CI3825 - Sistemas de Operación I
Proyecto 1

Abril 29, 2015

1. Introducción

En el mundo de la programación paralela y concurrente, el modelo de *Map-Reduce* es uno de los más importantes y de mayor uso. Su uso se vuelve indispensable al procesar una cantidad de datos tal que hacerlo de manera lineal y con un solo proceso haría el problema intratable. Básicamente, el modelo consta de dos funciones: Un `Map()`, que se encarga de filtrar la información y un `Reduce()`, que se encarga de reunir valores según algún criterio y devolver un resultado.

Para este proyecto, se desea que usted implemente dos versiones muy básicas de este modelo haciendo uso de procesos e hilos. El objetivo es que se familiarice con las librerías de programación concurrente y con los conceptos que son indispensables para ello.

2. Friend Find

Suponga que tiene una lista de usuarios de una red social (llamémosla *Facelook*) y que en dicha red social existe una relación entre usuarios la cual llamaremos «amistad». Es decir, el usuario `Ana` puede ser amiga del usuario `Bernardo`. Con el propósito de brindar a cada usuario la oportunidad de incrementar la cantidad de amigos que tiene, el sistema debe ofrecer una lista de amigos en comunes que tengan dos personas.

Se desea diseñar un algoritmo que, dada una lista de usuarios cada una con su lista de amigos, retorne para cada par de usuarios, la lista de amigos que tienen en común.

Así, si `Ana` es amiga de `Bernardo`, `Cristina` y `David`, y `Bernardo` es amigo de `Cristina`, `Gabriel` y `Helena`, el algoritmo debe poder determinar que `Ana` y `Bernardo` tienen como amiga en común a `Cristina`.

3. Implementación

Para ajustarse a nuestras versiones simplificadas del modelo *Map-Reduce*, se debe implementar ambas funciones de manera que se puedan ejecutar con múltiples procesos ó hilos.

3.1. Entrada

La entrada a su programa consta de un archivo en texto plano el cual tendrá un número arbitrario de líneas de la siguiente estructura:

```
[Nombre] -> [Nombre] [Nombre] [Nombre] ... [Nombre]
```

Donde `Nombre` es una cadena de caracteres compuesto por las letras minúsculas (a-z) o mayúsculas (A-Z). Ningún nombre tendrá números o caracteres especiales. La cantidad de usuarios a la derecha del `->` es también arbitraria.

Si un usuario no tiene amigos, esa línea de la entrada sería

```
[Nombre] -> -None-
```

3.2. Salida

La salida consistirá en un archivo con todas las parejas posibles en el conjunto total de usuarios presentes en la entrada y para cada una, la lista asociada de los amigos que tienen en común. La estructura debe ser la siguiente:

```
([Nombre] [Nombre]) -> [Nombre] [Nombre] [Nombre] ... [Nombre]
```

De no tener amigos en común, la salida sería

```
([Nombre] [Nombre]) -> -None-
```

3.3. Map

La función de `Map` deberá partir de la entrada y construir listas que asocien a dos usuarios a las listas de amigos donde aparecen. Es decir, si se tiene:

```
Ana -> Bernardo Cristina David
Bernardo -> Ana Cristina David Ernesto
```

La función `Map` debe ejecutarse sobre cada entrada y devolver[1]:

[1] En este sentido «devolver» se refiere a construir (o agregar elementos a las) listas, no a imprimir cadenas con el formato de la entrada o la salida general del programa

```
Map(Ana -> Bernardo Cristina David):
    (Ana Bernardo) -> Bernardo Cristina David
    (Ana Cristina) -> Bernardo Cristina David
    (Ana David) -> Bernardo Cristina David
Map(Bernardo -> Ana Cristina David Ernesto):
    (Ana Bernardo) -> Ana Cristina David Ernesto
    (Bernardo Cristina) -> Ana Cristina David Ernesto
    (Bernardo David) -> Ana Cristina David Ernesto
    (Bernardo Ernesto) -> Ana Cristina David Ernesto
```

Note que en el caso de `(Ana Bernardo)`, se debe ir construyento los elementos de la lista de manera incremental. Es decir, al finalizar la pasada por toda la entrada, dicha tupla debe estar asociada a `Bernardo Cristina David` y `Ana Cristina David Ernesto`, sin que exista reducción de redundancia (piensen en listas de listas o simplemente elementos repetidos).

Esta función deberá procesar la lista obtenida de la entrada usando una cantidad determinada de hilos/procesos siendo divido el trabajo equitativamente. Note que debe manejar la posibilidad de que dos o más hilos agreguen información a la lista correspondiente a una misma tupla.

3.4. Reduce

Ahora es el turno de la función `Reduce`, la cual tomará las listas creadas por `Map` y la reducirá eliminando redundancias. En concordancia con el ejemplo anterior, `Reduce` devolvería:

```
Reduce((Ana Bernardo) -> Bernardo Cristina David, Ana Cristina David Ernesto):
    (Ana Bernardo) -> Cristina David
```

Esta función deberá procesar la lista obtenida de la entrada usando una cantidad determinada de hilos/procesos siendo divido el trabajo equitativamente.

4. Programa principal

Usted deberá implementar dos versiones del programa, una que usará procesos y otra que usará hilos: friendfindP y friendfindH respectivamente.

La llamada desde la línea de comandos es de la forma:

```
./friendfindP [-n #_de_procesos] <entrada> <salida>
./friendfindH [-n #_de_hilos] <entrada> <salida>

donde:

-n: Número de procesos o hilos que se ejecutarán sobre las funciones map y reduce.
<entrada>: Nombre del archivo de entrada
<salida>: Nombre del archivo de salida
```

Note que la opción -n es opcional, cuando no se especifica significa que sólo un proceso o hilo hace todo el trabajo. <entrada>y <salida>si son parámetros obligatorios.

5. Comunicación

La comunicación entre los procesos se hará a través de archivos intermedios cuyos nombres serán de la forma PID.txt de manera que el proceso padre los pueda leer fácilmente. Cada proceso hijo hace su trabajo parcial, lo guarda en archivos y el proceso padre genera las listas finales a partir de estos resultados parciales.

La comunicación entre los hilos se hará a través de estructuras de datos compartidas.

6. Entrega

La fecha límite para la entrega es el día lunes 18 de mayo del 2015 a las 11:59 PM mediante Aula Virtual. Se deberá entregar un archivo .tar.gz cuyo nombre será de la forma `Proyecto1_GXX.tar.gz` donde XX es el número de su grupo. El archivo debe contener:

- Código fuente.
- Makefile con el cual se debe compilar su código (debería compilarse simplemente con una llamada a `make`).
- Informe de no más de 3 páginas donde se explique su implementación, detalles de uso de hilos y procesos y una tabla comparativa de los tiempos de ejecución de hilos vs procesos (incluyendo el caso de 1 solo hilo/proceso) para un caso de su propia autoria.

7. Consideraciones adicionales

- La estructura contenedora para las distintas partes de la implementación debe ser una lista de asignación de memoria dinámica. No está permitido usar un arreglo «muy grande» y estático para la solución.
- Toda memoria que sea reservada debe ser liberada. Es decir, al terminar el programa, no debe existir *leaks* de memoria. Use Valgrind para verificar que lo están haciendo bien.
- Su código debe estar debidamente documentado. Cada función debe tener una breve descripción de lo que hace y de los parámetros que son pasados a la misma.