1. Qué es MPI y OpenMP? (10%)
   - **MPI**: Es una especificación para los desarrolladores y usuarios de librerías de paso de mensajes.
   - **OpenMP**: Un API que puede ser usado directamente a multi-hilos. Para paralelismo de memoria compartida.
2. ¿Cuál es el propósito de las siguientes funciones? (20%)
   - **MPI_Init:** Inicia el ambiente de ejecución MPI.
   - **MPI_Comm_rank:** Determina el identificador del proceso que llama en el comunicador.
   - **MPI_Comm_size:** Determina el tamaño del grupo asociado al comunicador.
   - **MPI_Reduce:** Utiliza una función dada para reducir la información contenida en una variable alojada en varios procesos, a sólo un proceso. Por ejemplo, si se utiliza la función MPI_SUM, utiliza todos los enteros alojados en N procesos y almacena la suma de todos los valores en el proceso 0.
   - **MPI_Finalize:** Termina el ambiente de ejecución MPI.
   - **MPI_Barrier:** Obliga a todos los nodos a empezar al mismo tiempo desde el punto donde se localiza esta instrucción. En otras palabras, bloquea a los proceso que van terminando hasta que todos terminan. Cuando esto sucede, todos comienzan con las instrucciones restantes.
   - **MPI_Wtime:** Devuelve el tiempo transcurrido desde que se manda a llamar por primera vez..

3. ¿Cuándo se utilizan las directivas? (10%)
   - **parallel for:** Asigna cada una de las iteraciones de un loop for a un hilo distinto. Se ejecutan de forma paralela.
   - **parallel section:** Se inicia un bloque seccions y dentro de ese bloque se definen secciones diferentes. Cada sección se ejecuta en un hilo distinto.
   - **Critical:** Se especifican los bloques de código que deben ser ejecutados por un hilo a la vaez.

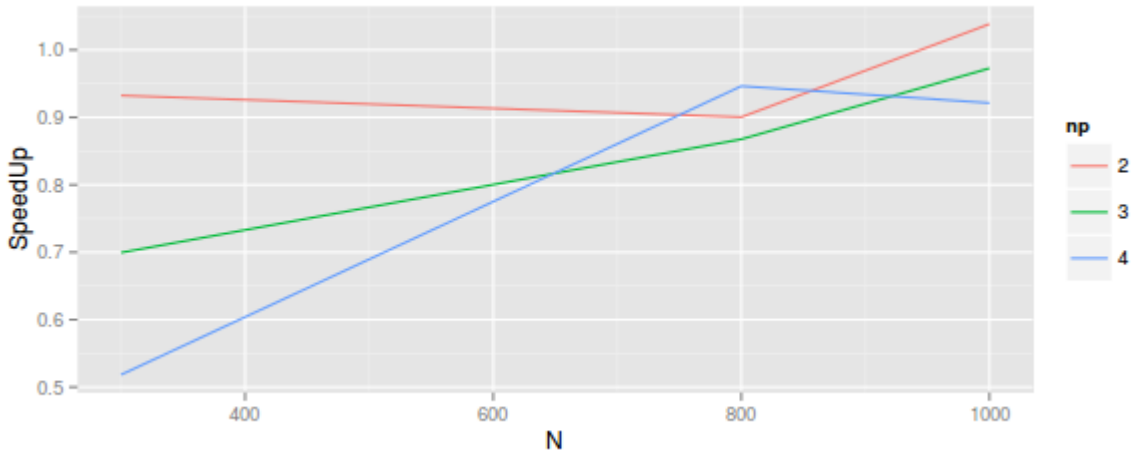
4. Qué se obtiene de las siguientes funciones? (20%)
   - **Omp_get_num_procs:** Regresa el número de procesadores disponibles.
   - **Omp_get_num_threads:** Regresa la cantidadd de hilos disponibles.
   - **Omp_get_thread_num:** Regresa el ID del hilo que corre la instrucción.
   - **Omp_set_num_threads:** Fija la cantidad de hilos que openMP utiliza en secciones paralelas.

5. Realiza en MPI la multiplicación de una matriz por un vector. (20%)

Los resultados son los siguientes:

N	Time	np	SpeedUp
1000	0.00365	1	
2000	0.01400	1	
3000	0.03110	1	
4000	0.05520	1	
1000	0.00183	2	1.995
2000	0.00717	2	1.953
3000	0.01620	2	1.920
4000	0.02870	2	1.923
1000	0.00179	4	2.039
2000	0.00695	4	2.014
3000	0.01560	4	1.994
4000	0.02760	4	2.000

donde N es tamaño de la matrix, Time es el tiempo transcurrido y np es el número de procesos utilizados. Lamentablemente la cantidad de filas y columnas requeridas en el examen no fue posible, ya que no cabían en RAM. La gráfica resultante es la siguiente:

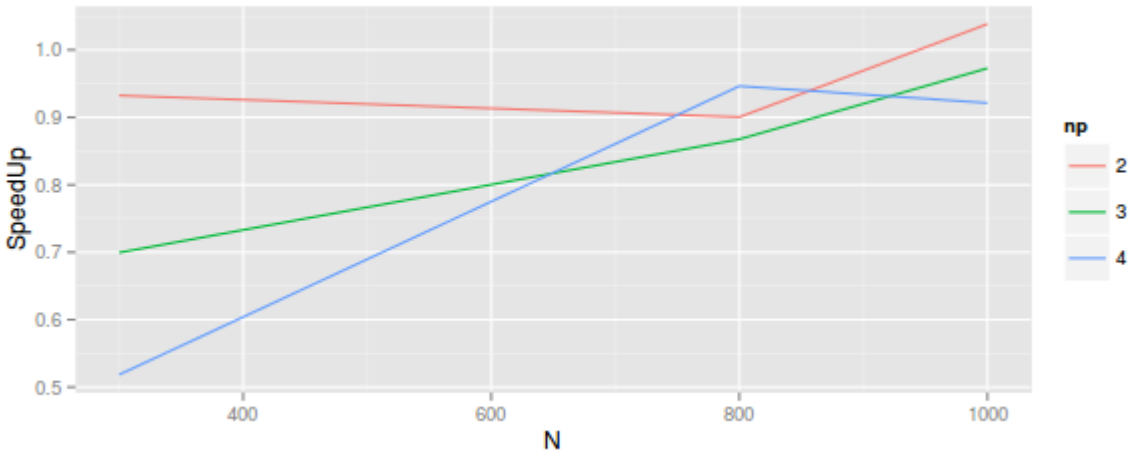


6. Realiza un programa híbrido (OpenMP y MPI) que implemente el algoritmo de Floy'd. Establecer el número de hilos igual al número de procesos disponibles en el programa. Contrasta el speedup obtenido en el programa con el algoritmo secuencial. (20%)

La tabla de resultados es la siguiente:

N	Time	np	SpeedUp
300	0.1626	1	
800	4.0214	1	
1000	5.8318	1	
300	0.1743	2	0.9324
800	4.4660	2	0.9004
1000	5.6161	2	1.0384
300	0.2324	3	0.6995
800	4.6349	3	0.8676
1000	5.9949	3	0.9728
300	0.3136	4	0.5184
800	4.2494	4	0.9463
1000	6.3297	4	0.9213

El SpeedUp resultante no fue el mejor, ya que sólo para N = 1000 y 2 hilos se rebaso al programa secuencial. Esto puede ser debido a que se generan muchos hilos y el tiempo en la creación de tantos hilos afecta el SpeedUp. Para obtener mejores resultados sería conveniente probar el algoritmo en una computadora con más de un procesador. El resultado se presenta en la gráfica siguiente:


SpeedUp
1.0
0.9
0.8
0.7
0.6
0.5
400
600
800
1000
N
np
2
3
4