Bueno Osorio Mauricio
Espinosa Guarin Nelson Esled
Lopez Garcia Yoan Esteban
Narvaez Erazo Luba Carolina

Análisis de rendimiento de la multiplicación de matrices de manera secuencial, por procesos (fork) e hilos.

21 de Septiembre del 2021

Presentado a:

Ramiro Andrés Barrios Valencia.

Asignatura:

High Performance Computing.

Universidad Tecnológica de Pereira
Programa de Ingeniería de Sistemas y Computación
Pereira, 2021

Resumen

Al momento de tomar la mejor decisión para realizar una operación como lo es la multiplicación de matrices es necesario poder analizar algunas de las posibles implementaciones y ver cual es la más óptima en términos de ejecución.

Para términos del desarrollo de este laboratorio se hizo un enfoque en la implementación secuencial, la implementación por hilos y la implementación por procesos, en el primer caso se hizo uso de ciclos que recorren por completo las matrices, en el segundo caso se hizo uso de la librería pthread y en el último se usó la función Fork() para crear la cantidad de procesos definidos.

Palabras Clave

Hilos, procesos, secuencial, librería,función, pthread, fork.

Abstract

When making the best decision to perform an operation such as matrix multiplication, it is necessary to be able to analyze some of the possible implementations and see which is the most optimal in terms of execution.

In terms of the development of this laboratory, a focus was made on sequential implementation, implementation by threads and implementation by processes, in the first case, cycles that completely traversed the matrices were used, in the second case, use was made of the pthread library and in the last one the Fork () function was used to create the processes.

Keywords

Threads, processes, sequential, library, function, pthread, fork.

1. Introducción

Este documento es la continuación de un primer análisis hecho entre una implementación secuencial y por hilos, en esta ocasión se hará una comparativa entre el tiempo de ejecución de manera secuencial, por hilos y por procesos.

Para lograr crear los procesos a bajo nivel se hizo uso de la función ***fork()*** la cual permite crear 2 procesos uno padre y otro hijo, finalmente para crear un hijo se implementaron bucles.

2. Marco Teórico.

Procesos.

Un proceso es cualquier programa en ejecución. Este necesita ciertos recursos para realizar satisfactoriamente su tarea:

- Tiempo de CPU.
- Memoria
- Archivos.
- Dispositivos de E/S.

Las obligaciones del SO como gestor de procesos son:

- Creación y eliminación de procesos.
- Planificación de procesos (procurando la ejecución de múltiples procesos maximizando la utilización del procesador).
- Establecimiento de mecanismos para la sincronización y comunicación de procesos.
- Manejo de bloqueos mutuos.

Estado de un proceso.

A medida que un proceso se ejecuta cambia de estado. Cada proceso puede estar en uno de los siguientes estados:

- **Nuevo (new):** El proceso se está creando.
- **En ejecución (running):** El proceso está en la CPU ejecutando instrucciones.
- **Bloqueado (waiting):** Proceso esperando a que ocurra un suceso.
- **Preparado (ready):** Esperando que se asigne un procesador.

- **Terminado (terminated):** Finalizó su ejecución, por tanto no ejecuta más instrucciones y el SO le retirará los recursos que consume.

Para que un programa se ejecute, el SO debe crear un proceso para él. En un sistema con multiprogramación el procesador ejecuta código de distintos programas que pertenecen a distintos procesos.

Creación de procesos.

Los procesos se crean mediante una llamada al sistema de “crear proceso”, durante el curso de su ejecución. El proceso creador se denomina proceso padre, y el nuevo proceso, hijo.

Cuando un proceso crea un proceso nuevo, hay dos posibilidades en términos de ejecución:

- Padre e hijo se ejecutan concurrentemente
- Padre espera por la finalización del hijo.

Función Fork.

Cuando se llama la función ***fork*** esta genera un duplicado del proceso actual. El duplicado comparte los valores actuales de todas las variables, ficheros y otras estructuras de datos. La llamada a ***fork*** retorna al proceso padre el identificador del proceso hijo y retorna un cero al proceso hijo.

PID.

Las siglas PID, del inglés Process IDentifier, se usan para referirse al identificador del proceso.

Hilos.

Los hilos son un concepto relativamente nuevo de los SO. En este contexto, un proceso recibe el nombre de proceso pesado, mientras que un hilo recibe el nombre de proceso ligero. El término hilo se refiere sintáctica y semánticamente a hilos de ejecución.

El término multihilo hace referencia a la capacidad de un SO para mantener varios hilos de ejecución dentro del mismo proceso.

En un SO con procesos monohilo (un solo hilo de ejecución por proceso), en el que no existe el concepto de hilo, la representación de un proceso incluye su PCB, un espacio de direcciones del proceso, una pila de proceso y una pila núcleo.

En un SO con procesos multihilo, sólo hay un PCB y un espacio de direcciones asociados al proceso, sin embargo, ahora hay pilas separadas para cada hilo y bloques de control para cada hilo.

Estructura de los hilos.

Un hilo (proceso ligero) es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila.

Los hilos dentro de una misma aplicación comparten:

- La sección de código.
- La sección de datos.
- Los recursos del SO.

Un proceso tradicional o pesado es igual a una tarea con un solo hilo.

Los hilos permiten la ejecución concurrente de varias secuencias de instrucciones asociadas a diferentes funciones dentro de un mismo proceso, compartiendo un mismo espacio de direcciones y las mismas estructuras de datos del núcleo.

Memoria Compartida.

La memoria compartida (Shared Memory) es una forma implícita de comunicación entre procesos (IPC). Es, por tanto, un método altamente eficiente para compartir información entre procesos.

Los procesos pueden comunicarse directamente entre sí compartiendo partes de su espacio de direccionamiento virtual, por lo que podrán leer y/o escribir datos en la memoria compartida. Para conseguirlo, se crea una región o segmento fuera del espacio de direccionamiento de un proceso y cada proceso que necesite acceder a dicha región, la incluirá como parte de su espacio de direccionamiento virtual.

El sistema permite que existan varias regiones compatibles, cada una compartida por un subconjunto de procesos. Además, cada proceso puede acceder a varias regiones

Para trabajar con memoria compartida se utilizan básicamente las siguientes llamadas al sistema:

- **Shmget:** Crea una nueva región de memoria compartida o devuelve una existente.
- **shmat:** Une lógicamente una región al espacio de direccionamiento virtual de un proceso.
- **shmdt:** Separa una región del espacio de direccionamiento virtual de un proceso.
- **shmctl:** Manipula varios parámetros asociados con la memoria compartida.

Para leer y escribir en la memoria compartida, un proceso utiliza las mismas instrucciones que en el caso de memoria no compartida, pero debe haber incluido la región en su espacio de direccionamiento.

3. Marco contextual

Características del entorno de desarrollo.

El programa desarrollado se ejecutó en una máquina con las siguientes características:

- Intel(R) Core(TM) i3-2120 CPU
- 4 núcleos 4 hilos
- Taza de frecuencia 3.3 GHz
- 8 GB RAM
- SO Linux

Desarrollo.

Para el análisis de este laboratorio se implementó el código de multiplicación de matrices *nxn* en lenguaje C, este programa llena de manera aleatoria una matriz cuadrada con numeros que estan en un rango 0 - 10000, las variables que hace relación a la cantidad de hilos y tamaño de la matriz fueron parametrizados.

En esta ocasión se tomó como base el código secuencial de multiplicación de matrices para crear los procesos (padre - hijo) con la función fork(), debido a que en este caso los datos no son compartidos porque los procesos no comparten memoria, fue necesario hacer uso de la memoria compartida la cual permite que el proceso padre acceda a los datos de los procesos hijos.

Por otro lado para realizar el ciclo de ejecución de las diferentes matrices con sus respectivos tamaños, se implementó un script que facilitó el proceso de toma de tiempos de ejecución (7 datos de prueba es decir tamaños, cada uno 10 veces), esto está debidamente consignado en las tablas 1, 2, 3, 4 y 5.

En la figura 1. y en la figura 2. se puede visualizar el proceso secuencial y el paralelo. En la figura 3. se muestra el segmento de código del script.

```
clock_t start = clock();

for (int  c = 0; c < N; c++)
    {
        for(int b=0; b<N; b++)
        {
            matrix_3[c][b] = 0;
            for (int a = 0; a < N; a++)
                {
                    matrix_3[c][b] = matrix_3[c][b] + (matrix_1[c][a] * matrix_2[a][b]);
                }
        }
    }

clock_t end = clock();
double elapsed = (double)(end - start)/CLOCKS_PER_SEC;
```

Figura 1. Código secuencial

```
clock_t start = clock();

for (int i = 0; i < numHilos; i++)
{
    pthread_create(&misHilos[i], NULL, (void *) multi, i*SEGMENTO);
}

for (int i = 0; i < numHilos; i++)
{
    pthread_join(misHilos[i], NULL);
}

clock_t end = clock();

double elapsed = (double)(end - start)/CLOCKS_PER_SEC;
```

Figura 2. Código con hilos.

```
73      clock_gettime(CLOCK_REALTIME, &start2);
74      for (int i = 0; i < numHijos; i++)
75      {
76          shift = i*SEGMENTO;
77          pidC = fork();
78          if(pidC==0){
79              int a, b, c;
80
81              for (c = shift; c < SEGMENTO+shift; c++)
82                  {
83                      for(b = 0; b < TAMANIO; b++)
84                      {
85                          matrix[c][b] = 0;
86                          for (a = 0; a < TAMANIO; a++)
87                              {
88                                  matrix[c][b] = matrix[c][b] + (matrixA[c][a] * matrixB[a][b]);
89                              }
90                      }
91                  }
92              shmdt(matrix);
93              exit(EXIT_SUCCESS);
94          }
95      }
96
97      wait(NULL);
```

Figura 3. Código con procesos.

```
48
49      size_t sizeMatrix = sizeof_dm(TAMANIO,TAMANIO,sizeof(int));
50      shmId = shmget(IPC_PRIVATE, sizeMatrix, IPC_CREAT|0600);
51      int shmId2 = shmget(IPC_PRIVATE, sizeMatrix, IPC_CREAT|0600);
52      int shmId3 = shmget(IPC_PRIVATE, sizeMatrix, IPC_CREAT|0600);
53
54      matrix = shmat(shmId, NULL, 0);
55      create_index((void*)matrix, TAMANIO, TAMANIO, sizeof(int));
56      matrixA = shmat(shmId2, NULL, 0);
57      create_index((void*)matrixA, TAMANIO, TAMANIO, sizeof(int));
58      matrixB = shmat(shmId3, NULL, 0);
59      create_index((void*)matrixB, TAMANIO, TAMANIO, sizeof(int));
```

Figura 4. Memoria Compartida

```
Actividad03 > $_ script.sh
1  #!/bin/bash
2
3  lista=( 100 200 400 800 1200 1600 2000 );
4
5
6  for i in ${lista[@]};
7      do
8      variable="$i,"
9      for (( j=0; j<10; j++ ));
10         do
11         variable+="$(./mxm.out $i),"
12         done
13     echo $variable>>result.csv
14     done
```

Figura 5. Código Script

[5] Se anexa link del repositorio creado.

Análisis de resultados.

Para lograr determinar la implementación óptima, se analizó el rendimiento de las tres implementaciones, el tiempo de ejecución de cada una se midió sometiendo cada una a los mismos parámetros, toda esta información de los promedios obtenidos se consignó en las tablas respectivas.

Tamaño de matriz	Tiempo de ejecución serial										Promedio
100	0.007	0.009	0.009	0.008	0.009	0.009	0.009	0.008	0.009	0.006	0.0083
200	0.049	0.046	0.048	0.043	0.042	0.042	0.042	0.043	0.046	0.042	0.0443
400	0.431	0.411	0.415	0.475	0.407	0.535	0.586	0.431	0.429	0.428	0.4548
800	5.283	4.687	4.585	4.611	4.695	4.704	4.647	4.67	4.629	4.619	4.713
1200	20.662	20.579	20.687	21.098	20.81	20.915	21.215	20.722	21.08	20.788	20.855
1600	50.395	51.845	51.119	51.19	50.79	51.283	50.529	50.793	50.162	51.374	50.948
2000	102.14	101.38	101.59	101.54	103.66	107.02	110.05	105.06	101.71	103.27	103.74

Tabla 1. Tiempo de ejecución secuencial.

Tamaño de matriz	Tiempo de ejecución con 2 hilos.										Promedio
100	0.01	0.008	0.008	0.009	0.009	0.01	0.008	0.009	0.009	0.009	0.0089
200	0.071	0.065	0.054	0.056	0.049	0.051	0.06	0.05	0.086	0.052	0.0594
400	0.475	0.53	0.741	0.555	0.522	0.504	0.508	0.557	0.731	0.518	0.5641
800	7.214	8.037	7.232	7.243	6.604	6.586	6.635	7.163	6.675	6.98	7.0369
1200	33.225	28.871	27.484	24.663	26.245	26.922	27.62	26.027	25.83	27.466	27.435
1600	64.254	65.917	63.055	68.229	65.856	64.877	61.76	68.136	68.128	66.141	65.635
2000	129.17	124.43	135.16	131.55	129.85	132.09	127.87	126.66	131.25	128.35	129.64

Tabla 2. Tiempo de ejecución con 2 hilos

Tamaño de matriz	Tiempo de ejecución 4 hilos										Promedio
100	0.011	0.01	0.009	0.008	0.01	0.01	0.009	0.01	0.009	0.011	0.0097
200	0.073	0.07	0.068	0.083	0.082	0.081	0.075	0.08	0.081	0.08	0.0773
400	0.788	0.745	0.698	0.758	0.852	0.793	0.695	0.729	0.755	0.755	0.7568
800	8.525	8.449	8.533	8.401	8.561	8.723	8.601	8.652	8.506	8.531	8.5482
1200	32.736	32.292	32.736	32.37	32.559	33.367	33.994	30.74	30.684	30.841	32.2319
1600	78.348	75.013	75.829	75.277	75.345	75.169	77.299	76.264	75.326	75.131	75.9001
2000	151.05	151.23	151.98	154.19	153.66	151.24	152.88	150.45	149.83	153.38	151.992

Tabla 3. Tiempo de ejecución 4 hilos.

Tamaño de la matriz	Tiempo de ejecución 2 procesos										Promedio
100	0.004	0.006	0.006	0.005	0.005	0.004	0.009	0.005	0.006	0.006	0.0056
200	0.042	0.05	0.045	0.048	0.028	0.029	0.03	0.027	0.026	0.044	0.0369
400	0.255	0.36	0.307	0.324	0.293	0.29	0.363	0.284	0.258	0.368	0.3102
800	4.235	3.202	3.659	2.949	2.965	3.562	3.152	4.097	3.242	3.225	3.4288
1200	11.682	12.999	10.237	10.695	9.79	10.158	10.414	10.461	10.375	10.341	10.7152
1600	27.836	27.227	29.544	31.17	30.325	27.691	28.439	29.686	26.765	28.724	28.7407
2000	52.945	50.4	53.666	56.457	57.357	54.753	55.748	52.454	54.039	54.681	54.25

Tabla 4. Tiempo de ejecución 2 procesos.

Tamaño de matriz	Tiempo de ejecución con 4 procesos										Promedio
100	0.003	0.003	0.003	0.003	0.003	0.002	0.003	0.002	0.002	0.003	0.0027
200	0.046	0.023	0.031	0.023	0.014	0.019	0.022	0.023	0.023	0.023	0.0247
400	0.216	0.218	0.212	0.209	0.218	0.249	0.259	0.286	0.27	0.213	0.235
800	2.462	2.226	2.334	2.378	2.299	2.46	2.298	2.443	2.23	2.079	2.3209
1200	8.875	8.863	9.064	9.754	9.297	10.108	9.574	9.207	8.845	8.787	9.2374
1600	23.575	23.087	24.821	22.954	23.801	23.004	24.509	22.643	23.553	23.307	23.525
2000	48.466	51.149	71.022	59.379	51.706	51.92	51.903	51.864	53.437	55.648	54.649

Tabla 5. Tiempo de ejecución 4 procesos.

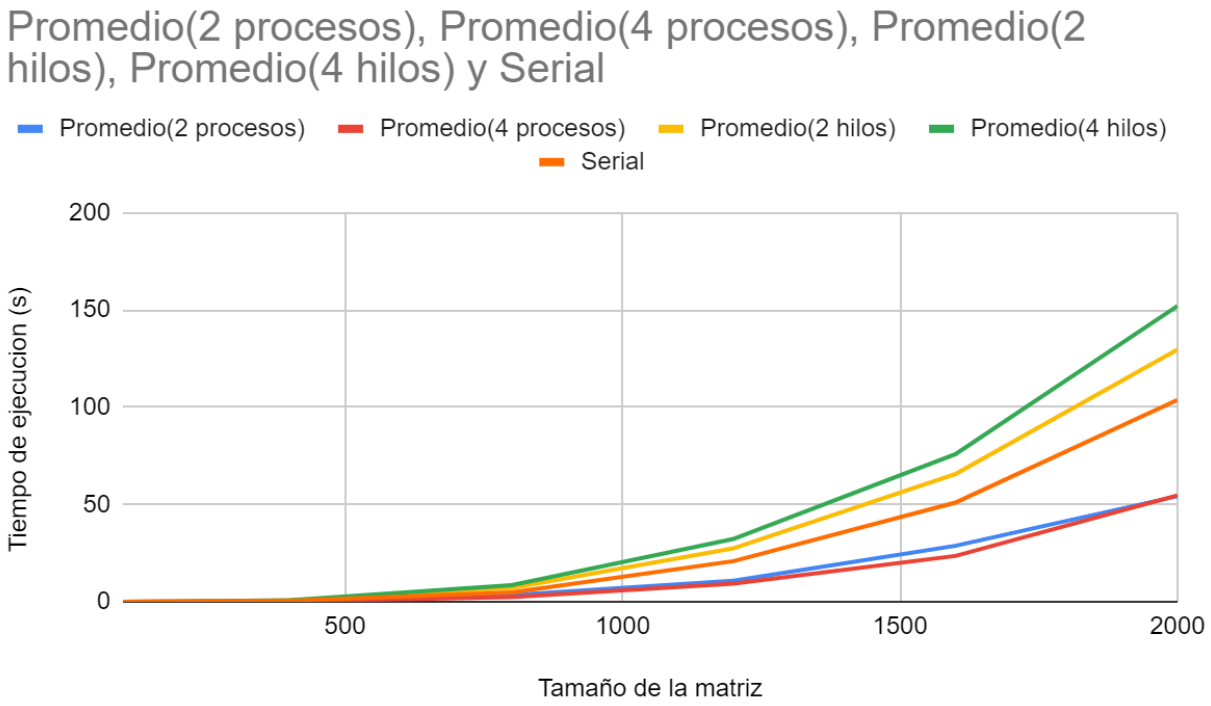

Gráfico 1. Tiempo de ejecución en función del tamaño de la matriz, secuencial, hilos y procesos.

El anterior gráfico permite visualizar cómo fue el comportamiento del programa a la hora de la ejecución serial, concurrente con 2 y 4 procesos y con 2 y 4 hilos. Se puede apreciar claramente que la implementación más óptima en cuanto a tiempo de ejecución es la de procesos, por otro lado se puede observar que enfocado en las dos pruebas hechas con 2 y 4 procesos la gráfica muestra una ligera diferencia en tiempo de ejecución.

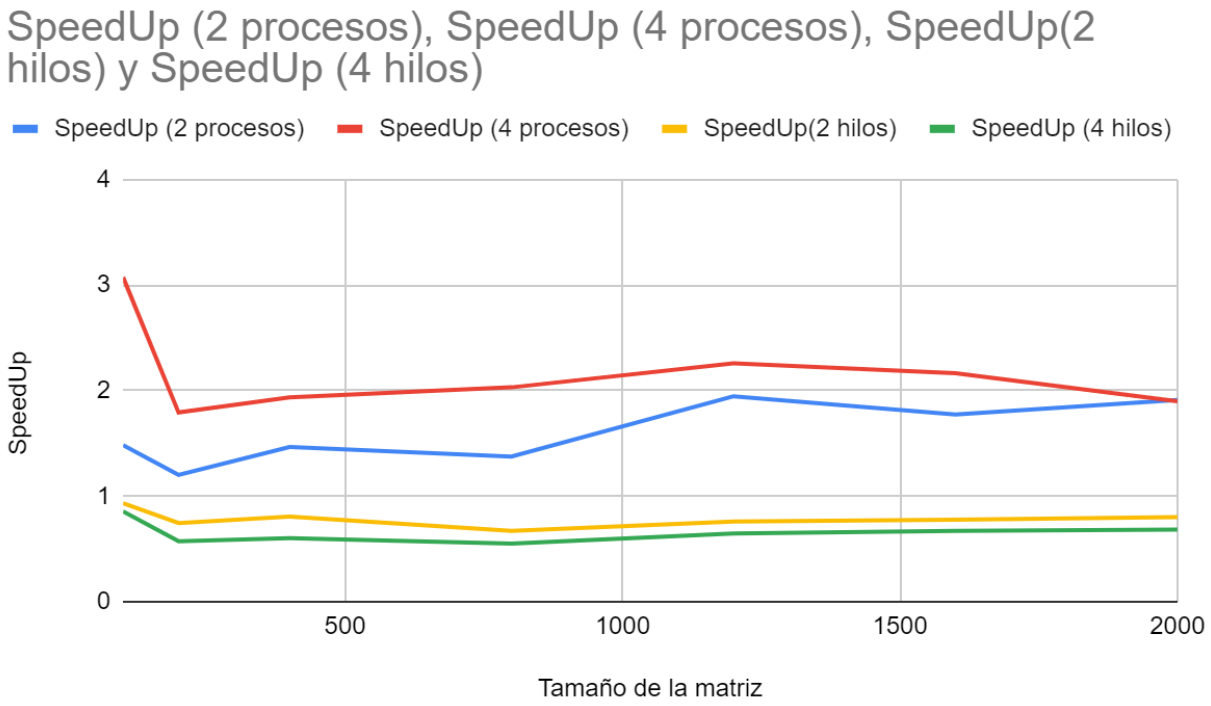

Gráfico 2. SpeedUP, hilos y procesos.

En la anterior gráfica se aprecia el cálculo de la aceleración observada de los algoritmos paralelos frente a la versión secuencial, la implementación por procesos arroja un speedup mucho más satisfactorio lo que hace incidir que la optimización fue exitosa.

Conclusiones.

Durante la actividad se observó un claro rendimiento al implementar procesos en comparación con las anteriores implementaciones (serial e hilos), aunque si se abusa de la cantidad de procesos este rendimiento se puede ver afectado por los cambios de contexto. Esta práctica tuvo una complejidad mayor a las prácticas anteriores debido a que los procesos no comparten memoria y es estrictamente necesario que estos procesos se comuniquen entre sí, por tal motivo inicialmente implementamos tuberías para lograr la comunicación pero se nos presentaron algunos inconvenientes por el tamaño de las matrices, debido a esto decidimos implementar algoritmos y funciones para compartir la memoria de cada uno de los procesos.

Debido a la naturaleza de los procesos la toma de tiempos cambió con respecto a anteriores implementaciones, ya que la función que teníamos previamente sólo registraba el tiempo del proceso padre y no tenía en cuenta los procesos hijos.

Bibliografía.

[1] Lasha Khintibidze, “Medir el tiempo del sistema con la función getrusage en C,” Delft Stack, Mar. 30, 2021. https://www.delftstack.com/es/howto/c/getrusage-example-in-c/ (accessed Sep. 22, 2021).

[2] “Introduction to Parallel Computing Tutorial | High Performance Computing,” Llnl.gov, 2018. https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial (accessed Sep. 10, 2021)

[3] K. Köhntopp, “fork, exec, wait and exit,” Percona.community, Jan. 04, 2021. https://percona.community/blog/2021/01/04/fork-exec-wait-and-exit/ (accessed Sep. 22, 2021).

[4] user1904731, “Using shared memory with matrices,” Stack Overflow, May 04, 2013. https://stackoverflow.com/questions/16375894/using-shared-memory-with-matrices (accessed Sep. 22, 2021).

Anexo del repositorio del código implementado

[5] ZeldrisG, “HPC/Actividad04 at main · ZeldrisG/HPC,” GitHub, Sep. 22, 2021. https://github.com/ZeldrisG/HPC/tree/main/Actividad04 (accessed Sep. 22, 2021).

.