DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 3

Algoritmos en sistemas distribuidos

Primer Cuatrimestre de 2017 — Sistemas Operativos

Integrante	LU	Correo electrónico
Langberg, Andrés	249/14	andreslangberg@gmail.com
Sticco, Patricio	337/14	patosticco@gmail.com
Cadaval, Matías	345/14	matias.cadaval@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. Informe de correcciones

- Se modificó la forma de realizar envíos entre los nodos, siendo ahora todos estos no bloqueantes. De esta manera, evitamos una espera innecesaria por parte del nodo que emite el mensaje.
- Se modificó la función `addAndInc`. Ahora el nodo consola espera una única respuesta (correspondiente al nodo más veloz) para saber qué nodo será el encargado de realizar la operación.
- Se agilizó el tiempo de respuesta de la función `member` cuando ya se conoce la misma. Si se sabe que la clave está presente. se devuelve el resultado inmediatamente, sin necesidad de esperar a que todos los nodos terminen de buscar la clave (como era antes).
- Se detalló más precisamente la finalidad y los resultados esperados por los tests.
- Se ajustaron algunas explicaciones del informe de acuerdo a los cambios introducidos en la implementación.

2. Introduccion

En el siguiente trabajo práctico, se pide desarrollar un `DistributedHashMap`, una nueva versión del conocido `ConcurrentHashMap` del TP2, donde se busca poner en práctica el conocimiento sobre sistemas distribuidos a través del envío de mensajes para varias de sus funcionalidades.

Se utilizara la interfaz `MPI`, provista por `C++`. La idea es poder ejecutar el cómputo distribuido en varias máquinas, donde cada una de ellas podrá correr más de un proceso o *nodo*.

Entrando en detalle sobre la estructura del sistema, el mismo contará con un *nodo* distinguido (la **consola**), que se encargará de la entrada y salida de los comandos y sus resultados. A continuación, presentamos dichas funcionalidades:

- `load`(list<string> *params*): Toma como parámetro una lista de archivos de texto y carga las palabras de estos archivos en el `DistributedHashMap`. Deberá utilizar un nodo para cada archivo. Si hay más archivos que nodos, deberá enviarle un nuevo archivo a cada nodo que vaya liberándose.

- `addAndInc`(string *key*): Si *key* existe en el `DistributedHashMap`, incrementa su valor. Si no existe, crea el par (*key*, 1) en algún nodo. No tiene por qué ser procesada siempre por el mismo nodo, ni por uno que ya contenga la clave. Deberá enviarse el comando a todos los nodos; el primer nodo que lo tome deberá avisar que lo ha hecho, y el resto no deberá ejecutarlo.

- `member`(string key): Dado un string, determina si el mismo existe o no en el sistema.

- `maximum`(): Busca el string que más veces aparece y su valor. Luego, imprime el resultado por pantalla. La consola tendrá que pedir a cada nodo las palabras allí almacenadas, y deberá poder recibirlas de distintos nodos, sin que necesariamente todas las de un mismo nodo lleguen juntas. El `HashMap` dado provee un iterador (`HashMap::iterator`) que permite obtener una a una las palabras almacenadas. Una vez que la consola tenga el HashMap completo podrá calcular el resultado. Si un mismo nodo utiliza múltiples mensajes para enviar sus palabras a la consola, debe realizar una única llamada a `trabajarArduamente`(), en lugar de una por cada mensaje.

- `quit`(): Avisa a todos los nodos que deben finalizar y liberar sus recursos.

Más adelante en este informe será detallada la implementación de cada uno de estos comandos.

El resto de los *nodos* serán indistinguibles, y cada uno de ellos tendrá su propio `HashMap`, cuya implementación es provista por la cátedra.

3. Desarrollo

A continuación, haremos una breve descripción sobre la implementación de cada una de las funcionalidades previamente mencionadas, enfatizando sobre la manera en la que se realiza el cómputo distribuido, utilizando la interfaz `MPI`.

En cuanto al llamado de cada función, la implementación de cada una de ellas empieza mediante un envío no bloqueante, a través del cual el *nodo* consola le avisa al conjunto de nodos la tarea a realizar. Por su parte, el resto de los nodos reciben de manera bloqueante el mensaje, para luego ejecutar el handler indicado.

La implementación que construimos siempre utiliza envíos no bloqueantes y recepciones bloqueantes. Los envíos son no bloqueantes ya que en ninguna situación es necesario que el nodo emisor espere a que le llegue el mensaje al receptor para poder continuar con su ejecución. Por el otro lado, las recepciones sí son bloqueantes ya que en todos los casos resulta estrictamente necesario el mensaje esperado para poder continuar con la tarea.

Por otra parte, el nodo consola siempre espera a que los nodos que están ejecutando tareas finalicen, antes de devolverle el control al usuario. Decidimos hacerlo de esta manera para evitar que se superpongan operaciones, ocasionando resultados indeseados (por ejemplo si un llamado a `load` y otro a `member` se superponen).

3.1. `load`

En esta función el nodo consola itera sobre la lista de archivos recibida por parámetro y, en cada iteración, le asigna un archivo al primer nodo de la cola de nodos libres. Si la cola está vacía, el nodo consola espera (recibiendo de manera bloqueante) a que un nodo le avise mediante un envío no bloqueante, que finalizó de procesar un archivo, y que por ende ya está disponible. Una vez recibido el mensaje, el nuevo nodo libre es encolado en la cola previamente mencionada.

Ya asignados todos los archivos, el nodo consola espera a que todos los nodos le informen, que terminaron la ejecución de la tarea de carga que tenían asignada. Cuando todos terminan, se le indica al usuario que el load fue realizado con éxito.

3.2. `addAndInc`

Para realizar un `addAndInc` en nuestro `DistributedHashMap`, necesitamos de alguna manera "elegir" el nodo en el cual vamos a agregar la nueva clave, de manera no determinística. Por lo tanto, en primer lugar el nodo consola recibe de manera bloqueante un único mensaje proveniente de cualquiera de los otros nodos. El mensaje enviado, es el *rank* del nodo emisor. Entonces, el primer mensaje en llegar, contendrá el *rank* del nodo en el que se agregará la clave.

A este nodo, se le enviará la clave mediante un envío no bloqueante y el mismo la agregará en su `HashMap` local. Finalmente, se le avisa a la consola que ya se terminó con el agregado.

Con respecto al resto de los nodos, se les comunica que no van a tener que incorporar la clave a su diccionario.

3.3. `member`

Como previamente mencionamos, esta función se encarga de indicarle al usuario si la clave dada como parámetro esta definida o no en el `DistributedHashMap`. Para ello, el nodo consola les envía mediante *broadcasting* (usando `MPI_Bcast`) la clave a buscar, a todos los nodos.

Por su parte, los nodos reciben la clave y verifican si está definida en su `HashMap` mediante la función `member`. Luego, le envían el resultado al nodo consola mediante un mensaje no

bloqueante.

Mientras va recibiendo las respuestas de manera bloqueante (ya que el dato es necesario para continuar la ejecución), el nodo consola aplica sucesivamente la operación lógica `OR` entre el resultado guardado hasta el momento, y el nuevo resultado proveniente de algún nodo. En caso de que el resultado ya quede determinado (es decir, si ya sabemos que la clave está incluida), inmediatamente se imprime el resultado. Caso contrario no imprime nada y sigue recibiendo mensajes de los nodos. Una vez recibidos los mensajes de todos los nodos, verifica si aún no se imprimió el resultado (si no se imprimió, entonces no está presente la clave), y lo hace de ser necesario.

3.4. `maximum`

Inicialmente el nodo consola crea un `HashMap`, en el que irá agregando las claves (y sus repeticiones) pertenecientes a los `HashMaps` de cada uno de los nodos. Para lograrlo, itera indefinidamente en un ciclo, hasta que todos los nodos le informan que ya enviaron todas las palabras de sus respectivos `HashMaps`. En cada iteración, consola recibe de manera bloqueante, un `string` proveniente de alguno de los nodos. Si el `string` es *"fin de hashmap"*, consola incrementa en uno una variable local, en la que almacena la cantidad de nodos que ya enviaron todas sus claves (esta sirve para determinar la condición de terminación del ciclo). Caso contrario, el nodo consola agrega mediante `addAndInc` el `string` recibido, a su propio `HashMap`.

Una vez que todos los nodos ya enviaron el mensaje de finalización, el ciclo finaliza, y el nodo consola busca el máximo en su `HashMap`, mediante la operación `maximum`. Finalmente le informa al usuario el resultado arrojado por dicha función.

3.5. `quit`

Esta operación es muy simple. Cuando los nodos reciben el mensaje de `quit` salen del ciclo en el que están iterando continuamente a la espera de nuevas tareas, mediante un `break`.

El nodo consola sólo le avisa a los nodos que finalicen, y termina.

4. Tests

Para probar el correcto funcionamiento de nuestra implementación para la clase `DistributedHashMap` realizamos una serie de tests que ejecutan las operaciones ofrecidas por la misma, en diferentes situaciones. Todos los tests disponibles están definidos en el archivo *Makefile*. Para entender más fácilmente su funcionamiento y finalidad, a continuación presentamos una breve explicación de cada uno de ellos. En algunos casos, adjuntaremos los comandos que ejecutan y los explicaremos uno a uno, para entender cada paso.

Además, para realizar estos tests agregamos el comando `print`, para fácilmente poder imprimir por stdout el contenido del `DistributedHashMap`.

4.1. Modo de uso

Antes de ejecutar cualquiera de los tests se debe compilar el código mediante `make`. Por su parte, los tests se puede ejecutar de varias formas:

- Todos: `make test-all` ejecuta todos los tests, uno tras otro.
- Por función: `make test-load` ejecuta todos los tests para la función `load`, uno tras otro. Es análogo para `addAndInc`, `member` y `maximum`.
- Test particular: `make <nombre-test>`. Ejemplo: `make test-maximum-3`

4.2. Tests para la función `load`

Presentamos tres tests en los que probamos la función `load` con distintas cantidades de archivos y nodos. Para todos ellos se espera que la función `print` imprima los mismos valores que contiene el archivo *corpus-post*, que fue generado mediante otros programas, y sabemos que es correcto. Para cada test, este archivo contiene todas las palabras (con sus correspondientes cantidades de apariciones) incluidas en los archivos de claves que se le dan al `load`.

- `test-load-1`: invoca el comando `load` con una cantidad de archivos menor que la cantidad de nodos trabajadores (2 archivos, 3 nodos). Luego se llama a `print` y se compara el output con el resultado esperado.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 4 ./dist_hashmap
  load corpus-0 corpus-1
  print
  q
  ```

- `test-load-2`: invoca el comando `load` con una cantidad de archivos mayor que la cantidad de nodos trabajadores (5 archivos, 2 nodos). Luego se llama a `print` y se compara el output con el resultado esperado.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 3 ./dist_hashmap
  load corpus-0 corpus-1 corpus-2 corpus-3 corpus-4
  print
  q
  ```

- `test-load-3`: invoca el comando `load` con una cantidad de archivos igual que la cantidad de nodos trabajadores (4 archivos, 4 nodos). Luego se llama a `print` y se compara el output con el resultado esperado.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 5 ./dist_hashmap
  load corpus-0 corpus-1 corpus-2 corpus-3
  print
  q
  ```

Explicación paso a paso de `test-load-2`:

```
test-load-2:
  ########## test-load-2 ##########

  # Genera el archivo de salida que es esperado como resultado, para luego compararlo
  awk -f corpus.awk corpus | sort > corpus-post

  # Genera archivos de entrada usados para el load a partir de corpus (divide corpus en 5)
  for i in 0 1 2 3 4; do sed -n "$$((i * 500 + 1)),$$(((i + 1) * 500))p"
  corpus > corpus-"$$i"; done

  # Genera lista de comandos que recibe el nodo consola y los guarda en input
  printf " load corpus-0 corpus-1 corpus-2 corpus-3 corpus-4 \n print \n q \n" > input

  # Ejecuta el codigo con la lista de comandos guardada en input y luego procesa el output mediante
  # sed y sort, para compararlo contra corpus-post usando diff. El awk es para mostrar un mensaje
  # de Test OK si es exitoso, o Test FAILED en caso contrario
  cat input | mpiexec -np 3 ./dist_hashmap | sed -e '1,/finalizado/d' | sed '/>/d' | sort
  | diff -u - corpus-post | awk '{ print } END { if (NR) print "Test FAILED"; else print "Test OK" }'

  # Elimina todos los archivos generados
  rm -f input corpus-post corpus-[0-4]
```

4.3. Tests para la función `addAndInc`

Presentamos tres tests en los que probamos la función `addAndInc` en distintas circunstancias.

- `test-addAndInc-1`: invoca el comando `addAndInc primerstring` sobre un `DistributedHashMap` vacío (np = 3). Luego se llama a `print` y se compara el output con el resultado esperado. La idea es ver cómo se comporta `addAndInc` cuando tiene que definir una nueva clave sobre un `DistributedHashMap` vacío.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 3 ./dist_hashmap
  addAndInc primerstring
  print
  q
  ```

  Salida esperada: el comando `print` debe imprimir la única clave del `DistributedHashMap`, que es *primerstring*, con una sola aparición.

- `test-addAndInc-2`: invoca los comandos `addAndInc nuevapalabrauno` y `addAndInc nuevapalabrados` sobre un `DistributedHashMap` con los archivos *corpus-0* y *corpus-1* cargados (np = 3). Luego se llama a `print` y se compara el output con el resultado esperado. Con este test buscamos ver si `addAndInc` funciona correctamente cuando se quieren definir nuevas claves sobre un `DistributedHashMap` que ya tiene definidas muchas otras claves.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 3 ./dist_hashmap
  load corpus-0 corpus-1
  ```

```
addAndInc nuevapalabrauno
addAndInc nuevapalabrados
print
q
```

Salida esperada: el comando `print` debe imprimir todas claves aportadas por los archivos *corpus-0* y *corpus-1* con sus correspondientes cantidades de apariciones, y además las claves *nuevapalabrauno* y *nuevapalabrados*, ambas con una única aparición.

- `test-addAndInc-3`: invoca los comandos `addAndInc archivo`, `addAndInc estabilidad` y `addAndInc funcional` sobre un `DistributedHashMap` con el archivo *corpus* cargado (np = 3). Luego se llama a `print` y se compara el output con el resultado esperado. Con este test buscamos ver si `addAndInc` funciona correctamente cuando se quieren incrementar claves ya definidas sobre un `DistributedHashMap` que ya tiene definidas muchas claves.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 3 ./dist_hashmap
  load corpus
  addAndInc archivo
  addAndInc estabilidad
  addAndInc funcional
  print
  q
  ```

  Salida esperada: el comando `print` debe imprimir todas claves aportadas por los archivo *corpus* con sus correspondientes cantidades de apariciones, con la excepción de las claves *archivo*, *estabilidad* y *funcional*, que deben tener una aparición adicional.

Explicación paso a paso de `test-addAndInc-2`:

```
test-addAndInc-2:
  ########## test-addAndInc-2 ##########

  # Genera lista de comandos que recibe el nodo consola y los guarda en input
  printf " load corpus-0 corpus-1 \n addAndInc nuevapalabrauno \n addAndInc nuevapalabrados
  \n print \n q \n" > input

  # Le agrega a corpus las dos nuevas claves y lo guarda en corpus-aux
  cat corpus > corpus-aux && printf "\nnuevapalabrauno\nnuevapalabrados" >> corpus-aux

  # Genera el archivo de salida que es esperado como resultado, para luego compararlo
  awk -f corpus.awk corpus-aux | sort > corpus-post

  # Genera archivos de entrada usados para el load a partir de corpus (divide corpus en 2)
  for i in 0 1; do sed -n "$$((i * 1250 + 1)),$$(((i + 1) * 1250))p" corpus > corpus-"$$i"; done

  # Ejecuta el codigo con la lista de comandos guardada en input y luego procesa el output mediante
  # sed y sort, para compararlo contra corpus-post usando diff. El awk es para mostrar un mensaje
  # de Test OK si es exitoso, o Test FAILED en caso contrario
  cat input | mpiexec -np 3 ./dist_hashmap | sed -e '1,/nuevapalabrados/d' | sed '/>/d' | sort
  | diff -u - corpus-post | awk '{ print } END { if (NR) print "Test FAILED"; else print "Test OK" }'

  # Elimina todos los archivos generados
  rm -f input corpus-aux corpus-post corpus-[0-1]
```

4.4. Tests para la función `member`

Presentamos cuatro tests en los que probamos el funcionamiento de `member` en distintas circunstancias.

- `test-member-1`: invoca los comandos `member noestoy` y `member yotampocoestoy` (no definidas) sobre un `DistributedHashMap` vacío (np = 3). Luego se filtra el output y se comparan los resultados arrojados por la función contra los resultados esperados. Con este test buscamos ver si `member` funciona correctamente cuando se quiere saber si están definidas claves ausentes sobre un `DistributedHashMap` vacío.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 3 ./dist_hashmap
  member noestoy
  member yotampocoestoy
  q
  ```

  Salida esperada: el output resultante de las llamadas a `member` debe ser:
  La clave noestoy NO ESTA DEFINIDA
  La clave yotampocoestoy NO ESTA DEFINIDA

- `test-member-2`: invoca los comandos `member noestoy` y `member yotampocoestoy` (no definidas) sobre un `DistributedHashMap` con los archivos *corpus-0*, *corpus-1*, *corpus-2* y *corpus-3* cargados (np = 5). Luego se filtra el output y se comparan los resultados arrojados por la función contra los resultados esperados. Con este test buscamos ver si `member` funciona correctamente cuando se quiere saber si están definidas claves ausentes sobre un `DistributedHashMap` que ya tiene definidas muchas claves, y distribuidas en distintos nodos.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 5 ./dist_hashmap
  load load corpus-0 corpus-1 corpus-2 corpus-3
  member noestoy
  member yotampocoestoy
  q
  ```

  Salida esperada: el output resultante de las llamadas a `member` debe ser:
  La clave noestoy NO ESTA DEFINIDA
  La clave yotampocoestoy NO ESTA DEFINIDA

- `test-member-3`: invoca los comandos `member archivo` y `member estabilidad` (sí definidas) sobre un `DistributedHashMap` con los archivos *corpus-0*, *corpus-1*, *corpus-2* y *corpus-3* cargados (np = 5). Luego se filtra el output y se comparan los resultados arrojados por la función contra los resultados esperados. Con este test buscamos ver si `member` funciona correctamente cuando se quiere saber si están definidas claves presentes sobre un `DistributedHashMap` que ya tiene definidas muchas claves, y distribuidas en distintos nodos.

Comandos que se ejecutan (entrada):

```
mpiexec -np 5 ./dist_hashmap
load load corpus-0 corpus-1 corpus-2 corpus-3
member archivo
member estabilidad
q
```

Salida esperada: el output resultante de las llamadas a `member` debe ser:
La clave archivo ESTA DEFINIDA
La clave estabilidad ESTA DEFINIDA

- `test-member-4`: invoca el comando `member nuevaclave`, luego `addAndInc nuevaclave` y `member nuevaclave` (primero no definida, luego sí) nuevamente, sobre un `DistributedHashMap` con el archivo *corpus* cargado (np = 2). Luego se filtra el output y se comparan los resultados arrojados por la función contra los resultados esperados. Con este test buscamos ver si `member` funciona correctamente antes y después de hacer un llamado a `addAndInc` para una clave que en principio no está definida.
  Comandos que se ejecutan (entrada):

```
mpiexec -np 3 ./dist_hashmap
load corpus
member nuevaclave
addAndInc nuevaclave
member nuevaclave
q
```

Salida esperada: el output resultante de las llamadas a `member` debe ser:
La clave nuevaclave NO ESTA DEFINIDA
La clave nuevaclave ESTA DEFINIDA

Explicación paso a paso de `test-member-4`:

```
test-member-4:
  ########## test-member-4 ##########

  # Genera lista de comandos que recibe el nodo consola y los guarda en input
  printf " load corpus \n member nuevaclave \n addAndInc nuevaclave \n member nuevaclave \n
  q \n" > input

  # Genera el archivo de salida que es esperado como resultado, para luego compararlo
  printf "La clave nuevaclave NO ESTA DEFINIDA\nLa clave nuevaclave ESTA DEFINIDA\n" > output

  # Ejecuta el codigo con la lista de comandos guardada en input y luego procesa el output mediante
  # sed y grep, para compararlo contra ouput usando diff. El awk es para mostrar un mensaje
  # de Test OK si es exitoso, o Test FAILED en caso contrario
  cat input | mpiexec -np 2 ./dist_hashmap | grep DEFINIDA | sed 's/> //' | diff -u - output
  | awk '{ print } END { if (NR) print "Test FAILED"; else print "Test OK" }'

  # Elimina todos los archivos generados
  rm -f input output
```

4.5. Tests para la función `maximum`

Presentamos tres tests en los que probamos el funcionamiento de `maximum`.

- `test-maximum-1`: invoca el comando `maximum` sobre un `DistributedHashMap` con el archivo *corpus-max* cargado (np = 3). Luego se filtra el output y se compara el resultado arrojado por la función contra el resultado esperado. Con este test buscamos ver si `maximum` funciona correctamente sobre un `DistributedHashMap` que ya tiene definidas muchas claves, todas en un único nodo.
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 3 ./dist_hashmap
  load corpus-max
  maximum
  q
  ```

  Salida esperada: el output resultante de la llamada a `maximum` debe ser:
  El maximo es <clavemaxima, 9>

- `test-maximum-2`: invoca el comando `maximum` sobre un `DistributedHashMap` con los archivos *corpus-max-0*, *corpus-max-1*, *corpus-max-2*, *corpus-max-3* y *corpus-max-4* cargados (np = 6). Luego se filtra el output y se compara el resultado arrojado por la función contra el resultado esperado. Con este test buscamos ver si `maximum` funciona correctamente sobre un `DistributedHashMap` que ya tiene definidas muchas claves, y distribuidas en distintos nodos. Además, *clavemaxima* no es máximo en ninguno de los archivos localmente, pero si globalmente (podemos comprobarlo cargando los archivos separados y haciendo `maximum`).
  Comandos que se ejecutan (entrada):

  ```
  mpiexec -np 6 ./dist_hashmap
  load corpus-max-0 corpus-max-1 corpus-max-2 corpus-max-3 corpus-max-4
  maximum
  q
  ```

  Salida esperada: el output resultante de la llamada a `maximum` debe ser:
  El maximo es <clavemaxima, 9>

- `test-maximum-3`: invoca el comando `maximum` sobre un `DistributedHashMap` con el archivo *corpus-max* cargado (np = 3). Hecho esto, ejecuta diez veces seguidas `addAndInc nuevomax`, y nuevamente ejecuta `maximum`. Luego se filtra el output y se compara el resultado arrojado por la función contra el resultado esperado. Con este test buscamos ver si `maximum` funciona correctamente luego de hacer múltiples llamados a `addAndInc` para una misma clave, de manera que se convierta en la nueva clave máxima.

Comandos que se ejecutan (entrada):

```
mpiexec -np 3 ./dist_hashmap
load corpus-max
maximum
addAndInc nuevomax
...
addAndInc nuevomax
maximum
q
```

Salida esperada: el output resultante de las llamadas a `maximum` debe ser:
El maximo es <clavemaxima, 9>
El maximo es <nuevomax, 10>

Explicación paso a paso de `test-maximum-3`:

```
test-maximum-3:
  ########## test-maximum-3 ##########

  # Genera lista de comandos que recibe el nodo consola y los guarda en input
  printf " load corpus-max \n maximum \n" > input && for i in 0 1 2 3 4 5 6 7 8 9;
  do printf " addAndInc nuevomax \n" >> input; done && printf " maximum \n q \n" >> input

  # Genera el archivo de salida que es esperado como resultado, para luego compararlo
  printf "El maximo es <clavemaxima, 9>\nEl maximo es <nuevomax, 10>\n" > output

  # Ejecuta el codigo con la lista de comandos guardada en input y luego procesa el output mediante
  # sed y grep, para compararlo contra ouput usando diff. El awk es para mostrar un mensaje
  # de Test OK si es exitoso, o Test FAILED en caso contrario
  cat input | mpiexec -np 3 ./dist_hashmap | grep maximo | sed 's/> //' | diff -u - output
  | awk '{ print } END { if (NR) print "Test FAILED"; else print "Test OK" }'

  # Elimina todos los archivos generados
  rm -f input output
```