Universidad del Valle De Guatemala

Facultad de Ingeniería

Computación paralela y distribuida

Óscar Canek

Lab 4: MPI

Javier Mombiela: 20067
Jose Hernández: 20053
Pablo González: 20362

Guatemala, 28 de septiembre 2023

1. (10 pts) Explique por qué y cómo usamos comunicación grupal en las siguientesfunciones de mpi_vector_add.c:

    A. Check_for_error(): Esta función se utiliza para verificar si algún proceso ha encontrado un error. Se utiliza la función MPI_Allreduce para obtener el valor mínimo del local_ok en todos los procesos del comunicador si alguno de los procesos encuentra un error es decir que el valor del local_ok es igual a 0, significa que ha habido un error en al menos uno de los proceso, en nuestro caso se imprime un mensaje de error en proceso 0 y se llama a MPI_Finalize para terminar todos los procesos debido al error, como resumen se puede mencionar que la comunicacion grupal en este método se utiliza para determinar si algún proceso ha encontrado un error mediante MPI_allredyce que se utiliza para obtener el valor mínimo de local_ok asegurando que todos los proceso conozcan si algún proceso ha encontrado un error o si todos están bien para su ejecución.

    B. Read_n(): Esta función se utiliza para obtener el valor de n desde el proceso 0 y luego difundirlo a todos los demás procesos en el comunicador. La comunicación grupal aquí se realiza mediante MPI_Bcast, que permite que el proceso 0 comparta el valor de n con todos los demás procesos de manera eficiente. Esto garantiza que todos los procesos tengan acceso al mismo valor de n para su uso posterior.

    C. Read_data(): Esta función se utiliza para leer un vector desde el proceso 0 y distribuirlo a todos los procesos en el comunicador. Se utiliza la función MPI_Scatter para distribuir porciones del vector desde el proceso 0 a cada proceso en el comunicador. La comunicación grupal aquí permite la distribución eficiente de los datos entre todos los procesos de manera sincronizada.

    D. Print_vector(): : Esta función se utiliza para recopilar y mostrar un vector desde todos los procesos en el comunicador en el proceso 0. Se utiliza la función MPI_Gather para recopilar las porciones de vector locales de todos los procesos en el proceso 0. La comunicación grupal aquí permite la recopilación eficiente de los datos en el proceso 0 para su posterior impresión.

2. (15 pts) Descargue y modifique el programa vector_add.c para crear dos vectores de al menos 100,000 elementos generados de forma aleatoria. Haga lo mismo con mpi_vector_add.c . Imprima únicamente los primeros y últimos 10 elementos de cada vector (y el resultado) para validar. Incluya captura de pantalla.

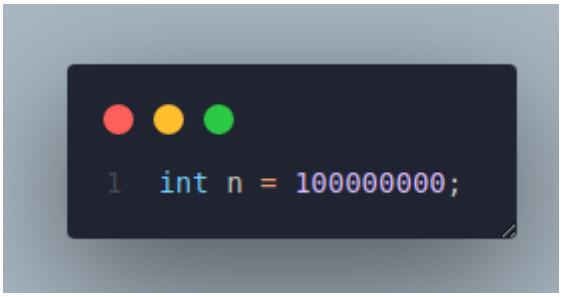
vector_add.c

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$ ./add
Primeros 10 de x
0.294578 0.176852 0.412460 0.660536 0.435056 0.128983 0.002290 0.308890 0.652654 0.971102

Primeros 10 de y
0.210926 0.666104 0.426216 0.492945 0.849067 0.637267 0.555108 0.541802 0.605627 0.181014

Ultimos 10 de de x
0.118039 0.350330 0.504577 0.392478 0.893669 0.033240 0.455500 0.558202 0.565366 0.844132

Ultimos 10 de de y
0.062981 0.672165 0.789312 0.070694 0.956670 0.961755 0.411165 0.179942 0.383245 0.588160

Primeros 10 de z
0.505504 0.842956 0.838676 1.153481 1.284123 0.766250 0.557398 0.850691 1.258281 1.152116

Ultimos 10 de de z
0.181020 1.022495 1.293889 0.463172 1.850339 0.994995 0.866665 0.738144 0.948611 1.432292
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$
```

mpi_vector_add.c

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$ mpiexec -n 4 ./mpi

Primeros 10 de local_x:
0.424358 0.298136 0.446364 0.910261 0.685287 0.559051 0.566802 0.935953 0.331818 0.676051

Primeros 10 de local_y:
0.801451 0.074169 0.365802 0.272249 0.163878 0.426549 0.390103 0.732243 0.503603 0.972931

Ultimos 10 de local_x:
0.603525 0.476504 0.060335 0.881525 0.109323 0.770366 0.398468 0.724051 0.081910 0.185794

Ultimos 10 de local_y:
0.539517 0.747996 0.897014 0.213823 0.793527 0.751775 0.717167 0.869782 0.820699 0.491367

Primeros 10 de local_z:
1.225809 0.372305 0.812166 1.182510 0.849165 0.985600 0.956906 1.668195 0.835421 1.648982

Ultimos 10 de local_z:
1.143043 1.224500 0.957348 1.095348 0.902850 1.522141 1.115634 1.593833 0.902609 0.677161
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$
```

3. (5 pts) Mida los tiempos de ambos programas y calcule el speedup logrado con la versión paralela. Realice al menos 10 mediciones de tiempo para cada programa y obtenga el promedio del tiempo de cada uno. Cada medición debe estar en el orden de los ~5 segundos para asegurar valores estables (utilice una cantidad de elementos adecuada para que a su máquina le tome por lo menos ~5 cada corrida). Utilice esos promedios para el cálculo del speedup. Incluya capturas de pantalla.



Para poder realizar este ejercicio se usaron 10^8 elementos, ya que es el mayor número de elementos que permite utilizar la VM. Se intentó utilizar 10^9 elementos, pero la VM no permite y mata la ejecución.

Ejemplos de corridas con el tiempo de ejecución.

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$ ./add
Primeros 10 de x
0.912588 0.057258 0.727794 0.149549 0.528436 0.298161 0.181014 0.440585 0.243214 0.853636
Primeros 10 de y
0.349133 0.444601 0.888594 0.755680 0.984462 0.591909 0.250197 0.762788 0.505474 0.092235
Ultimos 10 de de x
0.286768 0.318291 0.820398 0.963987 0.245163 0.991594 0.603840 0.166436 0.519276 0.366592
Ultimos 10 de de y
0.890479 0.866944 0.218448 0.674673 0.203423 0.347772 0.960035 0.539645 0.906324 0.921907
Primeros 10 de z
1.261721 0.501859 1.616388 0.905229 1.512898 0.890070 0.431210 1.203373 0.748688 0.945870
Ultimos 10 de de z
1.177247 1.185235 1.038847 1.638659 0.448586 1.339366 1.563875 0.706081 1.425599 1.288499
Tiempo de ejecución: 0.496371 segundos
```

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$ mpiexec -n 4 ./mpi
Primeros 10 de local_x:
0.597626 0.078802 0.787158 0.952671 0.295779 0.602323 0.184331 0.803442 0.579654 0.830658
Primeros 10 de local_y:
0.564340 0.668725 0.252334 0.605184 0.886562 0.339736 0.991242 0.023954 0.015057 0.324000
Ultimos 10 de local_x:
0.323204 0.093713 0.314885 0.953366 0.567013 0.619133 0.303980 0.837514 0.544832 0.925242
Ultimos 10 de local_y:
0.830614 0.981787 0.067992 0.944171 0.504945 0.262853 0.276642 0.019174 0.691911 0.530388
Primeros 10 de local_z:
1.161965 0.747527 1.039492 1.557856 1.182342 0.942059 1.175573 0.827396 0.594711 1.154659
Ultimos 10 de local_z:
1.153818 1.075500 0.382877 1.897538 1.071958 0.881986 0.580621 0.856688 1.236743 1.455630
Tiempo de ejecución: 0.227000 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$
```

Tabla con comparaciones de tiempo entre MPI y normal

Programa sin MPI	**Programa con MPI**
1.539127	1.231146
0.538733	0.178251
0.528878	0.155375
0.540695	0.172249
4.880140	0.131332
0.525225	0.266011
0.600376	0.172138
0.536926	0.186168
0.576883	0.145840
0.496371	0.157992

Tabla de promedios

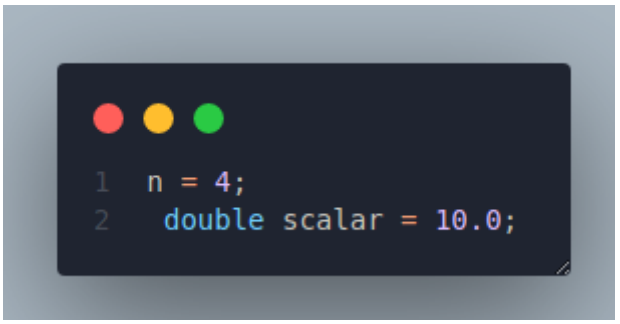
Promedio sin MPI	Promedio con MPI
1.072443	0.2796502

$$Speedup = \frac{tiempo\ secuencial}{tiempo\ paralelo}$$

$$Speedup = \frac{1.072443}{0.2796502} = 3.83494451$$

El speedup del programa con MPI en comparación con el programa sin MPI es aproximadamente 3.83. Esto significa que el programa con MPI es 3.8 veces más rápido en promedio que el programa sin MPI en las condiciones y tamaños de entrada dados.

4. (55 pts) Modifique el programa mpi_vector_add.c para que calcule de dos vectores 1) el producto punto, 2) el producto de un escalar por cada vector (el mismo escalar para ambos). Verifique el correcto funcionamiento de su programa (para ello puede probar con pocos elementos para validar). Incluya captura de pantalla.



Para poder verificar si el producto punto y el producto de la multiplicación escalar era correcta, probamos con n = 4, y con scalar = 10, ya que estos valores facilitaban la verificación de los resultados.

Vector X	1.064806	0.934740	0.300938	0.800718
Vector Y	0.954973	0.972221	1.095438	0.150783

$$Producto\ Punto = X1 * Y1 + X2 * Y2 + X3 * Y3 + X4 * Y4$$
$$= 1.01686098 + 0.9087738575 + 0.3296589208 + 0.1207346622$$
$$= 2.376028421$$

$$Escalar = 10$$
$$Multiplicaciones\ Escalares:\ X = (X1 * 10, X2 * 10, X3 * 10, X4 * 10)$$
$$Y = (Y1 * 10, Y2 * 10, Y3 * 10, Y4 * 10)$$

$$X = (10.648055,\ 9.347399,\ 3.009376,\ 8.007180)$$
$$Y = (9.549727,\ 9.722210,\ 10.954378,\ 1.507829)$$

Resultados reales del código

```
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$ mpicc -o mpi2 mpi_vector_punto.c
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$ mpiexec -n 4 ./mpi2
Elementos originales de local_x : 1.064806 0.934740 0.300938 0.800718
Elementos originales de local_y : 0.954973 0.972221 1.095438 0.150783

Producto punto: 2.376027

Resultado de la multiplicación escalar:
Elementos modificados de local_x: 10.648055 9.347399 3.009376 8.007180
Elementos modificados de local_y: 9.549727 9.722210 10.954378 1.507829

Tiempo de ejecución: 0.003301 segundos
javimombiela@javimombiela-VirtualBox:~/Documents/GitHub/LAB4$
```

5. (15 pts) Finalmente, escriba una reflexión del laboratorio realizado en donde hable de las técnicas aplicadas, lo que se aprendió y pudo repasar, elementos que le llamaron la atención, ediciones/mejoras que considera que son posibles y cualquier otra cosa relevante que tengan en mente. (No hay mínimo de palabras/párrafos, pero si desarrollan poco o de forma superficial seguramente tendrán nota baja en este inciso).

   En este laboratorio, hemos tenido la oportunidad de explorar las técnicas y conceptos generales de la programación en paralelo utilizando OpenMPI y programando en C. Durante este lab, aprendimos aún más cosas que no sabíamos de labs pasadas y hemos tenido la oportunidad de repasar conceptos esenciales. Lo que pudimos ver más claro durante la elaboración de este trabajo fue:

   Aprendizaje de programación en paralelo: Este laboratorio nos brindó una introducción bastante clara a la programación en paralelo para hacer cálculos de vectores. Lo más importante es que aprendimos a gestionar la comunicación entre ellos y aprovechar la potencia de cálculo de sistemas distribuidos en nuestras máquinas virtuales de Ubuntu.

   OpenMPI lo tomamos como una herramienta poderosa para la programación en paralelo aparte de OpenMP en general. Aprendimos a utilizar las funciones proporcionadas por la librería extendida de OpenMPI para la comunicación entre procesos y la asignación de tareas.

   Implementamos el cálculo del producto punto en paralelo, lo que nos permitió comparar el rendimiento de una versión secuencial con la versión paralela. Esto nos ayudó a comprender la importancia de la paralelización en tareas intensivas donde un solo core podría tener más problemas.

   Aprendimos a medir el tiempo de ejecución de nuestros programas. Esto es fundamental para evaluar el rendimiento y la eficiencia de los programas paralelos y se logró utilizando la función MPI_Wtime en el caso de OpenMPI y un simple clock en la versión secuencial.

Nos pudimos dar cuenta que la programación en paralelo puede ofrecer mejoras significativas en el rendimiento cuando se trata de tareas intensivas en cálculos, especialmente en sistemas con múltiples núcleos o nodos.

Comprendimos la importancia de diseñar cuidadosamente los algoritmos para la programación en paralelo. La distribución de tareas y nuestra gestión de nuestra comunicación son aspectos críticos del diseño que deben abordarse con atención.

En resumen, este laboratorio de programación en paralelo con OpenMPI y C ha sido una experiencia buena para seguir indagando en las capacidades del paralelismo. Hemos aprendido cosas esenciales para diseñar y desarrollar programas paralelos para cálculos vectoriales, así como una comprensión más profunda de cómo aprovechar la capacidad de nuestros núcleos de sistemas paralelos.