Trabajo Práctico N°3

Integrantes:

- **Austin Myles Barker (19299/4)**
- **Nicolas Bonoris (19413/6)**

- **Parte 1**
    - Los códigos blocking.c y non-blocking.c siguen el patrón master-worker, donde los procesos worker le envían un mensaje de texto al master empleando operaciones de comunicación bloqueantes y no bloqueantes, respectivamente.
        - Compile y ejecute ambos códigos usando P={4,8,16} (no importa que el número de núcleos sea menor que la cantidad de procesos). ¿Cuál de los dos retorna antes el control?
            - En todos los casos el non-blocking retornara el control antes que el algoritmo blocking. Aunque sea por unos pocos milisegundos.
        - En el caso de la versión no bloqueante, ¿qué sucede si se elimina la operación MPI_Wait() (línea 52)? ¿Se imprimen correctamente los mensajes enviados? ¿Por qué?
            - Al eliminar el MPI_Wait() el proceso Master no se bloqueará hasta que la operación en el request haya finalizado. Por ende al no demorarse para recibir el mensaje del proceso worker este imprimirá la cadena previamente almacenada en el buffer "No debería estar leyendo esta frase."

    - Los códigos blocking-ring.c y non-blocking-ring.c comunican a los procesos en forma de anillo empleando operaciones bloqueantes y no bloqueantes, respectivamente. Compile y ejecute ambos códigos empleando P={4,8,16} (no importa que el número de núcleos sea menor que la cantidad de procesos) y N={10000000, 20000000, 40000000, ...}. ¿Cuál de los dos algoritmos requiere menos tiempo de comunicación? ¿Por qué?
        - El algoritmo que requiere menos tiempo es el algoritmo **non-blocking-ring.c**. Esto se debe a que el algoritmo hace uso de las sentencias **MPI_Isend()** y **MPI_Irecv()**, ambas sentencias no bloqueantes que como está expresado en el mismo algoritmo, reduce la probabilidad de que ocurra una situación de deadlock, no obliga a que los procesos tengan que turnarse o esperar a que previamente su anterior les haya mandado el mensaje que deben recibir. Todos los procesos realizan el envío no bloqueante y a su vez la recepción del mensaje, al no ser bloqueante casi todos los procesos intercambian la información al mismo tiempo. Al final todos los procesos esperaran en los dos **MPI_Wait()** expresados al final a la espera del evento de envío y recepción de los mensajes.

- **Parte 2**
    - **Secuencial**
        - **TIEMPO TOTAL**

Dim - bs	**512 - 16**	**1024 - 16**	**2048 - 16**	**4096 - 16**
Time	2.807195	22.444279	178.965912	1429.277103

    - La solución para el algoritmo planteado empleando MPI:

        - Declaramos las funciones que vamos a utilizar para realizar la multiplicación de las matrices, en main declaramos las variables que vamos a usar, las matrices y verificamos los parámetros que en este caso será la dimensión de las matrices y el tamaño de bloque.
        - Además de obviamente declarar las variables necesarias para que MPI funcione adecuadamente.
        - Hacemos un **MPI_Init()** para inicializar el entorno, indicamos la cantidad de procesos en el comunicador e indicamos el rank de cada proceso.
        - Verificamos si los parámetros son correctos, además de verificar si la dimensión de la matriz es múltiplo del número de procesos, en caso contrario finalizamos con **MPI_Finalize()** e informamos del error.
        - Calculamos la porción de matriz que recibirá cada proceso worker que deberá alocar en su memoria local mientras el coordinador alocara toda la matriz(Solo la matriz B y D es alocada en su totalidad en ambos casos).
        - Una vez inicializada las matrices se realiza un **MPI_Scatter()** con las matrices A y C distribuyendo la matriz de forma equitativa (en porciones) entre todos los procesos. Mientras que las matrices B y D se distribuyen enteras con un **MPI_Bcast()**.
        - Calculamos máximos, mínimos y promedio para luego realizar una reducción de los valores locales de cada proceso a las respectivas variables, esta reducción debe ser recibida por todos los procesos ya que usarán este valor para el cálculo del escalar. La reducción se realiza mediante **MPI_Allreduce()**. Siendo la mejor opción en vez de utilizar comunicación punto a punto que generaría un mayor overhead de comunicación.
        - Una vez realizada la multiplicación de matrices haremos un **MPI_Gather()** de las matrices que poseen los respectivos productos, el Gather se encargará recolectar la porción de matriz de cada proceso en orden almacenandose completa en el proceso coordinador, resultando mucho más eficiente que la comunicación punto a punto ya que no se tendrá que depender de comunicación bloqueante.
        - Mismo procedimiento para el cálculo de R.
        - Realizamos la reducción de las variables commTimes en min y max CommTimes para obtener el tiempo final de ejecución y el tiempo de comunicación.
        - Hacemos un MPI_Finalize() para cerrar el entorno MPI. Y Luego el proceso coordinador se encargará de liberar memoria e imprimir los tiempos.

    - **1-MPI**
        - **TIEMPO TOTAL**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	0.296932	2.224413	17.450281	138.997774
2	0.213937	1.387858	9.714052	73.360319
4	0.182605	0.936669	5.790792	40.30532

- **TIEMPO DE COMUNICACIÓN**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	0.031464	0.092435	0.389853	2.831899
2	0.106111	0.332478	1.201757	5.295061
4	0.136524	0.480956	1.579709	6.314471

- **SPEEDUP**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	9.454	10.089978	10.255761	10.282734
2	13.121597	16.171884	18.423405	19.482973
4	15.373046	23.961804	30.905256	35.461252

- **EFICIENCIA**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	1.18175	1.261247	1.28197	1.285342
2	0.8201	1.010743	1.151463	1.217686
4	0.480408	0.748806	0.965789	1.108164

- **ESCALABILIDAD**
  En lo que Escalabilidad hablamos el algoritmo resuelto con el estándar MPI, al incrementar el número de procesos resulta necesario también aumentar el tamaño de la dimensión de las matrices sobre las que vamos a operar. Como podemos notar la con la dimensión 512 al aumentar la cantidad de procesos la eficiencia decae. Esto demuestra que posee **Escalabilidad débil**.

- **OVERHEAD DE COMUNICACIÓN**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	10.596365	4.155478	2.234079	2.03737
2	49.599181	23.956197	12.371326	7.217882
4	74.764656	51.347488	27.279671	15.666594

- La solución del algoritmo planteado empleando MPI + OMP

    - Es bastante similar a la solución planteada con el estándar MPI, la diferencia aquí, radica en que al estar usando 8 hilos en la porción paralela del algoritmo se tuvo un especial cuidado con las sentencias de comunicación, principalmente para evitar problemas de rendimiento o una posible falla sobre hardware utilizado.
    - Las porciones en las que se usan comunicaciones colectivas se utilizaron "**#pragma omp master**" para que solo un proceso, en este caso el maestro, se encargue de realizar los **MPI_Reduce()**, **MPI_Gather()**, **MPI_Scatter()** y **MPI_Bcast()**.
    - Los for declarados con "**#pragma omp for**" todos poseen su scheduler en static, y la región paralela está definida como en el Trabajo Práctico #2.

- **2-Híbrido**
    - **TIEMPO TOTAL**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	0.354618	1.54816	9.93065	73.76916
2	0.343669	1.139706	5.928041	41.670566

    - **TIEMPO DE COMUNICACIÓN**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	0.260184	0.664257	1.575865	6.699855
2	0.289611	0.777482	1.978124	8.181925

    - **SPEEDUP**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	7.91611	14.49739	18.021571	19.374995
2	8.16831	19.693043	30.189722	34.299441

    - **EFICIENCIA**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	0.989514	1.812174	2.252696	2.421874
2	0.510519	1.230815	1.886858	2.143715

    - **ESCALABILIDAD**
      En este caso resulta similar a la solución planteada con MPI, la eficiencia se reduce al aumentar la cantidad de procesadores sobre

un mismo. Demostrando Escalabilidad débil.

- **OVERHEAD DE COMUNICACIÓN**

Nucleos	512 - 16	1024 - 16	2048 - 16	4096 - 16
1	73.370218	42.906224	15.868699	9.08219
2	84.27033	68.217768	33.368933	19.634782

- Caso P=(8):
    - **Cálculo de Eficiencia OpenMP(TP2):**

Threads - Dimensión	**512 - 64**	**1024 - 64**	**2048 - 64**	**4096 - 64**
2	0.838916	0.840962	0.837627	0.834972
4	0.837068	0.840667	0.836145	0.835017
8	0.834376	0.837455	0.834429	0.83253

    - Al caso de OpenMP solo le vamos hacer caso al caso de 8 threads.
    - **Cálculo de Eficiencia MPI(TP3):**

Núcleos/ Dimensión	**512 - 16**	**1024 - 16**	**2048 - 16**	**4096 - 16**
1 - 8	0.902229	0.961042	0.979745	0.992377

    - Al comparar el grado de eficiencia de ambos algoritmos podemos notar que el superior en este caso es el algoritmo que emplea el uso del estándar MPI. MPI al ser un estándar más centrado al pasaje de mensajes/comunicación entre sistemas con memoria distribuida, al estar haciendo todas las pruebas sobre el cluster resulta ser más eficiente. Aunque es cierto que OpenMP posee un menor overhead de comunicación, esto no logra que supere el rendimiento del algoritmo desarrollado con MPI.
    - Si el algoritmo fuera ejecutado en una sola máquina que hace uso de memoria compartida, lo más probable es que el algoritmo OpenMP sea el más eficiente.

- Caso P=(16,32);
        - (Hacer un análisis de la solución MPI e Híbrida comparándolos)
    - **EFICIENCIA**

Núcleos/ Dimensión	512 - 16	1024 - 16	2048 - 16	4096 - 16
2 - 16	0.603283	0.757720	0.874081	0.925025
4 - 32	0.364938	0.573577	0.740024	0.848138

- **EFICIENCIA**

Núcleos/ Dimensión	512 - 16	1024 - 16	2048 - 16	4096 - 16
2 - 16	0.333531	0.664301	0.837867	0.895456
4 - 32	0.207265	0.444312	0.716663	0.822941

- Por último la comparativa entre el algoritmo que implementa MPI y el modelo híbrido de MPI y OMP.
- Es notable la caída de rendimiento del algoritmo híbrido en comparación con el que solamente utiliza MPI, esto puede deberse a la necesaria implementación y utilización de hilos internos para cada proceso. En un principio con MPI solo se debe dividir los recursos que se tendrán que compartir entre procesos, pero al agregar OMP es necesario nuevamente dividir la carga de trabajo entre los 8 hilos que forman cada proceso, esto en una escala grande no supone problema, pero como vemos en las dimensiones 512 y 1024 vemos que el modelo híbrido resulta ser ineficiente en comparación a MPI en este caso. La ineficiencia del modelo híbrido es debido principalmente al overhead que genera la utilización de los hilos, temas relacionados a la compilación y el manejo de la memoria compartida.