# CC332A - Programación Paralela - Tarea 8.

# Walter J. Felipe T. - 20171714F

## 1. Parallelizing Bucket Sort with MPI

**Source file** : *buckesort-mpi.cpp*

**Compiling and Running** : *mpic++ bucketsort-mpi.cpp && mpirun --oversubscribe -np 4 ./a.out*

Utilizando 2^{19} floats generados aleatoriamente tenemos los siguientes cálculos de tiempos totales de ejecución, se ha ejecutado 6 veces por proceso.

| Número de procesos | Tiempo total de ejecución (segundos) |
|---|---|
| 1 | 0.011605, 0.010265. 0.010527, 0.010339, 0.009722, 0.012100 |
| 4 | 0.007861, 0.007317, 0.008656, 0.011423, 0.009973, 0.008814 |
| 8 | 0.019460, 0.020466, 0.026152, 0.013920, 0.013131, 0.016967 |

El bucket sort es un algoritmo de ordenamiento que funciona dividiendo una arreglo en varios buckets. Luego, cada bucket se ordena individualmente.

Hacemos que las entradas se distribuyan con una variable aleatoria uniformeme (misma probabilidad para cada valor) en (0, 1). Para producir la salida, simplemente se ordena los números en cada bucket y luego revise el grupo de buckets en orden, enumerando los elementos en cada uno.

Para arreglos pequeños, el paralelismo no aumenta el rendimiento del algoritmo. Eso es porque el tiempo empleado para sincronizar el proceso es mayor que el tiempo empleado para ordenar la matriz. Para arreglos largos, como el que trabajamos, el paralelismo muestra un aumento significativo en el rendimiento.

Para realizar en algoritmo con open_mp revisar el archivo bucket-openmp.cpp. Referencia : https://github.com/adricarda/parallel-Bucket-Sort , se observa que al

compararlo con el qsort hay una mejor en la escabilidad presentando una complejidad dominada por el término (n\p log(n\p)) donde p es el número de hilos, y mientras se incremente la cantidad de buckets por hilo se incrementa la eficiencia, debido que se reduce el tiempo de comunicación, ya que se trabaja en memoria compartida.

## 2. Parallelizing Bitonic Algorithm with MPI

**Source file** : *bitonic-mpi.cpp*

**Compiling and Running** : *mpic++bitonic-mpi.cpp && mpirun --oversubscribe -np  4 ./a.out*

Utilizando 2^{19} floats generados aleatoriamente tenemos los siguientes cálculos de tiempos totales de ejecución.

| Número de procesos | Tiempo total de ejecución (segundos) |
|---|---|
| 1 | 0.054167,  0.052814, 0.055147, 0.052503, 0.052888, 0.052271 |
| 4 | 0.101098, 0.083378, 0.103138, 0.265658, 0.337409, 0.301193 |
| 8 | 0.103848, 0.270168, 0.286321, 0.097512, 0.125508, 0.106375 |

Se observa que en 4 y 8 procesos se reduce bastante el tiempo de ejecución, pero se descompensa un poco cuando se trabaja con más procesos puesto que el tiempo de comunicación y bloqueos se incrementa. Un buen balance entre memoria distribuida y compartida sería lo ideal para este algoritmo.