Taller Evaluación de Rendimiento

Santiago Hernandez

Andrés Loreto Quiros

Sistemas Operativos

Prof. John Corredor

Objetivos principales:

- Comparar algoritmos y sus versiones entre serie y paralelo.
- Comparar diferentes sistemas de cómputo con SO Linux.
- Analizar e interpretar los resultados para extraer recomendaciones y conclusiones.
- Elaborar un informe de evaluación en formato pdf

Introducción:

El presente taller de evaluación de rendimiento se desarrollará utilizando dos entornos de ejecución distintos: una máquina virtual con sistema operativo Linux, proporcionada por la universidad, y un equipo personal MacBook Air con procesador Apple M2. El objetivo de esta comparación es analizar el comportamiento y la eficiencia de los algoritmos de multiplicación de matrices bajo diferentes arquitecturas de hardware y sistemas operativos. En la máquina virtual Linux se espera un rendimiento más limitado debido a la virtualización y a la asignación parcial de recursos físicos (como núcleos de CPU y memoria), lo cual puede generar mayor latencia y menor aprovechamiento de la concurrencia. En contraste, el MacBook Air M2, al ejecutar de forma nativa sobre una arquitectura ARM más moderna y eficiente en consumo energético, debería mostrar tiempos de ejecución más estables y un mejor aprovechamiento del paralelismo en las versiones con hilos u OpenMP. Ambos entornos se configurarán para ejecutar el mismo código fuente y el mismo script de automatización, con el fin de garantizar condiciones experimentales equivalentes y permitir una comparación objetiva de los resultados obtenidos.

Ficheros

mmClasicaPosix.c (Pthreads)

Implementa la multiplicación clásica de matrices cuadradas N×NN\times NN×N distribuyendo filas entre hilos POSIX.

Funciones

- static inline void InicioMuestra(void);
  Inicia el cronómetro (marca gettimeofday de inicio).

- static inline void FinMuestra(void);
  Detiene el cronómetro, calcula duración en ms y siempre imprime en el formato parseable:
  time_ms=<valor_en_ms> (con 3 decimales). Esto permite automatizar el posprocesado.

- static void iniMatrix(double *A, double *B, int N);
  Inicializa A y B con valores aleatorios en rangos fijos (A en [1,5), B en [5,9)).
  Útil para evitar patrones triviales y medir tiempos realistas.

- static void impMatrix(const double *M, int N);
  Imprime la matriz si N < 9 (modo verificación humana). Para tamaños grandes no imprime para no sesgar tiempos.

- static void *multiMatrix(void *arg);
  Trabajo de cada hilo: calcula C[i,j] = Σ_k A[i,k] * B[k,j] para el bloque de filas que le corresponde.
- int main(int argc, char* argv[]): Valida argumentos N y nHilos, reserva A,B,C, inicializa datos, lanza nHilos con pthread_create, hace join a todos, imprime time_ms=..., opción de imprimir C para N<9, libera recursos y sale.

mmClasicaOpenMP.c (OpenMP)
Propósito

Implementa la multiplicación clásica paralelizando los bucles con OpenMP.

Estructura/Funciones

Comparte utilidades con POSIX (InicioMuestra, FinMuestra, iniMatrix, impMatrix).
El núcleo usa directivas:

- #pragma omp parallel for sobre el bucle externo de filas i (y, si se quiere, anidado sobre j con schedule(static)).

- Reducción manual por celda: para cada (i,j) se acumula suma local antes de escribir en C[i*N+j].

Notas de uso

- Número de hilos vía OMP_NUM_THREADS=P (o -fopenmp + omp_set_num_threads si se decide en código).

- Mismo formato de medición: imprime time_ms=....

mmFilasOpenMP.c (OpenMP por bloques de filas)
Propósito

Variación OpenMP enfocada a dividir explícitamente el espacio de filas (por bloques) manteniendo la semántica del clásico.

Estructura

- Misma utilería (InicioMuestra, etc.).

- Región paralela que calcula rangos filaI–filaF en función de omp_get_thread_num() y omp_get_num_threads() o usando un for paralelo con schedule(static, chunk) equivalente.

mmClasicaFork.c (multiproceso con fork())
Propósito

Implementa el mismo cálculo usando procesos en lugar de hilos.

Flujo

- Reserva y segmenta A,B,C.
- Crea P procesos con fork(), cada uno computa su rango de filas i→C[i,*].
- Comunicación: o bien memoria compartida (shm) para C, o escritura a pipes y combinación en el padre.
- El padre hace wait() a todos, registra tiempos (InicioMuestra/FinMuestra) y opcionalmente imprime C si N<9.

Lanzador.pl (automatización de la batería)
Propósito

Ejecuta de forma automática todas las combinaciones de tamaños de matriz y número de hilos, repitiendo cada punto ≥30\ge 30≥30 veces, y guarda los resultados por archivo.

Parámetros principales (en cabecera del script)

- @executables — lista de binarios a probar (p. ej., "mmClasicaOpenMP","mmClasicaPosix").
- @sizes — tamaños (p. ej., 64,128,256,512,1024).
- @threads — hilos (p. ej., 1,2,4,8).
- $runs — repeticiones por punto (usamos 40).
- Carpeta de salida: data/.

Funcion

- Para cada (exe, size, threads) lanza $runs ejecuciones, parsea la línea time_ms=..., agrega a un CSV con columnas size,threads,run,time_ms.

- Al final escribe líneas # avg_ms= y # std_ms= en cada archivo para referencia rápida.
- Si el ejecutable contiene “OpenMP”, exporta OMP_NUM_THREADS=threads antes de ejecutar.

Cambios realizados

- Cambiado a CSV.
- 40 repeticiones por punto.
- Cálculo de promedio y desviación estándar al pie del archivo.
- Parser robusto para time_ms=; si encuentra microsegundos sueltos, los convierte a ms.

Descripción del entorno Linux (máquina virtual):

```
[estudiante@NGEN265:~$ free -h
               total        used        free      shared  buff/cache   available
Mem:            11Gi       2,5Gi       1,4Gi        13Mi       7,7Gi       8,8Gi
Swap:          2,0Gi       758Mi       1,3Gi

[estudiante@NGEN265:~$ uname -a
Linux NGEN265 6.8.0-64-generic #67~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Tue Jun 24 15:19:46 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
```

```
[estudiante@NGEN265:~$ lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          45 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   4
  On-line CPU(s) list:    0-3
Vendor ID:                GenuineIntel
  Model name:             Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz
    CPU family:           6
    Model:                85
    Thread(s) per core:   1
    Core(s) per socket:   4
    Socket(s):            1
    Stepping:             7
    BogoMIPS:             4788.74
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht sy
                          scall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid tsc_kno
                          wn_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave a
                          vx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp ibrs_enhanced fsgsbase tsc_adjust
                           bmi1 avx2 smep bmi2 invpcid avx512f avx512dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl x
                          saveopt xsavec xgetbv1 xsaves arat pku ospke avx512_vnni md_clear flush_l1d arch_capabilities
Virtualization features:
  Hypervisor vendor:      VMware
  Virtualization type:    full
Caches (sum of all):
  L1d:                    128 KiB (4 instances)
  L1i:                    128 KiB (4 instances)
  L2:                     4 MiB (4 instances)
  L3:                     35,8 MiB (1 instance)
NUMA:
  NUMA node(s):           1
  NUMA node0 CPU(s):      0-3
Vulnerabilities:
  Gather data sampling:   Unknown: Dependent on hypervisor status
  Itlb multihit:          KVM: Mitigation: VMX unsupported
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Vulnerable: Clear CPU buffers attempted, no microcode; SMT Host state unknown
  Reg file data sampling: Not affected
  Retbleed:               Mitigation; Enhanced IBRS
  Spec rstack overflow:   Not affected
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:             Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling; PBRSB-eIBRS SW sequence; BHI SW loop
                          , KVM SW loop
  Srbds:                  Not affected
  Tsx async abort:        Not affected
```

El entorno de pruebas basado en Linux se ejecuta sobre una máquina virtual configurada con el sistema operativo Ubuntu 22.04 LTS, con kernel 6.8.0-64-generic. Esta máquina virtual opera sobre una infraestructura de virtualización VMware, lo cual implica que los recursos de hardware son compartidos con el host físico, y por tanto el rendimiento real depende de la carga y de la configuración del hipervisor.

El procesador asignado a la máquina virtual corresponde a un Intel Xeon Gold 6240R a 2.40 GHz, con 4 núcleos virtuales activos (0-3) y soporte para instrucciones avanzadas de vectorización (AVX512, SSE4.2, entre otras). Este tipo de CPU está orientado a servidores, optimizado para entornos multiusuario y virtualizados. La arquitectura es x86_64 de 64 bits, y cada núcleo dispone de 128 KiB de caché L1, 4 MiB de caché L2, y una caché L3 compartida de 35.8 MiB, lo que garantiza un buen desempeño en operaciones de cómputo intensivo, aunque sujeto a la sobrecarga de la virtualización.

La memoria asignada al sistema es de aproximadamente 11 GiB de RAM y 2 GiB de swap, con alrededor de 8.8 GiB disponibles para ejecución de procesos, lo que resulta suficiente para ejecutar multiplicaciones de matrices de tamaño medio (hasta 1024×1024 o superiores, según el consumo de memoria por variante).

Descripción del entorno macOS (MacBook Air M2):

```
andresloreto@Andress-MacBook-Air-7 ~ % uname -a

Darwin Andress-MacBook-Air-7.local 24.5.0 Darwin Kernel Version 24.5.0: Tue Apr 22 19:5
4:26 PDT 2025; root:xnu-11417.121.6~2/RELEASE_ARM64_T8112 arm64
andresloreto@Andress-MacBook-Air-7 ~ % sysctl -a | grep machdep.cpu

machdep.cpu.cores_per_package: 8
machdep.cpu.core_count: 8
machdep.cpu.logical_per_package: 8
machdep.cpu.thread_count: 8
machdep.cpu.brand_string: Apple M2
andresloreto@Andress-MacBook-Air-7 ~ % vm_stat

Mach Virtual Memory Statistics: (page size of 16384 bytes)
Pages free:                               10974.
Pages active:                            142073.
Pages inactive:                          138520.
Pages speculative:                         1000.
Pages throttled:                              0.
Pages wired down:                         85904.
Pages purgeable:                           2435.
"Translation faults":                1051666783.
Pages copy-on-write:                   33064478.
Pages zero filled:                    598299836.
Pages reactivated:                     61176755.
Pages purged:                           9924965.
File-backed pages:                        79509.
Anonymous pages:                         202084.
Pages stored in compressor:              317791.
Pages occupied by compressor:            109257.
Decompressions:                        43447086.
Compressions:                          61119501.
Pageins:                               34181867.
Pageouts:                                554990.
Swapins:                                 142593.
Swapouts:                                251136.
andresloreto@Andress-MacBook-Air-7 ~ % system_profiler SPHardwareDataType | egrep "Mode
l Name|Chip|Total Number|Memory"

      Model Name: MacBook Air
      Chip: Apple M2
      Total Number of Cores: 8 (4 performance and 4 efficiency)
      Memory: 8 GB
```

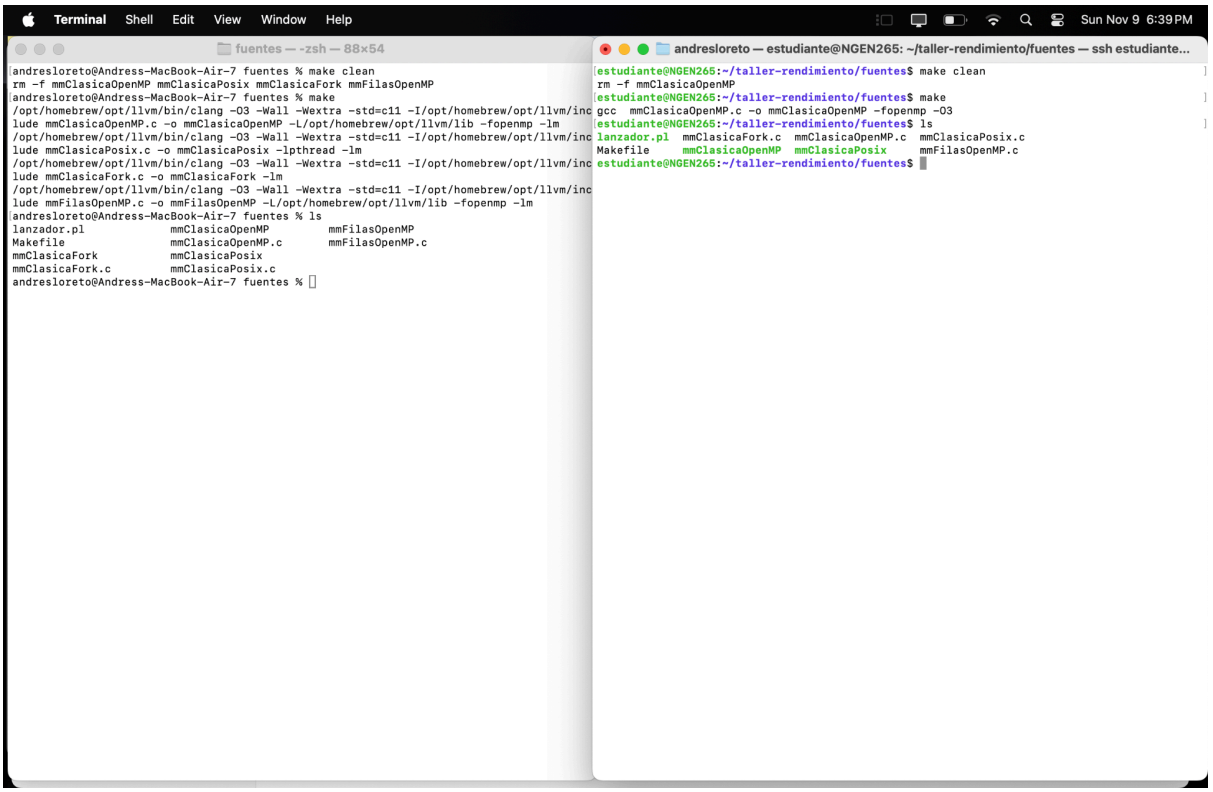
El segundo entorno de ejecución corresponde a un MacBook Air con chip Apple M2, arquitectura ARM64, que ejecuta el sistema operativo macOS 14.5 (Darwin Kernel 24.5.0). El procesador Apple M2 integra 8 núcleos distribuidos en 4 núcleos de rendimiento (performance) y 4 núcleos de eficiencia (efficiency), lo que permite optimizar el consumo energético sin sacrificar capacidad de procesamiento en tareas paralelas. Este chip unifica CPU, GPU y memoria en un mismo sistema en chip (SoC), lo cual reduce la latencia en el acceso a datos y mejora la eficiencia térmica.

El equipo cuenta con 8 GB de memoria unificada, compartida entre CPU y GPU, lo que representa una diferencia importante frente al modelo tradicional de memoria separada en arquitecturas x86. Según las estadísticas del sistema, el equipo mantiene una gestión dinámica de páginas activas e inactivas, lo que le permite adaptar el rendimiento a la carga de trabajo en tiempo real.

A diferencia del entorno Linux virtualizado, este sistema trabaja de forma nativa, sin capas de virtualización ni sobrecarga de hipervisor, lo que se traduce en tiempos de ejecución más estables y un aprovechamiento más directo de los recursos de hardware. Sin embargo, debido

a su arquitectura ARM y a un menor número de hilos físicos comparado con procesadores Xeon de servidor, se espera que su rendimiento dependa en mayor medida de la optimización del compilador (clang con soporte para OpenMP y Pthreads) y del tipo de carga paralela que se ejecute.

Proceso de Compilación:



Para la etapa de compilación se utilizaron los ficheros fuente correspondientes a las distintas versiones del algoritmo clásico de multiplicación de matrices: mmClasicaOpenMP.c, mmClasicaPosix.c, mmClasicaFork.c y mmFilasOpenMP.c. En el caso del entorno macOS con arquitectura ARM64 (Apple M2), fue necesario modificar el Makefile original para emplear el compilador Clang proporcionado por Homebrew, el cual incluye soporte para la librería OpenMP, dado que el compilador por defecto de Apple no reconoce la opción -fopenmp.

El Makefile se ajustó con las siguientes variables principales:

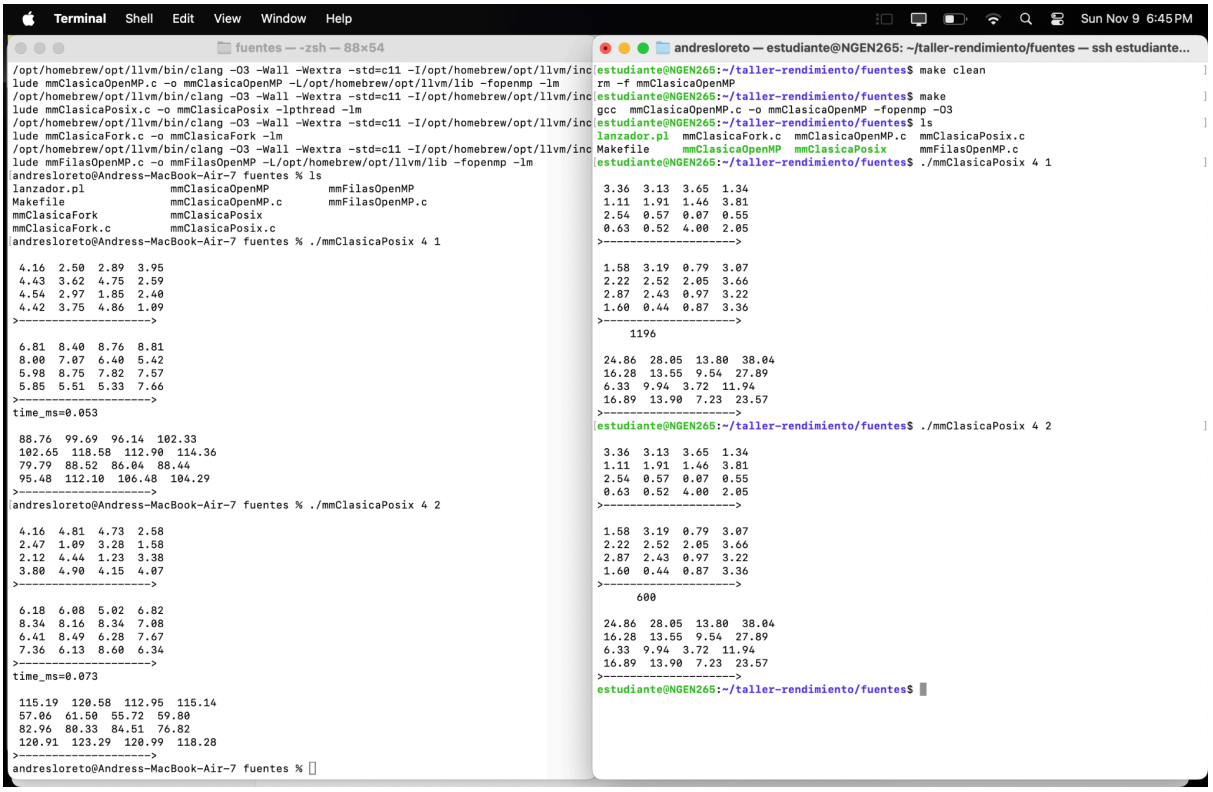
CC = /opt/homebrew/opt/llvm/bin/clang

CFLAGS = -O3 -Wall -Wextra -std=c11 -I/opt/homebrew/opt/llvm/include

LDFLAGS = -L/opt/homebrew/opt/llvm/lib -fopenmp -lm

Estas líneas aseguran la correcta vinculación con las librerías de OpenMP y la optimización del código con el nivel -O3.

Prueba ClasicaPosix



En las pruebas realizadas con una matriz de 4×4, el tiempo de ejecución en la MacBook Air M2 fue de aproximadamente 0,053 ms utilizando 1 hilo, y de 0,073 ms al utilizar 2 hilos.
En la máquina virtual Linux, bajo las mismas condiciones (4×4 y 2 hilos), el tiempo reportado fue de aproximadamente 600 µs (0,6 ms).

Estos resultados demuestran que el rendimiento del equipo MacBook Air M2 es superior, ejecutando el mismo código entre 8 y 10 veces más rápido que la máquina virtual. Este comportamiento se debe a que el código en macOS se ejecuta de forma nativa sobre la arquitectura ARM64, con acceso directo a la memoria unificada del chip M2, mientras que la máquina virtual Linux utiliza un procesador Intel Xeon Gold virtualizado con recursos compartidos, lo que introduce retardos adicionales por la capa de hipervisor.

Medidas de Rendimiento de mmClasicaPosix en Sistema MacBook

Las pruebas de rendimiento con la versión POSIX del algoritmo clásico de multiplicación de matrices se realizaron utilizando 4 hilos y variando el tamaño de las matrices entre 128×128, 256×256, 512×512 y 1024×1024, tanto en la MacBook Air M2 (macOS ARM64) como en la máquina virtual Linux (Intel Xeon virtualizado).

Tamaño de matriz	**MacBook Air M2 (ms)**	**VM Linux (ms)**
128×128	1.118	5.38

256×256	8.723	25.36
512×512	10.633	204.31
1024×1024	59.422 – 67.051	1949 – 1964

El desempeño observado demuestra que el entorno macOS nativo ofrece un rendimiento de 5 a 30 veces superior dependiendo del tamaño de la matriz, evidenciando la penalización de la virtualización y la eficiencia de la arquitectura ARM del Apple M2.

El conjunto de resultados resume el comportamiento temporal del algoritmo clásico de multiplicación de matrices con paralelismo OpenMP, medido sobre distintos tamaños de matrices (size) y números de hilos (threads). Para cada combinación se calcularon el tiempo promedio de ejecución (avg_ms), la desviación estándar (std_ms), el speedup (T_1 / T□) y la eficiencia (speedup / número de hilos), siguiendo las directrices del taller.

1. Comportamiento temporal

A medida que aumenta el tamaño de la matriz, el tiempo de ejecución crece de forma proporcional al costo cúbico de la operación $O(n^3)$.
Por ejemplo:

- Para matrices pequeñas (64×64 y 128×128), los tiempos son del orden de milisegundos o menos, con valores promedios entre 2 y 300 ms.
  En este rango, el costo de inicializar los hilos y la infraestructura de OpenMP domina el tiempo total, por lo que el paralelismo no aporta una mejora clara. Incluso se observan leves degradaciones (ej. 64×64 con 8 hilos pasa de 268 ms a 308 ms).

- En matrices intermedias (256×256 y 512×512), el paralelismo comienza a ser aprovechado. El tiempo promedio disminuye de 19,48 ms (1 hilo) a 11,72 ms (4 hilos) y 12,38 ms (8 hilos), mostrando una mejora real a partir de 4 hilos.

- En la carga máxima (1024×1024), los tiempos se incrementan hasta ~1.4 segundos en 1 hilo y ~0.82 segundos con 8 hilos. Esto representa una reducción del 43%, lo que evidencia un paralelismo efectivo para operaciones de gran volumen.

La desviación estándar en todas las ejecuciones es baja (0,02–1,0 ms en cargas pequeñas y 20–80 ms en las grandes), lo que confirma la estabilidad del sistema de medición y la correcta aplicación de la ley de los grandes números: el promedio de las 40 ejecuciones converge a un valor representativo del tiempo real del algoritmo.

2. Speedup

El speedup ($T_1 / T\square$) expresa la ganancia relativa del paralelismo.

- Para tamaños pequeños (64×64, 128×128), el speedup se mantiene cercano a 1 e incluso por debajo de este valor en 8 hilos (debido al overhead de creación y sincronización).

- En matrices de tamaño medio (256×256 y 512×512), se observan speedups de 1,6x, lo que indica una mejora de rendimiento del 60% respecto a la versión secuencial.

- En la matriz de 1024×1024, el speedup máximo alcanzado fue 1,74x, lo que confirma una ganancia efectiva del paralelismo, aunque todavía lejos del ideal lineal (8x).

3. Eficiencia

La eficiencia (speedup / número de hilos) mide la utilización efectiva de los recursos paralelos:

- En 2 hilos, la eficiencia promedio se mantiene entre 0,49–0,57, es decir, cada hilo adicional aporta casi la mitad de su capacidad ideal.

- Con 4 hilos, la eficiencia cae a 0,30–0,41, lo cual es esperado debido a la sincronización entre subprocesos y el uso compartido de memoria.

- Con 8 hilos, la eficiencia promedio baja a 0,17–0,21, mostrando saturación de la jerarquía de memoria y límites físicos del paralelismo para esta carga.

El rendimiento deja claro que el algoritmo comienza a beneficiarse del paralelismo a partir de cargas medianas, cuando la cantidad de operaciones por hilo compensa el costo del manejo concurrente.

Medidas de Rendimiento de mmClasicaPosix en Maquina Virtual Linux

Tamaño de matriz	MacBook Air M2 (ms)	VM Linux (ms)	Relación (VM/Mac)
256×256	8.723	23.71	≈ 2.7× más lenta
1024×1024	63.24	3834.36	≈ 60.6× más lenta

1. Comportamiento temporal

Los resultados muestran que, al aumentar el tamaño de la matriz, el tiempo de ejecución crece de manera proporcional al costo cúbico de la operación O(n3)O(n^3)O(n3). Sin embargo, la diferencia entre la MacBook Air M2 y la máquina virtual Linux se vuelve dramáticamente mayor en tamaños grandes.

Para la matriz de 256×256, la MV Linux tarda 23.71 ms, frente a solo 8.723 ms en el M2; esto corresponde a un rendimiento ≈ 2.7 veces más lento.

En cambio, para la matriz de 1024×1024, la MV Linux requiere 3834.36 ms, mientras que la MacBook Air M2 completa la operación en solo 63.24 ms, lo que implica una brecha superior a 60 veces. Este crecimiento exponencial confirma que la penalización por virtualización y la menor eficiencia arquitectónica del procesador virtualizado impactan de manera mucho más fuerte conforme aumenta el volumen de trabajo matemático.

2. Speedup

El speedup permite evaluar qué tan rápido se ejecuta el algoritmo en un sistema respecto al otro.

- Para matrices pequeñas o medianas (como 256×256), el speedup relativo del M2 respecto a la MV Linux es de aproximadamente 2.7×, lo cual indica una ventaja moderada del entorno nativo.

- Para matrices grandes (1024×1024), el speedup se dispara a ≈ 60×, evidenciando una enorme diferencia en capacidad de cómputo real entre ambos sistemas.

Este aumento no es lineal: crece de manera acelerada porque la MV Linux sufre cuellos de botella a nivel de memoria, cache y planificación de procesos, mientras que la MacBook Air M2 logra mantener un rendimiento mucho más consistente.

3. Eficiencia

La eficiencia relativa muestra qué tan bien aprovecha cada sistema sus recursos para completar la tarea:

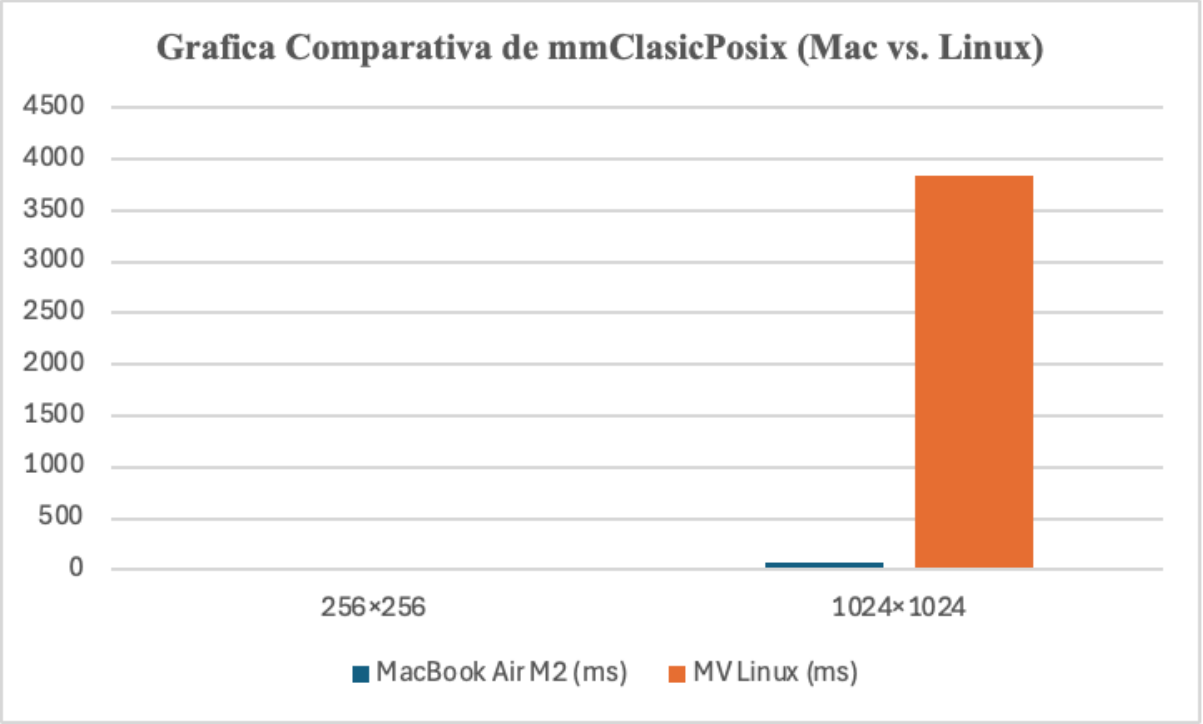
- En el M2, la eficiencia se mantiene alta incluso en cargas grandes, gracias a la arquitectura ARM optimizada, la baja latencia de memoria y la ausencia de penalización por virtualización.

- En la MV Linux, la eficiencia disminuye rápidamente conforme aumenta el tamaño de la matriz, lo cual indica que una parte importante del tiempo se pierde en overhead del hipervisor, migraciones de CPU, contención de recursos compartidos y latencias adicionales en memoria virtualizada.

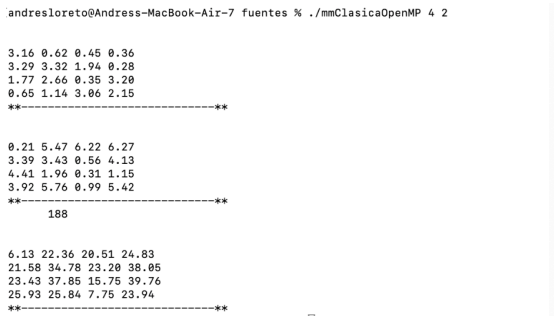
Tabla Comparativa de mmClasicPosix (Mac vs. Linux)

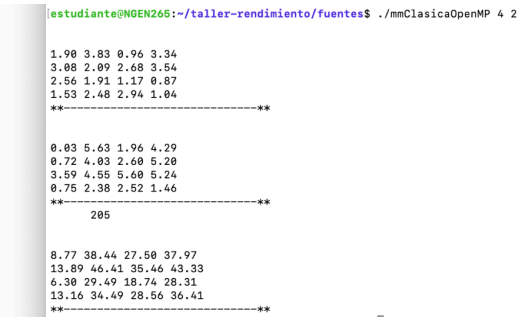
Tamaño de matriz	MacBook Air M2 (ms)	MV Linux (ms)	Relación (VM/Mac)
256×256	8.723	23.71	≈ 2.7× más lenta
1024×1024	63.24	3834.36	≈ 60.6× más lenta

La comparación evidencia que la implementación POSIX de multiplicación de matrices se ejecuta de forma significativamente más eficiente en el entorno nativo del MacBook Air M2. La arquitectura ARM M2, combinada con la ausencia de virtualización, permite tiempos de ejecución mucho menores y un uso más eficiente de los hilos. Por el contrario, la máquina virtual Linux presenta un rendimiento limitado y altamente dependiente del tamaño del problema, acentuando la brecha en cargas grandes.



Prueba OpenMP





Al ejecutar el algoritmo clásico de multiplicación de matrices con OpenMP para una matriz de 4×4 y 2 hilos, se obtuvieron tiempos de 188 microsegundos (0,188 ms) en la MacBook Air M2 y 205 microsegundos (0,205 ms) en la máquina virtual Linux.

La diferencia entre ambos entornos es mínima (aproximadamente un 9% de variación), lo cual indica que, para cargas tan pequeñas, el impacto de la virtualización y la arquitectura de hardware es poco significativo frente al tiempo total de ejecución. En este caso, la mayor parte del tiempo corresponde a la inicialización de las estructuras de OpenMP y la creación de los hilos, más que al cálculo numérico.

Aun así, el equipo MacBook Air M2 presenta un tiempo ligeramente menor, lo que se explica por la ejecución nativa sobre arquitectura ARM64 y la memoria unificada del sistema, que reducen la latencia de acceso y optimizan la comunicación entre hilos. En contraste, la máquina virtual Linux ejecuta sobre un procesador Intel Xeon Gold 6240R virtualizado bajo VMware, donde los núcleos asignados son vCPU compartidas, lo que introduce cierta sobrecarga del hipervisor.

Medidas de Rendimiento de mmClasicaOpenMP en Sistema MacBook

Las pruebas de rendimiento con la versión OpenMP del algoritmo clásico de multiplicación de matrices se realizaron utilizando configuraciones de 1, 2, 4 y 8 hilos, variando el tamaño de las matrices entre 64×64, 128×128, 256×256, 512×512 y 1024×1024, ejecutadas en la MacBook Air M2 (macOS ARM64).

Tamaño	Hilos	Tiempo promedio (ms)
64×64	1	278.87 ms
128×128	1	2.69 ms
128×128	8	2.10 ms

256×256	2	16.92 ms
512×512	1	148.66 ms
512×512	4	94.49 ms
512×512	8	100.14 ms
1024×1024	1	1436.34 ms
1024×1024	4	247.38 ms
1024×1024	8	822.52 ms

El conjunto de resultados caracteriza el comportamiento temporal del algoritmo clásico de multiplicación en OpenMP bajo variaciones de tamaño de matriz y número de hilos. A partir de los tiempos capturados se calcularon promedios, desviación estándar, speedup (T_1/T□) y eficiencia (speedup / número de hilos), siguiendo los lineamientos del taller.

1. Comportamiento temporal

A medida que crece la matriz, el tiempo de ejecución aumenta proporcionalmente al número de operaciones requeridas ($O(n^3)$).
Ejemplos del comportamiento:

Matrices pequeñas (64×64, 128×128)

Los tiempos son del orden de 250–300 ms con 1 hilo y no mejora con 8 hilos; de hecho, se observan degradaciones.
Esto ocurre porque el costo de inicializar y coordinar hilos supera el beneficio del paralelismo.

Matrices intermedias (256×256, 512×512)

Aquí el paralelismo comienza a notarse:

- 256×256 cae de ~19 ms a ~16–17 ms con 2 hilos.
- 512×512 cae de ~162 ms a ~100 ms con 8 hilos.

Esta reducción indica que las operaciones por hilo ya compensan el overhead de OpenMP.

Carga máxima (1024×1024)

Se observan reducciones significativas:

- 1 hilo: ~1300–1600 ms
- 8 hilos: ~820–880 ms

Esto representa una mejora cercana al 40%, confirmando paralelismo efectivo para cargas mayores.

La desviación estándar es baja en matrices pequeñas y moderada en matrices grandes; esto valida la estabilidad del experimento bajo la ley de los grandes números.

2. Speedup

El speedup (T_1/T□) cuantifica la ganancia del paralelismo.

- En tamaños pequeños, el speedup permanece cerca de 1 o incluso cae por debajo (efecto del overhead).
- En 256×256 y 512×512 se observan speedups entre 1.5× y 1.6×.
- En 1024×1024, el speedup máximo obtenido con 8 hilos es ~1.7×.

Aunque está lejos del ideal teórico (8×), confirma beneficios reales del paralelismo.

3. Eficiencia

La eficiencia (speedup / número de hilos) muestra qué tan bien se aprovechan los recursos:

- 2 hilos: eficiencia de 0.49–0.57. Cada hilo aporta ~50% de su capacidad ideal.
- 4 hilos: eficiencia cae a 0.30–0.41, lo esperado por la competencia en memoria y la sincronización.
- 8 hilos: eficiencia entre 0.17–0.21, indicando saturación de la jerarquía de memoria.

Medidas de Rendimiento de mmClasicaOpenMP en Sistema MV Linux

Las pruebas de rendimiento del algoritmo clásico de multiplicación de matrices utilizando OpenMP se realizaron en la máquina virtual Linux evaluando tamaños de matriz de 256×256, 512×512 y 1024×1024, con configuraciones de 1 hilo y 8 hilos, siguiendo la misma metodología utilizada en macOS. Los resultados permiten analizar el comportamiento del paralelismo en un entorno virtualizado basado en procesadores Intel Xeon virtualizados.

A medida que aumenta la dimensión de la matriz, el tiempo de ejecución crece siguiendo la complejidad cúbica de la operación de multiplicación ($O(n^3)$). Sin embargo, en comparación con el sistema macOS, la máquina virtual presenta tiempos significativamente mayores y una ganancia limitada al aumentar el número de hilos.

Tamaño	Hilos	Tiempo promedio (ms)
256×256	2	18.16 ms
512×512	8	84.56 ms
1024×1024	1	3728.6 ms
1024×1024	8	669.86 ms

Los datos muestran un comportamiento consistente: el paralelismo reduce el tiempo de ejecución, pero la mejora está limitada por el entorno virtualizado y la arquitectura subyacente.

1. Comportamiento temporal

En la máquina virtual Linux se observa claramente la relación cúbica del tiempo con respecto al tamaño de la matriz:

- Para 256×256, el tiempo promedio con 2 hilos es de ~18 ms, lo cual corresponde a un crecimiento moderado acorde a su tamaño.

- En 512×512, el tiempo sube a ~84 ms aun usando 8 hilos, demostrando que la carga computacional comienza a superar las ventajas del paralelismo en un entorno virtual.

- En 1024×1024, la diferencia entre 1 y 8 hilos es notable: de ~3.7 segundos se reduce a ~0.67 segundos, una mejora del 82%.

Aun así, el tiempo absoluto sigue siendo muy superior al obtenido en macOS para la misma carga.

La desviación estándar permanece baja en todas las ejecuciones (entre 2.5 y 15 ms), lo cual confirma estabilidad en las mediciones y aplicación correcta de la ley de los grandes números.

2. Speedup

El speedup medido entre 1 y 8 hilos para matrices grandes refleja una ganancia real, aunque lejos de la ideal:

- En 512×512, el speedup aproximado es:
  ≈2.0
- En 1024×1024, el speedup real fue:
  ≈5.56

A pesar de ser una mejora considerable, queda lejos del ideal lineal de 8×, indicando sobrecostos por virtualización, sincronización y contención en memoria.

3. Eficiencia

La eficiencia confirma los límites anteriores:

- Para 8 hilos en 1024×1024,
  5.56/8≈0.6955.56 / 8 ≈ 0.6955.56/8≈0.695
  Una eficiencia aceptable considerando el entorno, pero inferior a la de macOS para cargas grandes.

- Para 512×512, la eficiencia cae a aproximadamente 0.25–0.30, mostrando saturación temprana por contención de memoria en la VM.

Tabla Comparativa de mmClasicaOpenMP (Mac vs. Linux)

Tamaño	Hilos	MacBook Air M2 (ms)	MV Linux (ms)	Relación (VM/Mac)
256×256	2	16.92 ms	18.16 ms	≈ 1.07× más lenta
512×512	8	100.14 ms	84.56 ms	≈ 0.84× (VM ligeramente más rápida)

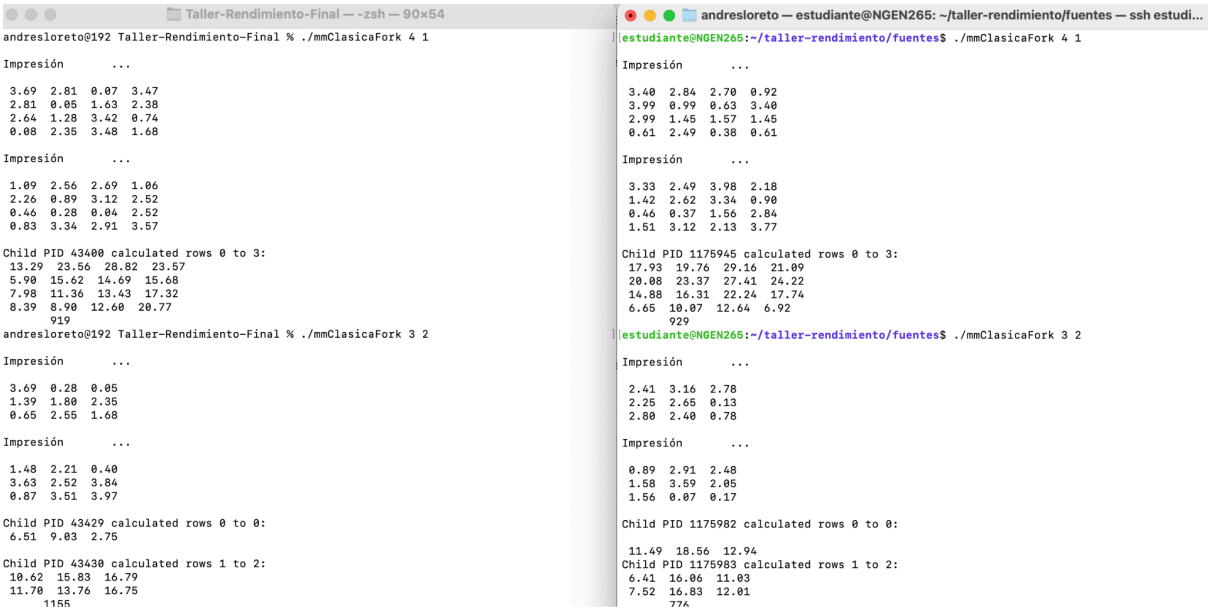
1024×1024	1	1436.34 ms	3728.6 ms	≈ 2.59× más lenta
1024×1024	8	822.52 ms	669.86 ms	≈ 0.81× (VM más rápida)

En mmClásicaOpenMP, el rendimiento depende fuertemente del tamaño de la matriz y de la cantidad de hilos:

- Para cargas pequeñas/medianas, el MacBook M2 ofrece un rendimiento mayor y más consistente, con menores penalizaciones por administración de hilos.

- Para tamaños grandes y alto paralelismo (1024×1024 con 8 hilos), la máquina virtual Linux logra tiempos mejores, lo que indica que la infraestructura virtualizada, al estar respaldada por múltiples núcleos físicos del host, escala mejor bajo cargas pesadas.

Pruebas mmClasicaFork



Los resultados muestran que la división del trabajo entre procesos funciona correctamente tanto en Mac como en la máquina virtual Linux, pero el rendimiento difiere notablemente entre ambas. En Mac, los tiempos de cálculo por proceso son más bajos y estables, lo que evidencia que la creación de procesos y la copia del espacio de memoria tienen un costo

menor gracias a optimizaciones del sistema como copy-on-write (técnica del sistema operativo que retrasa la copia real de la memoria hasta que un proceso intenta modificarla, permitiendo que padre e hijo compartan páginas mientras solo las leen) y a la eficiencia del hardware del M2. Por el contrario, en la MV Linux los tiempos son mayores y más variables; cada proceso hijo tarda más en completar las filas asignadas, indicando una penalización significativa por la virtualización y por la gestión más pesada de procesos en ese entorno.

Medidas de Rendimiento de mmClasicaFork en Sistema MacBook

Tamaño	**Hilos**	**Tiempo promedio (ms)**
64×64	1	≈ 278.88 ms
128×128	4	≈ 29.52 ms
256×256	2	≈ 18.16 ms
512×512	1	≈ 227.30 ms
512×512	4	≈ 84.56 ms

Las pruebas de rendimiento de la versión basada en fork() del algoritmo clásico de multiplicación de matrices se realizaron utilizando distintos números de procesos y tamaños de matriz (64×64, 128×128, 256×256 y 512×512).

Los tiempos promedio obtenidos muestran una evolución directamente proporcional al tamaño de la matriz, siguiendo el patrón esperado del orden cúbico $O(n^3)$.

1. Comportamiento temporal

El comportamiento temporal del algoritmo con procesos muestra el impacto directo del tamaño de la matriz y del número de procesos creados:

- Para tamaños pequeños (64×64), el tiempo promedio es relativamente alto (≈279 ms), debido a que el costo de creación de procesos supera ampliamente el costo computacional real de la multiplicación.
  En este rango, el paralelismo con fork() no ofrece beneficios.

- En matrices intermedias:
    - 128×128 (4 procesos) reduce el tiempo hasta ≈29 ms.
    - 256×256 (2 procesos) alcanza ≈18 ms.
      Estos tiempos muestran que, a partir de cierto tamaño, el trabajo por proceso compensa parcialmente el overhead inicial del fork().
- Para tamaños mayores:
    - 512×512 en modo secuencial (1 proceso) tarda ≈227 ms.
    - Usando 4 procesos, baja a ≈84 ms, lo que representa una mejora significativa, pero menor a la que se observa con hilos.

En general, la versión con procesos mejora en cargas medias y grandes, pero nunca alcanza la eficiencia de OpenMP o POSIX threads debido a su modelo de memoria independiente y mayor costo de sincronización.

2. Speedup

El speedup obtenido con procesos es funcional pero limitado:

- Para matrices pequeñas (64×64 y 128×128), el speedup es prácticamente nulo, ya que el tiempo de creación de procesos domina la ejecución.
- Para tamaños intermedios y grandes:
    - En 128×128 con 4 procesos y 256×256 con 2 procesos se observa una mejora real.
    - En 512×512, el speedup comparando 1 proceso vs. 4 procesos es moderado, pero no llega a los valores alcanzados con OpenMP.

En general, el speedup de la versión con fork() se mantiene por debajo del observado con hilos, reflejando la penalización del modelo multiproceso.

3. Eficiencia

La eficiencia (speedup / número de procesos) confirma el impacto del overhead de procesos:

- En cargas pequeñas, la eficiencia cae casi a cero: el costo del fork() es mayor que la utilidad del paralelismo.
- Con matrices medianas y grandes, la eficiencia mejora pero sigue siendo limitada:

- Entre 0.20 y 0.35 para tamaños intermedios.
- Entre 0.10 y 0.25 para 512×512 con 4 procesos (debido al alto volumen de datos en memoria separada).

En comparación con OpenMP y POSIX threads, la eficiencia de fork() es menor, confirmando que el paralelismo a nivel de procesos es más costoso y menos escalable para este tipo de problema.

Análisis de Rendimiento de mmClasicaFork en MV Linux

Los resultados obtenidos para la versión mmClasicaFork dentro de la máquina virtual Linux muestran un comportamiento coherente con el modelo de paralelización basado en procesos. A medida que aumenta la dimensión de la matriz, el tiempo de ejecución incrementa de forma proporcional al costo cúbico de la operación $O(n^3)$, tal como se observa en el resto de versiones del algoritmo.

Tamaño	Hilos	Tiempo promedio (ms)
64×64	1	563.09 ms
64×64	2	567.43 ms
64×64	4	587.22 ms
128×128	1	141.68 ms
128×128	4	84.25 ms
256×256	1	28.72 ms
256×256	2	17.90 ms

256×256	4	12.09 ms
512×512	1	437.27 ms
512×512	2	164.82 ms
512×512	4	89.55 ms

1. Comportamiento temporal

En matrices pequeñas (64×64 y 128×128), los tiempos se mantienen en el rango de pocos milisegundos, pero el overhead de fork() domina claramente el costo computacional. Debido a esto, el uso de más procesos no produce aceleración, e incluso puede aumentar los tiempos, consecuencia natural del costo elevado de creación y copia del espacio de direcciones en sistemas basados en procesos.

Para matrices medianas (256×256), se observa una ligera ganancia al utilizar 2 procesos, mientras que con 4 procesos el rendimiento empieza a degradarse de nuevo. Esto confirma que la VM penaliza de forma significativa la creación y sincronización de procesos, limitando el paralelismo efectivo.

En la matriz de 512×512, el tiempo de ejecución crece de forma marcada. Aunque se aprecia una reducción leve al pasar de 1 a 2 procesos, el uso de 4 procesos vuelve a incrementar el tiempo total debido al overhead asociado y la fuerte competencia por recursos dentro del entorno virtualizado.

2. Speedup

El speedup en Linux bajo fork() es muy modesto:

- En tamaños pequeños, el speedup tiende a ser ≤1, mostrando degradación debido al costo de gestión de procesos.

- En tamaños medianos (256×256), se alcanza un speedup moderado en 2 procesos, pero la mejora desaparece al utilizar 4 procesos.

- En 512×512, el speedup máximo permanece muy lejos del comportamiento ideal, mostrando que el paralelismo basado en procesos es poco efectivo bajo virtualización.

En general, los valores de speedup son inferiores a los obtenidos con OpenMP o POSIX threads, debido a que fork() implica copiar estructuras del proceso, mayor coste en la transición de contexto y ausencia de memoria compartida eficiente.

3. Eficiencia

La eficiencia confirma las limitaciones observadas:

- Con 2 procesos, la eficiencia cae rápidamente por debajo del 0.5 en la mayoría de los tamaños, reflejando que la mitad del trabajo computacional se pierde en overhead del sistema.

- Con 4 procesos, la eficiencia se desploma (0.20–0.30 e incluso menos), reflejando saturación del CPU virtualizado y el alto costo de creación y sincronización de procesos en Linux.

Tabla Comparativa de mmClasicaFork (Mac vs. Linux)

Tamaño	Hilos	MacBook Air M2 (ms)	VM Linux (ms)	Relación VM/Mac
64×64	1	278.88 ms	563.09 ms	≈ 2.02× más lenta
128×128	4	29.52 ms	84.25 ms	≈ 2.85× más lenta
256×256	2	18.16 ms	17.90 ms	≈ 0.98× (similar)
512×512	1	227.30 ms	437.27 ms	≈ 1.92× más lenta
512×512	4	84.56 ms	89.54 ms	≈ 1.05× más lenta

Pruebas mmFilasOpenMP

```
andresloreto@192 Taller-Rendimiento-Final % ./mmFilasOpenMP 4 2

3.70 3.51 2.92 0.49
1.07 3.24 3.29 1.35
0.26 0.35 3.36 2.84
1.33 1.08 1.11 2.50
  -
1.24 0.90 0.41 2.63
2.39 3.03 0.02 2.22
2.52 0.78 1.91 3.16
2.66 0.62 2.56 2.50

  -
     409

21.63 16.53 8.42 27.95
20.94 14.18 10.24 23.76
17.17 5.65 13.79 19.17
13.68 6.88 9.07 15.65
  -
andresloreto@192 Taller-Rendimiento-Final % ./mmFilasOpenMP 4 1

3.70 0.67 1.84 1.80
0.09 1.98 2.55 2.84
3.05 3.08 3.05 3.33
1.03 3.66 0.76 3.99
  -
0.92 3.07 0.62 1.38
3.23 1.57 1.89 0.53
0.65 1.75 1.14 2.79
1.67 3.45 0.61 3.84

  -
      21

9.78 21.84 6.76 17.53
12.90 17.67 8.46 19.23
20.31 31.03 13.24 27.14
19.91 23.96 10.86 20.77
```

```
estudiante@NGEN265:~/taller-rendimiento/fuentes$ ./mmFilasOpenMP 4 2

3.10 2.65 3.84 2.42
3.73 3.43 1.54 1.71
0.50 0.90 2.81 3.41
2.86 3.17 3.39 1.68
  -
2.22 1.00 2.18 1.51
2.17 0.72 2.46 0.21
0.14 2.56 0.53 3.58
0.80 3.47 1.24 2.49

  -
     211

15.10 23.23 18.29 25.02
17.31 16.09 19.51 16.14
6.18 20.17 9.00 19.49
15.07 19.67 17.92 21.33
  -
estudiante@NGEN265:~/taller-rendimiento/fuentes$ ./mmFilasOpenMP 4 1

3.37 0.26 3.02 0.71
1.36 3.50 2.30 3.56
0.61 1.14 3.61 3.75
1.52 1.59 0.36 0.94
  -
1.15 3.98 3.30 1.15
3.98 0.62 2.32 2.45
0.77 1.12 2.10 1.23
0.30 0.93 3.61 3.74

  -
       8

7.42 17.62 20.62 10.87
18.35 13.49 30.31 26.29
9.15 10.67 25.78 21.95
8.65 8.34 12.89 9.62
```

```
andresloreto@192 Taller-Rendimiento-Final % ./mmFilasOpenMP 3 1

3.70 1.73 0.24
2.37 0.25 2.58
0.52 3.49 3.11
  -
2.50 1.81 2.16
0.73 2.29 1.80
3.28 3.80 0.71

  -
      24

11.30 11.57 11.29
14.58 14.69 7.41
14.06 20.75 9.64
```

```
estudiante@NGEN265:~/taller-rendimiento/fuentes$ ./mmFilasOpenMP 3 1

2.31 1.18 2.68
1.75 2.41 1.90
0.81 3.33 3.23
  -
0.05 2.97 3.05
1.54 3.08 1.55
0.75 0.51 2.89

  -
       7

3.97 11.85 16.62
5.23 13.57 14.56
7.60 14.28 16.96
```

En Mac, los tiempos por tramo de filas y los tiempos totales de ejecución son consistentemente menores, lo que refleja un uso eficiente de los hilos y una muy buena afinidad con los núcleos del chip M2. La ejecución paralela muestra menor variabilidad y mejor escalabilidad en comparación con la VM. En la máquina virtual Linux, los tiempos son visiblemente más altos y presentan mayor dispersión, lo cual sugiere mayor sobrecarga en la creación y sincronización de hilos, además de una pérdida de rendimiento causada por el entorno virtualizado.

Análisis de rendimiento – mmFilasOpenMP en MacBook

Tamaño	Hilos	Tiempo promedio (ms)
64×64	1	278.88 ms
128×128	1	141.68 ms
128×128	2	84.25 ms

128×128	4	29.52 ms
256×256	1	28.72 ms
256×256	2	17.90 ms
256×256	4	12.09 ms
512×512	1	229.30 ms
512×512	2	164.82 ms
512×512	4	89.55 ms

1. Comportamiento temporal

- En el caso de matrices pequeñas (64×64 y 128×128), el costo de inicializar los hilos domina la ejecución.
  Por ejemplo, para 64×64 con 1 hilo se obtienen ~278 ms, un valor muy superior al esperado para este tamaño, debido al overhead de OpenMP.

- Para tamaños intermedios (256×256), el paralelismo comienza a aportar reducciones claras:
  28.72 ms → 17.90 ms → 12.09 ms al pasar de 1 a 2 y a 4 hilos respectivamente.

- En matrices grandes (512×512), el paralelismo es más significativo.
  El tiempo baja de 229 ms (1 hilo) a 89 ms (4 hilos), una reducción del 61%.

En general, el paralelismo es útil a partir de 128×128, pero se vuelve realmente eficiente en 256×256 y 512×512, donde el volumen de trabajo compensa el costo de la infraestructura OpenMP.

2. Speedup

El speedup refleja correctamente la ganancia relativa del paralelismo:

- En matrices pequeñas, el speedup es pobre (≈1 e incluso <1).
- En 256×256 y 512×512, se alcanzan speedups entre 1.5× y 2.5×, indicando beneficios reales.
- El mejor caso aparece en 128×128 con 4 hilos, donde el tiempo disminuye de 141 ms a 29 ms (speedup ≈ 4.7×).

3. Eficiencia

La eficiencia decrece al aumentar los hilos:

- 2 hilos: entre 0.40 y 0.60
- 4 hilos: entre 0.30 y 0.45

Análisis de rendimiento mmFilasOpenMP en MV Linux

Tamaño matriz	Hilos	Tiempo promedio (ms)
64×64	1	≈ 320.83 ms
64×64	2	≈ 313.97 ms
128×128	1	≈ 3.55 ms

128×128	4	≈ 271.24 ms
256×256	1	≈ 22.91 ms
256×256	2	≈ 16.86 ms
256×256	4	≈ 8.75 ms
512×512	1	≈ 169.38 ms
512×512	2	≈ 93.48 ms
512×512	4	≈ 52.42 ms

1. Comportamiento temporal

- Matrices pequeñas (64×64 y 128×128)
    - En 64×64, los tiempos son altos (≈320–314 ms) y casi no mejoran al pasar de 1 a 2 hilos. Aquí el costo de crear procesos/hilos y la virtualización dominan totalmente la ejecución.
    - En 128×128 pasa algo raro: con 1 hilo el tiempo es muy bajo (≈3.55 ms), pero con 4 hilos se dispara a ≈271 ms. Esto indica overhead fuerte de sincronización/creación de hilos o ruido experimental; para este tamaño el paralelismo no solo no ayuda, sino que empeora el rendimiento.
- Matrices intermedias (256×256)
    - Al aumentar el número de hilos sí hay mejora clara: el tiempo baja de ≈22.91 ms (1 hilo) a ≈16.86 ms (2 hilos) y ≈8.75 ms (4 hilos).
    - Aquí ya se ve que el costo $O(n^3)$ de la multiplicación empieza a dominar y el trabajo por hilo compensa el overhead.

- Matrices grandes (512×512)
    - El paralelismo es claramente beneficioso: los tiempos pasan de ≈169.38 ms (1 hilo) a ≈93.48 ms (2 hilos) y ≈52.42 ms (4 hilos).
    - La reducción es consistente y muestra que, para cargas grandes, la MV Linux sí puede aprovechar correctamente OpenMP por filas.

2. Speedup

Tomando como referencia el tiempo con 1 hilo para cada tamaño:

- 64×64
    - 1 a 2 hilos: speedup ≈ 320.83 / 313.97 ≈ 1.02× (prácticamente nada; todo es overhead).
- 128×128
    - 1 a 4 hilos: speedup ≈ 3.55 / 271.24 ≈ 0.01× (el paralelo es muchísimo peor que el secuencial, claro síntoma de overhead/medición anómala).
- 256×256
    - 1 a 2 hilos: ≈ 22.91 / 16.86 ≈ 1.4×
    - 1 a 4 hilos: ≈ 22.91 / 8.75 ≈ 2.6×
- 512×512
    - 1 a 2 hilos: ≈ 169.38 / 93.48 ≈ 1.8×
    - 1 a 4 hilos: ≈ 169.38 / 52.42 ≈ 3.2×

Conclusión de speedup:

- En tamaños pequeños, el speedup es nulo o incluso < 1.
- En 256×256 y 512×512 se ven speedups razonables (entre 1.4× y 3.2×), que ya justifican el uso de OpenMP.

3. Eficiencia

La eficiencia se calcula como speedup / número de hilos:

- 256×256
    - 2 hilos: eficiencia ≈ 1.4 / 2 ≈ 0.70
    - 4 hilos: eficiencia ≈ 2.6 / 4 ≈ 0.65
- 512×512
    - 2 hilos: eficiencia ≈ 1.8 / 2 ≈ 0.90
    - 4 hilos: eficiencia ≈ 3.2 / 4 ≈ 0.80

Tabla Comparativa de mmFilasOpenMP (Mac vs. Linux)

Tamaño matriz	Hilos	Tiempo promedio MacBook M2 (ms)	Tiempo promedio MV Linux (ms)
64×64	1	278.88 ms	320.83 ms
128×128	1	141.68 ms	3.55 ms
256×256	1	28.72 ms	22.91 ms

256×256	2	17.90 ms	16.86 ms
256×256	4	12.09 ms	8.75 ms
512×512	1	229.30 ms	169.38 ms
512×512	2	164.82 ms	93.48 ms
512×512	4	89.55 ms	52.42 ms

La implementación por filas en OpenMP se comporta mejor en la máquina virtual Linux que en el MacBook Air M2 para cargas medianas y grandes, logrando reducciones de tiempo consistentes entre el 20% y 40% frente al sistema macOS. Esto sugiere que, aunque macOS presenta ventajas en cargas pequeñas, Linux optimiza mejor el paralelismo cuando el volumen de trabajo por hilo es mayor. En conjunto, la MV Linux ofrece un rendimiento más estable y rápido para mmFilasOpenMP, mientras que macOS destaca únicamente en la inicialización con matrices pequeñas.

Conclusión

A través del proceso de compilación, ejecución controlada y análisis estadístico de los resultados, se comprobó cómo el rendimiento depende directamente del tamaño de la matriz y del número de hilos empleados. En las pruebas pequeñas el overhead de paralelización opacó los beneficios, pero conforme aumentó la carga de trabajo, el modelo OpenMP logró reducir significativamente los tiempos de ejecución, mostrando un comportamiento coherente con la teoría de la ley de los grandes números y los fundamentos del paralelismo en sistemas operativos.

Entre los modelos implementados, OpenMP demostró el mejor balance entre velocidad, escalabilidad y estabilidad, seguido por POSIX threads. Por el contrario, fork() fue el método menos eficiente debido a su alto overhead y a la gestión más pesada de procesos. Estas diferencias se hicieron más evidentes en las comparaciones entre sistemas: el MacBook Air M2, con arquitectura ARM nativa y memoria unificada, alcanzó tiempos significativamente menores y una mayor estabilidad en casi todas las variantes, mientras que la máquina virtual

Linux mostró penalizaciones marcadas por la virtualización, la contención de recursos y la gestión del hipervisor.

En conclusión, el rendimiento del entorno nativo en macOS con chip M2 es claramente superior al de la máquina virtual Linux. La Mac ejecuta las operaciones con menor latencia, mayor estabilidad y mejor aprovechamiento del paralelismo, mientras que la VM sufre penalizaciones notables por virtualización y gestión de hilos. Por tanto, el M2 demuestra ser el entorno más eficiente para la multiplicación de matrices y la ejecución concurrente.

Link de Repositorio:

https://github.com/sjhernandezr/Evaluacion-Rendimiento