DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Sistemas Operativos
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Balboa, Fernando	246/15	fbalboa95@gmail.com
Lopez Valiente, Patricio Nicolas	457/15	patricio454@gmail.com
Zdanovitch, Nikita	520/14	3hb.tch@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El siguiente Trabajo Práctico consistió en implementar un `DistributedHashMap`, que no es más que la clase `ConcurrentHashMap` de forma distribuida utilizando el standard *Message Passing Interface (MPI)*. Ésto significa que tendremos una red de nodos (que simulan computadoras diferentes) y que cada uno de ellos almacena su propio `HashMap`. Dentro de este grupo, hay uno destacado con el que interactúa el usuario y que llamamos $consola$. Este último no almacena un HashMap, sino que se encarga de manejar la comunicación entre el usuario y el resto de los nodos. Salvo esta excepción, los otros serán indistinguibles, pero podrán ser identificados por el atributo $rank$, un natural que es único a ellos. Por ejemplo, el $rank$ de la consola siempre es 0. Llamaremos np a la cantidad total de nodos, incluyendo a la consola.

Para la implementación particular de este Trabajo, utilizamos el lenguaje `C++` y la libreria `mpi.h`[1]. Como aclaración, diremos que en los pseudocódigos presentados, en la mayor parte de los casos, se ignoran o cambian las aridades reales de las funciones, así como el manejo de memoria para el lenguaje en particular, para facilitar la comprensión del algoritmo. Ante cualquier duda, consultar el código fuente provisto.

[1] Para más información, consulte https://computing.llnl.gov/tutorials/mpi/

2. Implementación de la consola

En esta sección explicaremos la implementación de las funciones que el usuario puede pedirle a la consola que maneja del `DistributedHashMap`. Ésta se encarga de recibir los comandos, interpretarlos y pedirle al resto de los nodos la información que necesite para devolverle al usuario una respuesta correcta.

2.1. Función member()

Esta función debe preguntarle a todos los nodos si tienen el string pasado por parámetro. Si alguno responde que si, entonces la consola imprime por pantalla que la palabra se encuentra en el HashMap. De lo contrario, escribe que no. El pseudocódigo es:

Algorithm 1 Implementación de la función `member` de la consola

void member(`string` key)

```
1:  bool esta = false
2:  int myRank = MPI_GetRank()
3:  char [] key_buffer
        // Esta función copia el string key en el buffer key_buffer
4:  copy_to_buffer(key_buffer, key)
5:  for i = 1 a np do
        // Envío de mensaje no bloqueante
        // Enviamos al nodo i un buffer de chars que contiene la key pedida, e indicamos cuál es el tamaño de la misma. El Tag se utiliza para indicarle al nodo qué hacer con la palabra que lee. En este caso, buscarla en su HashMap
6:      MPI_Isend(&key_buffer, key.length(), MPI_CHAR, i, TAG_MEMBER)
7:  end for
8:  bool res
9:  for i = 1 a np do
        // Lectura de mensajes bloqueante
        // Esperamos a que todos los nodos envíen si encontraron o no la palabra. El mensaje recibido es true si lo encontró y false si no. Utilizamos MPI_ANY_SOURCE para recibir mensajes de cualquier nodo en forma no determinística. Además, solo leemos mensajes con el tag TAG_MEMBER.
10:     MPI_Recv(&res, 1, MPI_BOOL, MPI_ANY_SOURCE, TAG_MEMBER)
11:     if res then
        // Algún nodo tiene la palabra
12:         esta = true
13:     end if
14: end for
        // Imprime por pantalla si la palabra está o no en el HashMap
15: ImprimirResultado(esta)
```

2.2. Función addAndInc()

Esta función debe envíarle un mensaje a todos los nodos para agregar el string pasado por parámetro. Como sólo queremos agregar la palabra una vez, necesitamos que sólo un nodo la cargue en su HashMap. La consola le asigna la tarea al primero que responda a su mensaje, mientras que al resto les indica que no hagan nada.

Algorithm 2 Implementación de la función `addAndInc` de la consola

```
void addAndInc(string key)
    // Aviso a los nodos que quiero agregar una palabra. Lo importante no es el mensaje en sí, sino el tag utilizado.
 1: char add_char = 'a'
 2: for i = 1 a np do
 3:     MPI_Send(&add_char, 1, MPI_CHAR, i, TAG_ADD_START)
 4: end for
 5: int receptor
 6: char skip_char = 's'
 7: char [] palabra
 8: for i = 1 to np do
    // Espero la respuesta de alguno. En receptor queda guardado quién envió el mensaje (su rank).
 9:     MPI_Recv(&receptor, 1, MPI_INT, MPI_ANY_SOURCE, TAG_ADD_RECEPTOR)
10:     if i ==1 then
    // Fue el primero en responder. Le asigno la tarea con TAG_ADD_WORD
11:         copy_to_buffer(palabra, key)
12:         MPI_Isend(&palabra, key.length(), MPI_CHAR, receptor, TAG_ADD_WORD)
13:     else
    // No fue el primero, le indico que no haga nada con TAG_SKIP. Nótese que el mensaje es irrelevante, sólo importa el tag.
14:         MPI_Isend(&skip, MPI_CHAR, MPI_CHAR, receptor, TAG_SKIP)
15:     end if
16: end for
    // Asumo que el nodo elegido agregó bien la palabra. No es necesario que avise que lo hizo.Si el nodo todavía no terminó de agregarlo y justo le piden una función como member, dado que el nodo es un ciclo y no tiene varios threads, va a terminar de agregar la palabra antes de responder a member.
```

2.3. Función maximum()

Esta función debe buscar el máximo de todo el HashMap. Como éste se encuentra distribuido entre todos los nodos, primero es necesario armarlo entero. Para ésto, la consola le pide a cada nodo que le envíe palabra por palabra el contenido de su HashMap. La consola agrega entonces cada una a un HashMap local. Una vez completo, busca el máximo y lo imprime por pantalla. El pseudocódigo es:

Algorithm 3 Implementación de la función `maximum` de la consola

```
void maximum()
    // Aviso a los nodos que quiero que me envíen el contenido de sus HashMaps. Lo importante no es el mensaje en sí, sino el tag utilizado.
 1: char maximum_char = 'm'
 2: for i = 1 a np do
 3:     MPI_Send(&maximum_char, 1, MPI_CHAR, i, TAG_MAXIMUM_START)
 4: end for
 5: HashMap h                                                    // El HashMap local
 6: int completados = 0
 7: char [] palabra
 8: int cant_chars
 9: MPI_Status status
10: while completados < np - 1 do
    // Me fijo de forma bloqueante si llegó algún mensaje. En status queda guardada información del mismo (quién lo envío, tag, etc.)
11:     MPI_Probe(MPI_ANY_SOURCE, MPY_ANY_TAG, &status)
    // Obtengo cuánto mide el mensaje. Almaceno el resultado en cant_chars
12:     MPI_Get_count(&status, MPI_CHAR, &cant_chars)
    // Leo el mensaje en cuestión
13:     MPI_Recv(&palabra, cant_chars, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG)
    // El nodo puede haber enviado una palabra para agregar (TAG_PALABRA) o indicarme que ya no tiene más palabras (TAG_MAXIMUM_END)
14:     if status.MPI_TAG == TAG_PALABRA then
15:         string key = buffer_to_string(palabra, cant_chars)
16:         h.addAndInc(key)
17:     else
18:         if status.MPI_TAG == TAG_MAXIMUM_END then
19:             completados++
20:         end if
    // Nunca debería entrar acá, pero si lo hago, simplemente descarto el mensaje leído.
21:     end if
22: end while
    // Obtenemos el máximo y lo imprimimos
23: par<string, int> max = h.maximum()
24: ImprimirMaximo(max)
```

2.4. Función load()

Esta función toma como parámetro una lista de strings que son nombres de archivos. La idea es entonces asignarle cada uno a un nodo distinto para que carguen las palabras del archivo al HashMap. Si hay más archivos que nodos, cada vez que se libera uno, debe asignársele un nuevo archivo para cargar.

Algorithm 4 Implementación de la función `load` de la consola

```
void load(lista<string> archivos)
 1: int agregados = 0
 2: char [] archivo
 3: iterador it = archivos.begin()
 4: for nodo = 1 a np do
 5:     if it != archivos.end() then
 6:         copy_to_buffer(archivo, *it)
        // Le envío al nodo un archivo para cargar con el TAG_LOAD
 7:         MPI_Send(&archivo, *it.length(), MPI_CHAR, nodo, TAG_LOAD)
 8:     else
        // No quedan más archivos
 9:         break
10:     end if
11:     it++
12: end for
13: MPI_Status status
14: char ok_msg
15: while it != archivos.end() do
        // Cada vez que un nodo termina de cargar un archivo, envía un mensaje con TAG_-
    LOADED. Si todavía no cargue todos, le envío otro.
16:     MPI_Recv(&ok_msg, 1, MPI_CHAR, MPI_ANY_SOURCE, TAG_LOADED, &status)
17:     agregados++
        // Me fijo quién envió el mensaje y le mando otro archivo
18:     int nodo_libre = status.MPI_SOURCE
19:     copy_to_buffer(archivo, *it)
20:     MPI_Send(&archivo, *it.length(), MPI_CHAR, nodo, TAG_LOAD)
21:     it++
22: end while
        // Espero a que todos los nodos me respondan que agregaron los archivos que faltan.
    De esta forma no quedan mensajes colgados en la cola que puedan interferir con sucesivas
    llamadas a load.
23: while agregados < archivos.size() do
24:     MPI_Recv(&ok_msg, 1, MPI_CHAR, MPI_ANY_SOURCE, TAG_LOADED, &status)
25:     agregados++
26: end while
```

2.5. Funcion quit()

Esta función sólo se utiliza para indicarle a los nodos que liberen sus recursos y terminen su ciclo. La idea es entonces enviarles un mensaje sin importancia pero con un tag particular que el nodo pueda interpretar como una orden de finalización. El pseudocódigo es:

Algorithm 5 Implementación de la función `quit` de la consola

```
void quit()
1: char quit_char = 'q'
2: for i = 1 a np do
       // Envío las señales de forma no bloqueante
3:     MPI_Isend(&quit_char, 1, MPI_CHAR, i, TAG_MAXIMUM_START)
4: end for
```

3. Implementación de los nodos

Todos los nodos ejecutan el mismo código y tienen un HashMap propio. La idea es que cada uno escuche mensajes de la consola y realicen la tarea pedida en dicho mensaje. En la mayor parte de los casos, el mensaje en particular no importa, sino que el nodo utiliza el tag del mismo para interpretar la orden. La implementación propuesta consisten entonces en un ciclo infinito que sólo termina cuando la consola envía un mensaje con el tag TAG_QUIT.

Primero creamos las variables locales a utilizar, como el HashMap o el buffer donde almacenar los mensajes recibidos. Luego comienza el ciclo principal, que consiste en leer un mensaje y realizar alguna operación de acuerdo al tag del mismo. Como pide el enunciado, los nodos llaman a la función $trabajarArduamente()$ antes de enviar mensajes. Presentamos a continuación el pseudocódigo:

Algorithm 6 Implementación de un nodo

```
void nodo(uint rank)
1: HashMap h
2: MPI_Status status
3: int tam_msg, tag
4: char [] msg, palabra_msg
       // Ciclo principal
5: while true do
       // Espero algún mensaje de la consola
6:     MPI_Probe(CONSOLA_RANK, MPI_ANY_TAG, &status)
7:     MPI_Get_count(&status, MPI_CHAR, &tam_msg)
8:     MPI_Recv(&msg, tam_msg, MPI_CHAR, CONSOLA_RANK, MPI_ANY_TAG, &status)
9:     tag = status.MPI_TAG
```

```
    // Decido qué hacer de acuerdo al tag enviado por la consola
10:   if tag == TAG_QUIT then
    // Termino la ejecución
11:       return
12:   else if tag == TAG_MEMBER then
    // Busco la palabra en el HashMap local y le envío a la consola el resultado con el TAG_-
  MEMBER de forma no bloqueante
13:       string key = buffer_to_string(msg, tam_msg)
14:       bool esta = h.member(key)
15:       trabajarArduamente()
16:       MPI_Isend(&esta, 1, MPI_BOOL, CONSOLA_RANK, TAG_MEMBER)
17:   else if tag == TAD_ADD_START then
    // Le respondo a la consola que estoy disponible
18:       trabajarArduamente()
19:       MPI_Isend(&rank, 1, MPI_INT, CONSOLA_RANK, TAG_ADD_RECEPTOR)
20:   else if tag == TAG_ADD_WORD then
    // Fui el primero en responder y por lo tanto la consola me asignó la tarea de cargar la
  palabra al HashMap
21:       string key = buffer_to_string(msg, tam_msg)
22:       h.addAndInc(key)
23:   else if tag == TAG_SKIP then
    // No fui el primero en responder. No hago nada porque otro nodo agregó la palabra
24:   else if tag == TAG_LOAD then
    // Cargo el archivo que me mandó la consola y le envío un mensaje diciendo que lo hice.
  Sólo importa el TAG_LOADED
25:       string archivo = buffer_to_string(msg, tam_msg)
26:       h.load(archivo)
27:       char ok_msg = 'k'
28:       trabajarArduamente()
29:       MPI_Isend(&ok_msg, 1, MPI_CHAR, CONSOLA_RANK, TAG_LOADED)
30:   else if tag == TAG_MAXIMUM_START then
    // Creo un iterador del HashMap local y le envío las palabras a la consola con el TAG_-
  PALABRA. Cuando no hay más, le envío un mensaje vacío con el TAG_MAXIUM_END para
  avisarle que terminé de enviarle palabras
31:       trabajarArduamente()
32:       while Iterador it = h.begin() ≠ h.end() do
33:           copy_to_buffer(palabra_msg, *it)
34:           MPI_Isend(&palabra_msg, *it.length(), MPI_CHAR, CONSOLA_RANK, TAG_PA-
  LABRA)
35:           it++
36:       end while
37:       MPI_Isend(&palabra_msg, 0, MPI_CHAR, CONSOLA_RANK, TAG_MAXIMUM_END)
38:   end if
39: end while
```

4. Tests

Para comprobar la correctitud de nuestros algoritmos, desarrollamos los siguientes test:

- `TEST MEMBER`: Este test comprueba en primera instancia que un determinado set de palabras no pertenezcan al HashMap. Luego agrega algunas de ellas y verifica que pertenezcan al mismo.
- `TEST LOAD`: Este test carga un set de archivos. En caso de que la cantidad de archivos sea mayor que `np`, comprueba que el primer nodo que se libera sea el que cargue el siguiente archivo. Luego verifica que efectivamente cada una de las palabras pertenezcan al HashMap.
- `TEST ADD`: Este test agrega un set de palabras e imprime en el log qué nodo agregó cada una. Realiza ésto 4 veces y comprueba mediante la función `diff` que la selección del nodo que carga la palabra no es determinística.
- `TEST MAX`: Este test carga un set de archivos y comprueba que el máximo sea efectivamente el correcto. Para comprobar las demás funcionalidades de `maximum()` se provee el Log de una inspección manual, ya que éste es usualmente generado con líneas de texto en desorden, dificultando su análisis automático.

La función `maximum()` agrega las palabras a medida que las recibe. Por lo tanto, para que realmente simule un comportamiento real, depende de que el simulador provea un grado de concurrencia aceptable a nivel nodos. Comprobamos que éste mejora notablemente si en `maximum()` no se ejecuta `trabajarArduamente()`.

Aclaración: como los tests están diseñados en función de los logs y la herramienta diff, puede que su correctitud dependa del output del simulador. Esporádicamente podrían fallar los tests, porque la salida a veces se escribe en un orden que no es el esperado. En estos casos, será necesario evaluar manualmente la correctitud mirando el log creado por el test.