Ejercicios cortos para MPI / C++

Objetivos

Que el estudiante sea capaz de:

1. Diseñar un algoritmo paralalelo basado en memoria distribuida, identificando y combinando los patrones de programación paralela estructurada adecuados tales como: fork, join, map y reduction.
2. Implementar un algoritmo paralelo identificando y combinando funciones MPI en lenguaje C++.
3. Depurar sistemáticamente un programa en una estación de trabajo (ET) hasta lograr el funcionamiento esperado utilizando algún depurador simbólico (asociado al IDE de su preferencia) o haciéndolo desde la consola mediante la interfaz de "gdb" por "comandos".
4. Evaluar programas MPI /C++ usando criterios como el desempeño (MPI_Wtime()), la aceleración y la eficiencia por proceso en algún "cluster" .

Procedimiento

Para cada ejercicio, el procedimiento general es:

1. Diseñar un algoritmo paralelo basado en memoria distribuida mediante el método de Fosler: partición, comunicación, agregación y asignación de tareas. Haga un dibujo o pseudo-código. Tome en cuenta que los procesos NO pueden compartir memoria, sólo pueden pasarse mensajes con datos.

2. Elaborar el código preliminar.

3. Depurar el programa hasta asegurar su funcionamiento correcto. Se recomienda depurar en dos o tres fases:
3.1 Siempre que se pueda, depurar con un único proceso en una ET, de tal manera que la comunicación entre los procesos no interfiera con la funcionalidad.
3.2 Depurar con dos procesos en una ET. Esta fase es precisamente para depurar la comunicación entre los procesos.
3.3 En caso de que el programa incluya varias fases de procesamiento, con distinta cantidad de procesos en cada fase, haga pruebas en una ET con al menos dos fases diferentes.

4. Una vez depurado, ejecutar el programa en alguno de los "cluster" y realizar el análisis de desempeño, aceleración y eficiencia por proceso llenando la tabla correspondiente.

6. Dependiendo de la evaluación, continuar en el punto #2 o #3 o continuar con el siguiente.

Evaluación

Dado que son cinco ejercicios, y el peso total de esta actividad en la evaluación del curso es de 10%, cada ejercicio pesa 2%.

Ejercicio #1 (basado en la asignación 3.2 para MPI / C++ del texto de Peter Pacheco)

Elabore un programa paralelo mediante MPI / C para aproximar el valor del número PI mediante el método de MonteCarlo. El programa deberá recibir como parámetros la cantidad de hilos y la cantidad de términos que se sumarán para aproximar el valor de PI. Ver página 267 del texto de Peter Pacheco.

Las salidas del programa son:

1. La aproximación de PI que se deberá contrastar en su precisión con el que se obtenga mediante la expresión: 4.0*atan(1.0).
2. El tiempo de ejecución neto (MPI_Wtime()), es decir, eliminando la entrada y salida de datos.

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$.

We can use this formula to estimate the value of π with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
   x = random double between -1 and 1;
   y = random double between -1 and 1;
   distance_squared = x*x + y*y;
   if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4*number_in_circle/((double) number_of_tosses);
```

Análisis de desempeño, aceleración y eficiencia: ci = cantidad de intentos, en celeste MPI_Wtime() ejecutado en un "cluster", en amarillo la aceleración respecto de la versión serial y en verde la eficiencia respecto de la versión serial.

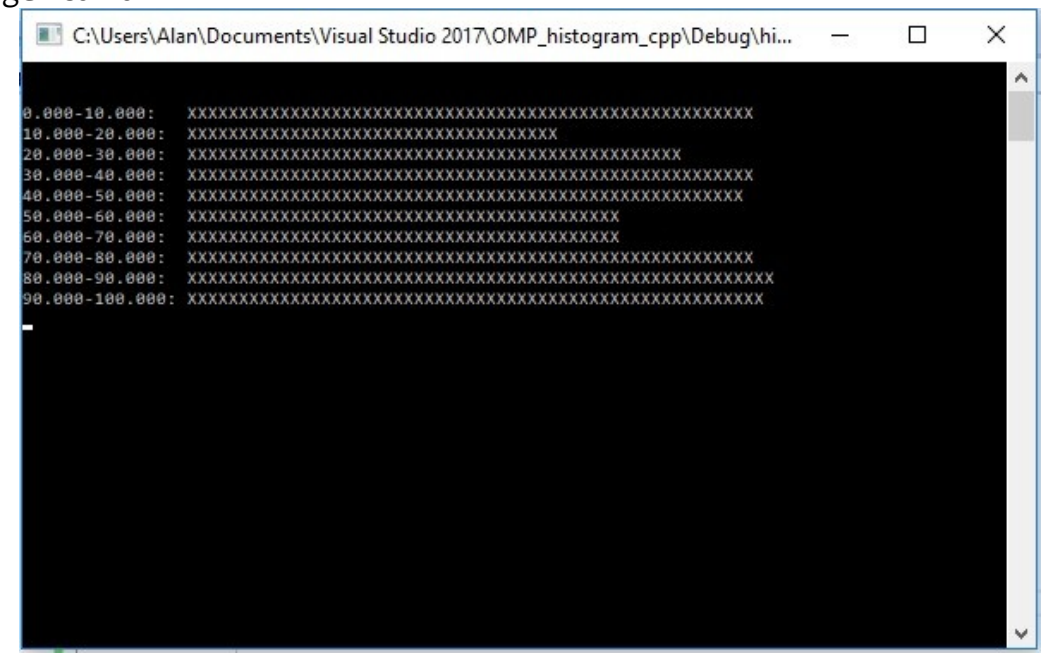
p	ci = 1000	ci = 10,000	ci = 100,000	ci = 1000	ci = 10,000	ci = 100,000	ci = 1000	ci = 10,000	ci = 100,000
1									
2									
4									
8									

Ejercicio #2 (basado en la asignación 3.1 para MPI / C++ del texto de Peter Pacheco)

Elabore un programa paralelo mediante MPI / C++ para el ejemplo que se usó para ilustrar el método de Fosler que genera un histograma con una muestra al azar de números enteros. El programa recibe parámetros de consola para:
1. la cantidad de rangos (y por ende de barras del histograma),
2. el valor mínimo del intervalo al que pertenecerán los números generados al azar,
3. el valor máximo del intervalo al que pertenecerán los números generados al azar,
4. la cantidad total de números al azar que se deberán generar.
Reutilice todo lo que pueda del código provisto para aligerar la solución.

Las salidas del programa son:

1. Una imagen como:



2. El tiempo de ejecución neto (MPI_Wtime()), es decir, eliminando la entrada y salida de datos.

Análisis de desempeño, aceleración y eficiencia: en todos los casos use 10 rangos y valores de 0 a 100, cv = cantidad de valores, en celeste MPI_Wtime() ejecutado en un "cluster", en amarillo la aceleración respecto de la versión serial y en verde la eficiencia respecto de la versión serial.

p	cv= 1000	cv = 10,000	cv = 100,000	cv = 1000	cv = 10,000	cv = 100,000	cv = 1000	cv = 10,000	cv = 100,000
1									
2									
4									
8									

Ejercicio #3: sobre la conjetura de Goldbach

Según wikipedia (https://en.wikipedia.org/wiki/Goldbach%27s_conjecture) la moderna conjetura de Goldbach (célebre matemático del siglo 18) establece que "cualquier entero mayor que cinco puede ser expresado como la suma de tres números primos". Elabore un programa paralelo mediante MPI / C++ que genere la suma de primos para todo entero mayor que cinco y menor que N. El programa recibe como parámetros:

1. N,
2. la cantidad de hilos,

El programa genera como salida, los tres primos para cada entero entre 5 y N:

1.
7 = 2 +5,
11 = 2 + 2 + 7
...
2. El tiempo de ejecución neto (MPI_Wtime()), es decir, eliminando la entrada y salida de datos.

Análisis de desempeño, aceleración y eficiencia: en celeste MPI_Wtime() ejecutado en un "cluster", en amarillo la aceleración respecto de la versión serial y en verde la eficiencia respecto de la versión serial.

p	N = 1000	N = 10,000	N = 100,000	N = 1000	N = 10,000	N = 100,000	N = 1000	N = 10,000	N = 100,000
1									
2									
4									
8									

Ejercicio #4 (basado en la asignación 3.8 para MPI / C++ del texto de Peter Pacheco)

Elabore un programa paralelo mediante MPI / C++ para ordenar mediante el algoritmo "merge-sort" un vector de números enteros generados al azar. El programa deberá recibir como parámetros del usuario la cantidad N de números que se deberán generar al azar.

Las salidas del programa son:

1. La secuencia ordenada de números generados al azar por consola (sólo para pruebas).
2. El tiempo de ejecución neto, es decir, eliminando la entrada y salida de datos.

NOTAS:

1. El programa **DEBERÁ** basarse en una estrategia "bottom-up" iterativa (NO recursiva).
2. El programa **DEBERÁ** usar los componentes de la STL (<vector>, merge, inplace_merge).
3. Puede suponer que la cantidad de elementos es divisible por la cantidad de procesos.

Procedimiento específico:

1. Elabore una primera versión en la que la intercalación o "merge" se cargue al proceso cero.
2. Elabore una segunda versión en la que la intercalación se haga por niveles y se distribuya a partes iguales entre los procesos diferente de cero en forma de árbol binario. En el siguiente diagrama, el primer nivel (0,...7) representa el ordenamiento mediante "sort" por parte de 8 (2 a la tres) procesos. En los siguientes niveles el símbolo de "+" representa la intercalación mediante "merge" o "inplace_merge" por cuatro, dos y un proceso. Puede suponer que la cantidad de procesos es potencia de dos y la cantidad de elementos es divisible por la cantidad de procesos.

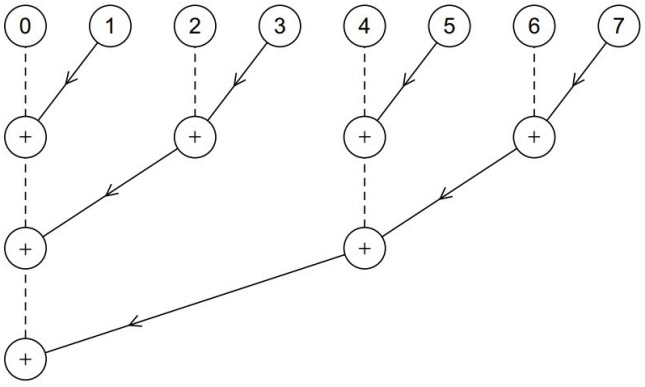


FIGURE 2.23

Adding the local arrays

Análisis de desempeño, aceleración y eficiencia: en celeste MPI_Wtime() ejecutado en un "cluster", en amarillo la aceleración respecto de la versión serial y en verde la eficiencia respecto de la versión serial. El valor de la sub-columna izquierda es para la primera versión, mientras que la sub-columna de la derecha para la versión dos.

p	N = 10,000		N = 100,000		N = 1,000,000		N = 10,000		N = 100,000		N = 1,000,000		N = 10,000		N = 100,000		N = 1,000,000	
1																		
2																		
4																		
8																		

Ejercicio #5

Modifique el programa basado en el algoritmo de Dijkstra para encontrar los caminos más cortos entre un nodo y todos los demás en un grafo no dirigido, así como sus longitudes, tomando como base la implementación para OpenMP / C++. Recuerde que NO se pueden comunicar datos "bool" con MPI, se deben representar mediante enteros: 1 == true, 0 == false.

Las entradas del programa son:

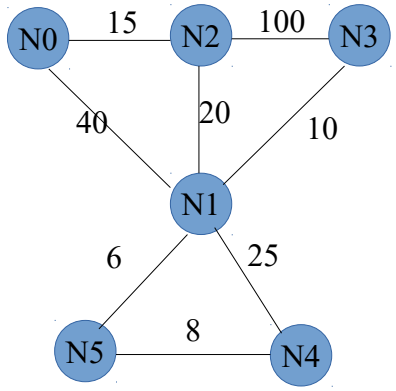
1. La cantidad de nodos del grafo.
2. La matriz de adyacencias de NxN entradas, para N nodos del 0 al N – 1.

La salidas del programa son:

1

1. Las longitudes de los caminos más cortos.
2. Los caminos más cortos representados como secuencia de nodos.

Para depurar el programa utilice el grafo de seis nodos que se usa en el código original:



Matriz de adyacencia:

0	40	15	Inf	Inf	Inf
40	0	20	10	25	6
15	20	0	100	Inf	Inf
Inf	10	100	0	Inf	Inf
Inf	25	Inf	Inf	0	8
Inf	6	Inf	Inf	8	0

antecesores:

0	2	0	1	5	1

Salidas

#n	dist	camino
0	0	
1	35	0,2,1
2	15	0,2
3	45	0,2,1,3
4	49	0,2,1,5,4
5	41	0,2,1,5

Análisis de desempeño, aceleración y eficiencia: N representa la cantidad de nodos en la red, "100.txt", "1000.txt" y "3500.txt" los archivos con las adyacencias. En todos los casos, dado un arco <i,j>, <j,i>, el valor asociado es 1, por tanto la matriz de adyacencias se genera con base en el archivo de adyacencias asignando uno como "costo" en cada adyacencia. En celeste MPI_Wtime() ejecutado en un "cluster", en amarillo la aceleración respecto de la versión serial y en verde la eficiencia respecto de la versión serial.

p	N = 100 100.txt	N = 1000 1000.txt	N = 3500 3500.txt	N = 100 100.txt	N = 1000 1000.txt	N = 3500 3500.txt	N = 100 100.txt	N = 1000 1000.txt	N = 3500 3500.txt
1									
2									
4									
8									