Pontificia Universidad Católica de Chile
Departamento de Computación
IIC-2333. Sección 1
Sistemas Operativos y Redes

Informe Tarea 2

Profesor: Cristian Ruz

Felipe Domínguez Claro.
1562188J

abril de 2019

Funcionamiento de la tarea

Para crear un programa que realmente use MapReduce, lo que hice fue separar el archivo de input en varios *buffers*, de tamaño fijo seteado como variable global. Cada *chunk* de información es procesado por un thread/proceso el cual realiza la función map para ese trozo de información. Cuando todos han terminado de procesar, se genera un proceso/thread por cada palabra, el cual toma la información recopilada por cada función map y saca la cuenta final de la palabra asociada.

Si bien en un sólo computador los beneficios de usar este algoritmo no se pueden apreciar, usando una red de muchos computadores, donde los procesos se pueden distribuir entre ellos, la eficiencia del programa aumenta considerablemente, ya que se ejecutan muchas tareas en paralelo.

Para construir la tarea, en un principio la estructuré usando hashtables, para hacerlo de manera eficiente. Durante el desarrollo me encontré con un problema que no pude resolver, así que opté por usar listas ligadas. La función para agregar nuevos nodos la modifiqué para que cada vez que inserte uno, revise si existe el elemento. Si este ya existe, a su contador se le suma uno.

La parte de threads fue lograda de manera completa. Los resultados de cada map, son una lista ligada con las palabras y frecuencia de cada uno. Es una lista global se guardan los punteros a las distintas listas de cada map. Esta última es usada por el thread reduce, que crea una lista ligada de las palabras finales y agrega elemento por elemento el outputs de los maps. Con mi estructura de lista ligada este proceso es bastante simple, ya que basta solo con agregar uno a uno las palabras.

Para procesos, luego de unos días intentarlo terminarlo, no pude debido a diversos problemas que surgieron en el camino. A pesar de esto, . Como la implementación usada en threads no era 100% compatible con esta versión, el proceso de guardar en memoria los datos difiere. Para procesos, las regiones de memoria compartidas fueron codificadas de la siguiente forma:

TOTAL_LEN, LEN_W1, W11, ..., W1f,C1 LEN_W2, W21, ..., W2f, C2, ...

Donde:

- TOTAL_LEN: Cantidad de palabras en memoria compartida
- LEN_Wi: Largo de la palabra i
- Ci: cantidad de veces que se repite la palabra

A modo de ejemplo, si tenemos las palabras: programe, mucho, mucho. Según la codificación usada sería:

3,8,p,r,o,g,r,a,m,e,1,5,m,u,c,h,o,2

Para reutilizar las funciones map y reduce, y que sean más compatibles entre las 2 versiones del programa, uso la función *shm_to_ll* que toma la region de memoria compartida codificada como describí anteriormente, y crea un array de listas ligadas para luego entregarle esto a reduce.

Finalmente, en ambas versiones, el resultado del reduce lo paso por la función *write_outup que* escribe un archivo con el formato pedido.

Los test realizados en este informe fueron ejecutados en un MacbookPro 13”, con un procesador Intel i5 2.4GHz de dos núcleos y 8gb RAM, corriendo en el sistema operativo MACOS 10.14.4.

Pregunta 1

Antes de realizar el análisis, cabe destacar que como terminé implementando listas ligadas en lugar de hashtable, el programa quedó menos eficiente de lo que debiera. Por lo tanto los datos pueden tener bastante ruido. Además, la implementación de procesos y threads no es igual, si no que difiere en el proceso de guardar los chunks de palabras en memoria.

En la tabla 1 podemos ver los tiempos de ejecución para ejecutar mapreduce sobre dos archivos usando ambas versiones del programa (no alcancé a realizar threads...). Para obtener más información, por cada archivo se usaron tamaños de buffer de 2.000, 8.000 y 24.000 palabras. Si observamos la tabla, podemos ver que el Systime aumenta a medida que disminuimos el porte del buffer. La principal razón de esto, es porque al llenarse más rápido el buffer, se crean más threads. Esto provoca que el sistema operativo dedique más tiempo a crearlos y realizar swap entre los distintos threads.

C. Palabras	Buffer size	Real (s)	User (s)	Sys (s)
10000	2000	0.0490	0.0675	0.0150
10000	8000	0.0587	0.0850	0.0125
10000	24000	0.0685	0.1012	0.0125
60000	2000	0.0940	0.2277	0.0244
60000	8000	0.0764	0.2255	0.0166
60000	24000	0.0842	0.2033	0.0100

Tabla 1: resultados para mapreduce con threads

C. Palabras	Buffer size	Real (s)	User (s)	Sys (s)
10000	-	-	-	-
10000	-	-	-	-
10000	-	-	-	-
60000	-	-	-	-
60000	-	-	-	-
60000	-	-	-	-

Tabla 2: resultados para mapreduce con procesos

Si bien no alcancé a terminar la parte de procesos. Lo más probable es que la version de threads sea más eficiente, ya que correr threads es mucho más eficiente para el sistema operativo. Y dado que cada threads o proceso debe realizar sólo una tarea determinada, la versión threads es más eficiente a nivel general.

Como crear procesos es más costoso, en esta versión probablemente el *Systime* suba considerablemente, mientras que el *Usertime* es probable que se mantenga dentro de un rango cercano al de threads. Dado esto, sería esperable observar que el *Realtime* es más pequeño en threads.

Pregunta 2.

Para esta pregunta, preferí usar como variable el tamaño del buffer en lugar de "cantidad de threads". Ambas variables tienen una relación directa, ya que la cantidad:

Cantidad_threads = ceil(file_size / buffer_size)

Por lo tanto, el rango de threads fue desde 658.844 hasta 1.

Buffer	Real (s)	Min (s)	Max (s)
2000	0.797555556	0.658	1.341
4000	0.618555556	0.605	0.634
8000	0.569777778	0.537	0.667
16000	0.550888889	0.54	0.559
32000	0.538222222	0.52	0.564
128000	0.571444444	0.553	0.617
512000	0.708222222	0.07	0.897
2048000	1.641555556	1.52	1.863

Tabla 3: resultados version threads para input de 1.317.688 palabras

Buffer	Real (s)	Min (s)	Max (s)
2000	-	-	-
4000	-	-	-
8000	-	-	-
16000	-	-	-
32000	-	-	-
128000	-	-	-
512000	-	-	-
2048000	-	-	-

Tabla 4: resultados version procesos para input de 1.317.688 palabras

En el gráfico 1 puse sólo el rango de buffer_size entre [2000, 32000], ya que de otra manera el eje del gráfico se disparaba.

De los datos podemos ver que si extrapolamos la cantidad de threads (o el tamaño de los buffer), podemos ver que en ambos casos extremos el rendimiento está muy por debajo del resto. El contexto que más me gustaría comparar es cuando se usa un sólo thread. En este experimento tomó 1.863 segundos, mientras que usando 3 threads, el tiempo de ejecución fue de 0.897, lo que es cercano a la mitad. Este aumento de performance es esperable, ya que como el procesador tiene 2 núcleos, la diferencia de realizar maps simultáneos ayuda mucho a realizar la tarea más eficiente.

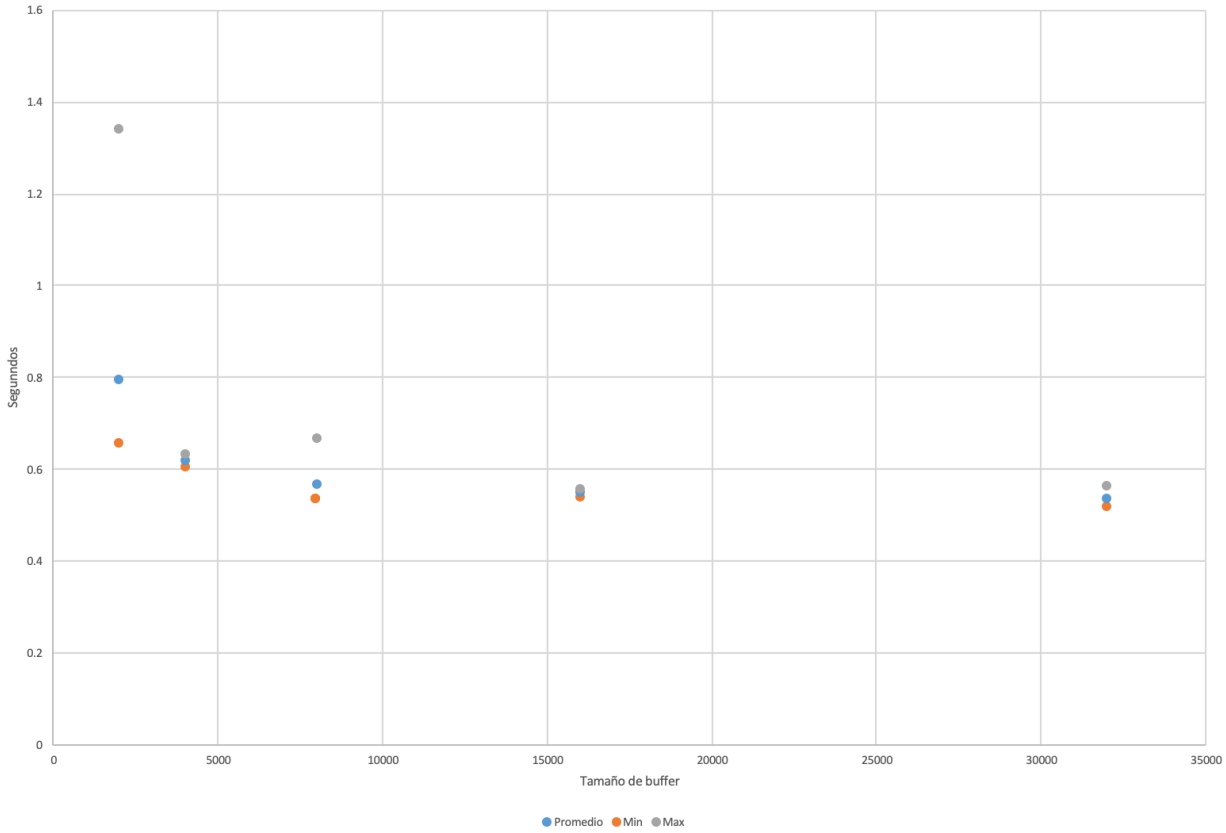

Gráfico 1: tiempos de ejecución usando distintos buffer en input de 1.317.688 palabras