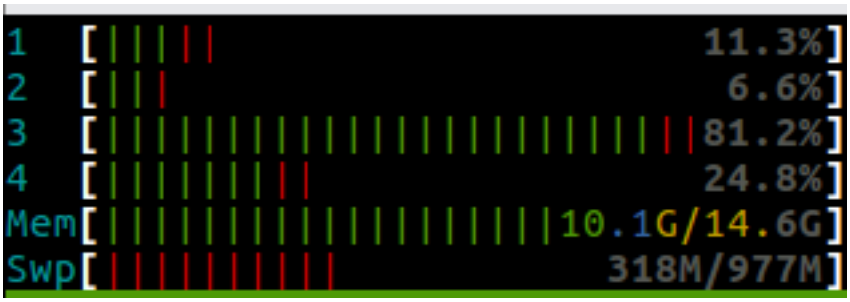


Figura 4: Memoria RAM durante la ejecución de las funciones

La función resta consiste en lo mismo que la suma (solo que en lugar de sumar se suma), por lo que omité este análisis ya que se esperan los mismos resultados.

- **Comparación de la Multiplicación.** Para este caso tenemos que la función para multiplicar dos matrices de forma secuencial y paralelo es:

```
// Función multiplicacion.
void multiplicacion_matriz(matriz mat_1, matriz mat_2, matriz resultado, int inf, int sup){  // sumamos dos matrices.
    for(int row = inf; row < sup; ++row) {
        for(int col = 0; col < resultado.m_columnas; ++col) {
            long int sum = 0.0;
            for(int k = 0; k < mat_1.m_columnas; ++k) {
                sum = sum + (mat_1.df[row][k] * mat_2.df[k][col]);
            }
        resultado.df[row][col] = sum;
        }
    }
}

// Función multiplicación para asignarla a un hilo.
void* multiplicacion_matriz_hilo(void *args) {
    Argumentos *_args = (Argumentos*) args;
    multiplicacion_matriz(_args->mat_1, _args->mat_2, _args->resultado, _args->inf, _args->sup);
    pthread_exit(NULL);
    return NULL;
}
```

Figura 5: Multiplicación matricial de forma secuencia y de forma paralelo.

Al igual que la función suma, se utiliza el mismo concepto en la parte secuencial y parte paralelo con la diferencia de los parámetros que determinan los limites de cada hilo. Para el caso secuencial no hay que dividir la matriz, por lo que se condirera $inf = 0$ y $sup = n$ (número de columnas de la matriz A) y la parte paralelo consideramos los mismo argumentos que los explicados en la suma. La idea del algoritmo paralelo es dividir las matrices a multiplicar entre submatrices más pequeñas (considerando el número de columnas de la primera matriz y el número de filas de la segunda matriz).

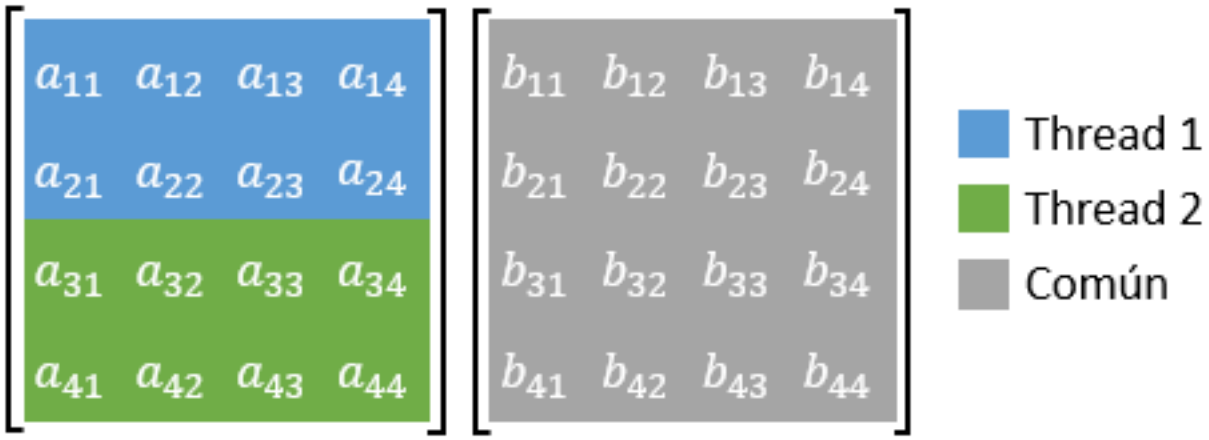


Figura 6: Idea de multiplicación en paralelo

A diferencia que la función suma, se consideraron tamaños de matrices más pequeñas ya que el tiempo de ejecución era extenso para probarlo con tamaños más grandes. Se consideraron 3 y 4 hilos en matrices cuadradas con tamaños: 100, 1000, 1500, 2500. Cada combinación de parámetros anteriores se simulación 4 veces y se obtuvo un tiempo promedio de ejecución. Los resultados son:

```
datos_multiplicacion <- datos %>%
  group_by(operacion, num_hilos, tamano_matriz, tamano) %>%
  summarise(tiempo_mean=mean(tiempo)) %>%
  filter(operacion %in% c("multiplicacion", "multiplicacion_p")) %>%
  mutate(cat=paste0(operacion, num_hilos))

datos_multiplicacion <- as.data.frame(datos_multiplicacion)

datos_multiplicacion %>%
  select(operacion, num_hilos ,tamano_matriz, tiempo_mean) %>%
  spread(key = "tamano_matriz", value = "tiempo_mean")
```

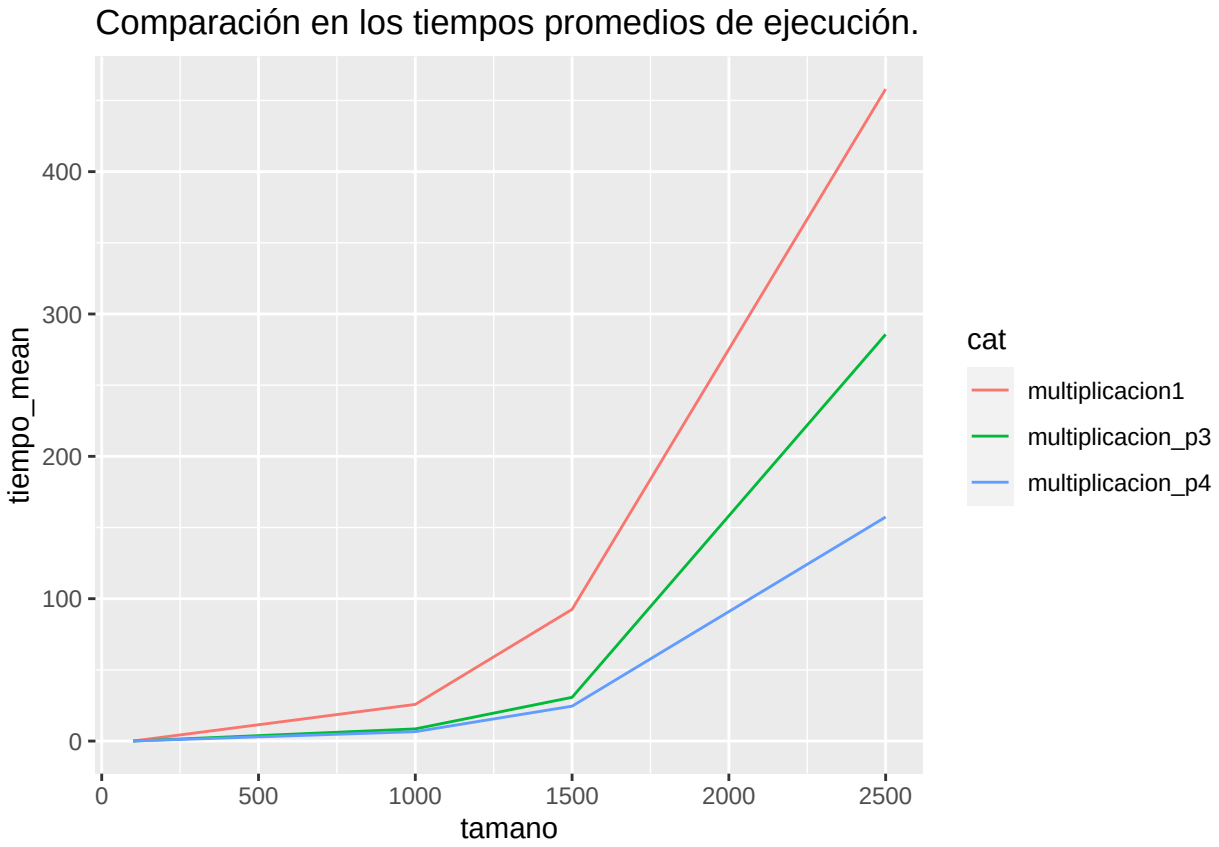
```
##          operacion num_hilos 1000x1000     100x100 1500x1500 2500x2500
## 1   multiplicacion         1 25.754225 0.012646850  92.47910   457.995
## 2 multiplicacion_p         3  8.536580 0.004705957  30.71770   285.659
## 3 multiplicacion_p         4  6.589602 0.005306000  24.49825   157.377
```

Es decir,

```
ggplot(datos_multiplicacion, aes(tamano,tiempo_mean, group=cat, color=cat))+
  geom_line()+
  labs(title="Comparación en los tiempos promedios de ejecución.")
```

Operación	Número de hilos	100	1000	1500	2500
secuencia		0.012646850	25.754225	92.47910	457.995
paralelo	3	0.004705957	8.536580	30.71770	285.659
paralelo	4	0.005306000	6.589602	24.49825	157.377

Cuadro 1: Resumen de tiempos multiplicación



Estos resultados me sorprendieron bastante, ya que a diferencia de la suma se observa una mayor diferencia en los tiempos de ejecución. Observamos claramente que cuando el tamaño de la matriz es pequeña la función paralela es más rápida que la secuencia, y conforme esta aumenta la diferencia se hace más notoria. Igual podemos notar que cuando el tamaño de la matriz es grande existe diferencias en los tiempos de ejecución de las funciones paralelas cuando se ocupan 3 y 4 hilos. La razón de estas diferencias es muy sencilla, una manera fácil de explicar: imaginemos que cada hilo estuviera realizando un multiplicación de submatrices de las matrices originales, es decir, partimos las matrices originales en submatrices las cuales cada hilo realizara la multiplicación de estas nuevas matrices por lo que la velocidad aumentara bastante.

- **Comparación de la Inversa.** Para este caso tenemos tenemos las funciones para calcular la inversa de una matriz cuadrada de forma secuencial y paralelo se pueden observar en las Figuras 7 y 8. Se ocupo la metodología de guass-jordan para determinar la inversa. La implementación de esta metodología consiste en 4 grandes pasos diferenciados. El primero de ellos es determinar el máximo valor que se encuentre en una columna el objetivo es siempre estar trabajando con el pivote más grande y que sea distinto de cero. Después de eso transformamos al pivote igual a 1, para ello dividimos todos los elementos del renglón en dónde se encuentre el pivote entre el pivote. Posteriormente restamos el renglón que contiene el pivote a los renglones que están abajo de el con el objetivo de tener ceros abajo del pivote. Lo anterior se itera por todas las columnas hasta que tengamos una matriz superior con unos en la diagonal. Y por último utilizamos eliminación hacia

atras hasta obtener la matriz identidad. Todo las operaciones realizadas a la matriz original también se le realizan a un matriz identidad inicial (matriz aumentada).

```
// Eliminación Gauss-Jordan.
matriz eliminacion_gauss_jordan(matriz mat){
    matriz aux_identidad = generar_matriz_identidad(mat.m_filas);
    int pivotet;
    if (mat.m_filas!= mat.m_columnas) {  // validamos si la matriz es cuadrada.
        cout << "La matriz no es cuadrada, por lo que no se puede calcular la inversa." << endl;
        return aux_identidad;  // retornamos la matriz aumentada
    }

    for (int i = 0; i < mat.m_filas; ++i) {
        pivotet = existencia_pivote(mat, i);
        if (pivotet==-1){   // validamos la existencia del pivote.
            cout << "La matriz no tiene inversa."<< endl;
            return aux_identidad;
        }
        else{      // intercambiamos el pivote de la matriz original y la matriz identidad.

            intercambia_pivote(mat , i, pivotet);
            intercambia_pivote(aux_identidad, i, existencia_pivote(mat, i));
        }

        dividir_matriz(mat, aux_identidad, i);
        eleminacion_parcial(mat, aux_identidad, i);
    }

    reduccion_hacia_atras(mat, aux_identidad);
    return aux_identidad;
}
```

Figura 7: Inversa secuencia

Esta operación matricial es un poco más complicada de paralelizar ya que tenemos que considerar que los resultados en algunos pasos perjudican las tareas de los otros hilos. Nosotros procedímos a paralelizar en 2 paso de la metodología planteada. A la hora de escoger el pivote máximo no es posible paralelizar (de forma fácil) ya que como deseamos obtener un máximo, si lo paralelizamos podemos obtener un máximo local por cada hilo lo que complica las cosas, por lo que este primer paso lo dejamos de igual forma. En segundo paso como depende del pivote elegido, entonces no podemos paralelizarlo. Una vez seleccionado el pivote, tenemos que esperar a que todos los hilos estén esperando el resultado de esta valor por lo que ocupamos una función para detener todas las tareas hasta que llegue el hilo que trabaja con el pivote máximo. El tercer paso como es resta el rengló del pivote a todos los inferiores a el, este paso si lo podemos paralelizar de forma sencilla ya que ocupamos la misma idea de sumar (o restan) dos matrices. Antes de avanzar al último paso, tenemos que volver a detener todas las tareas en este paso, ya que el resultado anterior se ocupa en los posteriores. Y por último, la eliminación hacia atras es la segunda parte que se paraleliza, ya que de igual manrea se ocupa la idea de suma (o restar) dos matrices pero en este caso son matrices de tamaño $1 \times m$.

Se consideraron 3 y 4 hilos en matrices cuadradas con tamaños: 100, 1000, 2000, 2500. Cada combinación de parámetros anteriores se simulación 4 veces y se obtuvo un tiempo promedio de ejecución. Los resultados obtenidos de las simulaciones se muestran en el Cuadro 3.

```
datos_inversa <- datos %>%
  group_by(operacion, num_hilos, tamano_matriz, tamano) %>%
  summarise(tiempo_mean=mean(tiempo)) %>%
  filter(operacion %in% c("inversa", "inversa_p")) %>%
  mutate(cat=paste0(operacion, num_hilos))
```

```
// Eliminación Gauss-Jordan.
int eliminacion_gauss_jordan_p(matriz mat, matriz aux_identidad, int NTHREADS, int rank){
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // inicializamos el objeto tipo mutex
    float tmp; // variables auxiliares
    int k;
    int first_row; // limites para las tareas de los hilos
    int last_row;
    for (int i = 0; i < mat.m_filas; ++i) {
        if (rank==0){
            k=i;
            for(int j=i+1; j < mat.m_filas; j++){
                if(abs(mat.df[k][i]<abs(mat.df[j][i]))){
                    k=j; // elegimos el renglón con el pivote más grande.
                }
            }

            intercambia_pivote_p(mat , i, k); // actualizamos el pivote.
            intercambia_pivote_p(aux_identidad, i, k); // actualizamos la identidad

            tmp = 1.0/mat.df[i][i]; // valor del pivote
            dividir_matriz_pivote_p(mat, aux_identidad, i, tmp); // dividimos sobre el pivote.
        }

        synchronize(NTHREADS); // esperamos a que todos los hilos llegue a este punto.

        first_row = (mat.m_filas- i - 1) * rank; // repartimos los limites de los hilos
        first_row = first_row/NTHREADS + i + 1;
        last_row = (mat.m_filas - i - 1) * (rank + 1);
        last_row = last_row/NTHREADS + i + 1;

        eliminacion_parcial_p(mat, aux_identidad, i, first_row, last_row); // sumamos el renglon del pivote
        synchronize(NTHREADS); // esperamos a que todos los hilos llegue a este punto.
    }
```

Figura 8: Inversa secuencia

```
datos_inversa <- as.data.frame(datos_inversa)

datos_inversa %>%
  select(operacion, num_hilos ,tamano_matriz, tiempo_mean) %>%
  spread(key = "tamano_matriz", value = "tiempo_mean")
```

```
##   operacion num_hilos 1000x1000    100x100 2000x2000 2500x2500
## 1   inversa         1 19.105250 0.02113908 152.74700  297.5100
## 2 inversa_p         3  7.810767 0.02152265  65.61510  128.6240
## 3 inversa_p         4  6.552650 0.08356430  52.77545   99.5397
```

Es decir,

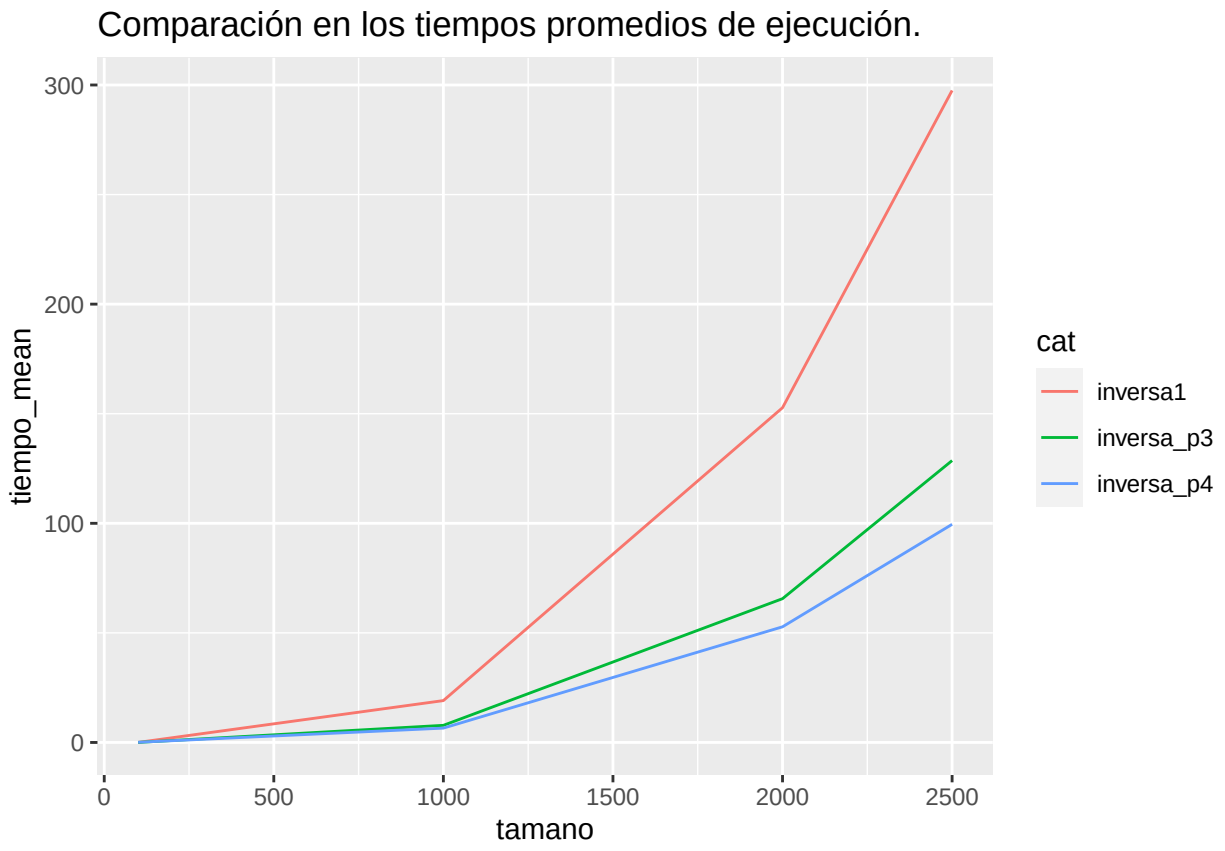
Operación	Número de hilos	100	1000	2000	2500
secuencia		0.02113908	19.105250	152.74700	297.5100
paralelo	3	0.02152265	7.810767	65.61510	128.6240
paralelo	4	0.08356430	6.552650	52.77545	99.5397

Cuadro 2: Resumen de tiempos inversa

Al igual que las operaciones anteriores observamos que los tiempos para tamaños de muestras pequeños son muy parecidos. Pero cuando se comparan los tiempos para tamaños de matrices grandes observamos una gran diferencia entre cada uno de las funciones, el secuencia es el que tarda más tiempo lo cual no es nada inusual. Después el paralelo utilizando 3 hillos es más lento y por último el más rápido es cuando se utilizando 4 hilos. Que es consistente con lo observado en las operaciones anteriores.

Algo que me hubiera probar es aumentar el número de hilos, pero como mi maquina no es tan potente no pude intentarlo. Ya que en teoría a más hilos se esperaría un tiempo de ejecución, pero por lo visto en clase también llegaremos a un punto en donde el número de hilos ya provocara un impacto en la velocidad.

```
ggplot(datos_inversa, aes(tamano,tiempo_mean, group=cat, color=cat))+
  geom_line()+
  labs(title="Comparación en los tiempos promedios de ejecución.")
```



Concluciones

Se ha probado con simulación las diferencias en los tiempos de ejecución para algoritmos de forma secuencia y paralelo. Vemos la importancia de usar una algoritmo en paralelo cuando se ocupan gran cantidad de operaciones (o datos), ya que para casos en donde se trabajan con pocas operaciones (o datos) el rendimiento de los tiempos es muy cercano al secuencial y si consideramos el tiempo de construcción de cada algoritmos, no tendría sentido el paralelo. Por otro lado, la cantidad de hilos impacta considerablemente en lso tiempos de ejecución para algoritmos paralelos. No pudimos observar la convergencia del número total de hilos a utilizar vs el rendimiento, pero sería adecuado mencionar que llegarás a un número de hilos en donde el impacto en la velocidad es practicamente nula.