Zdravko Dimitrov Arnaudov

--MEMORIA P2--

Comunicaciones punto a punto no bloqueantes.

Índice

1. Ejercicio 1

Recurriremos al apartado 1.D), donde planteábamos tres versiones de comunicación bloqueante: segura, insegura y bloqueante. En este caso, trabajaremos sobre la bloqueante y la modificaremos con herramientas de comunicación asíncrona para hacer que funcione.

Lo que hemos hecho en este apartado es hacer la recepción no bloqueante. De esta forma, dejamos la operación inicializada y continuamos haciendo otras instrucciones. Mientras la recepción se completa, cada proceso manda su respectivo y mensaje y después del Send, hacemos wait para asegurarnos de que hasta ese momento, se ha completado el trabajo. Después, podemos imprimir lo que se ha recibido del otro proceso.

mpirun -np 2 Ejer1
Mando desde 0
Mando desde 1
He recibido Hola desde proceso 0! desde 0
El proceso sender 1 ha terminado de recibir y enviar.
He recibido Hola desde proceso 1! desde 1
Mando desde 0
El proceso receiver 0 ha terminado de recibir y enviar.

2. Ejercicio 2

En este ejercicio, reutilizaremos el programa usado anteriormente del 1.D).
Lo que vamos a implementar es el envío de un número de enteros parametrizado.
Habrá dos versiones del programa: uno síncrono y otro asíncrono.

Interpretamos que en este programa, hay dos procesos trabajadores. El emisor, hace unos ciertos cálculos y se los manda al receptor. Este recibe los datos y procede a realizar otros cálculos con ellos. Este procedimiento se repite una vez más, entonces el emisor debe asegurarse, antes de mandar los segundos cálculos, que el receptor a recibido los primeros. El receptor por su parte, debe asegurarse que recibe correctamente cada uno de los cálculos mandados por el emisor para poder trabajar con ellos.

En resumen, la mecánica es la siguiente:

Emisor: computo 1s, mando datos 1, computo 6s, mando datos 2
Receptor: Computo 6s, recibo datos 1, computo 1s, recibo datos 2

Al terminar el código, es necesario responder a una serie de preguntas y hacer unas pruebas.

Cuestión 1: plantear cuánto tardará el programa síncrono.

En el síncrono, inicialmente deducimos que tardará 12s en ambos casos, porque en el primer envío en emisor se bloquea y cuando el receptor recibe, se continúa con el segundo cálculo de 6s. Después, el receptor se posicionaría a la espera de los datos siguientes.

Cuestión 2: Probar a ejecutar la versión síncrona con un T pequeño y otro grande.

mpirun -np 2 Ejer2_sin
La emisión ha tardado :7.003328

Ejecutando el programa síncrono vemos que el tiempo de ejecución son 7s para T pequeño. Consideramos que este resultado tiene sentido, ya que estamos en comunicación bloqueante y se emplea el buffer del sistema. Por lo tanto, la terminación es local y el emisor puede seguir con su trabajo.

mpirun -np 2 Ejer2_sin 100
La emisión ha tardado :12.001645

En el caso de T grande, cuando aumentamos el tamaño de los datos que enviamos, se debe esperar a que el receptor reciba el mensaje y por tanto, hay un tiempo de espera y el tiempo de ejecución tarda lo que esperábamos al principio.

Cuestión 3: Repetir el apartado 1 con la versión no bloqueante.

Lo que esperamos que tarde el programa asíncrono es 7 segundos, con T grande y pequeña. Esto es porque los procesos inicializan sus operaciones y siguen trabajando. Por muy grandes que sean los datos a enviar, se sigue operando hasta el wait.

El emisor, realiza su primer envío y hace el wait después de los 6s para aprovechar trabajo en segundo plano. Antes de mandar los segundos datos al receptor, nos aseguramos de que los primeros hayan llegado y por eso hacemos wait. Después, se hace el ultimo envío.El receptor después de cada recepción, comprueba que los datos hayann llegado para poder trabajar con ellos. En definitiva, no se esperan bloqueos y el tiempo deberían ser 7.

mpirun -np 2 Ejer2_asin
La emisión ha tardado :7.000954

Cuestión 4: repetir apartado 2 con la versión no bloqueante

Efectivamente, el tiempo sigue siendo **7s**.

3. Ejercicio 3

En este último ejercicio, compararemos un programa secuencial con los distintos paradigmas que existen para llevar a cabo un programa paralelo. Lo que haremos será aproximar el valor de una integral definida, sobre una función determinada por el usuario. Contamos con una serie de programas donde insertaremos ciertas métricas y anotaremos los resultados obtenidos.

La función que usaremos será f(x) = cos(x)*e^(sin(x)), la ofrecida por el guion. Será usada tanto en el modo secuencial, threads POSIX y MPI. Se han probado otras alternativas pero esta es cómoda, ya que el resultado tiende siempre a 0 y con extremos amplios, el cómputo puede llevar varios segundos.

3.1 Secuencial

Para este programa, compilamos con -lm (libería math)
Probemos algunas ejecuciones con este fichero:

./Ejer3_secuencial
Mete a ,b, n
0
200
3
Con 3 trapezoides la estimacion es de 45.710857

./Ejer3_secuencial
Mete a ,b, n
0
200
20
Con 20 trapezoides la estimacion es de 7.726662

./Ejer3_secuencial
Mete a ,b, n
0
200
5000
Con 5000 trapezoides la estimacion es de -0.584632

Como podemos ver, cuantos más trapezoides usemos, mayor será la aproximación de la integal.

3.2 Threads POSIX

En cuanto al modelo con threads, compilamos con -lm(librería math) y -pthread. Disponemos de 2 hilos trabajadores:

./Ejer3_threads Mete a ,b, n
0
400
100
rodaja y h 50 - 4.000000
inicio y fin 0.000000 - 200.000000
rodaja y h 50 - 4.000000
inicio y fin 200.000000 - 400.000000
total -0.010439
total 0.601718
total -0.214798
total 2.390792
Con 100 trapezoides la estimación es de 2.175994

3.3 MPI

Por último, probaremos la versión paralela MPI.
Como podemos ver, el scanf está comentado y el código nos pide que formemos una barrera MPI. Al probar el código, se genera lo siguiente:

mpirun -np 4 Ejer3_MPI
Mete a ,b,n
0
20
4
Soy 1 inicio y fin son 2.500000 - 5.000000 integral ini es 1.279845 h 0.010000
Soy 2 inicio y fin son 5.000000 - 7.500000 integral ini es 1.250452 h 0.010000
Soy 3 inicio y fin son 7.500000 - 10.000000 integral ini es 0.218677 h 0.010000
c
Soy 0 inicio y fin son 0.000000 - 5.000000 integral ini es -1.345066 h 5.000000
Con 4 trapezoides la estimación es de 1.403909

Lo que sucede es que el proceso 0 trabaja sobre su parte correspondiente pero el resto usan los datos que están inicializados en el código. Para que los demás procesos también dispongan de las métricas que se indican por consola, haremos que el proceso 0 las envíe a cada uno de los procesos que hay en el sistema y posteriormente ellos envíen el resultado de vuelta.

$ mpirun -np 4 Ejer3_MPI
Mete a ,b,n
0
120
4
Soy 0 inicio y fin son 0.000000 - 30.000000 integral ini es 30000078.000000 h 30.000000
A: 0.000000, B: 120.000000, N:4
Soy 1 inicio y fin son 30.000000 - 60.000000 integral ini es 30000012.000000 h 30.000000
A: 0.000000, B: 120.000000, N:4
Soy 2 inicio y fin son 60.000000 - 90.000000 integral ini es 29999940.000000 h 30.000000
A: 0.000000, B: 120.000000, N:4
Soy 3 inicio y fin son 90.000000 - 120.000000 integral ini es 30000170.000000 h 30.000000
Con 4 trapezoides la estimación es de 120000200.000000

Los repartos ahora se hacen correctamente.

3.4 Rendimiento

A estas alturas, las dimensiones del cómputo no han sido elevadas y por tanto el programa no ha tardado nada en terminar. Si aumentásemos tanto los límites como el número de trapezoides, la ejecución es razonablemente larga y se consigue observar Speedup de las versiones paralelas a la secuencial.

Lo que haremos ahora, será buscar unas métricas que hagan al secuencial tardar muchos segundos y repetirlas para el resto de programas.

SECUENCIAL: A=0, B=1000000000000000 , N=1000000000 (calculado por tanteo)

time ./Ejer3_secuencial
Mete a ,b, n
0
1000000000000000
1000000000
Con 1000000000 trapezoides la estimacion es de 33554432000000.000000

real 1m36.082s
user 1m24.313s
sys 0m0.012s

Nuestro programa tarda ahora en terminar 1min y 24s.

THREADS : A=0, B=1000000000000000 , N=1000000000

time ./Ejer3_threads
Mete a ,b, n
0
1000000000000000
1000000000
rodaja y h 500000000 - 1000000.000000
inicio y fin 0.000000 - 499999993495552.000000
rodaja y h 500000000 - 1000000.000000
inicio y fin 499999993495552.000000 - 999999986991104.000000
total 0.309567
total -0.048030
total -8388608000000.000000
total 33554432000000.000000
Con 1000000000 trapezoides la estimacion es de 25165824000000.000000

real 1m1.537s
user 1m27.463s
sys 0m0.008s

Nuestro programa tarda en acabar alrededor de **44 segundos**, la mitad de que se indica. Al andar con 2 hilos trabajadores, el cómputo se hace el doble de rápido. Esta máquina virtual tiene asociados 4 núcleos físicos, si añadiésemos más threads que cores el rendimiento se estabilizaría.

MPI : A=0, B=1000000000000000 , N=1000000000

En cuanto al modelo MPI, emplearemos el cluster de PCs del laboratorio 4 con slurm. Probaremos distintos números de CPUs y midiendo los tiempos, para finalmente sacar una gráfica de Speedup.

Antes de nada, lo que hacemos es trasladar el fichero .c para poder utilizarlo en unicanlabs. Luego, seguiremos las instrucciones dadas en el guion de la práctica 0.
Hay que tener en cuenta que cuando ejecutamos el script, no se nos pide por teclado introducir los parámetros del programa. Por este motivo, declararemos constantes y se las asociaremos a los extremos y al número de trapezoides.

Haciendo sinfo, vemos el número de particiones que existen junto con el número de nodos que tiene cada una. Mediante el script slurm.sh, podemos ajustar el número de nodos que ejecutarán nuestro programa.

>>2 NODOS

[prun] Master compute host = n232-174
[prun] Resource manager = slurm
[prun] Launch cmd = mpirun ./a.out (family=openmpi4)
A: 0.000000, B: 999999986991104.000000, N:1000000000
Soy 0 inicio y fin son 0.000000 - 499999993495552.000000 integral ini es 33554432000000.000000 h 1000000.000000
Con 1000000000 trapezoides la estimacion es de 25165824000000.000000
*El programa ha durado **60.434592 segundos***
Soy 1 inicio y fin son 499999993495552.000000 - 999999986991104.000000 integral ini es -8388608000000.000000 h 1000000.000000

La duración ronda 1 minuto. En este caso, no se da el doble de speedup frente a la versión secuencial que hemos ejecutado en nuestra máquina. Al final, las arquitecturas son distintas y el paso de mensajes conlleva su overhead. Además, nos hemos encontrado en complicaciones a la hora de ejecutar el programa secuencial y con threads en el laboratorio, pero asumiremos que bajo condiciones normales, el programa secuencial ejecutado desde slurm con un solo nodo tardaría alrededor de 2 minutos. Previamente, se hicieron pruebas independientes con otros parámetros en nuestra

máquina y el tiempo de ejecución de threads con 2 hilos y el de MPI con 2 procesos daba el mismo tiempo. Por esta razón, si obtenemos 1 minuto con dos nodos, intuimos que en secuencial se tardaría el doble.

Lo que sí vamos a apreciar es un rendimiento constante y escalable frente a la ejecución en MPI con 2 nodos. Lo cual nos aclara que la versión secuencial si dura alrededor de 2min y cada vez que duplicamos los procesos, el tiempo se reduce a la mitad.

>>4 NODOS

[prun] Master compute host = n232-177
[prun] Resource manager = slurm
[prun] Launch cmd = mpirun ./a.out (family=openmpi4)
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
Soy 1 inicio y fin son 249999996747776.000000 - 499999993495552.000000 integral ini es -16777216000000.000000 h 1000000.000000
Soy 3 inicio y fin son 750000007020544.000000 - 999999986991104.000000 integral ini es 2097152000000.000000 h 1000000.000000
Soy 0 inicio y fin son 0.000000 - 249999996747776.000000 integral ini es 33554432000000.000000 h 1000000.000000
Con 1000000000 trapezoides la estimacion es de 10485760000000.000000
*El programa ha durado **30.216276 segundos***
Soy 2 inicio y fin son 499999993495552.000000 - 750000007020544.000000 integral ini es -8388608000000.000000 h 1000000.000000

Con cuatro nodos computando el programa, ya observamos que el speedup es el doble frente a la versión previa con 2 nodos. Esta tendencia intuimos que seguirá manteniéndose, hasta que lleguemos a un punto donde no compense seguir añadiendo nodos porque el rendimiento se estabilizará, aunque por el momento, todo sigue mejorando.

>>8 NODOS

[prun] Master compute host = n232-174
[prun] Resource manager = slurm
[prun] Launch cmd = mpirun ./a.out (family=openmpi4)
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000

Soy 7 inicio y fin son 874999997005824.000000 - 999999986991104.000000 integral ini es -16777216000000.000000 h 1000000.000000
Soy 3 inicio y fin son 375000003510272.000000 - 499999993495552.000000 integral ini es 8388608000000.000000 h 1000000.000000
Soy 5 inicio y fin son 625000017035264.000000 - 750000007020544.000000 integral ini es 33554432000000.000000 h 1000000.000000
Soy 2 inicio y fin son 249999996747776.000000 - 375000003510272.000000 integral ini es -16777216000000.000000 h 1000000.000000
Soy 1 inicio y fin son 124999998373888.000000 - 249999996747776.000000 integral ini es 33554432000000.000000 h 1000000.000000
Soy 6 inicio y fin son 750000007020544.000000 - 874999997005824.000000 integral ini es 2097152000000.000000 h 1000000.000000
Soy 0 inicio y fin son 0.000000 - 124999998373888.000000 integral ini es 33554432000000.000000 h 1000000.000000
Con 1000000000 trapezoides la estimacion es de 69206016000000.000000
*El programa ha durado **15.106743 segundos***
Soy 4 inicio y fin son 499999993495552.000000 - 625000017035264.000000 integral ini es -8388608000000.000000 h 1000000.000000

De nuevo, el tiempo es la mitad que el previo. Solo nos quedan dos nodos más por añadir, veamos el efecto de la siguiente iteración.

>>10 NODOS

[prun] Master compute host = n232-174
[prun] Resource manager = slurm
[prun] Launch cmd = mpirun ./a.out (family=openmpi4)
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
A: 0.000000, B: 999999986991104.000000, N:1000000000
Soy 7 inicio y fin son 700000011026432.000000 - 800000003014656.000000 integral ini es 8388608000000.000000 h 1000000.000000
Soy 3 inicio y fin son 300000009519104.000000 - 400000001507328.000000 integral ini es 33554432000000.000000 h 1000000.000000
Soy 1 inicio y fin son 100000000376832.000000 - 200000000753664.000000 integral ini es 16777216000000.000000 h 1000000.000000
Soy 4 inicio y fin son 400000001507328.000000 - 499999993495552.000000 integral ini es 4194304000000.000000 h 1000000.000000

Soy 6 inicio y fin son 600000019038208.000000 - 700000011026432.000000 integral ini es 33554432000000.000000 h 1000000.000000
Soy 9 inicio y fin son 899999995002880.000000 - 999999986991104.000000 integral ini es -33554432000000.000000 h 1000000.000000
Soy 5 inicio y fin son 499999993495552.000000 - 600000019038208.000000 integral ini es -8388608000000.000000 h 1000000.000000
Soy 8 inicio y fin son 800000003014656.000000 - 899999995002880.000000 integral ini es -4194304000000.000000 h 1000000.000000
Soy 0 inicio y fin son 0.000000 - 100000000376832.000000 integral ini es 33554432000000.000000 h 1000000.000000
Con 1000000000 trapezoides la estimacion es de 50331648000000.000000
*El programa ha durado **12.135790 segundos***
Soy 2 inicio y fin son 200000000753664.000000 - 300000009519104.000000 integral ini es -33554432000000.000000 h 1000000.000000

La mejora en este caso no es tan grande como antes, tendríamos que haber probado con 16 nodos para verificar que el speedup sigue siendo el doble.

4. Observaciones

Es importante recalcar que las métricas indicadas al programa han sido escogidas con mucho cuidado, para que el programa secuencial dure un tiempo razonable y pueda compararse cómodamente con las versiones paralelas. Además, al probar slurm, hay veces que se producen desbordamientos si la extensión de los límites o el número de trapezoides es excesiva, los tiempos pasan a durar 0s y los extremos se convierten negativos.

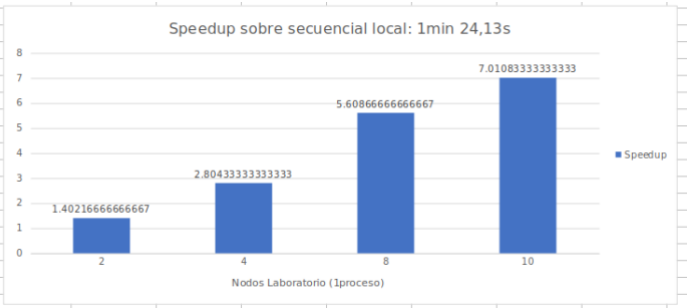
De nuevo, recordar que hemos computado el modelo secuencial y con threads en nuestra máquina y por eso el rendimiento al pasar a MPI no es el doble. Hubiese sido interesante probar todos los programas desde el laboratorio pero también ha sido beneficioso contemplar otra perspectiva distinta. La arquitectura juega un papel importante y en este caso se ve reflejado cómo mi máquina local termina antes que las pruebas con los nodos del laboratorio.

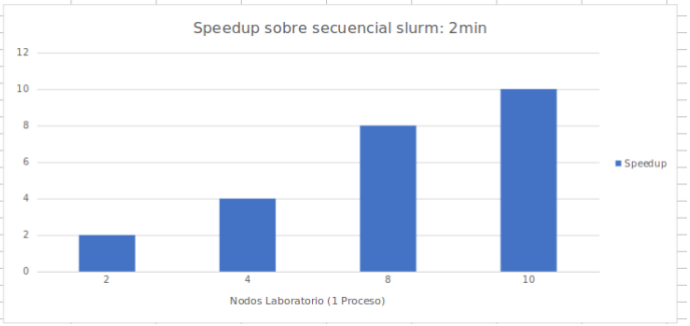
Por este motivo, realizaremos dos gráficas: una de ellas compara tiempos con el modelo secuencial desde mi máquina y otra comparando con el tiempo esperado que se obtendría desde slurm, basándonos en pruebas adicionales. Los recursos empleados del laboratorio se encuentran en la carpeta 'medidas_SLURM' y para las gráficas consultar el excel adjuntado. También, se añadirá una imagen de las gráficas igualmente.

Speedup secuencial local

Secuencial	threads	MPI	Nodos	Speedup
84.13	44	60	2	1.40216667
		30	4	2.80433333
		15	8	5.60866667
		12	10	7.01083333

Secuencial	threads	MPI	Nodos	Speedup
120	60	60	2	2
		30	4	4
		15	8	8
		12	10	10





Desde mi punto de vista, esta práctica ha sido muy interesante y completa. Con los dos primeros ejercicios, la comunicación MPI no bloqueante se entiende muy bien.
Lo que más me ha gustado ha sido el tercer ejercicio. Se ha abordado un tema interesante, que es la aproximación de integrales discretas y además hemos podido comparar con detalle un modo de ejecución secuencial con los paradigmas que existen sobre el desarrollo de programas paralelos.