Informe de Trabajo Práctico Nº 3

6 de mayo de 2016

75.42 Taller de Programación I

Cátedra Veiga

Facultad de Ingeniera - UBA

Autor: Martín Stancanelli

Padrón: 95188

Objetivo

El objetivo de este trabajo practico fue la implementación de el algoritmo *MapReduce* en C++, utilizando la librería pthread para el multithreading y aplicando lo aprendido en trabajos anteriores sobre sockets.

Configuración del entorno

El entorno de trabajo elegido para la realización del trabajo practico fue una PC con SO Linux Mint 17.3 y compilador gcc 4.8 configurado para utilizar el estándar de C++98 y la librería pthread.

Explicación de enunciado

El fin del trabajo es realizar un programa que permita recibir los datos de las temperaturas de muchas ciudades para todos los días del mes de marzo y, a través de la implementación del algoritmo MapReduce poder definir que ciudad tuvo la temperatura mas alta en cada día.
La idea es que los mappers se ejecuten en cada cliente y luego los datos mapeados se envíen al servidor, quien por cada día de marzo ejecutara un reducer que encontrara la ciudad que tuvo mayor temperatura.

Desarrollo

Para comenzar con el trabajo se portaron las librerías de sockets y threading usadas en trabajos previos a C++. Luego nos enfocamos en la realización del algoritmo MapReduce.

Como salida de los mappers se tomo al día como clave, ya que es el valor por el que los reducers van a separar los datos. Por lo tanto los mappers van a recibir los datos de la forma BuenosAires 25 31 y van a retornar 31 BuenosAires 25, donde el 31 en este ejemplo es el día de marzo. Una vez que se tuvo a los mappers funcionando correctamente pasamos a los reducers.

Los reducers entonces reciben un par clave, valor del tipo día, [ciudad, temperatura], siendo esta ultima una clave compuesta. Entonces el algoritmo del reducer, que recibe solo las claves de un determinado día, se va a basar en recorrer la lista de claves y tomar la ciudad que tenga la temperatura mas alta. En caso de empate se guardan todas las ciudades con esa temperatura.

Una vez que el algoritmo MapReduce se comportaba correctamente se integro este con el modelo de cliente-servidor y la parte de sockets y threads que requería el trabajo.

Se incluye un diagrama (1) explicando la estructura del algoritmo MapReduce que finalmente se empleó, donde cada reducer recibe todos los valores de una misma clave, en este caso los días.

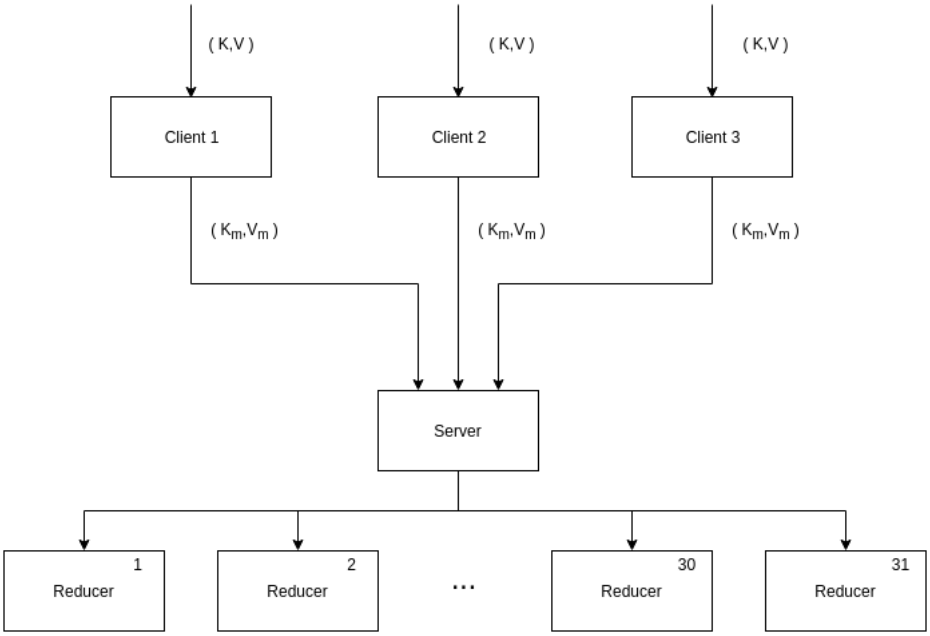

Figura 1: Diagrama de la estructura del algoritmo MapReduce empleado

Clases utilizadas

A continuación se incluye una breve descripción de las clases que modelan el dominio del problema

- *Client*: Procesamiento de los datos de entrada, preparación de los mismos para envío al *Mapper*.
- *Mapper*: Mapeo de los datos de entrada al formato esperado de salida, en este caso recibe (ciudad, temperatura, día) y devuelve (día, temperatura, ciudad).
- *ServerProxy*: Encapsulamiento del envío de datos por parte del cliente al servidor.
- *InputParser*: Parseo de los datos de entrada recibidos, generando pares día, value.
- *Value*: Estructura de datos que almacena el valor compuesto necesario para nuestro problema (temperatura, ciudad).
- *AceptorWorker*: Hilo aceptador de nuevas conexiones, llama a los hilos que reciben.
- *ClientProxy*: Encapsulamiento de la recepción de datos del cliente.
- *MappedData*: Estructura de datos que almacena los datos recibidos desde el cliente.
- *ParsedData*: Almacena los datos parseados provenientes de *MappedData*.

- *DayValuesMap*: Hashmap que almacena por cada día (clave), una lista de valores. Es una estructura compartida.
- *ReceiverWorker*: Hilo que recibe los datos, del cliente, parsea y almacena en *DayValuesMap*.
- *Reducer*: Recibe todos los valores para un determinado día y los reduce a la ciudad con mayor temperatura.
- *ReducerWoker*: Hilo que reduce los datos de entrada, llamando a un *Reducer*. Se lanzan hasta 4 simultáneamente.

Conclusión

Si bien el algoritmo MapReduce ya era conocido, la implementación desde cero en C++ permitió que se incorporaran mejor los conocimientos de threading que se empezaron a ver en el trabajo practico anterior.

El tratar de resolver los conflictos que se presentan por culpa de la concurrencia, por ejemplo las race conditions, obliga a que se entiendan mejor los conceptos y se aprovechen mucho mas.

Respecto a las correcciones marcadas en la entrega anterior, se tomaron en cuenta todas y cada una para la presente versión. Sin embargo, el uso de select en el socket no puedo ser evitado, ya que el *SERCOM* mataba al proceso del servidor si no se incluía esta función en el socket. Aguardamos comentarios sobre posibles variantes o soluciones a al uso de select en el socket, para poder así aprender sobre el uso o no de este método.