Entrega 3

Andrés Milla - 14934/6
Rodrigo Jara - 14854/8

Ejercicio 1

Punto 2, Práctica 4

Tiempo

Usando 1 único nodo con 4 procesos | P = 4		Usando 2 nodos con 2 procesos c/u | P =4	
Retorno del control		Retorno del control	
Blocking	Non-blocking	Blocking	Non-blocking
2.000092	0.000011	1.999964	0.00001
4.000087	2.000089	4.020545	2.000109
6.000117	4.000066	6.010489	4.010842
Usando 1 único nodo con 8 procesos | P = 8		**Usando 2 nodos con 8 procesos c/u | P = 16**	
Retorno del control		Retorno del control	
Blocking	Non-blocking	Blocking	Non-blocking
2.000105	0.00001	2.000114	0.000016
4.00429	2.000081	4.000027	2.000183
6.004282	4.000087	6.000001	4.000133
8.000097	6.000075	8.000043	5.999989
10.000093	8.000287	9.999997	8.000119
12.000057	10.000265	12.000136	10.000251
14.000048	12.00008	14.000041	12.000097
		16.011687	14.000229
		18.010418	16.002053
		20.01054	18.000984
		22.010499	20.001134
		24.010605	22.000999
		26.010553	24.001207
		28.010544	26.001209
		30.010469	28.001105

Conclusión

Como vemos en la tabla, el algoritmo *non-blocking* retorna inmediatamente el control. Esto sucede porque retorna el control al haber depositado la solicitud en el buffer local del sistema, en cambio el algoritmo *blocking* debe esperar a que el receptor reciba el mensaje.

Podemos ver que en el algoritmo *non-blocking* siempre se devuelve aproximadamente 2 segundos antes el control ya que no debe esperar el *sleep* de 2 segundos del proceso emisor como lo hace el algoritmo *blocking*.

Tener en cuenta que ambos programas tardan el mismo tiempo en terminar ya que el proceso que recibe en el algoritmo *non-blocking* tiene una cláusula *wait* para esperar la recepción del mensaje completo antes de pasar al siguiente emisor.

Al quitar la línea *MPI_Wait()* los mensajes no se imprimen correctamente ya que el proceso master hará el *for* sin esperar a cada worker porque realiza un *receive no bloqueante*, es decir, sólo dejará la solicitud pendiente en el buffer local del sistema y seguirá ejecutando. Por lo tanto siempre imprimirá *"Este mensaje no se debería ver"*, el cual es el valor por defecto que tiene la variable *message*.

- Nota: *El contenido de esta variable nunca va a cambiar a menos que las sentencias de comunicación que realiza el proceso master tarden más que el tiempo que están dormidos los procesos.*

Punto 3, Práctica 4

Tiempos

a - Usando 1 único nodo con 4 procesos | P = 4		
N	**Tiempo de comunicación**	
	Blocking	**Non-blocking**
10000000	0.248657	0.221127
20000000	0.494817	0.440358
40000000	0.980674	0.88357
b - Usando 2 nodos con 2 procesos c/u | P =4		
N	**Tiempo de comunicación**	
	Blocking	**Non-blocking**
10000000	1.491586	0.749504
20000000	2.969807	1.544038
40000000	5.924863	3.045919
c - Usando 1 único nodo con 8 procesos | P = 8		
N	**Tiempo de comunicación**	
	Blocking	**Non-blocking**
10000000	0.551336	0.297088
20000000	1.098626	0.590614
40000000	2.183434	1.175593
d - Usando 2 nodos con 8 procesos c/u | P = 16		
N	**Tiempo de comunicación**	
	Blocking	**Non-blocking**
10000000	2.339335	0.94722
20000000	4.66059	2.028696
40000000	9.278954	3.964059

Conclusión

Podemos ver que el algoritmo *non-blocking* requiere menos tiempo de comunicación.
Esto se debe a la semántica de los dos tipos de comunicación que utiliza cada algoritmo:

- En el algoritmo *blocking* los procesos se comunican en forma de cascada, es decir, un proceso no puede continuar hasta que el anterior le envíe sus datos. El último proceso será el primero en enviar sus datos invirtiendo las sentencias *receive/send* para no generar deadlock.
  Al utilizar comunicación bloqueante cada proceso debe esperar a recibir el vector completo antes de poder enviar su vector al proceso siguiente, esto genera que el último proceso sea el último en recibir su vector, lo cual produce ocio.

- En cambio, en el algoritmo *non-blocking* una vez que las solicitudes de comunicación de recepción y envío fueron depositadas en el buffer local del sistema, se devuelve inmediatamente el control al proceso que las realizó como vimos en el ejercicio anterior. Por lo tanto no se debe esperar al proceso anterior para enviar su vector al siguiente como sucede en el algoritmo *blocking*.
  En este caso, todas las solicitudes de comunicación se realizan en paralelo, solo se debe esperar al final con la sentencia *wait* para garantizar que los envíos y las recepciones fueron recibidas por el proceso receptor.

Ejercicio 2

Dada la siguiente expresión:

$$D = d.ABC$$

$$d = \frac{maxA \cdot maxB \cdot maxC - minA \cdot minB \cdot minC}{avgA \cdot avgB \cdot avgC}$$

donde:

- *A, B, C* y *D* son matrices *NxN***.**
- *minA* y *maxA* son el mínimo y el máximo valor de los elementos de la matriz *A*, respectivamente; ídem para las matrices *B* y *C***.**
- *avgA* es el valor promedio de los elementos de la matriz *A*; ídem para *B* y *C***.**

Algoritmo paralelo distribuido

Para realizar el programa usamos la misma forma de resolución que en la entrega anterior, pero esta vez utilizando procesos en vez de threads. Debido a esto, la forma en la que se reparten las diferentes matrices cambia ya que se requiere de pasaje de mensajes para realizar esta tarea.

Inicialmente el proceso coordinador repartirá los datos de la matriz *A* en tamaños iguales con un *scatter* sobre la matriz *A* a cada proceso, incluido él.

Luego, la matriz *B* y *C* son comunicadas con un *broadcast*, ya que se necesitarán todos sus elementos para realizar el producto.

Una vez distribuidos los datos comienzan las multiplicaciones en cada proceso en *forma local* de los *strips* correspondientes.
Para ello se aprovechan las iteraciones y localidad de memoria que realizamos en la segunda entrega:

- Primero se obtiene el producto *AB* mientras se aprovecha el recorrido para calcular el mínimo, máximo y suma total de las matrices *A* y *B*.

- Luego, se obtiene el producto *ABC* mientras se aprovecha el recorrido para calcular el mínimo, máximo y suma total de la matriz *C*.

➥ Nota: Para calcular el máximo, mínimo y sumarización de las matrices comunicadas por *Broadcast* (*B* y *C*) tuvimos que calcular el strip que le corresponde a cada proceso en base a su id, de forma similar a como lo hicimos en el algoritmo *Pthreads* de la entrega 2.

Una vez finalizado el procesamiento local se comunican los resultados locales de los máximos, mínimos y sumarizaciones entre todos los procesos mediante la sentencia *reduceAll*. La cual le permitirá a cada proceso obtener los máximos, mínimos y sumarizaciones globales de las matrices *A*, *B* y *C* para obtener *d.*

Luego, cada proceso calcula el escalar *d* y realizan la operación *d.D* en forma local de su strip correspondiente.

Al terminar el cálculo cada proceso realiza un *gather* para comunicar *D* al proceso coordinador.

Se puede ver que las comunicaciones se dan en 3 secciones del código:
- Al repartir las matrices *A*, *B* y *C*.
- Al reducir los máximos, mínimos y sumarizaciones.
- Al recolectar *D*.

Pseudocódigo

1º sección de comunicación:

Scatter de A *desde el proceso coordinador hacia el resto de los procesos*
Broadcast de B y C *desde el proceso coordinador hacia el resto*

Procesamiento local:

Calcular AB = A.B junto con mínimo, máximo y suma total de A y B:

```
for i=1..N
      for j=1..N
            // Calcular Mínimo, Máximo y Suma total B
            // Calcular Mínimo, Máximo y Suma total de A.
            for k=1..N
                  // Calcular multiplicación AB = A.B
```

Calcular ABC = AB.C junto con mínimo, máximo y suma total de C

```
for i=1..N
      for j=1..N
            // Calcular Mínimo, Máximo y Suma total C
            for k=1..N
                  // Calcular multiplicación ABC = AB.C
```

2º sección de comunicación:

ReduceAll de mínimos, máximos y sumarizaciones de A, B, C

Procesamiento local:

Calcular d
Calcular D=d.ABC del strip correspondiente

3º sección de comunicación:

Gather de D hacia el proceso coordinador

Tiempos

a - Usando 1 único nodo con 8 procesos | P = 8					
N	**Algoritmo Secuencial**	**Algoritmo Distribuido**			
	Tiempo total	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	2.189936	0.302519	0.030244	7.23900317	0.9048753963
1024	17.772299	2.900573	0.088093	6.127168322	0.7658960402
2048	142.09077	23.069267	0.794399	6.159310133	0.7699137667
4096	1144.89004	181.05659	1.940812	6.323382319	0.7904227899
b - Usando 2 nodos con 8 procesos c/u | P = 16					
N	**Algoritmo Secuencial**	**Algoritmo Distribuido**			
	Tiempo total	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	2.189936	0.203666	0.075268	10.75258511	0.6720365697
1024	17.772299	1.703325	0.308401	10.43388608	0.6521178797
2048	142.09077	12.354865	1.164037	11.50079503	0.7187996894
4096	1144.89004	94.695928	5.879159	12.09017182	0.7556357386
c - Usando 4 nodos con 8 procesos c/u | P = 32					
N	**Algoritmo Secuencial**	**Algoritmo Distribuido**			
	Tiempo total	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	2.189936	0.154064	0.103562	14.21445633	0.4442017603
1024	17.772299	0.984378	0.322286	18.05434396	0.5641982488
2048	142.09077	6.888457	1.349847	20.62737272	0.6446053975
4096	1144.89004	50.093907	5.747594	22.85487614	0.7142148795

Algoritmo paralelo híbrido

Para realizar el algoritmo híbrido optamos por un esquema que se conforma por un proceso *MPI* por nodo y por cada proceso *MPI* se generan tantos threads como cantidad de núcleos posea el nodo. Decidimos multhilar las siguientes secciones de código:

En el caso del proceso coordinador, el *Broadcast* de las matrices *B* y *C* sólo lo realizará un thread, para ello empleamos la cláusula *single nowait* que le permite a los demás threads del proceso coordinador comenzar con el cómputo del producto *ABC* mientras el otro está comunicando.

En el caso de los procesos workers, sólo un thread recibirá el *Broadcast* de las matrices *B* y *C* mediante la cláusula *single* aprovechando la barrera implícita del constructor.

Luego, todos los threads de los procesos *MPI* realizan el cálculo de *AB* dónde se utiliza el constructor *for* con schedule *static sin chunk* especificado para que se repartan las misma cantidad de iteraciones a los mismos threads para calcular la multiplicación *AB = A.B*. Además, se aprovecha este *for*, para realizar las reducciones de suma, máximo y mínimo de las matrices *A* y *B*, utilizando la cláusula *reduction*. También se especifica la cláusula *nowait* para que los threads puedan continuar luego de haber terminado de resolver esta parte.

Luego, se utiliza el constructor *for* con schedule s*tatic sin chunk* especificado para que se repartan las misma cantidad de iteraciones a los mismos threads para calcular la multiplicación *AB.C*. También se aprovecha este *for* para realizar la reducción de suma, máximo y mínimo de la matriz *C*, utilizando la cláusula *reduction*. En este caso no se especifica la cláusula *nowait*, ya que necesitamos la barrera implícita que proporciona el constructor *for* para luego comunicar los datos obtenidos entre todos los procesos.

Una vez terminado el cálculo de *ABC*, se comunican los máximos, mínimos y sumarizaciones locales entre todos los procesos mediante la sentencia *reduceAll*. Para ello, se emplea la cláusula *single* para que sólo lo haga un thread y los demás queden esperando en la barrera implícita del constructor. Después de recibir la reducción, el thread que realizó la comunicación también realiza el cálculo del escalar *d*.

Finalmente se utiliza el constructor *for* con schedule *static sin chunk* especificado para que se repartan las misma cantidad de iteraciones a los mismos threads para calcular la multiplicación *D = d.ABC*. No se especifica la opción *nowait* ya que se necesita la barrera implícita del constructor para esperar a que todos los threads terminen de hacer el cálculo.

Cuando todos los threads terminen la operación, se comunica *D* al proceso coordinador mediante la sentencia *gather.*

Pseudocódigo

1º sección de comunicación:
Scatter de A *desde el proceso coordinador hacia el resto de los procesos*

// omp parallel **{**

2º sección de comunicación:
if (pid = pid_coordinador) {
// omp single nowait {
Broadcast de B y C *desde el proceso coordinador hacia el resto*
}
} else {
// omp single {
Recibir broadcast de B y C del proceso coordinador
}
}

Procesamiento local:
// omp for static nowait con reducción de suma, max, min A y B
Calcular AB=A.B y max/min/suma de A y B

// omp for static con reducción de suma, max, min C
Calcular ABC=AB.C y max/min/suma de C

// omp single {
3º sección de comunicación:
ReduceAll de los mínimos, máximos y sumarizaciones de A, B, C.

Procesamiento local:
Calcular d
}

Procesamiento local:
// omp parallel for static
Calcular D=d.ABC del strip correspondiente

}

4º sección de comunicación:
Gather de D hacia el proceso coordinador

Tiempos

b - 2 nodos y 1 proceso MPI con 8 threads por nodo - P = 16					
N	**Algoritmo Secuencial**	**Algoritmo Híbrido**			
	Tiempo total	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	2.189936	0.204572	0.067153	10.70496451	0.669060282
1024	17.772299	1.390552	0.26205	12.7807511	0.7987969436
2048	142.09077	10.067745	1.020671	14.11346533	0.8820915831
4096	1144.89004	77.27085	4.022699	14.816584	0.9260365002

c - 4 nodos y 1 proceso MPI con 8 threads por nodo - P = 32					
N	**Algoritmo Secuencial**	**Algoritmo Híbrido**			
	Tiempo total	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	2.189936	0.14746	0.08893	14.85105113	0.4640953479
1024	17.772299	0.866728	0.335122	20.50504772	0.6407827412
2048	142.09077	5.793633	1.402869	24.52533151	0.7664166098
4096	1144.89004	41.545202	5.683134	27.55769583	0.8611779948

Análisis de rendimiento individual

Algoritmo distribuido

Se puede ver que a medida que aumenta *N* el tiempo de comunicación también aumenta, ya que el tamaño de los datos a enviar por cada proceso va a crecer provocando que la comunicación tarde mucho más por enviar datos de gran tamaño.

También se puede ver que a medida que aumenta la cantidad de nodos también aumenta el porcentaje de comunicación con respecto al tiempo total del algoritmo para un mismo *N*:

Utilizando 1 nodo	
N	**Porcentaje de comunicación respecto al tiempo total**
512	9.99
1024	3.03
2048	3.44
4096	1.07
Utilizando 2 nodos	
N	**Porcentaje de comunicación respecto al tiempo total**
512	36.95
1024	18.1
2048	9.42
4096	6.2
Utilizando 4 nodos	
N	**Porcentaje de comunicación respecto al tiempo total**
512	67.22
1024	32.74
2048	19.59
4096	11.47

Se puede ver también que el porcentaje de comunicación corridos con la misma cantidad de procesos va disminuyendo a medida que aumenta *N*. Esto se debe a que a medida que el *N* es más grande cada uno de los procesos tienen una mayor cantidad de computo con respecto a la comunicación.

También podemos notar que el algoritmo con P = 8, el cual corre en un único nodo el *speedup* y la *eficiencia* se mantienen constantes. Esto se debe a que la comunicación *intra-nodo* se hace mucho más rápida que la comunicación *inter-nodo*. Esto causa que al aumentar el *N* lo que se vea más afectado es el tiempo de procesamiento y no el tiempo de comunicación, lo cual deriva en un *speedup* que estará cercano al óptimo real, ya que el tiempo de comunicación es despreciable en comparación del tiempo de procesamiento. Como el *speedup* se mantiene constante esto causa que la *eficiencia* también se mantenga.

En cambio al utilizar más de un nodo (P={16, 32}), la *eficiencia* y el *speedup* no se pueden mantener porque el tiempo de comunicación es una gran parte del tiempo total. Se puede ver que entre más nodos también más tiempo de comunicación se va a necesitar, esto causa que para poder llegar al *speedup* óptimo se tenga que aumentar mucho el *N* para que el cómputo supere a la comunicación notablemente. Por lo tanto a medida que se aumente *N*, el *speedup*, y por consiguiente la *eficiencia* aumentarán.

Algoritmo híbrido

En este algoritmo podemos notar que suceden los mismos patrones que en el algoritmo distribuido con respecto a la comunicación, *eficiencia* y *speedup* cuando comparamos P = 16 con P = 32. Con la diferencia de que el híbrido aprovecha los 2 tipos de comunicaciones vistos: por una parte la comunicación por memoria compartida que se utiliza entre los threads pertenecientes a un mismo nodo, mientras que por otra parte la comunicación por pasaje de mensajes se utiliza para comunicarse nodo a nodo permitiendo que procesos de diferentes nodos se sincronicen e intercambien datos. Esto hace que los tiempos sean mucho menores con respecto al distribuido ya que se están aprovechando diferentes niveles de paralelismo propios al problema planteado.

Análisis de rendimiento comparativo

MPI vs. Pthreads

Tiempos

P = 8							
N	**Algoritmo Distribuido MPI** **Usando 1 único nodo con 8 proceso MPI**				**Pthreads** **8 threads**		
	Tiempo total	**Tiempo de comunicación**	**Speedup**	**Eficiencia**	**Tiempo total**	**Speedup**	**Eficiencia**
512	0.302519	0.030244	7.23900317	0.9048753963	0.275790	7.9405924797	0.99257405997
1024	2.900573	0.088093	6.127168322	0.7658960402	2.243278	7.9224683699	0.99030854624
2048	23.069267	0.794399	6.159310133	0.7699137667	18.238183	7.7908402388	0.97385502985
4096	181.05659	1.940812	6.323382319	0.7904227899	144.919299	7.9001902983	0.98752378729

Conclusión

Se puede ver en la tabla que el algoritmo de *Pthreads* es mucho más eficiente en todos los tamaños de las matrices probados. Esto se debe al método de comunicación, *Pthreads* utiliza memoria compartida para comunicar sus datos y *MPI* utiliza pasaje de mensajes. Como vimos en la teoría, la comunicación por memoria compartida genera mucho menos ocio que la comunicación por pasaje de mensajes. Por lo tanto, utilizar pasaje de mensajes aumentará el ocio en los procesos disminuyendo el uso de cómputo, lo que provoca que el speedup óptimo real que se ve en algoritmo *Pthreads* disminuya en el *algoritmo distribuido* y consecuentemente la *eficiencia*.

- Nota: Tener en cuenta que la ventaja del algoritmo distribuido es que se puede extender a más nodos, en cambio el algoritmo *Pthreads* solo se ve acotado a un solo nodo ya que no tiene forma de comunicarse con procesos de nodos diferentes.

Algoritmo distribuido vs. Algoritmo híbrido

Tiempos

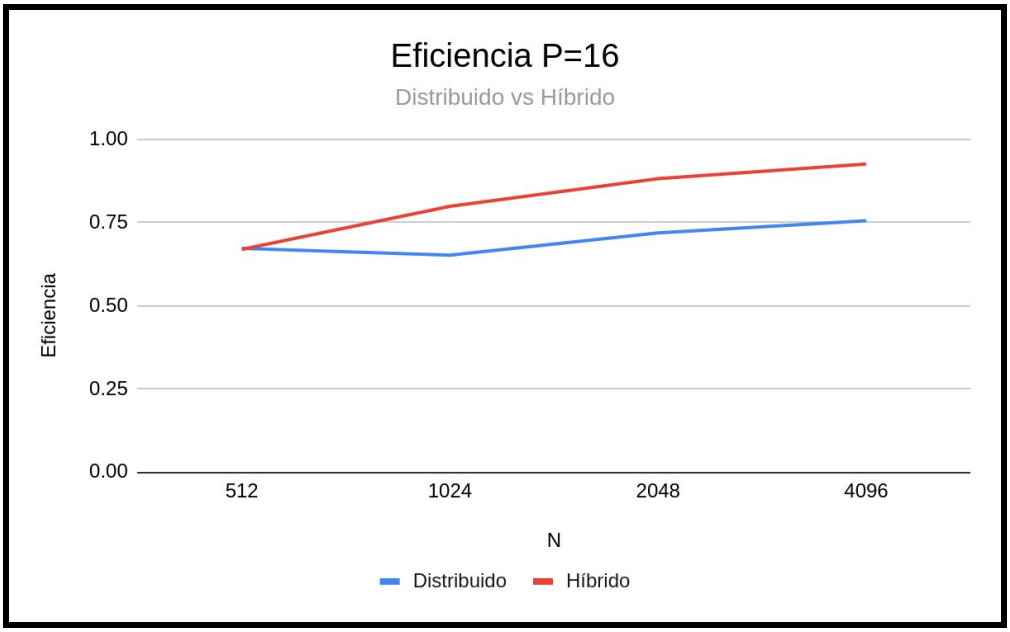
P = 16								
N	**Algoritmo Distribuido MPI** **Usando 2 nodos con 8 procesos MPI c/u**				**Algoritmo Híbrido** **2 nodos y 1 proceso MPI con 8 threads por nodo**			
	Tiempo total	**Tiempo de comunicación**	**Speedup**	**Eficiencia**	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	0.203666	0.075268	10.75258511	0.6720365697	0.204572	0.067153	10.70496451	0.669060282
1024	1.703325	0.308401	10.43388608	0.6521178797	1.390552	0.26205	12.7807511	0.7987969436
2048	12.354865	1.164037	11.50079503	0.7187996894	10.067745	1.020671	14.11346533	0.8820915831
4096	94.695928	5.879159	12.09017182	0.7556357386	77.27085	4.022699	14.816584	0.9260365002
P = 32								
N	**Algoritmo Distribuido MPI** **Usando 4 nodos con 8 procesos MPI c/u**				**Algoritmo Híbrido** **4 nodos y 1 proceso MPI con 8 threads por nodo**			
	Tiempo total	**Tiempo de comunicación**	**Speedup**	**Eficiencia**	**Tiempo total**	**Tiempo de comunicación**	**Speedup**	**Eficiencia**
512	0.154064	0.103562	14.21445633	0.4442017603	0.14746	0.08893	14.85105113	0.4640953479
1024	0.984378	0.322286	18.05434396	0.5641982488	0.866728	0.335122	20.50504772	0.6407827412
2048	6.888457	1.349847	20.62737272	0.6446053975	5.793633	1.402869	24.52533151	0.7664166098
4096	50.093907	5.747594	22.85487614	0.7142148795	41.545202	5.683134	27.55769583	0.8611779948

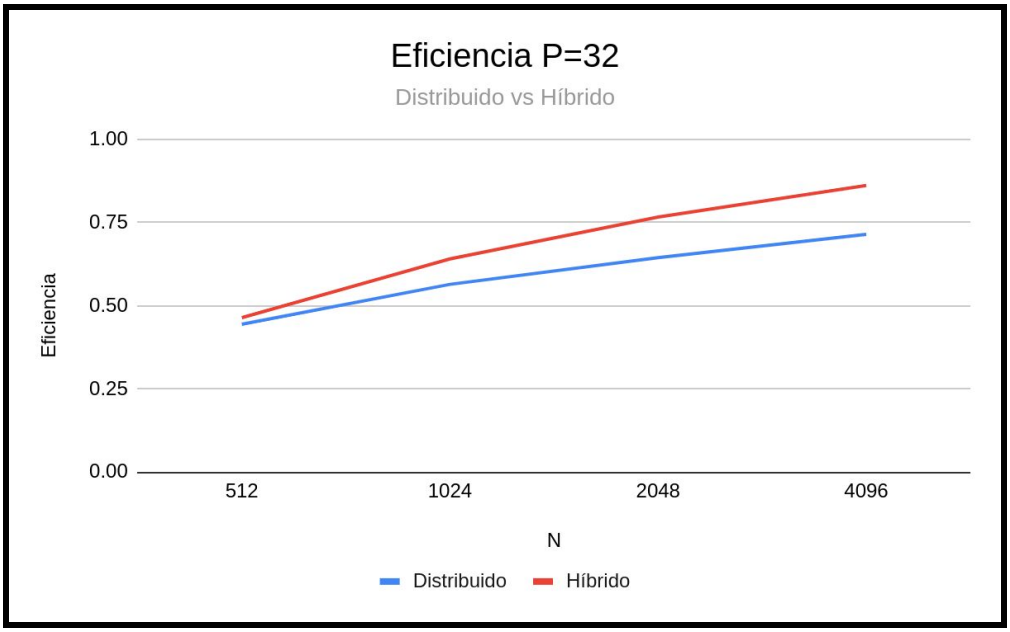
Conclusión

El algoritmo híbrido, permite explotar los diferentes niveles de paralelismo presentes en el problema lo cual lo hace más eficiente que el algoritmo distribuido. Esto se puede notar en que la cantidad de mensajes es menor ya que el híbrido sólo se comunicará con un proceso cada 8 threads, mientras que en el algoritmo distribuido se comunicarán todos los procesos por igual. Por lo tanto, el híbrido disminuye el ocio por tantas comunicaciones y aumenta el cómputo, por consiguiente la *eficiencia*.

También se puede ver que en el algoritmo híbrido el tiempo de comunicación es menor, lo que permite un mayor tiempo útil de cómputo, esto causa que el *speedup* crezca con mayor velocidad a medida que se incrementa *N* con respecto al algoritmo distribuido, y por lo tanto también la *eficiencia*.

Esto se puede visualizar en los siguientes gráficos:





En conclusión podemos ver que el algoritmo híbrido es más eficiente en todos los casos que el algoritmo distribuido.