TALLER DE EVALUACIÓN DE RENDIMIENTO

Ángel Daniel García Santana, Samuel Montaña, Mateo Guerra, Elier Ibarra

Pontificia Universidad Javeriana, Bogotá D.C, Colombia
Facultad de Ingeniería de Sistemas
Sistemas Operativos
13 de Noviembre de 2025

1 Introducción

Este documento presenta los resultados del taller de evaluación de rendimiento de algoritmos de multiplicación de matrices, comparando implementaciones en serie y paralelas utilizando diferentes técnicas de concurrencia en sistemas operativos Linux.

2 Metodología Experimental

2.1. Sistemas de Cómputo Utilizados

2.1.1. Máquinas Virtuales (3 sistemas)

- Arquitectura: x86_64.
- CPU: Intel Xeon Gold 6348 @ 2.60GHz.
- Núcleos: 4 cores, 4 hilos totales.
- Memoria RAM: 11.96 GB.
- Cachés: L1d 192KB, L1i 128KB, L2 5MB, L3 42MB.
- Hipervisor: VMware.

2.1.2. Computador Personal

- Arquitectura: x86_64.
- CPU: AMD Ryzen 5 7535HS con Radeon Graphics.
- Núcleos: 6 cores, 12 hilos totales (SMT habilitado).
- Memoria RAM: 14.74 GB.
- Cachés: L1d 192KB, L1i 192KB, L2 3MB, L3 16MB.
- Virtualización: AMD-V.

2.2 Algoritmos Evaluados

Se implementaron y evaluaron tres versiones del algoritmo de multiplicación de matrices:

1. mmClasicaFork: Paralelización usando procesos Fork.
2. mmClasicaPosix: Paralelización usando hilos POSIX.
3. mmClasicaOpenMP: Paralelización usando OpenMP.

3. Análisis de Algoritmos Implementados

3.1 mmClasicaFork paralelización con procesos fork

Utiliza creación de procesos mediante fork() para distribución de trabajo.

Función de multiplicación

```
void multiMatrix(double *mA, double *mB, double *mC, int D,
int filaI, int filaF) {
    double Suma, *pA, *pB;
    for (int i = filaI; i < filaF; i++) {
        for (int j = 0; j < D; j++) {
            Suma = 0.0;
            pA = mA+i*D;
            pB = mB+j;
            for (int k = 0; k < D; k++, pA++, pB+=D) {
                Suma += *pA * *pB;
            }
            mC[i*D+j] = Suma;
        }
    }
}
```

Parámetros: filaI y filaF definen el rango de trabajo y cada proceso calcula un bloque contiguo de filas.

Creación de y gestión de procesos

```
for (int i = 0; i < num_P; i++) {
    pid_t pid = fork();

    if (pid == 0) {
        // Código del proceso hijo
        int start_row = i * rows_per_process;
        int end_row = (i == num_P - 1) ? N : start_row +
rows_per_process;
        multiMatrix(matA, matB, matC, N, start_row,
end_row);
        exit(0);
    }
}
```

- Distribución: División equitativa de filas entre procesos.
- Proceso final: Maneja residuo si N no es divisible entre num_P.
- Sincronización: wait(NULL) para esperar la finalización de todos los hijos.

Consideraciones de memoria

- COW (Copy-On-Write): Los procesos hijos comparten memoria inicialmente.
- Escribir: Causa duplicación de páginas, aumentando el uso de memoria.

3.2 mmClasicaPosix paralelización con pthreads

Estructura de datos para hilos

```
struct parametros{
   int nH;     // Número total de hilos
   int idH;    // Identificador del hilo
   int N;      // Dimensión de la matriz
};
```

- Propósito: Paso de parámetros a función de hilos.
- Contenido: Información necesaria para distribución de trabajo.

Función del hilo:

```
void *multiMatrix(void *variables){
    struct parametros *data = (struct parametros
*)variables;

    int idH = data->idH;
    int nH = data->nH;
    int D = data->N;
    int filaI = (D/nH)*idH;
    int filaF = (D/nH)*(idH+1);

    // Algoritmo de multiplicación (similar a otras
versiones)
    // ...

    pthread_exit(NULL);
}
```

- Cálculo de rangos: Similar a versión Fork pero dentro del mismo proceso.
- Compartición: Todos los hilos acceden directamente a matrices globales.

Gestión de hilos:

```
pthread_mutex_init(&MM_mutex, NULL);
pthread_attr_init(&atrMM);
pthread_attr_setdetachstate(&atrMM,
PTHREAD_CREATE_JOINABLE);

for (int j=0; j<n_threads; j++){
    struct parametros *datos = malloc(sizeof(struct
parametros));
    // Inicialización de parámetros
    pthread_create(&p[j], &atrMM, multiMatrix, (void
*)datos);
}

for (int j=0; j<n_threads; j++)
    pthread_join(p[j], NULL);
```

- Atributos: Hilos joinable para sincronización.
- Memoria: Asignación dinámica para estructura de parámetros.
- Sincronización: pthread_join para esperar finalización.

3.3 mmClasicaOpenMP paralelización con OpenMP

Este programa implementa la multiplicación de matrices utilizando directivas de OpenMP para la paralelización automática.

Funciones principales

```
void InicioMuestra(){
    gettimeofday(&inicio, (void *)0);
}

void FinMuestra(){
    gettimeofday(&fin, (void *)0);
    fin.tv_usec -= inicio.tv_usec;
    fin.tv_sec  -= inicio.tv_sec;
    double tiempo = (double) (fin.tv_sec*1000000 +
fin.tv_usec);
    printf("%9.0f \n", tiempo);
}
```

- Propósito: Medición precisa del tiempo de ejecución.
- Mecanismo: Utiliza gettimeofday() para capturar tiempo con resolución de microsegundos.
- Cálculo: Diferencia entre inicio y fin, convertida a microsegundos.

```
void iniMatrix(double *m1, double *m2, int D){
    for(int i=0; i<D*D; i++, m1++, m2++){
        *m1 = (double)rand()/RAND_MAX*(5.0-1.0);
        *m2 = (double)rand()/RAND_MAX*(9.0-2.0);
    }
}
```

- Propósito: Inicialización de matrices con valores aleatorios.
- Rango: Matriz A (1.0-5.0), Matriz B (2.0-9.0).
- Distribución: Valores flotantes uniformemente distribuidos.

```
void multiMatrix(double *mA, double *mB, double *mC, int D){
    double Suma, *pA, *pB;
    #pragma omp parallel
    {
    #pragma omp for
    for(int i=0; i<D; i++){
        for(int j=0; j<D; j++){
            pA = mA+i*D;
            pB = mB+j;
            Suma = 0.0;
            for(int k=0; k<D; k++, pA++, pB+=D){
                Suma += *pA * *pB;
            }
            mC[i*D+j] = Suma;
        }
    }
    }
}
```

- Algoritmo: Multiplicación clásica $O(n^3)$.
- Paralelización: Directiva #pragma omp parallel for.
- Estrategia: Descomposición por filas.
- Acceso a memoria: Patrón secuencial para optimizar caché.

3.5 Características comunes

Los tres programas implementan el algoritmo clásico de multiplicación de matrices

```
Para cada fila i en matriz A
    Para cada columna j en matriz B
        suma = 0
        Para cada elemento k
            suma += A[i][k] * B[k][j]
        C[i][j] = suma
```

Complejidad: $O(n^3)$ operaciones.

Gestión de memoria

```
double *matrixA = (double *)calloc(N*N, sizeof(double));
// Uso de matrices...
free(matrixA);
```

- Asignación: Uso de calloc para inicialización a cero
- Almacenamiento: Matrices unidimensionales con indexación manual
- Liberación: Correcta liberación de memoria al finalizar

Verificación y visualización

```
void impMatrix(double *matrix, int D){
    if(D < 9){
        printf("\n");
        for(int i=0; i<D*D; i++){
            if(i%D==0) printf("\n");
            printf("%.2f ", matrix[i]);
        }
        printf("\n**----------------------------**\n");
    }
}
```

- Condicional: Solo muestra matrices pequeñas (D < 9)
- Formato: Impresión organizada en forma matricial
- Propósito: Verificación visual de resultados correctos

4. Diseño experimental

Parámetros de evaluación

- Tamaños de matriz: 1000×1000, 2000×2000, 4000×4000.
- Número de hilos: 1, 2, 4, 8, 16.
- Repeticiones: 30 ejecuciones por configuración.

Justificación de parámetros

- Tamaños de matriz: Seleccionados para evaluar desde cargas moderadas hasta intensivas, considerando la jerarquía de memoria.

- Número de hilos: Cubre desde ejecución serial hasta paralelismo excediendo los cores físicos.
- Repeticiones: Suficientes para aplicar ley de grandes números y obtener medidas estadísticamente significativas.

5. Proceso de ejecución

Automatización de pruebas

Se utilizó el script lanzador.pl para automatizar la ejecución masiva:

```
$Path = `pwd`;
chomp($Path);

$Nombre_Ejecutable = "mmClasicaOpenMP";
@Size_Matriz = ("1000","2000","4000");
@Num_Hilos = (1,2,4,8,16);
$Repeticiones = 30;

foreach $size (@Size_Matriz){
       foreach $hilo (@Num_Hilos) {
              $file =
"$Path/$Nombre_Ejecutable-".$size."-Hilos-".$hilo.".dat";
              for ($i=0; $i<$Repeticiones; $i++) {
system("$Path/$Nombre_Ejecutable $size $hilo  >> $file");
                     printf("$Path/$Nombre_Ejecutable $size
$hilo \n");
              }
              close($file);
       $p=$p+1;
       }
}
```

Métrica de desempeño

- Tiempo de ejecución: Medido en microsegundos usando gettimeofday().
- Precisión: Captura de tiempo con resolución de microsegundos.

5.1 Ejecución por lotes

Para cada combinación (algoritmo × tamaño × hilos × sistema):

- Se ejecutó el programa 30 veces consecutivas.

- Los tiempos se almacenan en archivos .dat separados.
- Se generó un archivo por combinación: [ejecutable]-[tamaño]-Hilos-[número].dat.

5.2 Procesamiento de resultados

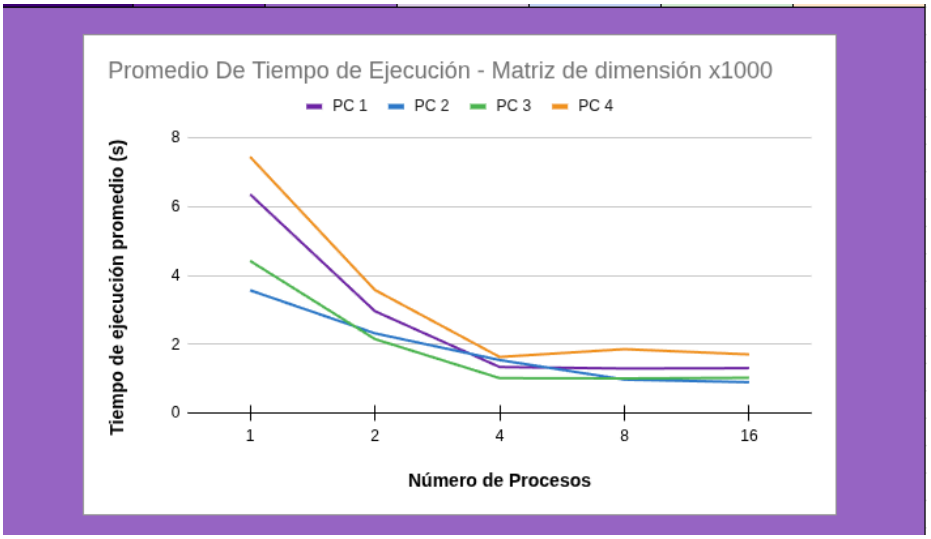
Los archivos .dat se procesaron utilizando el script *creadorCSV.c* para combinar los datos en un único .csv por categoría que se almacenó en una hoja de cálculo para posteriormente:

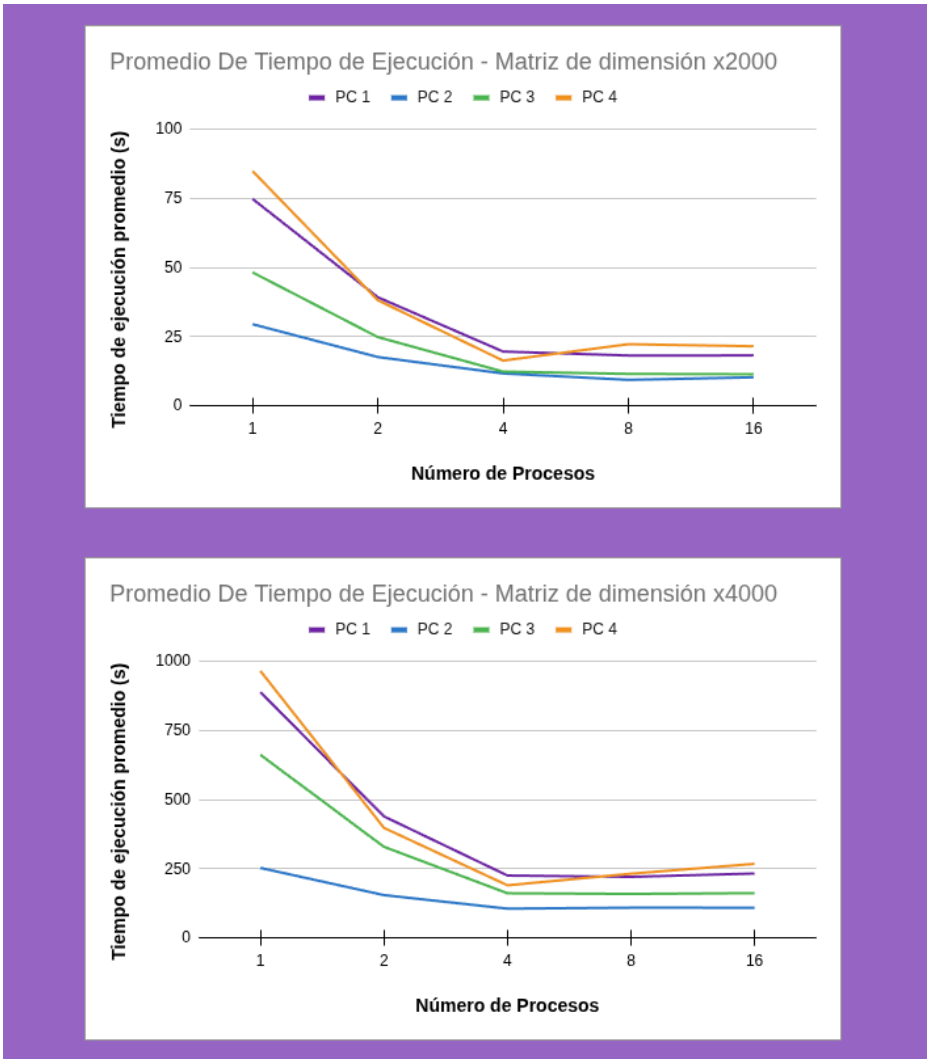
- Calcular promedios.
- Generar gráficas comparativas.
- Analizar tendencias de rendimiento.
- Contrastar los rendimientos de cada algoritmo.

Esta metodología permitió obtener datos confiables para el análisis comparativo del rendimiento de los diferentes algoritmos en diversas configuraciones de hardware.

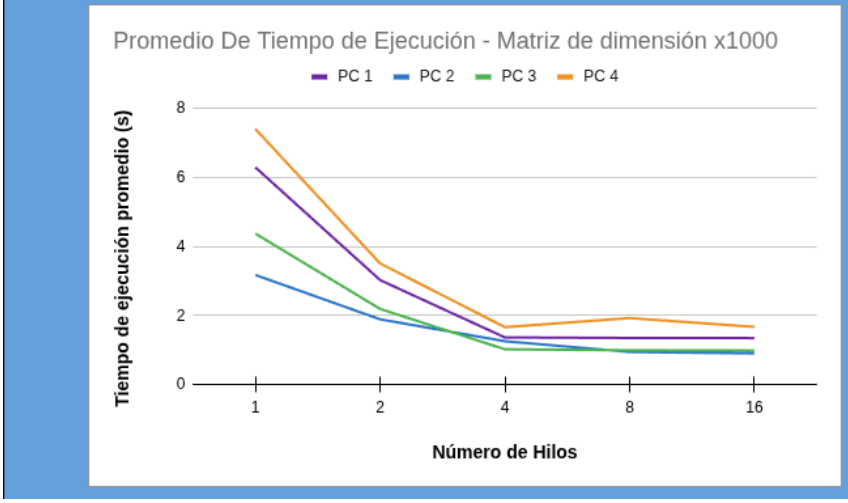
6. Resultados

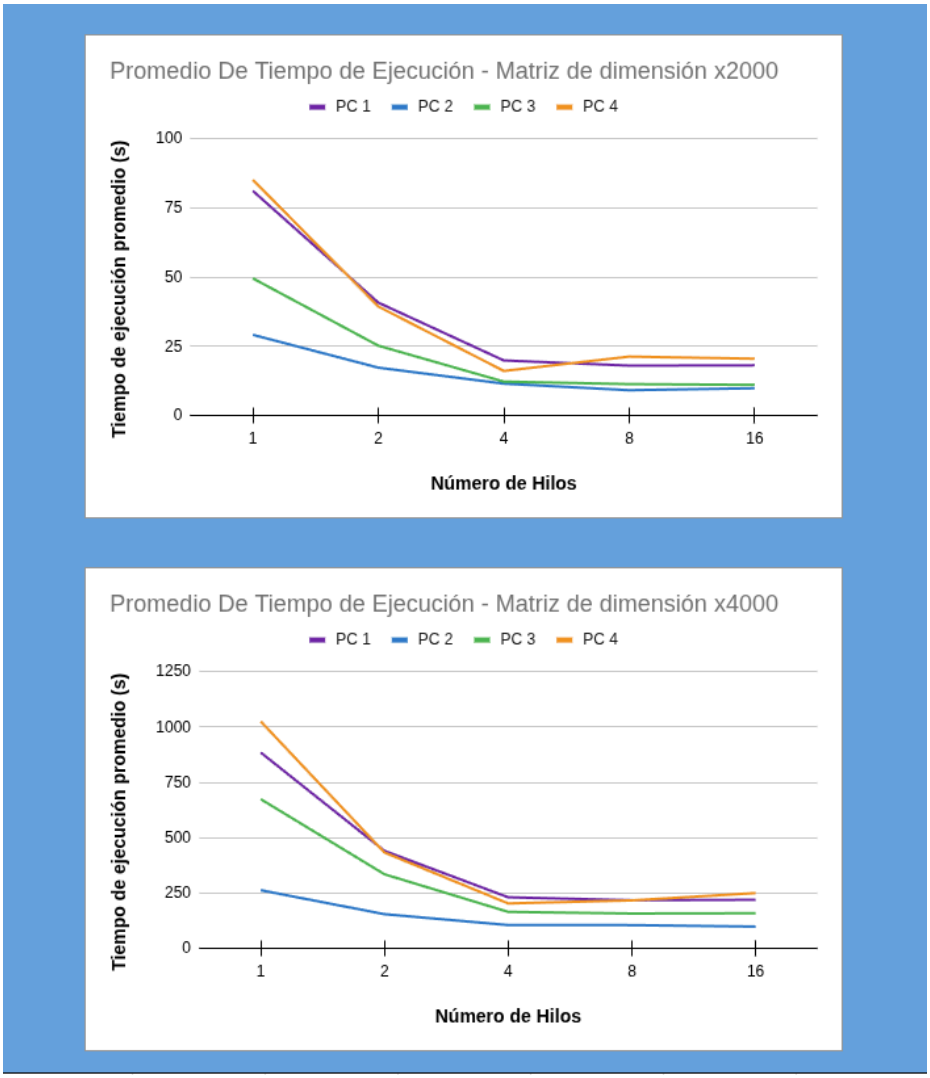
Comparación de Rendimiento Promedio entre computadoras (Fork)						
Computadora de Prueba			PC 1	PC 2	PC 3	PC 4
mmClasicaFork (s)	1000	1	6.365274467	3.5755694	4.429647933	7.455324667
		2	2.968376433	2.3235633	2.156304467	3.5844654
		4	1.3426279	1.5455961	1.019895967	1.635278567
		8	1.299835467	0.9775389	1.0115513	1.8622389
		16	1.309580567	0.9014237	1.0302299	1.710101267
	2000	1	74.9275754	29.57201483	48.32884413	85.01368687
		2	39.3193431	17.68168397	24.90440267	38.24132587
		4	19.6644248	11.73781737	12.40094803	16.38883013
		8	18.23421283	9.414794733	11.57849837	22.3287911
		16	18.2462164	10.3309232	11.46049597	21.57665913
	4000	1	889.8258121	253.5311916	663.0007108	966.4803974
		2	440.0282988	155.1073862	330.0359275	398.5452119
		4	225.9388386	106.0721688	161.7735024	190.5968136
		8	221.1493997	109.552794	159.4180685	232.5822434
		16	233.2654898	109.4724424	162.29786	268.3284273


Promedio De Tiempo de Ejecución - Matriz de dimensión x1000
PC 1
PC 2
PC 3
PC 4
Tiempo de ejecución promedio (s)
8
6
4
2
0
1
2
4
8
16
Número de Procesos


Promedio De Tiempo de Ejecución - Matriz de dimensión x2000
PC 1
PC 2
PC 3
PC 4
Tiempo de ejecución promedio (s)
100
75
50
25
0
1
2
4
8
16
Número de Procesos
Promedio De Tiempo de Ejecución - Matriz de dimensión x4000
PC 1
PC 2
PC 3
PC 4
Tiempo de ejecución promedio (s)
1000
750
500
250
0
1
2
4
8
16
Número de Procesos

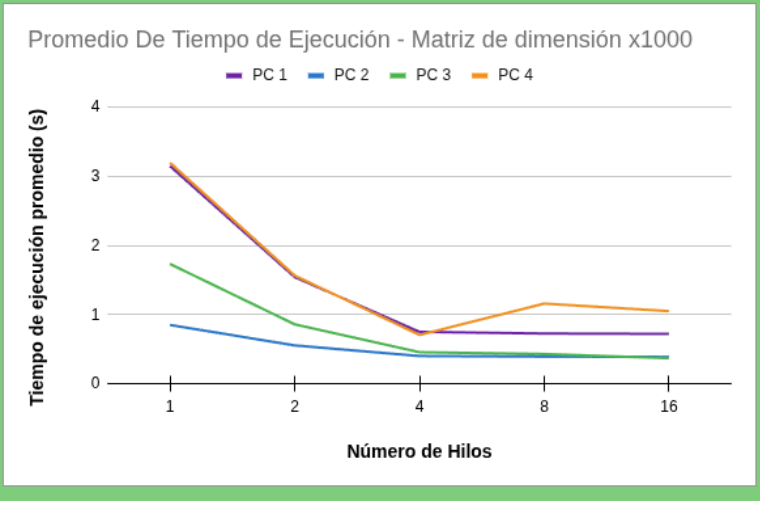
Comparación de Rendimiento Promedio entre computadoras (Posix)						
Computadora de Prueba			PC 1	PC 2	PC 3	PC 4
mmClasicaPosix (s)	1000	1	6.300146433	3.1785436	4.3784989	7.413243
		2	3.030908	1.894567833	2.194998867	3.513457
		4	1.368674833	1.2581265	1.025066333	1.666968
		8	1.353581367	0.9482432	0.9968321667	1.930338
		16	1.350205967	0.9073990667	0.9894789667	1.674198
	2000	1	81.23122957	29.21593907	49.6154541	85.232048
		2	40.84675227	17.34733243	25.2840043	39.443482
		4	19.9520393	11.5677809	12.30800667	16.154829
		8	18.0850495	9.1983194	11.39680647	21.340121
		16	18.18567513	9.906193733	11.13057567	20.584149
	4000	1	885.3836037	263.3701676	674.7832425	1025.918163
		2	441.2223392	155.2392325	335.4452751	433.646352
		4	231.487377	106.0323118	165.5675799	203.743153
		8	218.1566282	105.7421051	158.1208008	217.513674
		16	220.229127	99.09498277	159.4616514	250.498194

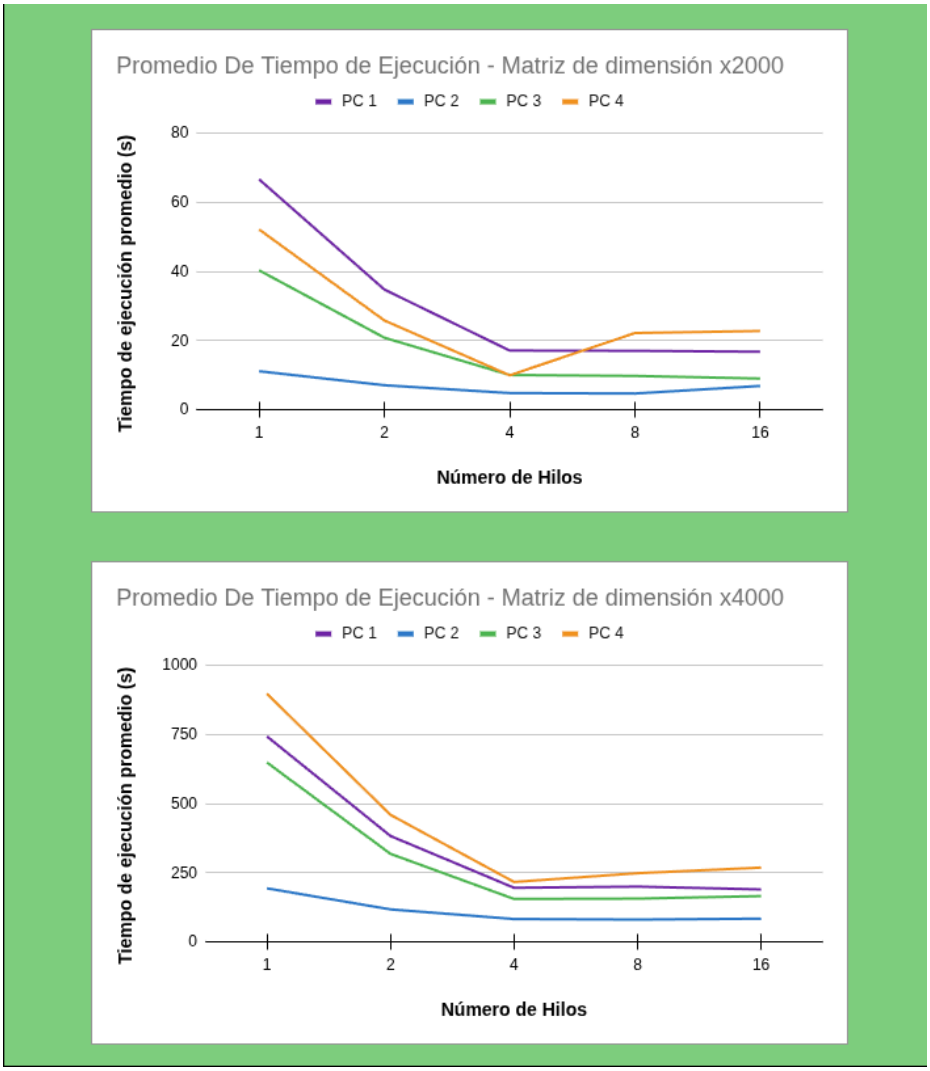



Promedio De Tiempo de Ejecución - Matriz de dimensión x2000
PC 1
PC 2
PC 3
PC 4
Tiempo de ejecución promedio (s)
100
75
50
25
0
1
2
4
8
16
Número de Hilos
Promedio De Tiempo de Ejecución - Matriz de dimensión x4000
PC 1
PC 2
PC 3
PC 4
Tiempo de ejecución promedio (s)
1250
1000
750
500
250
0
1
2
4
8
16
Número de Hilos

Comparación de Rendimiento Promedio entre computadoras (OpenMP)

Computadora de Prueba			PC 1	PC 2	PC 3	PC 4
mmClasicaOpenMP (s)	1000	1	3.152194533	0.8525532333	1.735568	3.200791
		2	1.5447374	0.5563190667	0.861166	1.567099
		4	0.7526897	0.4028276	0.458044	0.707923
		8	0.7282889333	0.3946736667	0.432278	1.161581
		16	0.7234354	0.3902683667	0.370895	1.052786
	2000	1	66.74396917	11.17970213	40.393593	52.240741
		2	34.83850193	7.131836967	20.890931	25.877792
		4	17.17047403	4.8582227	10.096516	10.005186
		8	17.09213393	4.728499067	9.824346	22.242482
		16	16.82145513	6.9262384	9.065475	22.836286
	4000	1	743.9539302	194.2735036	649.562556	898.296785
		2	383.7253225	118.129998	319.051409	460.254642
		4	196.2698588	83.23326363	155.842876	217.250836
		8	200.5449721	81.24480923	157.081095	249.076364
		16	190.3395153	84.4941647	166.228906	269.261375





7. Análisis de resultados

Al mirar las tablas de la hoja de cálculo, lo primero que se nota es que los tiempos de ejecución suben muchísimo cuando aumentamos el tamaño de la matriz. Para las matrices pequeñas los tiempos son manejables, pero cuando pasamos a 2000 y sobre todo a 4000, los valores se disparan. Esto coincide con lo que vimos en teoría: la multiplicación clásica de matrices tiene un costo que crece

muy rápido con nnn, así que no es raro que la diferencia de tiempos entre 1000 y 4000 sea tan grande.

También se ve claramente el efecto del número de hilos o procesos. En general, al pasar de 1 a 2 y de 2 a 4 hilos sí hay una mejora importante: los tiempos bajan de forma notoria para los tres enfoques (Fork, Pthreads y OpenMP). Sin embargo, cuando seguimos subiendo a 8 y 16 hilos ya no obtenemos el mismo beneficio; en algunos casos la mejora es muy pequeña y en otros incluso el tiempo empeora un poco. Eso muestra que en la práctica no basta con "poner más hilos" sino que hay un límite a partir del cual el overhead de sincronización y la competencia por la memoria empiezan a jugar en contra.

Comparando los tres métodos, los resultados del taller apuntan a que OpenMP suele ser el que se comporta mejor en la mayoría de configuraciones. Normalmente es el que registra los menores tiempos promedios, mientras que Pthreads y Fork quedan bastante parecidos entre sí, casi siempre por encima. Tiene sentido: con OpenMP el compilador y el runtime ayudan a gestionar los hilos y el reparto del trabajo, mientras que en Fork se crean procesos completos (con su propio espacio de memoria) y en Pthreads el manejo de hilos es más manual y propenso a tener algo más de sobrecosto.

Otro aspecto importante del análisis es la diferencia entre las máquinas usadas en las pruebas. En las tablas se aprecia que hay una computadora que de forma consistente obtiene mejores tiempos, especialmente con tamaños grandes y varios hilos, mientras que otras son claramente más lentas. Esto encaja con las características de hardware descritas en el informe: una máquina con más núcleos, mejor caché y sin tanta sobrecarga de virtualización aprovecha mejor el paralelismo, mientras que las máquinas virtuales o equipos más limitados se demoran más aunque el código sea exactamente el mismo.

8. Conclusiones

A partir de los resultados obtenidos en el taller se puede concluir, en primer lugar, que la multiplicación clásica de matrices es un problema muy costoso computacionalmente. A medida que aumenta el tamaño de la matriz, el tiempo de ejecución crece de forma muy notable, lo cual se ve claramente en todas las tablas. Esto refuerza la importancia de pensar en eficiencia cuando se trabaja con algoritmos que manejan grandes volúmenes de datos. Por otro lado, el uso de paralelismo sí aporta beneficios reales, pero no de manera ilimitada. Los experimentos muestran que pasar de 1 a 2 y de 2 a 4 hilos/procesos mejora bastante los tiempos, pero cuando se llega a 8 o 16 ya no se ve una reducción proporcional, e incluso en algunos casos el rendimiento empeora. Aquí se ve aplicada la idea de que siempre existen partes secuenciales, costos de coordinación y límites de hardware que impiden alcanzar un speedup ideal.

Otra conclusión importante es que OpenMP resultó ser la opción más conveniente dentro de las tres estrategias probadas. En la mayoría de escenarios fue el enfoque con mejor tiempo, mientras que Pthreads y Fork, aunque también permiten paralelizar, presentan más sobrecarga por la forma en que se crean y gestionan hilos o procesos. Desde el punto de vista práctico, esto sugiere que, cuando el lenguaje y el compilador lo permiten, vale la pena considerar OpenMP como una herramienta sencilla y efectiva para paralelizar ciertos algoritmos. Finalmente, el taller deja muy claro que el rendimiento no depende solo del código, sino también de la máquina donde se ejecuta. La diferencia entre las computadoras utilizadas fue evidente: algunas aprovecharon mejor el paralelismo y obtuvieron tiempos mucho menores. Esto recuerda que, al analizar resultados de rendimiento, siempre hay que tener en cuenta la arquitectura, el número de núcleos, el caché y posibles capas de virtualización.

Anexos

Anexo A. Resultados completos del laboratorio

El archivo “Taller_Rendimiento_SO.xlsx” contiene el registro completo de las mediciones obtenidas durante el desarrollo del laboratorio sobre POSIX, fork() y OpenMP.

En dicho archivo se incluyen:

- Tiempos de ejecución para cada método (POSIX threads, procesos con fork y paralelismo con OpenMP).
- Variaciones por número de hilos/procesos.
- Comparaciones de rendimiento entre técnicas.
- Desviaciones, promedios y datos obtenidos en múltiples iteraciones de cada experimento.

Debido a la extensión del conjunto de datos, estos resultados no se incluyen directamente en el cuerpo del informe.

Se puede consultar este archivo si requiere analizar en detalle el comportamiento cuantitativo de cada técnica de paralelismo.