Análisis comparativo de técnicas de micro benchmarking en MPI y Open MP para la multiplicación de matrices

Camilo Ernesto Cano, Daniel Duque, Juan Sebastián Pineda, Nicolas Mendez

Pontificia Universidad Javeriana

Maestría de Inteligencia Artificial

Abstract: Matrix multiplication is a fundamental operation in most scientific and engineering applications. Image processing, graphic design and machine learning are some of the areas commonly used. As matrix multiplication requires many operations, it can be computing and intensive and time consuming. In this document there will be shown a developing process of multiple parallelization process and its processing benchmark. We conclude that a good optimization of the matrix multiplication can save up to double the usual time of the process, however it shows diminishing returns in terms of number of processes.

Keywords: High performance computing, MPI, OMP, Parallel computing

La multiplicación de matrices es una operación fundamental en muchas aplicaciones científicas y de ingeniería, incluidos los gráficos por computadora, el procesamiento de imágenes y el aprendizaje automático. Como la multiplicación de matrices implica numerosos cálculos, puede ser computacionalmente intensivo y llevar mucho tiempo para matrices grandes. La computación paralela de Open MP se puede utilizar para acelerar la multiplicación de matrices mediante la distribución de la carga de trabajo entre varios subprocesos. Este enfoque permite el uso eficiente de varios núcleos dentro de una arquitectura de memoria compartida, lo que da como resultado tiempos de ejecución más rápidos y un rendimiento mejorado. Paralelizar la multiplicación de matrices usando Open MP puede reducir significativamente el tiempo de procesamiento requerido para cálculos de matrices grandes, lo que permite un procesamiento de datos más rápido y eficiente en una variedad de aplicaciones. Open MP proporciona una manera simple y eficaz de paralelizar la multiplicación de matrices y es una herramienta valiosa para mejorar el rendimiento de las aplicaciones informáticas intensivas.

1. Objetivos

Analizar el impacto de la paralelización de los algoritmos, evaluando el rendimiento de los mismos al variar la cantidad de hilos

2. Recursos

Todo el experimento se llevó a cabo en una sola maquina con las siguientes especificaciones.

```
estudiantes@worker5:~$ lscpu
Arquitectura:                    x86_64
modo(s) de operación de las CPUs:    32-bit, 64-bit
Orden de los bytes:              Little Endian
Address sizes:                   39 bits physical, 48 b
                                 its virtual
CPU(s):                          20
Lista de la(s) CPU(s) en línea:      0-19
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:          10
«Socket(s)»                      1
Modo(s) NUMA:                    1
ID de fabricante:                GenuineIntel
Familia de CPU:                  6
Modelo:                          165
Nombre del modelo:               Intel(R) Xeon(R) W-129
                                 0 CPU @ 3.20GHz
Revisión:                        5
CPU MHz:                         3200.000
CPU MHz máx.:                    5200,0000
CPU MHz mín.:                    800,0000
BogoMIPS:                        6399.96
Virtualización:                  VT-x
Caché L1d:                       320 KiB
Caché L1i:                       320 KiB
Caché L2:                        2,5 MiB
Caché L3:                        20 MiB
CPU(s) del nodo NUMA 0:          0-19
Vulnerability Itlb multihit:     KVM: Mitigation: VMX d
                                 isabled
```

Fig. 1. Lanzador

```
void Matrix_Init_col(int SZ, double *a, double *b, double *c){
int j,k;

        for (k=0; k<SZ; k++)
                for (j=0; j<SZ; j++) {
                        a[j+k*SZ] = 2.0*(j+k);
                        b[k+j*SZ] = 3.2*(j-k);
                        c[j+k*SZ] = 1.0;
                }
}
```

Fig. 2. Creación de las matrices

Sobre las entradas de las matrices ya construidos los parámetros vemos que se establecen en las anteriores líneas de código, tomando valores aparentemente arbitrarios diferentes para la matriz a y b, e inicializando la matriz c (que será el resultado de la multiplicación) con números 1:

2.1. *MM1F*

```
#pragma omp for
        for (i=0; i<SZ; i++)
                for (j=0; j<SZ; j++) {
                        double *pA, *pB, S;
                        S=0.0;
                        pA = a+(i*SZ); pB = b+(j*SZ);
                for (k=SZ; k>0; k--, pA++, pB++)
                        S += (*pA * *pB);
                c[i*SZ+j]= S;
        }

        Sample_Stop(THR);
        }

  Sample_End();
}
```

Fig. 3. Operación de matrices.

Nos centraremos en los siguientes fragmentos que son los que cambian entre los diferentes códigos. Tomamos como línea de base el MM1f en donde vemos que las matrices se multiplican por la iteración respectiva de los valores uno por uno en el loop que recorre las matrices y agrega los datos que encuentra en valores intermedios, luego en otro loop los agrega en una variable S, y por último reemplaza el valor S de esa operación en la ubicación correspondiente en la matriz de resultados. Por supuesto esto se hace para todas las operaciones necesarias en la iteración.

2.2. *MM1FU*

```
#pragma omp for
        for (i=0; i<SZ; i++)
                for (j=0; j<SZ; j++) {
                        double *pA, *pB, c0, c1, c2, c3;
                        c0=c1=c2=c3=0.0;
                        pA = a+(i*SZ);
                        pB = b+(j*SZ);
                        k=SZ;
                        while (k&3) { // in case SZ is not a multiple of 4
                                c0 += (*pA * *pB);
                                k--; pA++; pB++;
                        }
                        for (; k>0; k-=4, pA+=4, pB+=4) {
                                c0 += (*pA * *pB);
                                c1 += (*(pA+1) * *(pB+1));
                                c2 += (*(pA+2) * *(pB+2));
                                c3 += (*(pA+3) * *(pB+3));
                        }
                        c[i*SZ+j]= c0+c1+c2+c3;
                }

                Sample_Stop(THR);
        }

        Sample_End();
}
```

Fig. 4. Operación de matrices agilizando con cuatro datos a la vez

2.3. *MM2F*

```
#pragma omp for
        for (i=0; i<SZ; i+=2)
                for (j=0; j<SZ; j+=2) {
                        double *pA, *pB, c0, c1, c2, c3;
                        c0=c1=c2=c3=0.0;
                        pA = a+(i*SZ); pB = b+(j*SZ);
                        for (k=SZ; k>0; k-=2, pA+=2, pB+=2){
                                double a0, a1, a2, a3;
                                double b0, b1, b2, b3;

                                a0 = *pA;  a1 = *(pA+1);  a2 = *(pA+SZ);  a3 = *(pA+SZ+1);
                                b0 = *pB;  b1 = *(pB+1);  b2 = *(pB+SZ);  b3 = *(pB+SZ+1);

                                c0 += a0*b0 + a1*b1;
                                c1 += a0*b2 + a1*b3;
                                c2 += a2*b0 + a3*b1;
                                c3 += a2*b2 + a3*b3;
                        }
                        pB = c+i*SZ+j;
                        *pB= c0;  *(pB+1)= c1;  *(pB+SZ)= c2; *(pB+SZ+1)= c3;
                }

        Sample_Stop(THR);
        }

        Sample_End();
}
```

Fig. 5. Operación de matrices agilizando con sub matrices para las operaciones.

En este fragmento nos damos cuenta de que se utilizan los apuntadores de memoria para hacer los cálculos de la matriz en las siguientes filas, es decir le agrega el tamaño de la matriz para bajar de fila y no tener que pedirle al sistema que busque la ubicación del siguiente valor por fila, sino que se lo entrega directamente. Esto unido a que mantiene las 4 operaciones simultaneas de la estrategia anterior, debería mejorar el rendimiento por el buen manejo de la memoria. La ejecución de estas operaciones hace las veces de un subset de matriz, que permite calcular la totalidad de una parte de la matriz en dos franjas de línea.

3. MPI

Para hacer los cambios las funciones a MPI usamos varias caracterizaciones. Para explicarlas utilizaremos como ejemplo MM1F a detalle y luego explicaremos a grandes rasgos como cambiaron los otros dos scripts utilizados.

3.1. MM1F

```
double time1, time2, time3;  // for timing measurements
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

MPI_Request ireq[128]; // , asynch request, assume size<128
MPI_Status stat;       // status of asynch communication

// compute interval size each process is responsible for
```

Fig. 6. MM1F con MPI

En primer lugar, utilizamos MPI justo después de definir el tamaño de las matrices y sus rangos, Comenzamos inicializando la ejecución con el paquete MPI y agregando los valores de rango y tamaño para que pueda utilizar adecuadamente la comunicación entre el hardware. Al igual que permite las peticiones no bloqueadas y el status de la comunicación asincrónica.

```
if (rank==0){  // root/coordinator process

        Matrix_Init_col(N, a, b, c);

        time1 = MPI_Wtime();     // record time
        // For array B, we broadcast the whole array, however, the Bcast
        // operation is strange because it needs to be executed by all
        // processes.  There is no corresponding Recv for a Bcast
        MPI_Bcast(a,N*N,MPI_DOUBLE,0,MPI_COMM_WORLD); // send broadcast
        // printf("%d: Bcast complete\n",rank);

        // Send intervals of array A to worker processes
        for(i=1; i<size; i++)
                MPI_Isend(a+(i*interval), interval*N, MPI_DOUBLE, i, i, MPI_COMM_WORLD, ireq+i);
        for(i=1;i<size;i++)
                MPI_Waitany(size, ireq, &j, &stat); // join on sends

        mm1f_interval(0, interval, N);               // local work
        mm1f_interval(size*interval, remainder, N);  // remainder

        //get results from workers:
        for(i=1; i<size; i++)
                MPI_Irecv(c+(i*interval), interval*N, MPI_DOUBLE, i, i, MPI_COMM_WORLD, ireq+i);
        for(i=1;i<size;i++){
                MPI_Waitany(size, ireq, &j, &stat);
                //printf("received results from process %d\n",j);
        }

        /*================= conventional MM1c ========================*/
        time2 = MPI_Wtime();
```

Fig. 7. Ejemplo de proceso de paso de mensajes

Después y dentro del código, utilizamos mpi para varios procesos. Primero, inicializar el tiempo para posteriormente medir la demora. Segundó mandamos la primera matriz a todos los nodos (también lo haremos con la segunda en la siguiente operación). Tercero se envían intervalos del proceso, según el tamaño de la matriz, para que los diferentes nodos reciban la información y posteriormente la usen. Por último, en esta parte del código se ejecuta el trabajo en los nodos y se devuelve el resultado (además de imprimir el tiempo. En resumen, esta parte es el trabajo de coordinación mediante MPI del manager del proceso.

```
        }
else {          // worker process
        MPI_Bcast(b, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD); // receive broadcast
        // synchronous receive
        MPI_Recv(a+(rank*interval), interval*N, MPI_DOUBLE,0,rank, MPI_COMM_WORLD,&stat);
        mm1f_interval(rank*interval, interval, N);
        // send results back to root process, synchronous send
        MPI_Send(c+(rank*interval), interval*N, MPI_DOUBLE, 0, rank, MPI_COMM_WORLD);
}
MPI_Finalize();
```

Fig. 8. Operación de paso de mensajes

En esta parte del proceso los nodos trabajadores reciben los datos enviados por el Bcast, ejecutan el proceso en la matriz y lo envían ya procesado al nodo de manager. Por último, finalizan el proceso mpi.

3.2. MM1FU y MM2F

```
int main(int argc, char** argv){
        int rank, size, interval, remainder, i, j;
        int N  = (int) atof(argv[1]);

        a = MEM_CHUNK;
        b = a + N*N;
        c = b + N*N;

        double time1, time2, time3;  // for timing measurements
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,&size);

        MPI_Request ireq[128]; // , asynch request, assume size<128
        MPI_Status stat;       // status of asynch communication

        // compute interval size each process is responsible for,
        // rank 0 process will be responsible for the remainder
        interval = N/(size);
        remainder = N % (size);

        if (rank==0){  // root/coordinator process

                Matrix_Init_col(N, a, b, c);

                time1 = MPI_Wtime();     // record time
                // For array B, we broadcast the whole array, however, the Bcast
                // operation is strange because it needs to be executed by all
                // processes.  There is no corresponding Recv for a Bcast
                MPI_Bcast(a,N*N,MPI_DOUBLE,0,MPI_COMM_WORLD); // send broadcast
                // printf("%d: Bcast complete\n",rank);

                // Send intervals of array A to worker processes
                for(i=1; i<size; i++)
                        MPI_Isend(a+(i*interval), interval*N, MPI_DOUBLE, i, i, MPI_COMM_WORLD, ireq+i);
                for(i=1;i<size;i++)
                        MPI_Waitany(size, ireq, &j, &stat); // join on sends

                mm1f_interval(0, interval, N);               // local work
                mm1f_interval(size*interval, remainder, N);  // remainder

                //get results from workers:
                for(i=1; i<size; i++)
                        MPI_Irecv(c+(i*interval), interval*N, MPI_DOUBLE, i, i, MPI_COMM_WORLD, ireq+i);
                for(i=1;i<size;i++){
                        MPI_Waitany(size, ireq, &j, &stat);
                        //printf("received results from process %d\n",j);
                }

                /*================= conventional MM1c ==========================*/
                time2 = MPI_Wtime();

                /*==============================================================*/

                printf("%9.f\n", (time2-time1)*1000000);
                }
        else {          // worker process
                MPI_Bcast(b, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD); // receive broadcast
                // synchronous receive
                MPI_Recv(a+(rank*interval), interval*N, MPI_DOUBLE,0,rank, MPI_COMM_WORLD,&stat);
                mm1f_interval(rank*interval, interval, N);
                // send results back to root process, synchronous send
                MPI_Send(c+(rank*interval), interval*N, MPI_DOUBLE, 0, rank, MPI_COMM_WORLD);
        }
        MPI_Finalize();
} //main
```

Fig. 9. ejemplo MPI

Como se muestra el proceso de MPI es exactamente el mismo, y los cambios ocurren dentro del proceso de multiplicación de matrices que analizamos con anterioridad. Al igual que lo que ocurre con MM2f.

4. Lanzador

Se realizaron 3 micro benchmarks con las siguientes configuraciones experimentales, en las cuales se variaron los niveles de las variables threads (hilos) y tamaño de la matriz. A continuación, se presenta el input file empleado para ejecutar los experimentos.

```
##########################################################################################
# Inputs file experiments
##########################################################################################

Open MP                          // Kind of Experiment
MM1f,MM1fu,MM2f                  // Binarys
1,2,4,8,10,12,14,16,18,20        // Threads
30                               // Number of Repetitions per experiment


##########################################################################################
# Inputs file => Matrix Size NxN
##########################################################################################
:1000
:2000
:4000
:8000
```

5. Diseño experimental

Para el estudio del benchmarking de los procesos Open MP Y MPI, se realizarán análisis de anovas, como forma de contrastar influencias de las variables seleccionadas, dado que son demasiados niveles y deben ser evaluados al tiempo. No se optó por un diseño experimental de cuadro latino o 2^n. Se toman por muestra por nivel de 30 repeticiones, esto se hace con la finalidad de poder hacer un análisis por regresión y tener una cantidad de puntos deseables para revisar supuestos de normalidad.

Como resultado se plantean un total de 3600 experimentos, estos fueron repartidos de la siguiente manera:

- Métodos: 3 niveles (MM1f, MM1fu, MM2f)
- Tamaños de las matrices (1000,2000,4000,8000)
- Cantidad de hilos (1,2,4,8,10,12,14,16,18,20)

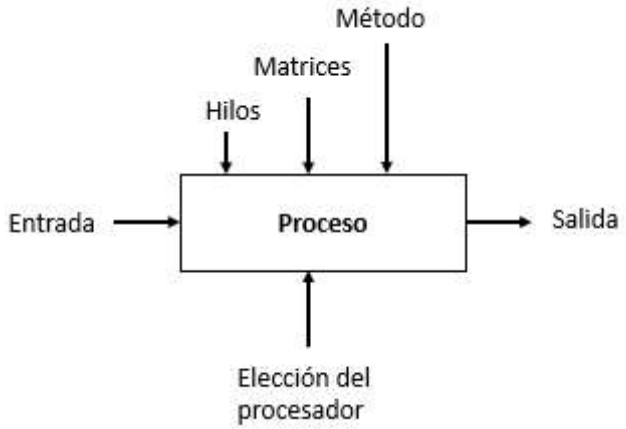


Fig. 10. Diagrama de experimentos

Con los resultados esperados, se espera responder las siguientes preguntas bases, de todo el trabajo, ¿Existe diferencia en los rendimientos de MPI y Open MP?, ¿El método de multiplicación de matrices presenta una diferencia en su tiempo de ejecución?, ¿Cómo afecta el tamaño de las matrices? Y ¿Cómo mejora el rendimiento de la ejecución acorde a la cantidad de hilos ejecutados?

6. Análisis de resultados

Como resultado del lanzador, encontramos la ejecución de 3.600 experimentos, el cual se validarán si los niveles descritos previamente tienen influencia en la cantidad de tiempo esperado en cada ejecución.

Method/ Size (x1000)	Time (min) TH									
	1	2	4	8	10	12	14	16	18	20
MM1f	2169	1096	561	302	256	217	188	167	152	151
1	13	7	3	2	2	1	1	1	1	1
2	115	59	30	16	14	12	10	9	8	8
4	939	477	245	132	111	94	82	73	66	65
8	7608	3842	1967	1058	899	759	661	586	533	532
MM1fu	1638	842	421	235	201	231	206	203	192	178
1	9	4	2	1	1	2	1	1	1	1
2	86	46	23	12	11	13	12	11	10	9
4	715	368	184	102	89	101	90	88	83	77
8	5744	2949	1476	824	703	807	719	713	672	624
MM2f	635	324	163	94	82	90	81	78	73	67
1	3	2	1	0	0	1	0	0	0	0
2	32	18	9	5	4	5	4	4	4	3
4	270	142	71	41	35	39	35	33	31	28
8	2234	1135	571	331	288	316	283	275	256	238

Tabla. 1. Resultados de los experimentos Open MP

Method/ Size (x1000)	Time (min) TH									
	1	2	4	8	10	12	14	16	18	20
MM1f		2393	1255	685	623	2385				
1		16	9	5	4					
2		131	69	37	33					
4		1053	549	305	272					
8		8372	4392	2453	2184	2385				
MM1fu	2354	1191	660	581	595	588	572	567	566	577
1	14	7	4	3	4	4	4	4	4	4
2	125	65	36	31	31	32	31	30	30	31
4	1030	528	288	254	257	254	248	245	245	251
8	8248	4243	2313	2035	2068	2062	1988	1989	1987	2005
MM2f	4184	2170	1126	645	615	690	638	611	595	593
1	27	14	8	4	4	5	4	4	4	4
2	224	117	62	34	33	38	36	33	32	32
4	1833	940	493	282	267	295	285	263	259	260
8	14651	7467	3943	2260	2154	2424	2227	2115	2084	2076

Tabla. 2. Resultados de los experimentos MPI

Al comparar los resultados de las matrices se evidencia que usando un tamaño 8.000, el tiempo utilizado tiene una relación exponencial con los hilos usados, haciendo esta una relación exponencial. Muestra que en un punto el proceso no tendrá mayor eficiencia a medida que se usen más hilos, dado que encontrará una asíntota en el rendimiento.

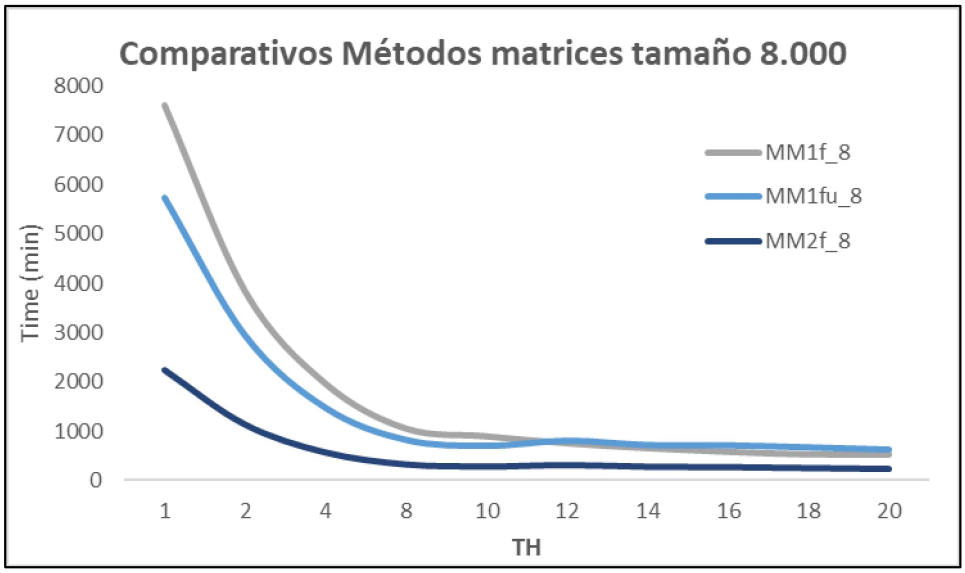
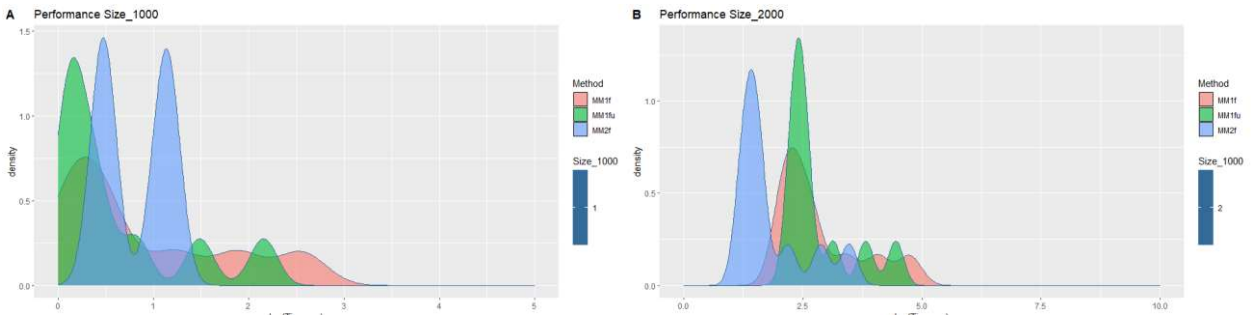


Fig. 11. Gráfica de resultados de los experimentos

6.1. Graficas de densidad

Open MP

Al linealizar los datos con logaritmo con la finalidad de poder graficar todos los hilos y los métodos, se encuentra que para matrices mayores a 2.000; la matriz MM2f es las más eficiente seguido de la MM1FU. Esto nos muestra que efectivamente el procesamiento entregándole la ubicación de memoria a buscar agiliza el proceso, al igual que la estrategia de pasar cada iteración del proceso con varios datos como mencionamos anteriormente.



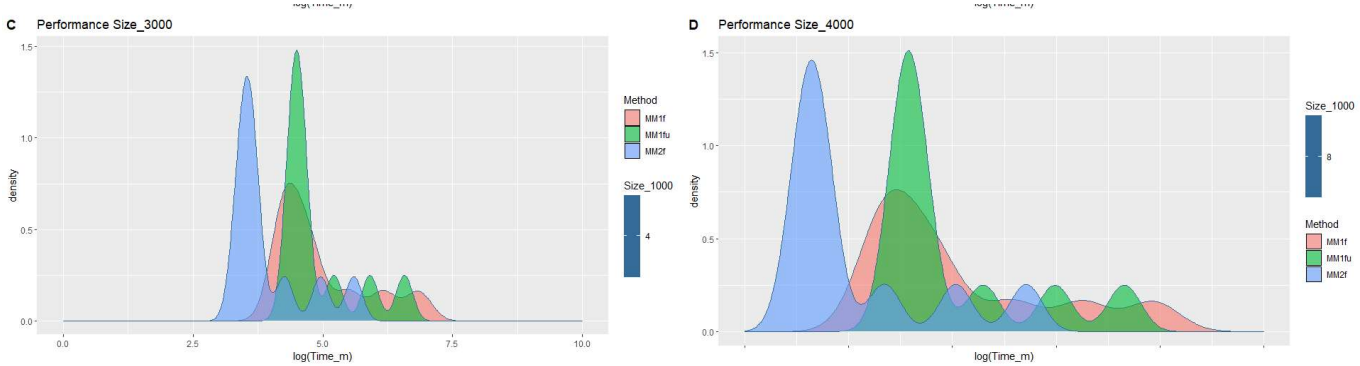


Fig. 12. Gráfica de resultados de los experimentos por tamaño de la matriz y método en Open MP.

MPI

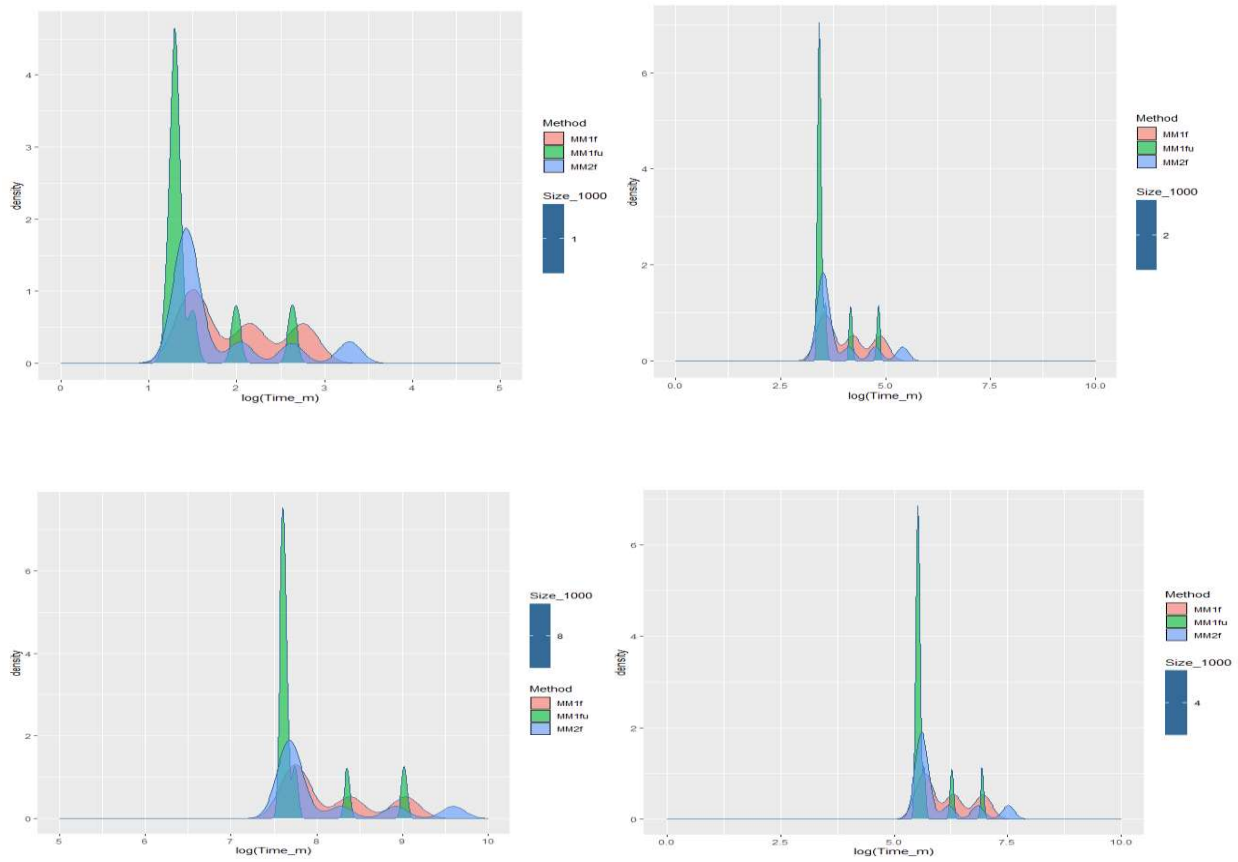


Fig. 13. Gráfica de resultados de los experimentos por tamaño de la matriz y método en MPI.

6.2. Boxplots

Al comparar estos resultados por medio del diagrama de caja y bigotes, evidenciamos que el método **MM2f** es el más eficiente, siempre y cuando los tamaños de las matrices sean mayores a 1000.

Open MP

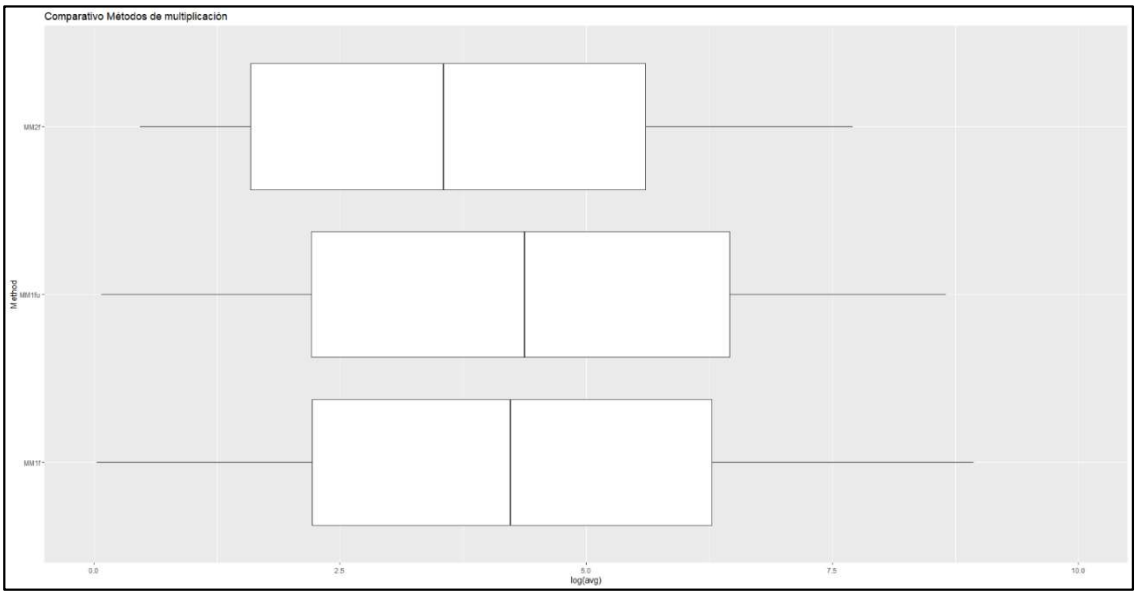


Fig. 14. Boxplot de resultados de los experimentos por método Open MP.

MPI

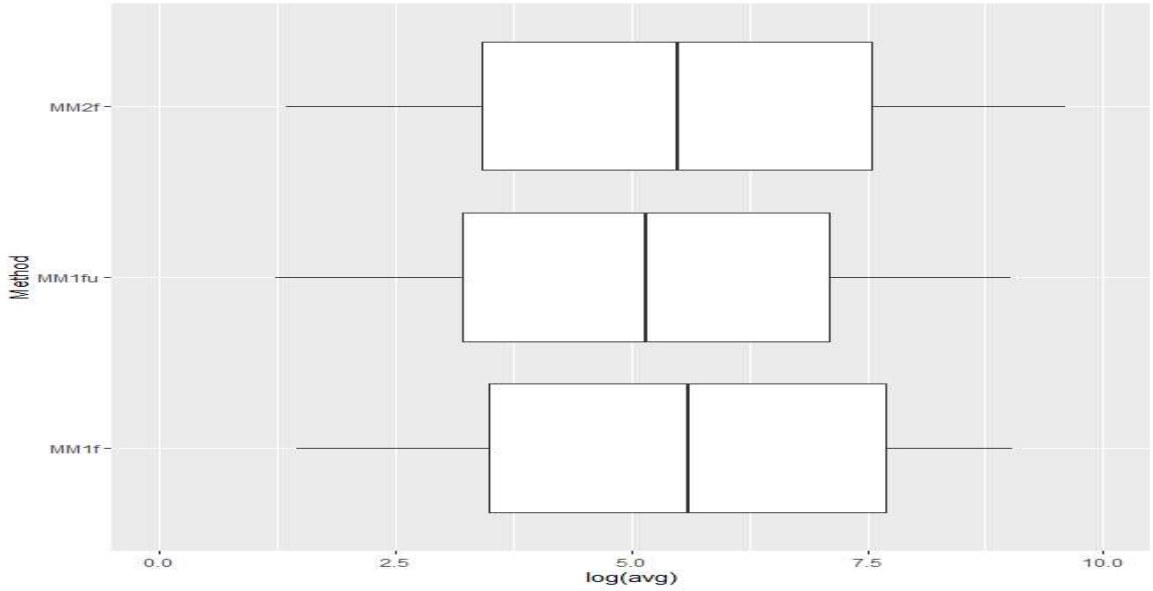


Fig. 15. Boxplot de resultados de los experimentos por método MPI.

Por lo o contrario al analizar los boxplot correspondientes a la ejecución usando Open MP no es observable una diferencia en las medidas de tendencia central de las ejecuciones para los diferentes métodos.

6.3. Mosaico temporal (Tamaño – método)

Open MP

Al hacer un análisis bivariado para mirar la relación entre métodos e hilos, se encontró por medio de la prueba de chi cuadrado, que no existe evidencia de que las medias se parezcan, por lo tanto, existen diferencias entre métodos e hilos.

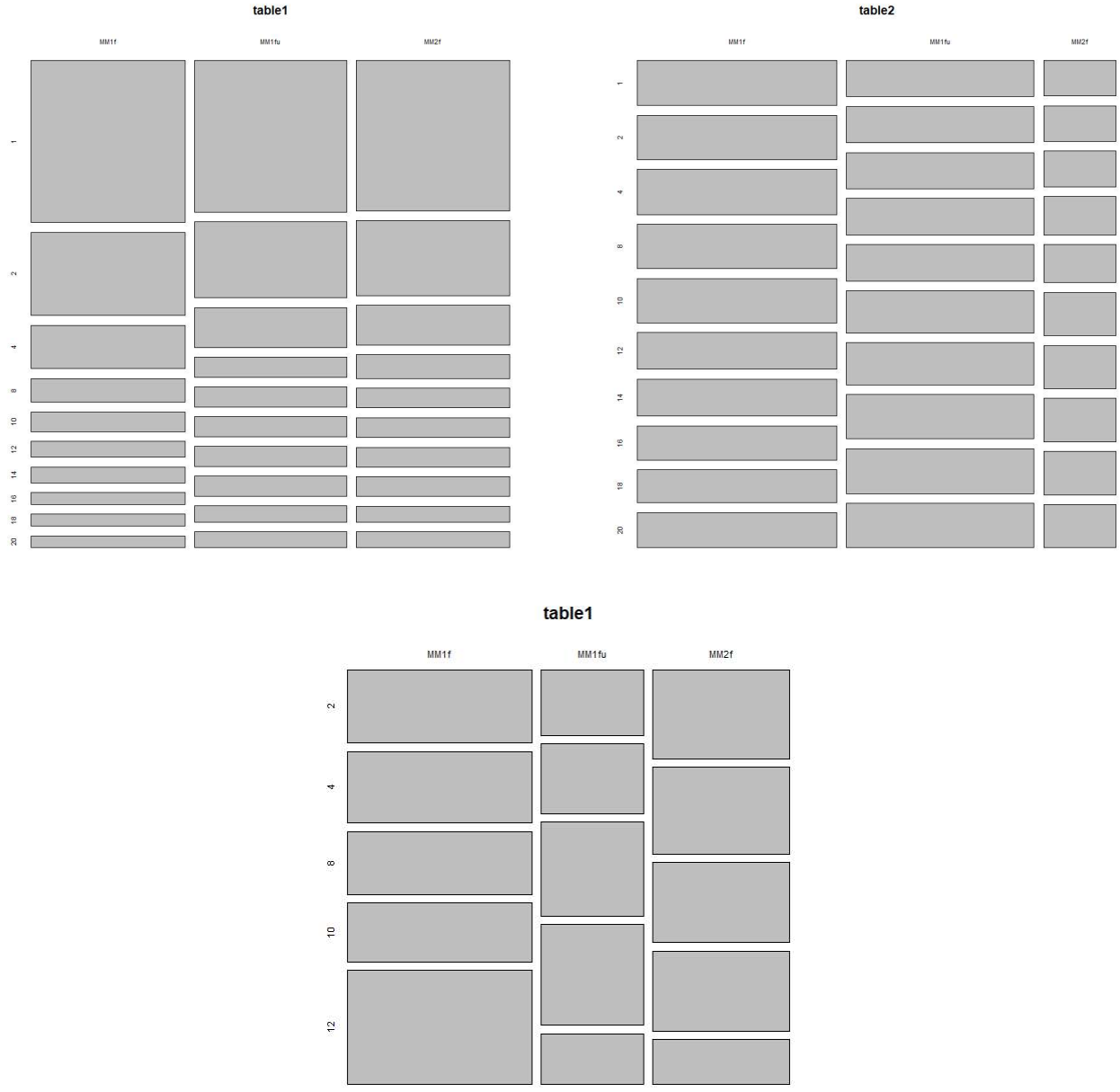


Fig. 16. Áreas temporales del proceso.

MPI

Esto lo evidenciamos también de otra manera, al comparar los promedios, encontramos que si existe una varianza enorme, causada por el número de hilos usados para procesar la información.

Size_1000	avg	median_time	std_time
1	2,1	1,2	2,7
2	20,1	10,9	25,1
4	163,5	87,9	205,9
8	1320,1	705,8	1662,6

Tabla 3. Resultados promedio, la media del tiempo y la desviación estándar

Comparando la diferencia de medias entre métodos y tamaño de matrices, la prueba chi cuadrado, que no existe evidencia de que las medias se parezcan, por lo tanto, existen diferencias entre métodos y tamaños.

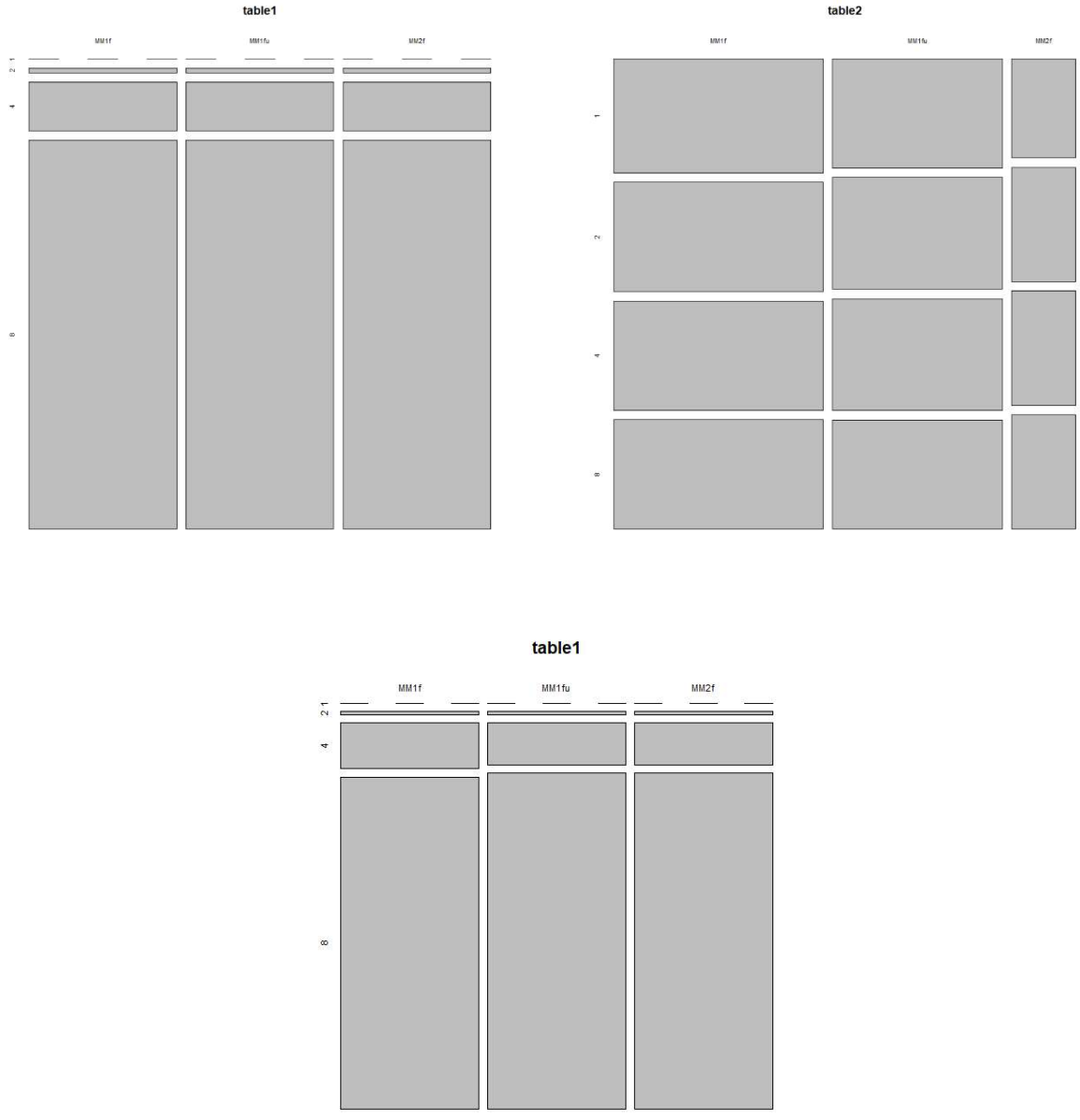


Fig. 17. Áreas temporales del proceso.

6.4. *Búsqueda de formula por medio de regresión*

Posterior a los análisis encontrados por medio de los análisis descriptivos, se buscaba encontrar por cada método una regresión, para explicar los resultados encontrados.

MM1F

```
Call:
lm(formula = log(Time_m) ~ log(Size_1000) + log(TH), data = df1)

Residuals:
      Min        1Q    Median        3Q       Max
-0.153193 -0.030596  0.001115  0.029366  0.156120

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)    2.541569   0.003582   709.6   <2e-16 ***
log(Size_1000) 3.061514   0.001716  1784.1   <2e-16 ***
log(TH)       -0.902923   0.001386  -651.3   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.04607 on 1197 degrees of freedom
Multiple R-squared:  0.9997,    Adjusted R-squared:  0.9997
F-statistic: 1.804e+06 on 2 and 1197 DF,  p-value: < 2.2e-16
```

Fig. 18. Resultados regresión lineal MM1F.

Open MP

Por medio de la prueba Shapiro, se encontró indicios de normalidad en los residuales de los errores, por lo que la regresión es apta para el análisis.

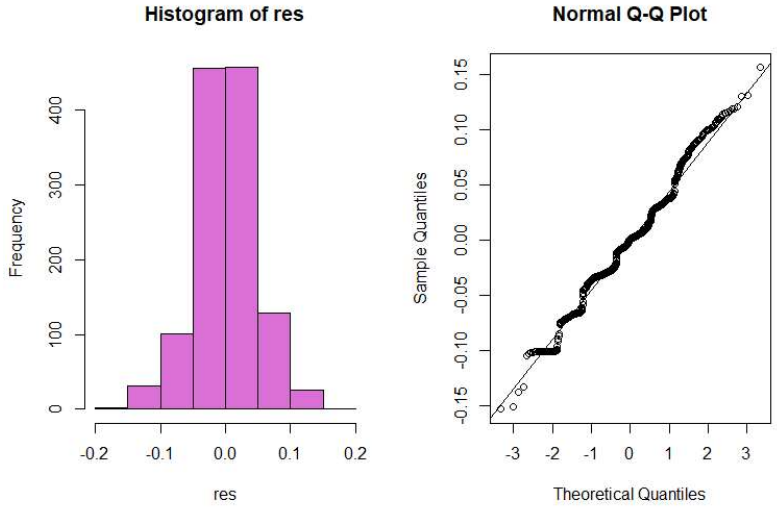


Fig. 19. Histograma de los residuos de la regresión Open MP.

MPI

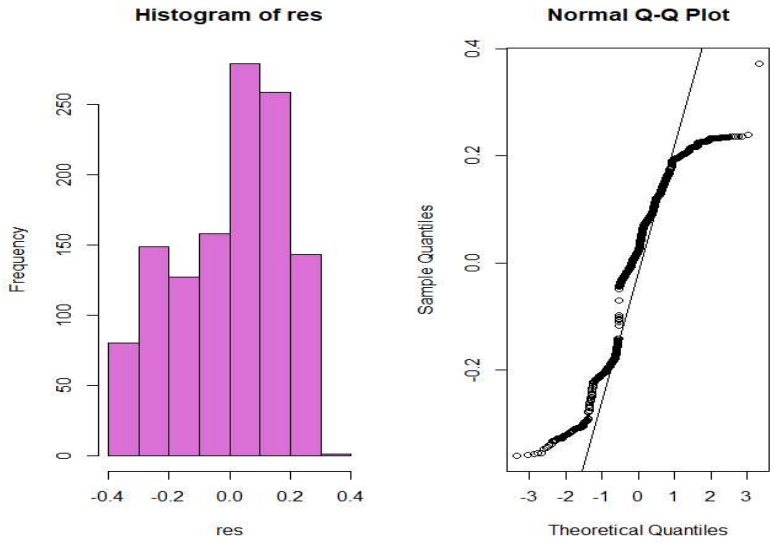


Fig. 20. Histograma de los residuos de la regresión MPI.

MM1fu

Con respecto al método M1fu, encontramos que se tiene problemas para generalizar las matrices de tamaño 8.000, siendo más eficiente en tamaños grandes que en menores.

```
Call:
lm(formula = log(Time_m) ~ log(Size_1000) * log(TH), data = df1)

Residuals:
     Min       1Q   Median       3Q      Max
-0.51397 -0.15038  0.08269  0.13578  0.28717

Coefficients:
                       Estimate Std. Error t value Pr(>|t|)
(Intercept)            2.009534   0.020000 100.476  < 2e-16 ***
log(Size_1000)         3.136010   0.015423 203.330  < 2e-16 ***
log(TH)               -0.677025   0.008928 -75.832  < 2e-16 ***
log(Size_1000):log(TH) -0.030693   0.006885  -4.458 9.05e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.1773 on 1196 degrees of freedom
Multiple R-squared:  0.9949,    Adjusted R-squared:  0.9949
F-statistic: 7.811e+04 on 3 and 1196 DF,  p-value: < 2.2e-16
```

Fig. 21. Regresión de resultados.

Open MP

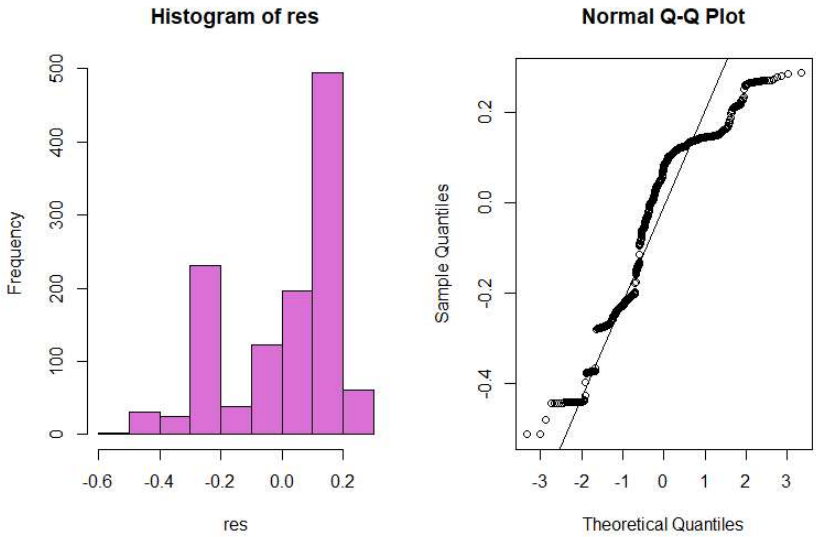


Fig. 22. Histograma de los residuos de la regresión.

MPI

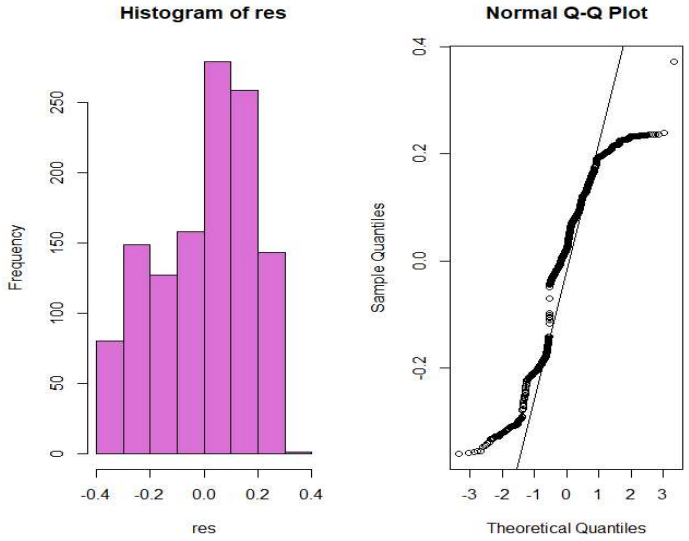


Fig. 23. Histograma de los residuos de la regresión.

MMf2

Con respecto a esta regresión, se encuentra que esta sesgada a la izquierda, mostrando que los resultados más complejos de generalizar son los asociados a matrices de tamaño 8000.

```
Call:
lm(formula = log(Time_m) ~ log(Size_1000) + log(TH), data = df2)

Residuals:
     Min       1Q   Median       3Q      Max
-0.39460 -0.13426  0.05662  0.11362  0.27884

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     1.075194   0.011607   92.64   <2e-16 ***
log(Size_1000)  3.122667   0.005561  561.55   <2e-16 ***
log(TH)        -0.726128   0.004493 -161.63   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.1493 on 1197 degrees of freedom
Multiple R-squared:  0.9965,    Adjusted R-squared:  0.9965
F-statistic: 1.707e+05 on 2 and 1197 DF,  p-value: < 2.2e-16
```

Fig. 24. Regresión de los resultados MMF2.

Open MPI

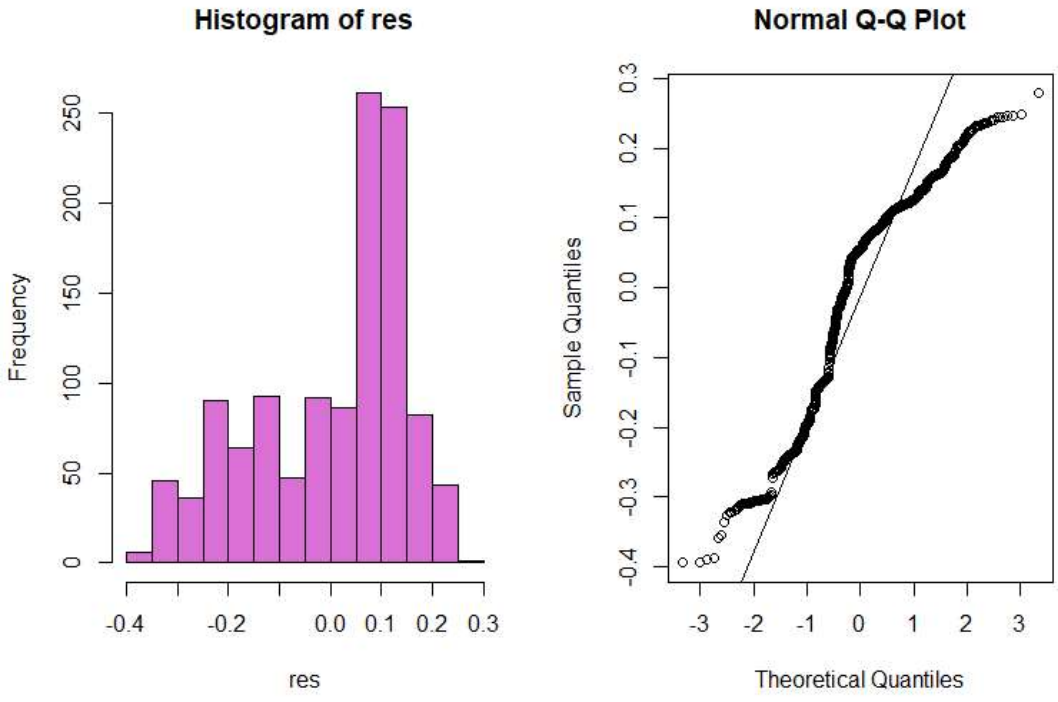


Fig. 25. Historial de los residuos de la regresión.

MPI

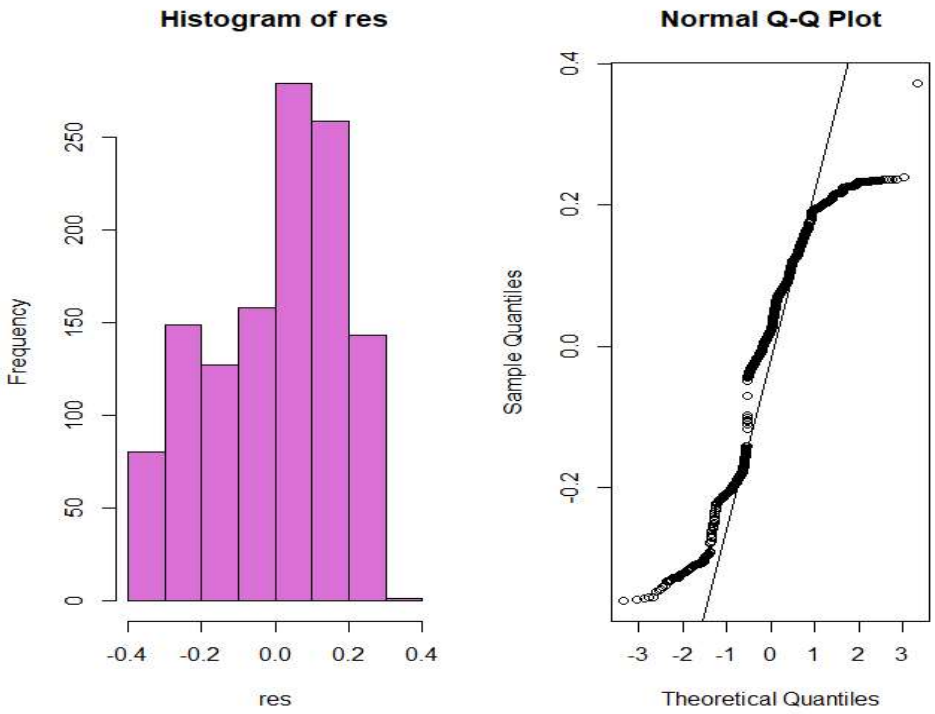


Fig. 26. Historial de los residuos de la regresión.

6.5. Regresión encontrada

Al ser una ecuación de la forma:

$$y = e^{\beta_0} + {\beta_1}^{Size} {\beta_2}^{TH}$$

Nos centramos en la primera expresión del intercepto, en donde el modelo MM2f, es el mejor modelo para realizar multiplicación de matrices.

7. Conclusiones

En este estudio, hemos llevado a cabo un análisis comparativo exhaustivo de técnicas de micro benchmarking en dos paradigmas de programación paralela ampliamente utilizados: MPI (Message Passing Interface) y Open MP (Open Multi-Processing), con el objetivo de evaluar el rendimiento de la multiplicación de matrices. A lo largo de nuestro análisis, hemos examinado aspectos clave como la escalabilidad, el tiempo de ejecución y el uso de recursos en ambas tecnologías, con el fin de proporcionar una visión comparativa y respaldada por evidencia empírica sobre las fortalezas y debilidades de cada enfoque. A continuación, presentamos las conclusiones obtenidas de nuestro estudio, que arrojan luz sobre las decisiones de implementación y selección de paradigmas de programación para la multiplicación de matrices en entornos de computación paralela:

1. Nuestro análisis revela que las estrategias implementadas pueden lograr ahorros significativos en el tiempo de ejecución de manera prácticamente exponencial en la máquina sobre todo con Open MP. Además, estos ahorros están estrechamente asociados con la reducción de energía utilizada y los costos de desarrollo asociados.
2. Open MP se destaca como una opción preferible a MPI para ejecutar un microbenchmark en una sola máquina, debido a su menor sobrecarga de comunicación y sincronización, lo que resulta en un mejor rendimiento y tiempos de ejecución más rápidos en entornos de una sola máquina.
3. En conclusión, el análisis de dispersión de los tiempos de ejecución en los Boxplots, reveló que todos los cuantiles de tiempo de ejecución para el modelo MM2f fueron consistentemente más bajos en comparación con los otros métodos evaluados. Esto indica una mayor eficiencia y rendimiento del modelo MM2f en la multiplicación de matrices, lo que lo posiciona como la opción preferida para obtener resultados más rápidos en esta tarea específica.