**Universidad ORT Uruguay**
**Facultad de Ingeniería**

# Framework basado en Map-Reduce utilizando x10

**Entregado como requisito para la obtención del título de Licenciado en Ingeniería de Software**

**Herman Persitz - 181084**

**Tutor: Sergio Yovine**

**2016**

Yo, Herman Persitz, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Puedo asegurar que:

- La obra fue producida en su totalidad mientras realizaba el Proyecto;
- Cuando he consultado el trabajo publicado por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros, he indicado las fuentes. Con excepción de estas citas, la obra es enteramente mía;
- En la obra, he acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué fue contribuido por otros, y qué fue contribuido por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Herman Persitz

**11-04-2016**

# Abstract

En los últimos años, el incremento en la *performance single-thread* disminuyo fuertemente. Actualmente la mayor ganancia en performance de los procesadores se encuentra en el uso de múltiples núcleos. De aquí surge la necesidad de aprovechar múltiples núcleos y nodos de procesamiento mediante aplicaciones paralelas y distribuidas.

Con el objetivo de investigar en este campo, se crea un *framework*, el que sigue el modelo de programación paralela *Map-Reduce*. Este pretende facilitar el desarrollo de aplicaciones que utilizan este modelo. Minimizando el esfuerzo que el usuario desarrollador debe dedicar al paralelismo y permitiendo centrarse en la "lógica de negocio".

Para esto se utiliza el lenguaje de programación x10. Este lenguaje orientado a objetos y funcional fue pensado con el foco en ser utilizado para aplicaciones paralelas y distribuidas, dándole acceso al desarrollado a potentes primitivas en este aspecto.

Como aplicaciones para probar el *framework*, se utiliza la búsqueda de sub string repetidos en un string y el algoritmo de generación de arboles de decisión c4.5.

Realizando comparaciones con implementaciones de referencia se logra validar un buen rendimiento y bajo costo de desarrollo para utilizar el *framework*, incluso cuando la aplicación no se presta para el modelo *Map-Reduce*.

# Palabras clave

x10; parallelismo; map-reduce; work-pile; c4.5;

# Índice general

# 1 Introducción

## 1.1. Contexto

Hasta hace algo más de 10 años, el incremento de la performance para un solo hilo de procesamiento superaba el 50 % anual. Tanto el aumento del número de transistores y de la frecuencia de operación impulsaban tales mejoras.

A consecuencia de esto, cada aproximadamente 2 años, la potencia de los procesadores se duplicaba, esto se transfería a que las aplicaciones duplicaban su rendimiento, sin necesidad de realizar ninguna modificación a las mismas.

A partir de 2004, esta tendencia cambió, pasando a mejorar la performance 20 % por año. Esto se debe a múltiples barreras que se encontraron en el desarrollo de los procesadores. Por un lado la velocidad de la memoria quedo atrás frente a la de los procesadores (*memory wall*). Por otro, el crecimiento de la dificultad para agregar más paralelismo a nivel de instrucción (*ILP wall*)[1]. Adicionalmente la frecuencia de los procesadores dejo de aumentar (*power wall*)[2].

Estos elementos causaron que la industria cambie el rumbo hacia a la utilización de arquitecturas de procesadores con múltiples núcleos. En consecuencia ya no ocurre que en un procesador más nuevo un software automáticamente corra mucho más rápido. Ahora es necesario realizar diseños de software que aprovechen los múltiples procesadores.

Adicionalmente para muchas aplicaciones, la potencia de un solo nodo de procesamiento no es ni remotamente suficiente, por lo que se utilizan *clusters*, los cuales distribuyen el procesamiento en múltiples nodos. Surge en consecuencia la necesidad de crear aplicaciones que puedan correr en múltiples nodos, de forma coordinada.

En este escenario vemos surgir lenguajes con un alto nivel de abstracción como x10[3], Chapel[4] y Julia[5] diseñados específicamente con el paralelismo en mente.

## 1.2. Objetivos

Siendo estos lenguajes tan nuevos, hay poca experiencia en el uso de ellos. Por estas razones se plantea experimentar en el uso de x10, un lenguaje diseñado con el paralelismo en mente.

Luego, utilizando las posibilidades que este ofrece, realizar una plataforma de programación paralela basada en los patrones *map-reduce* y *work-pile*. El diseño de esta estará focalizado en permitirle al desarrollador sacar provecho del paralelismo con la menor inversión posible de tiempo, abstrayendo muchas de las complejidades inherentes de la programación paralela y del lenguaje x10.

## 1.3. *Map-Reduce* y *Work-Pile*

*Map-reduce* es un modelo de programación paralela y distribuida[6]. Este cuenta con dos funciones principales que deben ser implementadas por el usuario: *Map* y *Reduce*.

La función *Map* que a partir de una entrada genera una o mas salidas. Luego se aplica *Reduce* sobre un conjunto de estas salidas generando un resultado agregado. Este resultado agregado es a su ves procesado por la función *reduce* nuevamente junto con otros resultados.

Las firmas de estas funciones son de la forma:

- `map(t:T):R[]`
- `reduce(rs:R[]):R[]`

*Work-Pile* es un patrón similar a *Map-Reduce*, pero agrega la posibilidad de al ejecutar la función *map*, se generen nuevas tareas[7].

## 1.4. Estructura del documento

En el capítulo 2, se comienza dando una idea general sobre las características de x10, exponiendo lo que lo hace diferente de otros lenguajes: sus primitivas de paralelismo y distribución. Para esto se utiliza el problema de sub string repetidos en un string como ejemplo.

Luego en el capítulo 3 se expone una primera implementación del *framework*. En el capítulo 4 se estudia el caso de su aplicación al problema utilizado como ejemplo en el capítulo 2.

En el capítulo 5 se implementa otro caso de estudio, el algoritmo c4.5. Este caso es retomado en el capítulo 6, en el cual se analizan diferentes propuestas de mejora al *framework*, junto con sus resultados.

# 2 Descripción del Lenguaje

## 2.1. Características

x10 es un lenguaje de programación orientado a objetos y funcional[8], diseñado específicamente para la computación paralela, utilizando el modelo APGAS. Este lenguaje es desarrollado por IBM bajo la licencia *Eclipse Public License*[9].

El modelo de programación Paralela APGAS *(asynchronous partitioned global address space)* tiene como elementos clave el asincronismo y el concepto de *place*. Un *place* representa un nodo de procesamiento, que tiene memoria y puede ejecutar múltiples actividades concurrentemente[10]. En la práctica un *place* se traduce generalmente a una computadora (virtual o real) que forma parte del grupo de las computadoras en las que se ejecuta.

## 2.2. Primitivas de paralelismo y distribución

### 2.2.1. Async y Finish

x10 cuenta con 2 primitivas que abstraen el uso de *threads*:

1. `Async`
2. `Finish`

#### Async

Esta instrucción indica que no es necesario esperar a ejecutar el código que encierra para continuar. Por lo que se ejecuta el contenido de la misma en simultaneo

con el código después de lo encerrado (por el `async`).

El *runtime* decide cómo se ejecuta, abstrayendo la creación o reutilización de *threads*. Los `async` pueden componerse con otros `async`. Una vez que se lanza una `async`, este no puede ser abortado.

Dentro de un `async` se puede acceder a todas las variables del *scope* local que sean definidas con `val` (lo que hace la referencia inmutable). Las variables a las que se les puede modificar la referencia son declaradas con `var`. Adicionalmente también se puede acceder a variables de objetos a los que se tienen acceso sin importar si estas fueron declaradas como `var` o `val`.[11]

### Finish

El `finish` indica que se debe esperar a todas las llamadas `async` creadas dentro del mismo, sin importar si son realizadas directamente en el cuerpo del método, dentro de métodos llamados desde este o dentro de otros `asyncs`. Es decir, un `finish` espera a que termine de ejecutarse todo su contenido.

### Ejemplo

En la figura 2.1 se ve un ejemplo de la aplicación de `finish` y `async`.

Debido a que la variable del `for i` fue declarada como `var` (no puede ser declarada `val` dado que su valor cambia), es necesario copiarla en una variable temporal declarada como `val` (de lo contrario se tienen errores de compilación).

Esto también funciona como protección ante errores de programación, no permitiendo acceder a varios hilos a variables locales.

`Operacion1` y `Operacion2` reciben como parámetro un `Long`.

```
1  public static def main(args:Rail[String]):void {
2    finish {
3      for (var i:Long = 0; i<3;i++){
4        val iTemp=i;
5        async{
6          Operacion1(iTemp);
7        }
8        async{
9          Operacion2(iTemp);
10       }
11     }
12   }
13   Operacion3();
14 }
15
16 public static def Operacion1(i:Long){
17   Console.OUT.println("Operacion1 "+i);
18 }
19
20 public static def Operacion2(i:Long){
21   Console.OUT.println("Operacion2 "+i);
22 }
23
24 public static def Operacion3(){
25   Console.OUT.println("Operacion3 ");
26 }
27
```

Figura 2.1: Ejemplo de utilización de Async y Finish

En el ejemplo se ejecuta primero lo que ésta dentro del finish. Por cada iteración del `for` se lanzan las tareas `Operacion1` y `Operacion2`. Las iteraciones del `for` no esperan a que terminen las operaciones lanzadas. Antes de ejecutar `Operacion3` se ejecuta todo lo que esta dentro del `finish`.

Al ejecutar, la salida es no determinista, un ejemplo de esta:

```
Operacion1 0
Operacion2 1
Operacion2 2
Operacion1 2
Operacion2 0
Operacion1 1
Operacion3
```

### 2.2.2. At y Place

x10 cuenta con una primitiva muy potente, el `at`. Esta permite cambiar el nodo (*place*) donde se está ejecutando. A diferencia de `async`, donde el mapeo a *thread* es realizado por el *runtime*, se debe elegir el nodo donde se desea continuar la ejecución. Esta instrucción no realiza cambios en cuanto al paralelismo. Es decir, en cuanto a concurrencia, se trata al código dentro del at como si estuviera ejecutándose dentro del mismo nodo.

En la figura 2.2 se puede ver un ejemplo simple con la sintaxis de `at`. El `at` recibe un `Place` que indica el lugar se va a continuar ejecutando. Existen múltiples formas de recuperar el objecto *place*, en este caso se recupera a partir del número de *place*. También puede ser utilizado `here` el que devuelve el place actual. En el caso de la figura 2.2, `here` es igual a `Place(1)`.

```
1  at (Place(0)){
2    Console.OUT.println("Printing in place 0");
3  }
4  at (Place(1)){
5    Console.OUT.println("Printint in place 1");
6  }
7  at (Place(1)){
8    Console.OUT.println("Printint in " + here);
9  }
10
```

Figura 2.2: Ejemplo de utilización `at`

El `at` captura a todas las variables utilizadas dentro del mismo y serializa el sub-grafo de objetos que son accesibles a partir de estas. Como restricción se tiene que no es posible referenciar variables del *scope* local que hayan sido declaradas utilizando `var`.

En la figura 2.3 vemos un ejemplo sencillo de cambio de nodo de ejecución utilizando `at` con serialización de objetos. Cabe resaltar que en caso que el `at` capturara objetos no inmutables (a diferencia del string), si se realiza una modificación dentro del `at`, esta no se verá reflejada fuera del mismo, dado que son instancias diferentes y no se realiza ningún tipo de sincronización al finalizar el `at`.

Dado que si se modifica un objeto dentro de un `at` el original no se ve afectado, existen patrones para "retornar" datos desde un `at`. Estos utilizan referencias globales, objetos que mantienen una referencia a un objeto determinado solo accesible dentro de su *place* original. Estas referencias pueden atravesar los `at` sin que

```
1 var str1:String = "var can be referenced inside at? False";
2 val str2 = "val can be referenced inside at? ";
3 at (Place(0)){
4   Console.OUT.println(str2 + "true");
5 }
6
```

Figura 2.3: Ejemplo de utilización `at` con serialización

sea necesario serializar y des serializar el objeto al cual tienen referencia. Podemos ver un ejemplo de esto en la figura 2.4. La ejecución de la misma tiene como salida "Hello". Para este ejemplo se crea una clase `StringWrapper` para poder "modificar" el string, dado que estos son inmutables.

```
1  var s:StringWraper = new StringWrapper();
2  val g:GlobalRef[StringWrapper]=new GlobalRef[StringWrapper](s);
3  at (Place(1)){
4    val h = "Hello";
5    at (Place(0)){
6      g().string = h;
7    }
8  }
9  Console.OUT.print(g().string);
10
```

Figura 2.4: Ejemplo de utilización `at` con `GlobalRef`

Al utilizar el `at` se deben tener algunas consideraciones, de forma de evitar la serialización excesiva de objetos, lo que puede tener un gran impacto en la performance. También se debe favorecer "pocas llamadas largas" sobre "muchas llamas cortas" dado el overhead que genera este cambio de contexto.

## 2.3. Ambiente de medición de rendimiento

### 2.3.1. Características Generales

A modo de poder variar fácilmente la cantidad de núcleos y simular el funcionamiento distribuido fueron utilizadas maquinas virtuales. Estas corren linux, más específicamente Debian.

Para configurar las mismas se utilizo Vagrant[12], una herramienta que permite definir las características de las maquinas junto con los programas a instalarse y automatizar el proceso de creación.

### 2.3.2. Características del *Host*

A continuación se detallan las características del equipo host:

| | |
|---|---|
| OS | Windows 10 |
| CPU | Intel 3770k 4 núcleos (8 virtuales) |
| RAM | 32GB |

### 2.3.3. Características del *Guest*

Dependiendo de la prueba a realizar se utilizaron diferentes configuraciones de CPU y RAM. En todas las pruebas se configuró para que los núcleos virtuales puedan usar hasta un máximo de 80 % del tiempo de ejecución de los reales. Esto se realizó para disminuir el efecto de los programas ejecutando en la máquina host durante las pruebas.

Se utilizó VirtualBox como software de virtualización. Cuando fue necesario hacer pruebas distribuidas con múltiples máquinas virtuales se utilizó el mismo *script* de creación vagrant.

| | |
|---|---|
| OS | Debian 7.9 |
| CPU | variable en cada prueba |
| RAM | variable en cada prueba |
| Network | Gigabit |

## 2.4. Implementación SubStringFinder

### 2.4.1. SubStringFinder

A modo de probar las primitivas de x10, se realiza una implementación de un algoritmo de búsqueda de sub strings repetidos en un string.

### 2.4.2. Implementación

#### 2.4.2.1. Implementación Secuencial

En la figura 2.5 se puede ver la implementación secuencial del algoritmo. Como parámetro se recibe una *array* con el string donde se debe buscar. Se itera con diferentes largos (primer `for`), buscando si cada substring del largo en cuestión (segundo `for`) se repite (`searchSubRep`). Para esto se utiliza un rolling hash y en caso de que de igual, se chequea la igualdad.

El string de entrada es generado de forma *random* (pero utilizando siempre la misma *seed*, para obtener resultados repetibles). En la figura 2.6 podemos ver los tiempos de ejecución de este algoritmo para diferentes largos de entradas. Como es de esperar, vemos que el tiempo aumenta de forma exponencial con respecto al tamaño de la entrada.

```
1  private static def searchSec(arr:Array_1[Char]){
2    for (var length:Long = arr.size-1; length>1;length--){
3      for (start in 0..(arr.size-length)){
4        if (searchSubRep(arr,start, length)){
5          Console.OUT.println("largo: " + length + " inicio " + start + " " +
       SubArray(arr,start, length));
6        }
7      }
8    }
9  }
10
```

Figura 2.5: Implementación Secuencial de búsqueda de sub strings repetidos en un string

#### 2.4.2.2. Implementación Paralela

Este algoritmo de búsqueda presenta grandes oportunidades de paralelismo. Esencialmente todo lo realizado dentro del segundo `for` puede ser realizado de forma paralela. Esto dado que solamente comparten el string a buscar y los parámetros de búsqueda, y no existe dependencia ni requerimiento de orden entre las diferentes ejecuciones de esta sección de código. El string a buscar no es editado, por lo que puede ser utilizado por todos de forma simultánea. En cuanto a los parámetros de búsqueda, estos deben ser copiados de forma sincrónica antes de ejecutar de forma asincrónica.

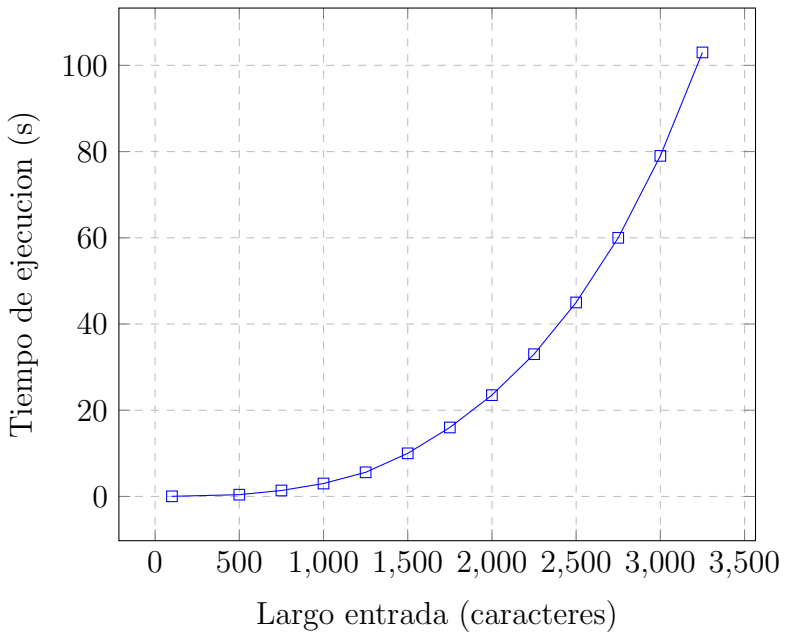


Figura 2.6: Tiempo de ejecución en función del largo de la entrada, algoritmo secuencial

En la figura 2.7 se puede ver una implementación paralela del algoritmo anterior, utilizando las primitivas provistas por x10. Se realizan los siguientes cambios a la implementación secuencial de la figura 2.5.

- Se agrega un `finish` de forma que se terminen de ejecutar todos los `async` lanzados.
- En las lineas 4 y 5 se copian las variables, de no hacerlo no compilaría dado que `length` fue declarada como `var` y no es posible acceder a `var` del *scope* local dentro de los `async`. Si se pudiera acceder habría que copiarlo, de lo contrario se accedería a el valor de las variables al momento de ejecutar y no al valor del momento en el que se lanzó el `async`, que es lo que se quiere.
- se encierra en un `async` la parte donde se hace la búsqueda

En todos los casos el uso del CPU fue superior al 90 % y el de memoria no supera los 700MB. En la figura 2.8 se detalla la mejora en los tiempos de ejecución en función de la cantidad de núcleos, para una entrada de 3000 caracteres de largo y con 6GB de memoria. Para esto se hace el calculo tiempo_secuencial/tiempo_n_nucleos.

Podemos ver que hay una mejora considerable por el uso de paralelismo. Como es de esperar, el porcentaje de mejora aumenta menos cuantos más núcleos se agregan.

```
1  private static def searchPar(arr:Array_1[Char]){
2    finish for (var length:Long = arr.size-1; length>1;length--){
3      for (start in 0..(arr.size-length)){
4        val length1=length;
5        val start1 = start;
6        async {
7          if (searchSubRep(arr,start1, length1)){
8            Console.OUT.println("largo: " + length1 + " inicio " + start1 + " " +
       SubArray(arr,start1, length1));
9          }
10       }
11     }
12   }
13 }
14
```

Figura 2.7: Implementación paralela del algoritmo

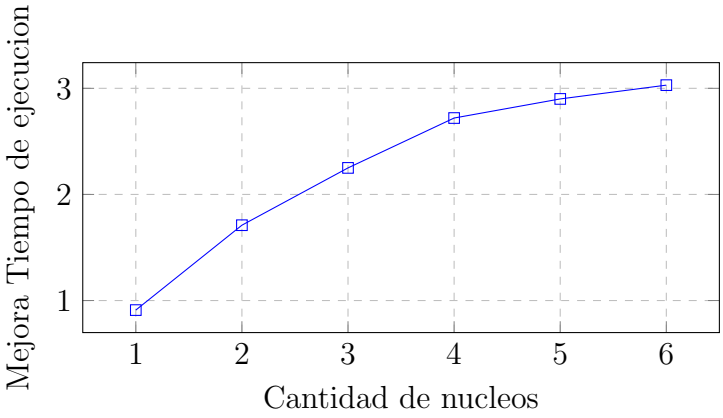


Figura 2.8: Mejora del tiempo de ejecución (comparado con secuencial) en función de la cantidad de núcleos (entrada de largo fijo, 3000 caracteres)

#### 2.4.2.3. Implementación Distribuida

**Primera implementación**

Luego de realizar la implementación paralela, se realiza una implementación distribuida de la misma, realizando pocas modificaciones. Para decidir en que nodo ejecutar se utiliza un criterio muy básico. Se realiza el módulo de la posición de inicio del string buscado con la cantidad de *places*. Si el string no es muy corto este método da una repartición bastante pareja, pero tiene como problema que no considera la diferencia de poder de procesamiento entre diferentes nodos. Esta implementación se encuentra en la figura 2.9.

```
1  private static def searchDist1(arr:Array_1[Char]){
2    finish for (var length:Long = arr.size-1; length>1;length--){
3      for (start in 0..(arr.size-length)){
4        val length1=length;
5        val start1 = start;
6        at (Place(start1%Place.places().size)){
7          async {
8            if (searchSubRep(arr,start1, length1)){
9              Console.OUT.println("largo: " + length1 + " inicio " + start1 + " "
       + SubArray(arr,start1, length1) + " " + here);
10           }
11         }
12       }
13     }
14   }
15 }
16
```

Figura 2.9: Primera implementación distribuida del algoritmo

Se ejecuta de forma distribuida en dos máquinas virtuales, ambas con 3 núcleos, pero con diferente memoria. La maquina que lanza el programa ejecutando el `main` (`Place(0)`) con 15GB y la segunda con 6GB.

La ejecución se completa en 266 segundos, un tiempo varias veces superior a cualquiera de los tiempos encontrados antes (implementación secuencial 79s). También se observa que en la máquina del `main`, se utiliza aproximadamente 10.7GB de memoria (en la otra no supera los 5GB) y en ambas máquinas la utilización del CPU ronda el 60 %.

Al ejecutarlo en una sola máquina (3 núcleos, 15GB de memoria), la ejecución no termina. Luego de más de 400s el programa tira una excepción:

`java.lang.OutOfMemoryError: GC overhead limit exceeded`.

Dentro del stacktrace se ve que el problema ocurre al reservar espacio para serializar. Parece que la causa es que en cada `at` se envía el string (lo que debe

generar el aumento en el tiempo de ejecución).

**Segunda implementación**
Partiendo de las observaciones anteriores, se modifica el método, de forma que utilice la estructura `DistArray`, que representa una *array* en la que el índice por el que se accede es el número de *place*. De esta forma se evita enviar el string en el que se busca antes de cada `at`. Este código se encuentra en la figura 2.10.

A diferencia de una *array* regular, en las `DistArray` no es posible acceder con índices correspondientes a *places* diferentes al que se esta sin hacer primero un `at`. En cuyo caso ya no se está en el *place* original.

```
1  private static def searchDist2(arr:Array_1[Char]){
2    val arrDis = new DistArray_Unique[Array_1[Char]]();
3
4    //se inicializa la array distribuida para cada place
5    for (val p in Place.places()){
6      at (p){
7        arrDis(p.id)=arr;
8      }
9    }
10
11   finish for (var length:Long = arr.size-1; length>1;length--){
12     for (start in 0..(arr.size-length)){
13       val length1=length;
14       val start1 = start;
15
16       at (Place(start1%Place.places().size)){
17         async {
18           if (searchSubRep(arrDis(here.id),start1, length1)){//here se refiere
     al place actual
19             Console.OUT.println("largo: " + length1 + " inicio " + start1 + " "
      + SubArray(arrDis(here.id),start1, length1) + " " + here);
20           }
21         }
22       }
23     }
24   }
25 }
26
```

Figura 2.10: Segunda implementación distribuida del algoritmo

Al correrlo en una máquina con 3 núcleos y 15GB de memoria, la ejecución

finaliza con un tiempo de 222 segundos, utilizando como máximo 8.5GB de memoria.

Cuando se corrió de forma distribuida utilizando una maquina de 3 núcleos con 15GB de memoria, y otra de 3 núcleos con 6 GB de memoria, el tiempo de ejecución fue 202 segundos. Se utilizo como máximo 4.5GB de memoria en la primera máquina y 3GB en la segunda.

### 2.4.3. Problemas

Podemos ver que una implementación naive de este problema no es suficiente para una aplicación distribuida por el tiempo de ejecución muy largo y alto uso de memoria.

Otro problema que surge es la repartición de trabajo entre los diferentes *places*. A diferencia de los `async`, en los que x10 decide cómo realizar el mapeo a *threads*, el `at` requiere que se indique el *place* especifico en el que se quiere ejecutar. Esto implica que el usuario es el responsable de crear mecanismos de repartición del trabajo entre los places.

# 3 Primera Implementación del Framework

## 3.1. Objetivos

El *framework* tiene como objetivo facilitar el uso de x10 para resolver problemas del tipo *Map-Reduce*, permitiendo que quien lo utiliza no tenga que preocuparse por los problemas inherentes al paralelismo, o por lo menos reducirlos al máximo.

De esta forma el usuario del *framework* puede aumentar su productividad implementando solo el código de "lógica de negocios" y no debiendo entrar en detalle sobre el funcionamiento de las primitivas de x10.

Adicionalmente se logra encapsular la lógica y se separa consumidores de productores.

## 3.2. Diferencias con otras implementaciones de *Map-Reduce*

Esta implementación de *Map-Reduce* es muy similar al utilizado en los lenguajes funcionales (donde se origina este patrón).

A diferencia de otras implementaciones de *Map-Reduce*[6] en las cuales los parámetros de las funciones son presentados como un conjunto clave/valor, en esta se representan por un solo objeto.

Adicionalmente, existen 2 funciones de *reduce*: una que reduce el resultado de *map* con un acumulador, retornando un acumulador y otra que reduce entre acumuladores retornando también un acumulador.

## 3.3. Descripción general

El *framework* permite ejecutar tareas definidas por el usuario programador de forma distribuida. Para esto se debe definir una clase que provea las tareas, las cuales son distribuidas por el *framework* entre los *places* disponibles.

## 3.4. Componentes

Los principales componentes del *framework* son:

1. `Manager`: elemento central de *framework*, coordina el accionar de los demás.
2. `TaskDistributor` : entrega tareas.
3. `Context`: contexto en el que se ejecutan las tareas.
4. `PlacesSpec`: indica el lugar ("físico") donde se ejecuta cada tarea.

En la figura 3.1 se puede ver un diagrama de clases simplificado de los principales elementos.

### 3.4.1. PlacesSpec

Esta clase tiene como responsabilidad la repartición de los recursos de procesamiento. Estos son representados por el `struct PlaceThread(placeID:Long, thread:Long)`. Donde el `placeID` representa el `Place` donde va a ejecutarse una tarea y `thread` representa el número de ejecución concurrente de tarea dentro de un `Place`.

Se considera que cada `Place` puede ejecutar un número de tareas concurrentes, generalmente similar (o en algunos casos un poco más) a la cantidad de núcleos que tiene. De esta forma se disminuye el *overhead* de tener muchas tareas ejecutándose de forma paralela. Pese a que esta estructura tiene un `Long thread`, no se controla en qué *thread* específica del `Place` se ejecuta la tarea, esto es decidido por el *runtime* de x10. Este número solo se utiliza para cuantificar la cantidad de tareas en ejecución.

Internamente, esta clase mantiene una cola con los `PlaceThread` que no están siendo utilizados.

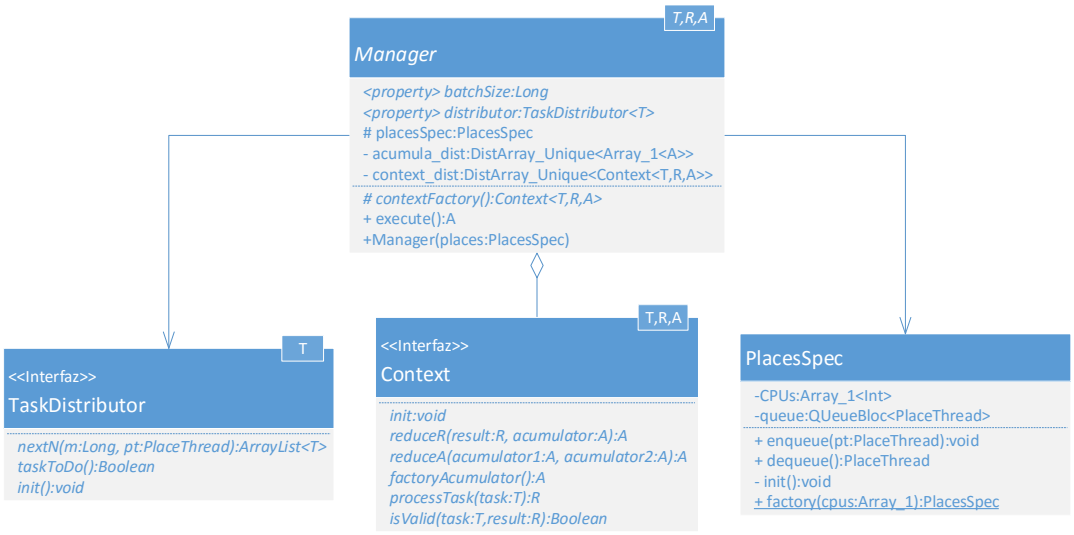


Figura 3.1: Principales componentes del *framework* (algunos atributos y métodos han sido omitidos por claridad)

#### 3.4.1.1. Métodos principales

**static factory(cpus:Array_1[Long]):PlacesSpec**
Recibe como parámetro una *array* de `Long` indexada por `placeId` indicando cuántos núcleos hay en cada `place`. Retorna un `PlacesSpec` con la configuración indicada.

**dequeue():PlaceThread**
Retorna un `PlaceThread` libre, en caso de que no exista uno, queda bloqueando la ejecución.
Este método es llamado antes de ejecutar una tarea.

**enqueue(pt:PlaceThread)**
Encola un `PlaceThread` que ya no está siendo utilizado para hacerlo disponible.
Este método es llamado cuando se finaliza de ejecutar una tarea.

### 3.4.2. Context

Esta interfaz contiene la firma del método que realiza la tarea. El usuario del *framework* debe crear una clase que la implemente. En esta se debe tener

todo lo necesario para ejecutar la tarea (ej: información pesada compartida entre tareas). Los métodos de esta deben ser *thread safe*, o preferentemente no deben modificar el estado de la clase. Se va a crear una instancias de la clase por cada `Place` (compartida por todas sus *threads*), más detalle en la subsección `Manager`. También contienen las funciones utilizadas durante los *reduces*.

Esta interfaz es *template* donde:

- `T`: Tarea
- `R`: Resultado de ejecutar la tarea
- `A`: Acumulador de resultados

#### 3.4.2.1. Métodos principales

**processTask(task:T):R**
Recibe como parámetro una tarea, la ejecuta y devuelve su resultado.

**isValid(task:T, resultado:R):Boolean**
Recibe como parámetro una tarea y el resultado de su ejecución. Retorna `true` si es válida o `false` sino. Es llamado luego de `processTask`. Usado para descartar resultados.

**reduceR(result:R, acumulator:A):A**
Recibe como parámetro el resultado de ejecución de una tarea y un acumulador, agrega el resultado al acumulador y lo retorna.

**reduceA(acumulator1:A, acumulator2:A):A**
Recibe como parámetro dos acumuladores, los combina y lo retorna.

**factoryAcumulator():A**
Retorna un nuevo acumulador.

**init():void**
Inicializa el contexto, es llamado en el `Place` donde este va a residir. Si es necesario leer de archivos debe hacerse en este método.

### 3.4.3. TaskDistributor

Esta interfaz contiene los métodos que devuelven tareas a realizar e indican cuando ya no hay tareas. El usuario del *framework* tiene la opción de implementar su propio `TaskDistributor` o utilizar (o extender) uno de los ya implementados que incluye el *framework*.

La implementación de los métodos de esta interfaz no tiene por qué ser *thread safe*; el *framework* no realiza llamadas concurrentes a estos.

Esta clase es template donde:

- `T`: Tarea
- `R`: Resultado de ejecutar la tarea
- `A`: Acumulador de resultados

#### 3.4.3.1. Métodos principales

**nextN(n:Long):ArrayList[T]**
Devuelve las próximas `n` tareas a realizar. En caso de que no existan tareas debe quedar bloqueado hasta que las haya. En caso de que no vayan a existir más tareas (se ejecutaron todas las tareas), puede retornar una `ArrayList[T]` vacía.

**taskToDO():Boolean**
Devuelve `true` si hay o no hay pero va a haber tareas a ejecutar y `false` en caso contrario. Una vez que retornó `false` se considera que no va a haber más tareas.

### 3.4.4. Manager

Esta clase abstracta es el elemento central de *framework*. El usuario del *framework* debe extenderla.

#### 3.4.4.1. Atributos y Propiedades principales

**abstract property batchSize():Long**
Esta propiedad abstracta indica cuántos elementos se debe ejecutar por *batch*. El

retorno de este método es pasado como parámetro al método `nextN` de `TaskDsitributor`.

No se guarda el resultado de esta, por lo que se puede hacer que el tamaño de tarea sea variable. Esta propiedad es llamada antes de la ejecución de cada *batch* de tareas, por lo que tampoco es recomendable que la ejecución se extienda.

**abstract property distributor():TaskDistributor[T]**
Esta propiedad abstracta devuelve la instancia (no una nueva, siempre la misma) de una clase que implementa el `TaskDistributor` usado.

**protected val placesSpec:PlacesSpec**
Este atributo contiene el `PlacesSpec` utilizado.

**private val acumula_dist:DistArray_Unique[Array_1[A]]**
Este atributo tiene una *array* distribuida (el indice es el `placeId` cada posición solo puede ser accedida desde el `Place` correcto). Dentro de cada posición de esta hay una *array* indexada por número de `thread` (ver `PlaceThread`).

Luego de ejecutar una *batch* de tareas, se reducen las mismas utilizando como punto de partida el contenido de esta *array* en la posición del `PlaceThread` actual y se guarda en la misma.

**private val context_dist:DistArray_Unique[Context[T,R,A]]**
Este atributo tiene una *array* distribuida similar a la de `acumula_dist` pero cada posición tiene el `Context` a utilizarse en el *place* especifico.

### Métodos principales

**abstract contextFactory():Context[T,R,A]**
Este método abstracto devuelve una nueva instancia de `Context`.

**execute():A**
Este método comienza con la ejecución del *framework*, y bloquea hasta finalizar la misma.

**this(places:PlacesSpec)**
Constructor de `Manager`, recibe como parámetro un `PlacesSpec` configurado (con la cantidad de *threads* por `Place`).

## 3.5. Funcionamiento

El método más importante del *framework* es `execute` de la clase `Manager`, que contiene el *loop* principal de procesamiento de tareas. En la figura 3.2 se puede ver el seudocódigo de este método.

A continuación se explica línea a línea:

1 Mientras existan tareas a realizar, se continúa el *loop*. Cuando ya no haya más, se espera a que toda las lanzadas finalicen.

2 Se desencola el `PlaceThread` donde se va a ejecutar, en caso de que todos los `PlaceThread` estén ocupados, se bloquea hasta que uno se libere.

4 Se consiguen tareas para ejecutar. Se recibe una `ArrayList` con un máximo de `batchSize()` tareas.

5 Se lanza un async que procesa las tareas. El *thread* que asigna recursos queda libre para repetir el proceso.

6 Se realiza un cambio de contexto de ejecución, esta continúa en el *place* indicado en 2.

7 Dentro de una *batch*, las tareas se ejecutan de forma secuencial de forma de no generar *overhead*.

8 Se accede por la id del *place* al contexto correspondiente del *place* en el que se está. Luego se realiza el mapeo (ejecución de la tarea).

11 Se realiza un *reduce* entre el acumulador correspondiente al *place* y *thread* con los resultados de la ejecución de la *batch* actual.

13 Se encola el `PlaceThread` para dejarlo disponible.

15 Dado que se utilizó un finish en 1, se espera a que todas las tareas finalicen antes de realizar los últimos pasos.

16 Se realiza un *reduce* entre los acumuladores de cada *thread* para cada *Place*.

17 Se realiza un *reduce* entre el resultado del paso anterior, es decir entre los acumuladores de cada *place*, y se retorna el mismo.

Cabe notar que solo hay una *thread* desencolando tareas y `PlaceThreads`. Cada *thread* utilizada para ejecutar tareas devuelve el `PlaceThreads` utilizado al finalizar. Esto se realiza en el *Place* donde corre la *thread* que distribuye las tareas. En la figura 3.3 se muestra una ejecución del *framework*.

```
1  finish while (distributor().taskToDO()){
2    val pt = placesSpec.dequeue();
3
4    val tasks = distributor().nextN(batchSize());
5    async{
6      at (pt.place){
7        for (task in tasks){
8          val result:R = context_dist(pt.placeID).processTask(task);
9          ...
10       }
11       ...//Reduce A
12     }
13     placesSpec.enqueue(pt);
14   }
15 }
16 ...//Reduce B
17 ...//Reduce C
18
```

Figura 3.2: Seudocódigo método `execute` de `Manager`

### 3.5.1. Mapeo del *Framework* para estrategias *Map-Reduce*

La clase que implementa `TaskDistributor` define como se realiza el *split*, generando tareas de tamaño reducido. No es necesario que todas las tareas estén definidas antes de comenzar, estas pueden ir definiéndose cuando son solicitadas (justo antes de ejecutarse).

La clase que implementa `Context` realiza el *Map*, procesando las tareas resultantes del *split*.

Luego se realizan múltiples etapas de *reduce*, al ejecutar una *batch*, entre los acumuladores de un *place* y entre los acumuladores resultantes (uno por cada *place*). En la figura 3.4 se ve cómo son realizadas las reducciones.

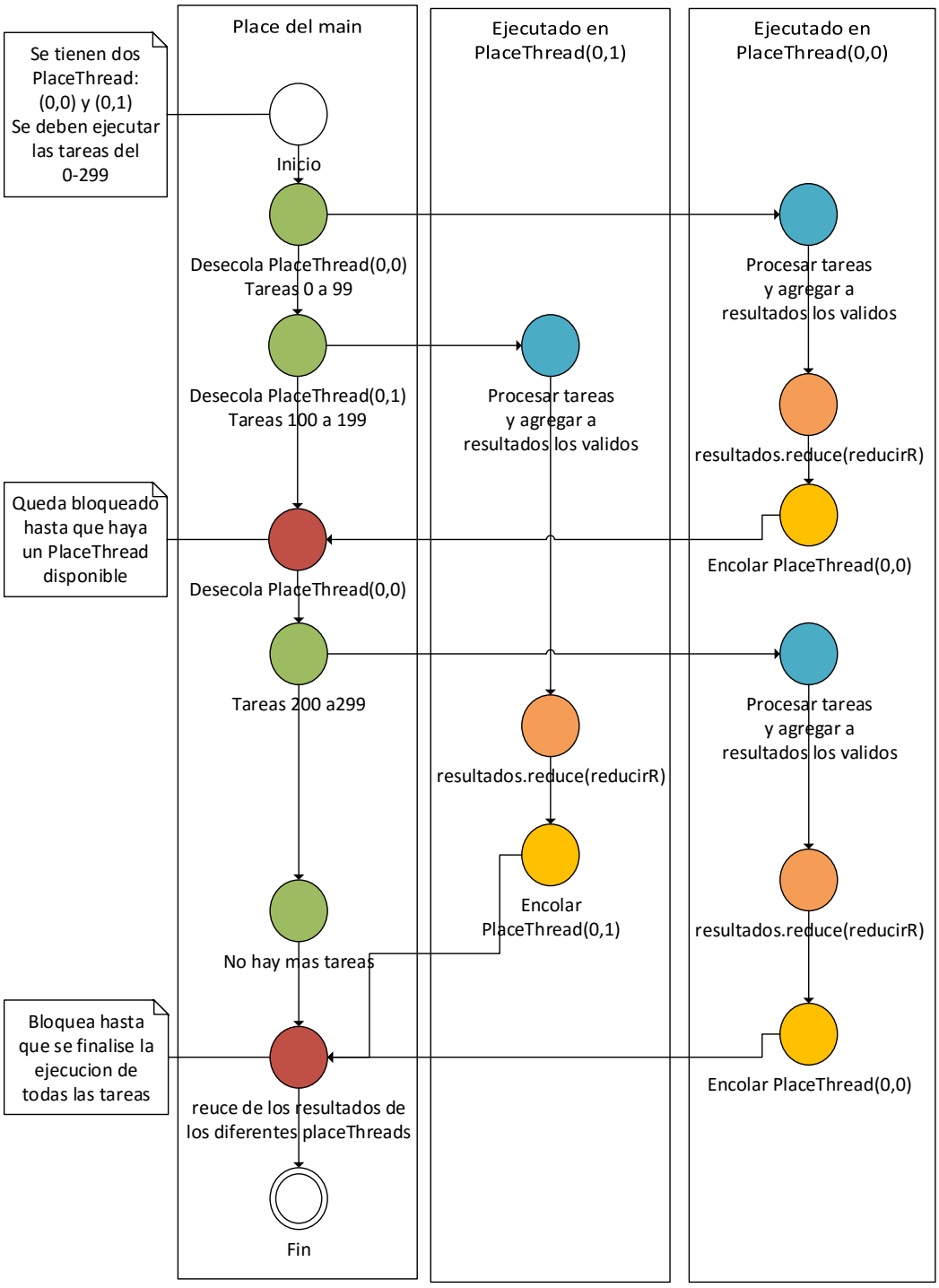


Figura 3.3: Ejemplo de ejecución: como se ve en el diagrama, no se espera a que se terminen de ejecutar las tareas para que otras se comiencen, solo es necesario que esté disponible un `PlaceThread`.

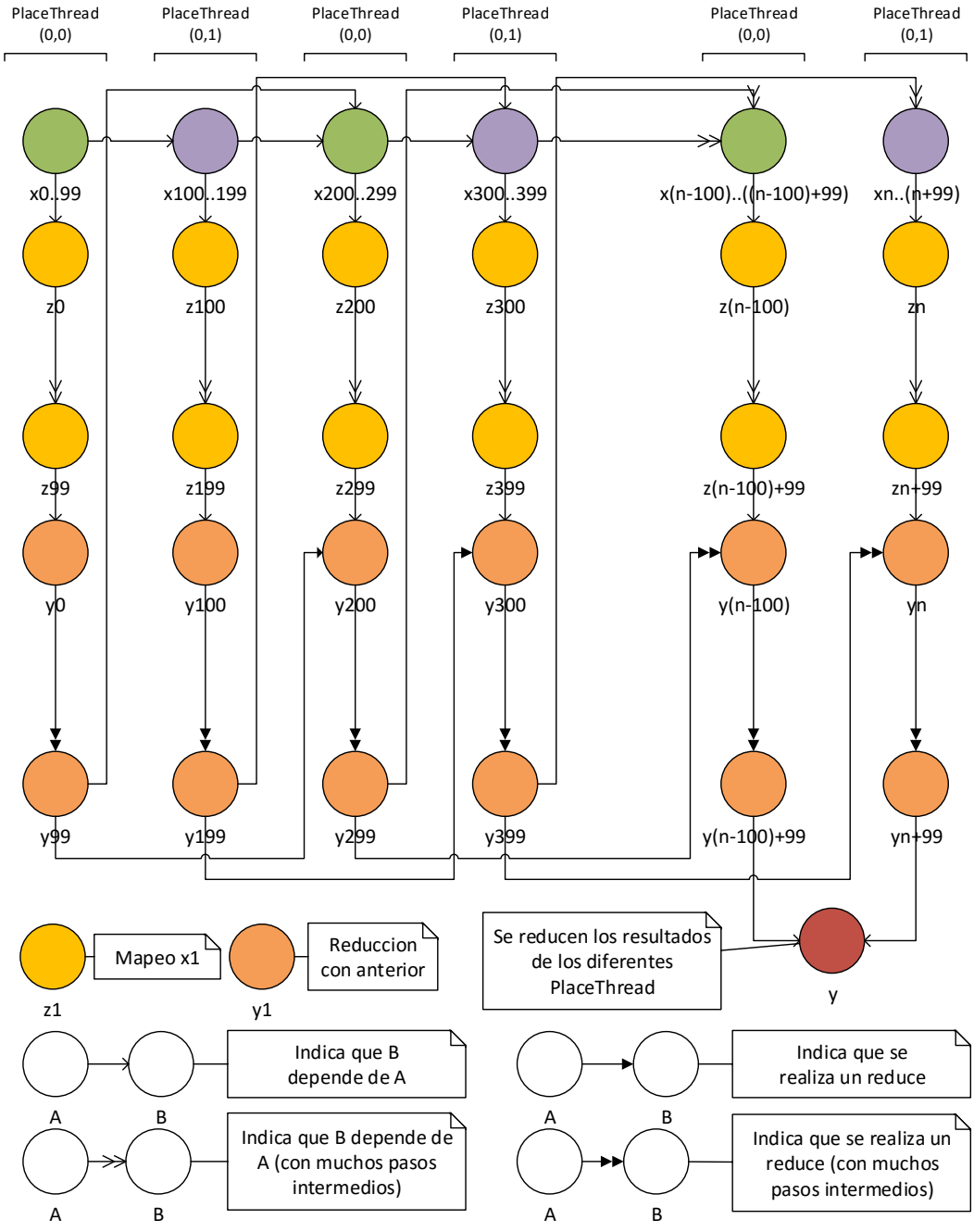


Figura 3.4: Dependencia entre ejecución de tareas y reducciones.

# 4 Primer caso de estudio: SubStringFinder

## 4.1. Descripción General

No habiendo tenido éxito con la implementación naive distribuida del `SubStringFinder`, se intentará aplicar el *framework* a este problema.

## 4.2. Detalles de la implementación

### 4.2.1. Elementos de soporte

Para poder utilizar el *framework* es necesario definir algunos elementos:

Tarea Se crea un `struct TaskSubString(start:Long, length:Long)` que contiene la posición donde arranca el sub string a buscar y su largo.

Resultado Se representa con la clase `ResultSubStringF` la que contiene el inicio, largo y sub string buscado

Acumulador Es representado por `ArrayListCustom[ResultSubStringF]`. Se opta por una implementación propia frente a la incluida en el lenguaje dado que esta no incluía una forma de agrandar la *array* interna antes de insertar múltiples elementos.

### 4.2.2. Implementación de las clases e interfaces del *Framework*

#### 4.2.2.1. TaskDistributor

Se crea la clase `DistributorSubString` que implementa `TaskDistributor[TaskSubString]`.

Esta tiene el método `nextN(n:Long) ArrayList[TaskSubString]` de tareas. Las tareas son generadas cuando son solicitadas, utilizando una lógica similar a la utilizada en las primeras implementaciónes.

También implementa el método `taskToDo()` que retorna true mientras el largo del string a buscar en la próxima tarea sea mayor a 1.

#### 4.2.2.2. Context

Se crea la clase `ContextSubString` que implementa `Context[TaskSubString, ResultSubStringF, ArrayListCustom[ResultSubStringF]]`.

Se implementan los siguientes métodos:

**processTask(task:TaskSubString):ResultSubStringF**
En este metodo se busca si el substring esta repetido.

**isValid(task:TaskSubString, result:ResultSubStringF):Boolean**
Retorna `true` si el resultado indica que se encontraron repeticiones.

**reduceR(res:ResultSubStringF, ac:ArrayListCustom[ResultSubStringF])**
Agrega `res` a `ac` y retorna `ac`.

**reduceA**
Recibe como parámetro dos `ArrayListCustom[ResultSubStringF]`, las combina en la más larga y la retorna.

**factoryAcumulator():ArrayListCustom[ResultSubStringF]**
Retorna una `ArrayListCustom[ResultSubStringF]` vacía.

**init():void**
No hace nada, no es necesario en este caso.

#### 4.2.2.3. Manager

Se crea una clase concreta `SubStringFinder` que extiende `Manager[TaskSubString, ResultSubStringF, ArrayListCustom[ResultSubStringF]]`. Se implementan las propiedades abstractas que configuran el *framework*, entre ellas `distributor` que retorna la instancia utilizada de `TaskDistributor`, un `DistributorSubString`.

También se implementa en método `contextFactory()` que retorna un nuevo `ContextSubString`.

### 4.2.3. Optimizaciones

#### 4.2.3.1. batchSize()

Uno de los problemas que se encontró al realizar la implementación naive distribuida es que las búsquedas se enviaban de a una, lo que genera un gran *overhead*.

Para reducir esto se realizaron múltiples pruebas con diferentes tamaños de *batchs*. Estas pruebas se realizaron ejecutando el *framework* sobre dos máquinas virtuales, cada una con 3 núcleos y 6GB de RAM. Los resultados se encuentran en la tabla 4.1. Podemos ver que el valor 10000 parece ser el óptimo. Quedaría pendiente realizar pruebas con tamaños que varíen durante la ejecución.

| batchSize() | Tiempo de ejecución (s) |
|---|---|
| 10 | 86 |
| 100 | 47.9 |
| 1000 | 34.8 |
| 5000 | 29.9 |
| 10000 | 28.3 |
| 50000 | 29 |
| 100000 | 29.4 |
| 500000 | 31.1 |

Tabla 4.1: Tamaños de `batch` y sus tiempos de ejecución

#### 4.2.3.2. Envio de string una sola vez

Se mantienen las optimizaciones realizadas en la segunda implementación distribuida, el string en el que se realiza la búsqueda es solo enviado una vez a cada nodo. Este es guardado en el `Context` y es distribuido por el *framework*.

#### 4.2.3.3. Distribución de PlaceThread

En las pruebas anteriores (en las cuales se configuraron los `PlaceThread` en forma similar a los procesadores, un thread por núcleo en el *place* y un *place* por maquina virtual) se notó que existen momentos en los cuales algunos núcleos quedaban sin trabajo. Esto puede ser causado por tiempos muertos entre que un `PlaceThread` termina una tarea y comienza una nueva.

En estos tiempos se realiza una vuelta al *place* principal, se encola el `PlaceThread`. Esto desbloquea la utilización de `PlaceThread`, para el cual se generan tareas, las cuales son serializadas, se realiza un cambio de *place*, los objetos serializados atraviesan la red y son deserializadas.

De forma de mitigar estos tiempos sin trabajo, se realizaron pruebas con diferentes cantidad de `PlaceThread` pero con la misma configuración de máquina virtual, 3 núcleos y 6GB de RAM. En la figura 4.1 se puede ver cómo la cantidad de optima de `PlaceThreads` en el `Place(0)` (donde se corre el `main`) es la cantidad de núcleos.

Se realizó una prueba similar pero trabajando de forma distribuida y los resultados fueron diferentes. Para esta se utilizaron dos máquinas con la misma configuración (3 núcleos y 6GB de RAM). Habiendo ajustado la cantidad de `PlaceThreads` del `Place(0)` se paso a ajustar la del `Place(1)`. En la figura 4.2 se puede ver que la cantidad de `PlaceThreads` óptimos para este caso parece ser 8.

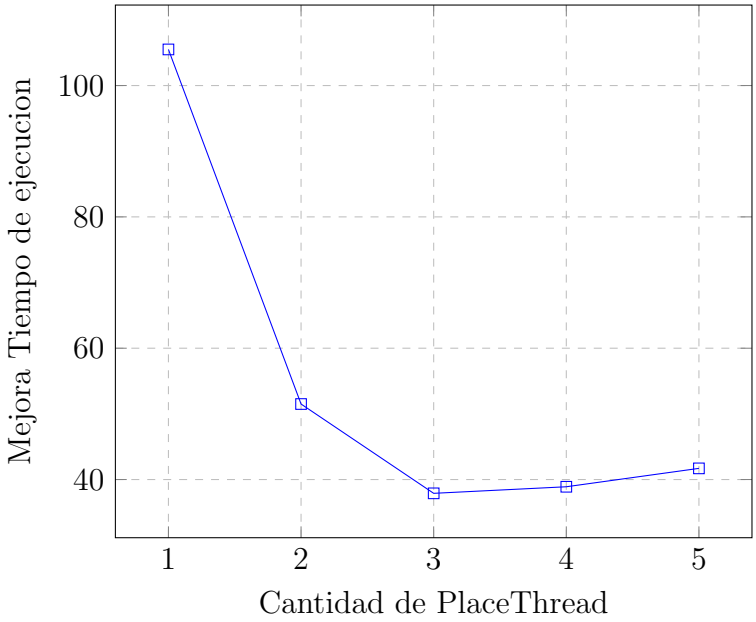


Figura 4.1: Tiempo de ejecución en función de la cantidad de `PlaceThreads`, en una máquina con 3 núcleos.

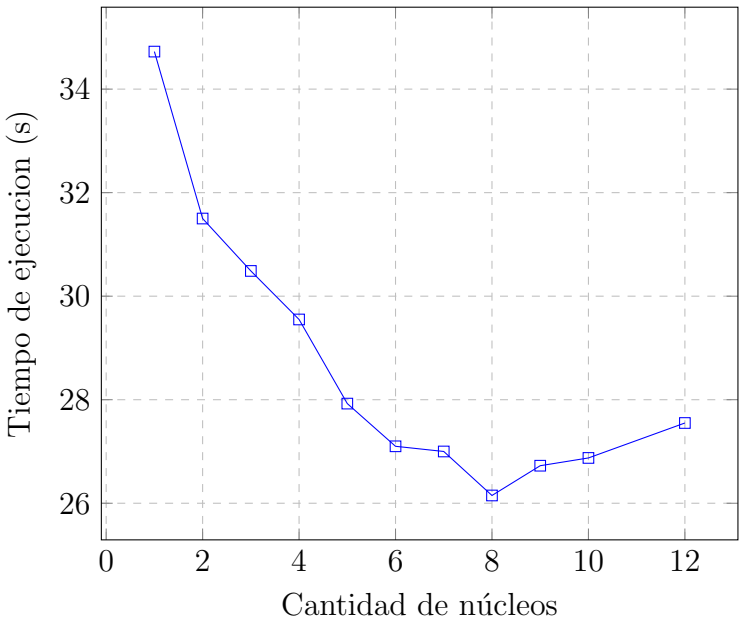


Figura 4.2: Tiempo de ejecución en función de la cantidad de `PlaceThreads` en el Place(1), en 2 máquinas con 3 núcleos cada una.

## 4.3. Referencia vs *Framework*

Se realizaron pruebas con diferentes cantidades de núcleos con el objetivo de comparar el rendimiento del *framework* vs la implementación paralela de referencia. En las figuras 4.3 y 4.4 se puede ver que la implementación que utiliza el *framework* de forma no distribuida (en rojo) es ligeramente más lenta (10 % para 6 núcleos) que la de referencia (en azul). Una de las posibles causas de esto es que la implementación de referencia no guarda los resultados, solo los imprime.

Adicionalmente, se observó que podría ser necesario que el tamaño de las *batches* se adapte a la cantidad de tareas restantes. Actualmente los últimos segundos de ejecución son realizados por pocos procesadores.

También se noto que el uso de memoria RAM de la implementación que utiliza el *framework* (270MB) es menor a la mitad que el de la implementación de referencia (680MB).

Luego se realizaron pruebas de forma distribuida, comenzando con una sola máquina virtual de 2 núcleos y 6GB de RAM, incrementando hasta 4 máquinas con estas características. Los resultados se pueden ver en las figuras 4.3 y 4.4 en verde. En la cantidad de núcleos se considera cuántos núcleos tiene la suma de las máquinas. El valor para 8 núcleos debe ser tomado como un techo, dado que a diferencia del resto de las pruebas donde se dejaba por lo menos un núcleo libre para el sistema *host*, en esta no se deja, por lo que sería de esperar que pueda llegar a ser un poco más bajo.

Contrariamente a lo esperado, la aplicación del *framework* de forma distribuida tuvo una mejor performance que de forma no distribuida. Esto pudiera deberse a que al finalizar la etapa de *map*, se realiza un *reduce* en paralelo en cada *place*, y al haber más *places* se realiza más en paralelo.

Como conclusión de la comparación del *framework* vs la implementación de referencia podemos ver que el *framework* escala de manera correcta en especial cuando se distribuye el trabajo en múltiples nodos.

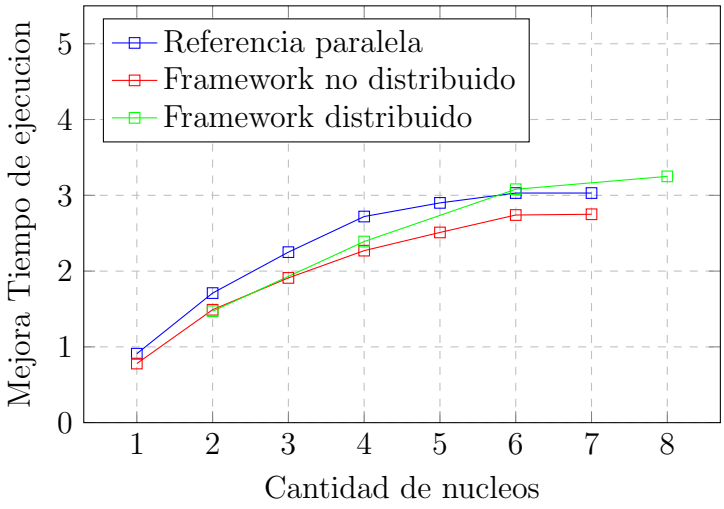


Figura 4.3: Comparación de la mejora del tiempo de ejecución entre la implementación paralela de referencia y las que utilizan el *framework*

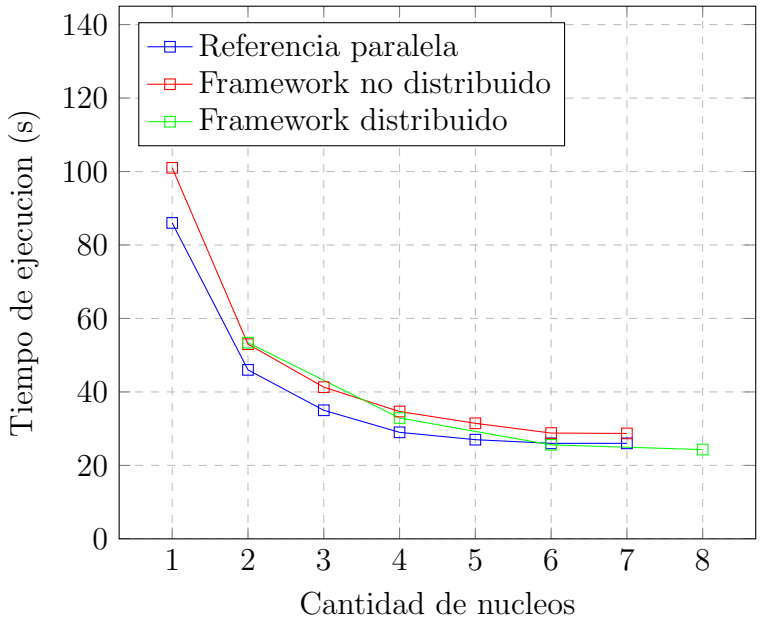


Figura 4.4: Comparación tiempo de ejecución entre implementación paralela de referencia y las que utilizan el *framework*

# 5 Segundo caso de estudio: c4.5

## 5.1. Descripción General

Una vez aplicado el *framework* al problema del string repetido más largo, se pasa a uno más real, un algoritmo de generación de árboles de decisión llamado c4.5. Este algoritmo tiene como entrada un set de datos con elementos que tienen diferentes valores para atributos (continuos o discretos) y una categoría (discreta).

Como salida genera un árbol de decisión en el que cada nodo filtra por un atributo. En el caso de los discretos se tiene un hijo por cada valor. Para el caso de los continuos se tiene dos hijos y se filtra por más chico y más grande o igual a un número determinado. En las hojas de este árbol se encuentran las categorías a las que pertenecen.

Este algoritmo es una evolución de ID3, ambos desarrollados por Ross Quinlan[13]. A diferencia de ID3, c4.5 permite la utilización de atributos con valores no discretos y atributos con valores desconocidos. En la implementación realizada no se soportan atributos con valores desconocidos.

Este problema no es el uso ideal del *framework*, debido a que no tiene la forma de *map-reduce* (no hay un *reduce*). Por esto va a servir para validar la versatilidad del *framework*.

## 5.2. Implementación

Debido a la mayor complejidad de este algoritmo y como forma de testear el *overhead* generado en cuando al tiempo de desarrollo para aplicar el *framework* a un problema, se codificó primero un núcleo que contiene la "lógica de negocios" de c4.5 y luego se creo un *wrapper* que le permite funcionar con el *framework*.

### 5.2.1. Consideraciones para la utilización de *Framework*

Al implementar la lógica, se tomaron solo dos consideraciones especiales de forma de facilitar su adaptación al *framework*.

Se aseguró que quedara bien definida lo que es una tarea, la creación de un nodo en el árbol de decisión.

Adicionalmente se tomó en cuenta que las tareas no debían tener referencias a los datos, de forma de que cuando estas sean serializadas no se serialicen también todos los datos de entrada. Para esto no se realizaron referencias directas entre muchos de los objetos, sino que se mantuvo una referencia por id y se recibe como parámetro en los métodos los objetos necesarios.

### 5.2.2. Diferencias con SubStringFinder

A diferencia del caso de SubStringFinder, en el que la siguiente tarea es "calculada", para c4.5 se utiliza una cola de tareas. Detectando esto como una oportunidad, se incluyó en el *framework* una clase que implementa `TaskDistributor` con una cola de tareas. Esta puede ser utilizada de forma directa o extendida.

La mayor diferencia entre este ejemplo y el del `SubStringFinder` es que en este el resultado de ejecución de una tarea incluye nuevas tareas que deben ejecutarse. Esto implica que debe procesarse el resultado de las tareas antes de la finalización de la ejecución.

Como solución a esto surgen dos opciones:

1. Ejecutar varias veces el *framework* teniendo como entrada el resultado de la ejecución anterior hasta que no existan más nodos a procesar.
2. A medida que se ejecutan las tareas, las nuevas tareas son agregadas como tareas pendientes.

Se optó por realizar la opción 2, debido a que la 1 requiere que todas las tareas de una ejecución finalicen para poder continuar, lo que implica que puedan quedar `PlaceThread` sin tareas mientras esperan que otros `PlaceThread` finalicen las suyas pendientes para que se vuelva a comenzar con las nuevas.

### 5.2.3. Modificaciones realizadas al *Framework*

Haber elegido la opción 2 implica que se debe proveer al usuario del *framework* con una forma de encolar nuevos elementos, pero sin afectar otros usos del *framework* (en los que quizás ni siquiera se utilizan colas).

Para esto se agregó un nuevo tipo a los *templates* de `Manager` y `Context` que ahora pasan a ser `[T,R,A,C]`, donde `C` es el tipo de un objeto que se pasa como parámetro en una llamada luego de realizar tareas.

#### 5.2.3.1. Context

Se agregó el siguiente método en la interfaz:

**def callbackValidsHome(results:ArrayListCustom[R], callObject:C):void**
Este método recibe los resultados y un objecto definido al implementar `Manager`. Se llama este método luego de realizar las tareas y el *reduce*.

Su ejecución ocurre en el *place* donde corre la thread principal del *framework*, por lo que se le pueden pasar como parámetro objetos de este (como `Manager`, `TaskDistributor`, etc).

#### 5.2.3.2. Manager

Se agregaron las siguientes propiedades:

**abstract property hasHomeCallback():Boolean**
Si es `true`, realiza un llamado el método `callbackValidsHome`, de lo contrario no se realiza.

**abstract property objetoCallback():C**
Esta propiedad devuelve la instancia del objeto llamar por parámetro en `callbackValidsHome`. Se lee solo una vez de esta propiedad, utilizando en todas las llamadas el mismo objeto.

### 5.2.4. Estructura general

En la figura 5.1 podemos ver el diagrama de paquetes de la implementación. Se puede observar que se agregaron algunos tipos template al *framework*. Dentro del paquete c45 tenemos 3 sub-paquetes: `tree` que contiene la estructura de árbol, `model` que modela el algoritmo c4.5 y `fwImp` que contiene la implementación del *framework* para c4.5.

Únicamente el código dentro del paquete `c45.fwImp` no es re utilizable si se cambia el mecanismo para ejecutar el algoritmo. Podemos ver que solo el paquete `fwImp` depende del *framework*, el resto es independiente. Dentro de los paquetes `c45.model` y `c45.tree` no se realiza paralelismo, lo que permite concentrase en la lógica de negocios.

En la figura 5.2 se ve la relación entre el paquete `c45.fwImp` y el *framework*.

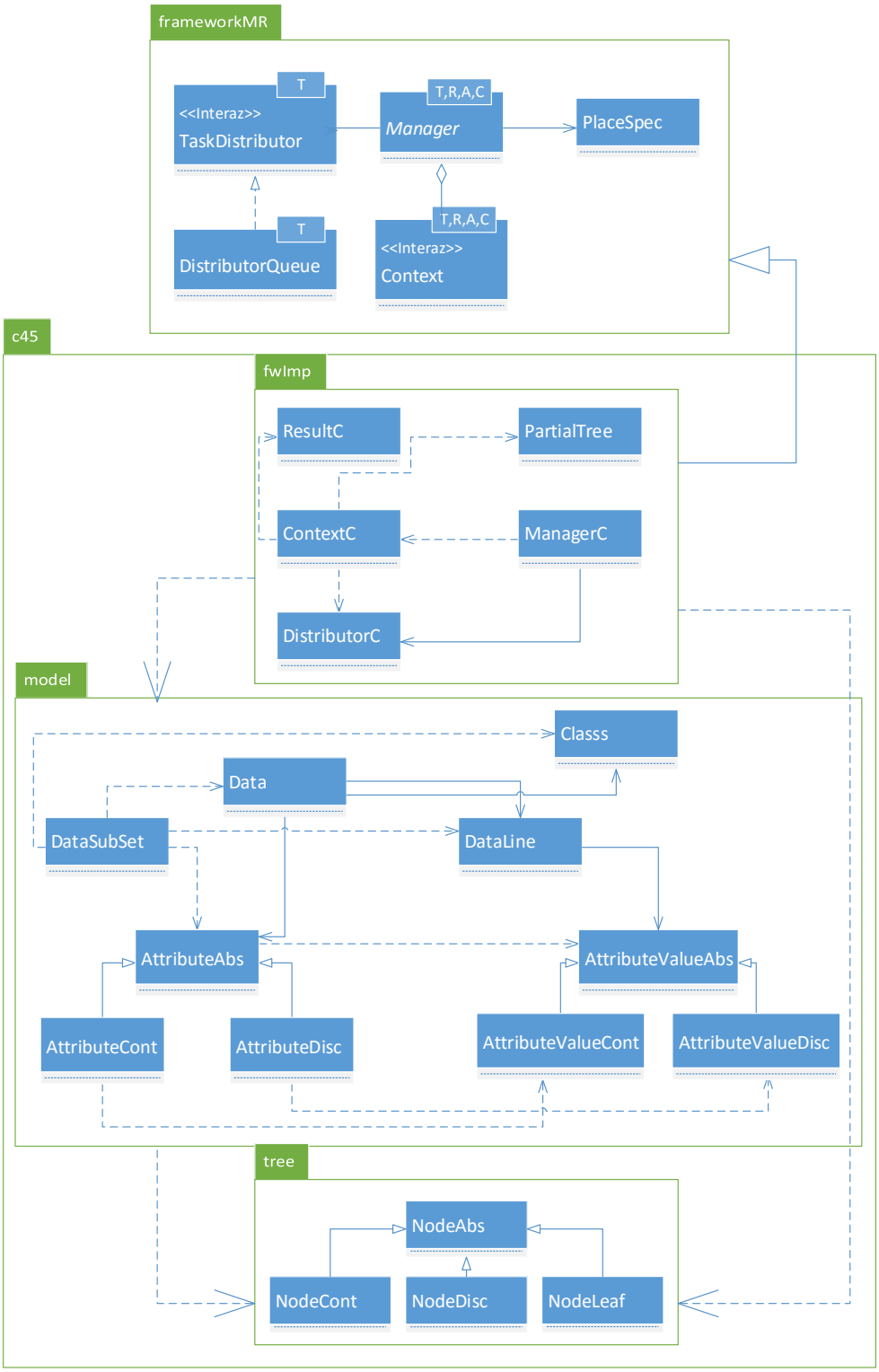


Figura 5.1: Diagrama de paquetes de la implementación de c4.5

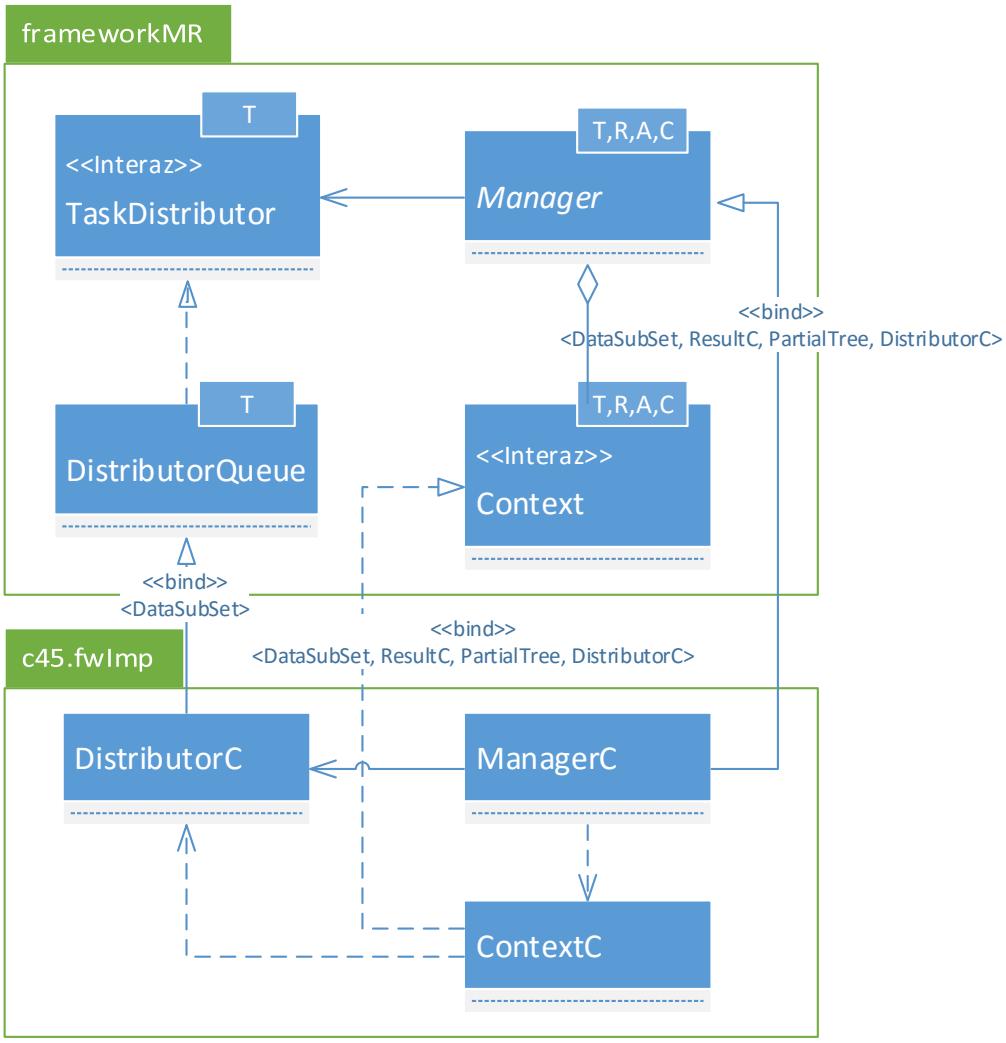


Figura 5.2: Diagrama de los paquetes que implementan el *framework* en c4.5

## 5.3. Aplicación de *Framework*

### 5.3.1. Elementos de soporte

Se definieron los siguientes tipos para utilizar con el *framework*:

Tarea `DataSubSet`: esta clase tiene una `ArrayList` con las ids de las líneas a ser procesadas y otra con los atributos no utilizados. También contiene la id del nodo que se creará a partir de estos datos. Para calcular el nodo recibe por parámetro un objeto `Data`.

Resultado `ResultC`: esta clase contiene un `NodeAbs` y una *array* de `DataSubSet`, que corresponden a los hijos del nodo creado. Luego de ejecutar la tarea cada uno de los `DataSubSet` ya contiene las lineas que le corresponden filtradas.

Acumulador `PartialTree`: esta clase contiene un `HashMap[Long,NodeAbs]` que mapea de la id de nodo a el nodo. Dentro de este se encuentran nodos del árbol que pueden o no estar relacionados.

Objecto *Callback* Se utiliza a `DistributorC` como este de forma de poder encolar nuevos `DataSubSets` al finalizar de ejecutar tareas.

### 5.3.2. Context

Se crea la clase `ContextC` que implementa `Context[DataSubSet, ResultC, PartialTree, DistributorC]`. Esta clase tiene un atributo del tipo `Data` con los datos a ser utilizado por las tareas. Al ser creado recibe como parámetro una función que al ejecutarse carga los datos.

#### 5.3.2.1. Métodos Principales

**processTask(task:DataSubSet):ResultC**
Llama a los siguientes métodos de DataSubSet:

- `buildNode(data:Data):NodeAbs` que devuelve el nodo óptimo.
- `subSets(node:NodeAbs, data:Data):Array_1[DataSubSet]` retorna los `DataSubSet` que una vez ejecutados darán como resultado los nodos hijos.

**isValid(task:DataSubSet, result:ResultC): Boolean**
Retorna siempre `true`, no se usa para este caso.

**reduceR(result:ResultC, acumulator:PartialTree):PartialTree**
Agrega los `NodeAbs` del resultado a el `HashMap` de nodos.

**reduceA(acumulator1:PartialTree, acumulator2:PartialTree):PartialTree**
Combina los `PartialTree`, el que tiene más elementos absorbe al otro y es retornado.

**factoryAcumulator():PartialTree**
Retorna un nuevo `PartialTree` sin elementos.

**init():void**
Ejecuta la función de carga de datos.

### 5.3.3. TaskDistributor

Se crea la clase `DistributorC` que extiende la clase `DistributorQueue[DataSubSet]` (provista por el *framework*) que implementa esta interfaz (`TaskDistributor`).

La clase `DistributorQueue` tiene una cola bloqueante e implementa el método `nextN` sacando elementos de esta.

`DistributorC` asigna las id de los `DataSubSets` (que luego son colocadas a los nodos), y también mantiene un conteo de cuántas tareas pendientes hay de forma de detectar cuándo se puede parar.

#### 5.3.3.1. Métodos y atributos Principales de DistributorQueue[T]

**stop:Boolean**
Indica si van a entrar nuevas tareas a la cola o si una vez que se acaben los que estén no habrán más tareas a realizar.

**nextN(n:Long):ArrayList[T]**
Si hay elementos devuelve como máximo los próximos `n` elementos a ejecutarse en la cola. Si en la cola hay menos de `n` no se bloquea, retorna los que haya.

En caso de que no haya tareas en la cola si todavía pueden entrar más (`stop` está en `false`), se espera hasta que haya tareas en la cola (se devuelven las tareas) o hasta que `stop` esté en `true` (se retorna una *array* vacía),

**taskToDO():Boolean**
Devuelve `true` si hay tareas en la cola o si no `stop`.

**stop():void**
Cambia el valor de `stop` a `true`.

#### 5.3.3.2. Métodos Principales de DistributorC

**nextN(n:Long):ArrayList[DataSubSet]**
`DistributorC` hace *override* del método. Dentro de este realiza una llamada al `nextN` de `DistributorQueue`, luego a coloca la id a cada tarea (no cambia el contador de tareas pendientes).

**addTasks(subSets:Array_1[DataSubSet], parentNodesNumber:Long)**
Este método recibe como parámetro una *array* de `DataSubSet` no procesados y los agrega a la cola de tareas. Adicionalmente modifica la cuenta de tareas pendientes, sumando la cantidad de `subSets` agregados y restando `parentNodesNumber`, la cantidad de tareas procesadas en esa *batch*.

Cuando la cantidad de tareas pendientes es 0, se llama a `stop()`.

### 5.3.4. Manager

Se crea la clase `ManagerC` que extiende `Manager[DataSubSet,ResultC, PartialTree,DistributorC]`.

#### 5.3.4.1. Elementos Principales

**property batchSize():Long**
Retorna la cantidad de elementos en `DistributorC` dividido la suma del número de `threads` de cada *place*.

**contextFactory():Context[DataSubSet, ResultC, PartialTree, DistributorC]**
Retorna un nuevo `ContextC`.

**property hasHomeCallback():Boolean**
Devuelve `true`.

## 5.4. Referencia vs *Framework*

### 5.4.1. Implementación de referencia

Con el objetivo de comparar la performance de la implementación de c4.5 con el *framework*, se crea una implementación recursiva paralela que no utiliza el *framework*.

Se puede ver en la figura 5.3 la implementación de referencia. Esta está compuesta de 2 métodos, uno recursivo privado `_refImpParallel` y uno público `refImpParallel` que llama al primero.

El método recursivo lanza el cálculo de los nodos hijos en paralelo. El no recursivo cuenta con un *finish* que asegura que termine la ejecución de todos los *asyncs* lanzados por el método recursivo.

Adicionalmente, para tener una linea base, se crea una implementación secuencia, esta se encuentra en la figura 5.4.

### 5.4.2. Performance

Para las pruebas de performance, se utiliza un set de datos con aproximadamente 59000 lineas. Se consideran 6 atributos continuos y 108 discretos. Al medir

```
1  public def refImpParallel(subSet:DataSubSet, data:Data):NodeAbs{
2    var ret:NodeAbs;
3    finish {
4      ret =_refImpParallel(subSet,data);
5    }
6    return ret;
7  }
8
9
10 private def _refImpParallel(subSet:DataSubSet, data:Data):NodeAbs{
11   val node = subSet.buildNode(data);
12   if (node.childrensN()>0){
13     val subSets = subSet.subSets(node, data);
14     for (indexM in node.childrens.indices()){
15       val index = indexM(0);
16       async{
17         node.childrens(index) = refImpParallel(subSets(index),data);
18       }
19     }
20   }
21   return node;
22 }
23
```

Figura 5.3: Implementación de c4.5 de referencia paralela

```
1  public def refImpNonParallel(subSet:DataSubSet, data:Data):NodeAbs{
2      val node = subSet.buildNode(data);
3      if (node.childrensN()>0){
4        val subSets = subSet.subSets(node, data);
5        for (indexM in node.childrens.indices()){
6          val index = indexM(0);
7          node.childrens(index) = refImpNonParallel(subSets(index),data);
8        }
9      }
10
11     return node;
12   }
13
```

Figura 5.4: Implementación de c4.5 de referencia secuencial

los tiempos no se consideró el tiempo que corresponde a la lectura de los datos de disco ni el de armado de las estructuras.

Se corrió primero un prueba con una implementación secuencial y se encontró

como tiempo 36.4s. Este tiempo fue usado como referencia para medir la mejora de las diferentes implementaciónes en sus configuraciones. La mejora fue calculada de la siguiente forma: tiempo_referencia/tiempo.

Se puede ver en la figura 5.5 el gráfico de la mejora y en 5.6 los tiempos. En estas gráficas la implementación de referencia esta en azul, utilizando el *framework* de forma local en rojo y utilizando el *framework* de forma distribuida en verde.

En las pruebas de referencia y del *framework* local se utilizo una maquina con 6GB de RAM. Para el caso del *framework* de forma distribuida la cantidad de núcleos es la suma de los núcleos de las maquinas utilizadas. Se partió de una maquina con 2 núcleos y 6GB de RAM, luego se ejecuto con 2 maquinas (ambas con 2 núcleos y 6 GB de RAM) y luego con 3 con la misma configuración.

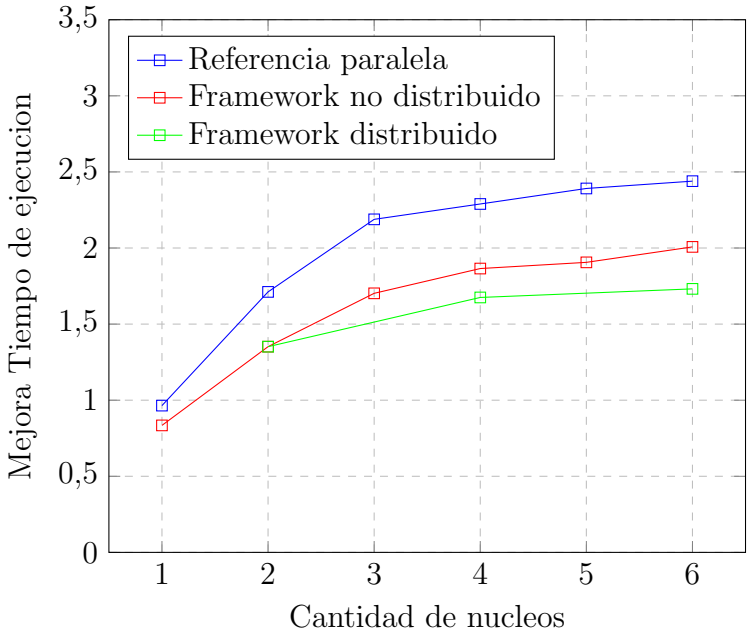


Figura 5.5: Mejora del tiempo de ejecución de las diferentes implementaciones de c4.5, en comparación con la implementación secuencial de referencia.

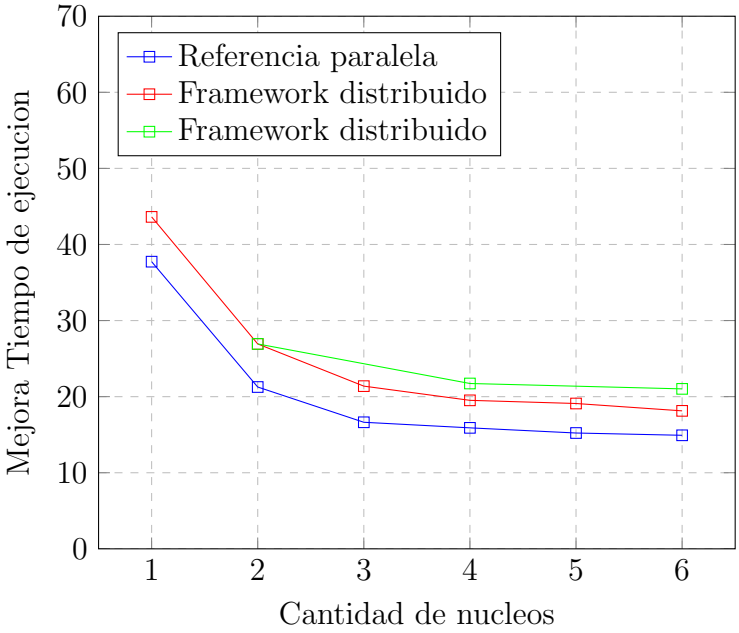


Figura 5.6: Tiempo de ejecución de las diferentes implementaciones de c4.5.

## 5.5. Problemas

Podemos ver que existe una diferencia bastante importante entre la implementación que utiliza el *framework* de forma no distribuida y la de referencia. Algunas de las algunas posibles causas son:

- Aunque no se este ejecutando de forma distribuida, el *framework* realiza un `at` antes de ejecutar las tareas. Esto genera un *overhead* al tener que serializar y deserializar las tareas antes de ejecutar y volver a serializar las nuevas a agregar.
- Al terminar de ejecutar una *batch* se debe agregar a la cola de tareas las resultantes de la ejecución. Esto implica la utilización de *locks* para sincronizar el acceso.
- A diferencia de la implementación de referencia, el *framework* realiza *reduce*, que en este caso no son necesarios, pero generan cierto *overhead*.

Adicionalmente se detectó como problema (de la implementación del *framework* y la de referencia) que dado que el paralelismo tiene como unidad el calculo de

un nodo del árbol, los primeros nodos que se calculan no logran utilizar todos los procesadores.

Como podemos ver en la figura 5.7, el tiempo de cálculo de un nodo es no lineal respecto al tamaño de los datos. Esto causa que el tiempo de ejecución de los primeros nodos (los que deben procesar mucho más datos) sea mucho mayor a los del resto. Por esta razón se ve como oportunidad de mejora el paralelizado a nivel de cálculo de nodo.

- Primero se debe calcular la información que aportan los datos parámetro para calcular ese nodo.

  Para esto se recorren todas las líneas y se cuenta la cantidad de elementos de cada clase (en principio hay una cantidad reducida, por lo que se considera constante).

  Orden: $O(n)$ donde $n$ es la cantidad de lineas.

- Luego se selecciona el atributo para filtrar. Para esto se calcula cual es el mejor atributo para filtrar.

  - Atributo Continuo:
    - Se recorren todas las lineas y se cuentan cuantos elementos hay de cada clase. Orden: $O(n)$
    - Se ordenan los valores del atributo. Orden: $O(n^2)$

    Orden: $O(n^2)$
  - Atributo Discreto:
    - Se recorren todas las lineas y se cuentan cuantos elementos hay de cada clase. Orden: $O(n)$

    Orden: $O(n)$

  Orden: $O(n^2 * m)$ donde $m$ es la cantidad de atributo parámetro

- Generación de los datos para calcular los hijos. Para esto se recorre los datos parámetro y se los asigna a uno de los datos de los hijos. Orden $O(n)$

Como resultado, el cálculo de un nodo tiene el orden:
$$O(n) + O(n^2 * m) + O(n) = O(n^2 * m)$$

Figura 5.7: Orden de peor caso para el calculo de un nodo

# 6 Segunda Implementación del *Framework*

Habiendo encontrado algunos problemas de performance en la primera implementación del *framework*, se trabaja en diferentes métodos para mejorar la misma.

## 6.1. Ejecución local

Al utilizar el *framework*, sin importar se realiza en una o múltiples máquinas, siempre se serializa y des serializa las tareas a ejecutar dado que la ejecución de las mismas está dentro de un `at`. En el caso de ejecuciones locales esto genera un *overhead* innecesario. Por estas razones se agregó una configuración al *framework* que permite correr en modo local.

Se debe implementar la siguiente propiedad en las clases que extiendan `Manager`:

**property runLocaly():Boolean**

Cuando esta propiedad es `true`, las tareas no se realizan dentro de un `at`. Si se configura en `true`, el `PlacesSpec` no debe tener múltiples *places*.

#### 6.1.0.1. Resultados

Como podemos ver en la tabla 6.1 el tiempo de ejecución disminuye considerablemente, acercándose a el tiempo de la implementación de referencia.

| | Tiempo promedio (s) | Mejora de tiempo |
|---|---|---|
| Referencia | 14.9 | 2.4 |
| *Framework* local con `at` | 18.1 | 2.0 |
| ***Framework* local sin `at`** | **15.8** | **2.3** |

Tabla 6.1: Tiempos de ejecución con 6 núcleos sin `at`

## 6.2. Work Stealing

Se detectó como posibilidad de problema en la implementación del *framework* que, luego de ejecutar una *batch* de tareas de forma paralela, se debía agregar las nuevas tareas a la cola. Esto implica realizar una sincronización debido a que se tienen muchas *threads* accediendo a la cola de forma simultanea.

Por esta razón se implementó work stealing. Esta técnica consiste en que cada linea de procesamiento tenga una cola de tareas. Cuando se generan nuevas tareas estas son agregadas a la cola de quien las genera. Al buscar tareas estas son buscadas primero en la cola de quien va a ejecutarlas. En caso de que no se tengan tareas se busca en la cola de otro y se "roban tareas".

En su implementación actual, se tiene una cola por `PlaceThread`. Estas están ubicadas en el *place* donde corre el *framework*.

### 6.2.1. Cambios al *Framework*

#### 6.2.1.1. TaskDistributor

Se modifica la firma del el siguiente método de la interfaz:

**init(pts:ArrayList[PlaceThread]):void**
Se agrega el parámetro `pts`. Este contiene los `PlaceThread` que van a ser utilizados, esto permite inicializar las colas necesarias.

#### 6.2.1.2. DistributorQueueSteal

Se crea una nueva implementación de `TaskDistributor`, `DistributorQueueSteal`, que actúa de *wrapper* a una cola bloqueante con *Work Stealing*.
Adicionalmente de los metodos de `TaskDistributor`, cuenta con el método:

**enqueue(es:Array_1[T],pt:PlaceThread)**
Este encola las tareas en la cola correspondiente al `PlaceThread`.

#### 6.2.1.3. Context

Se modifica la firma del el siguiente método de la interfaz:

**callbackValidsHome(results:ArrayListCustom[R], callObjetc:C, token:PlaceThread):void**
Se agrega el parámetro *token*. Donde *token* es el `PlaceThread` donde se ejecutaron las tareas.

### 6.2.2. Cambios a la implementación de c4.5

Se realizaron los siguientes cambios a la implementación de c4.5:

- `DistributorC`
  - Paso a extender `DistributorQueueSteal` en lugar de `DistributorQueue`.
  - Se agrega al método `addTasks` el parámetro `token:PlaceThread` que indica donde se debe encolar las nuevas tareas. Y se lo actualiza para que utilice el método `enqueue(es:Array_1[T],pt:PlaceThread)` al encolar las tareas.
- `ContextC`
  - Se actualizó la firma de `callbackValidsHome` agregando `token:PlaceThread` como parámetro, el cual es pasado a `addTask` de `DistributorC`.

### 6.2.3. Resultados

Como se puede ver en la tabla 6.2, los resultados no fueron buenos, para el caso con `at`, hubo una muy pequeña perdida en performance, esta perdida se encuentra dentro del margen de error de la prueba. Sin embargo los resultados sin `at`, indican con seguridad que hubo una baja en la performance por la inclusión del Work Stealing en su implementación actual.

Se realizaron pruebas distribuidas de esta modificación y no se encontraron mejoras. En el futuro podría implementarse una versión que utilice *Work Stealing*

y que aproveche la localización de los nodos. Es decir que si se generan tareas en un *place* determinado, estas queden en una cola dentro de este. Y se podría realizar que el proceso de robo se realice primero a colas de otros `PlaceThread` dentro del mismo *place* y luego si no se encuentra, en otros *places*.

| | Tiempo promedio (s) | Mejora de tiempo |
|---|---|---|
| Referencia | 14.9 | 2.4 |
| *Framework* local con `at` | 18.1 | 2.0 |
| ***Framework* local con `at` con *Work Stealing*** | **18.2** | **1.9** |
| *Framework* local sin `at` | 15.8 | 2.3 |
| ***Framework* local sin `at` con *Work Stealing*** | **17.4** | **2.1** |

Tabla 6.2: Tiempos de ejecución con 6 núcleos para *Work Stealing*

## 6.3. *Framework* Simplificado

Con el objetivo de facilitar el uso de del *framework* para tareas de menor tamaño, se crea una versión simplificada del *framework*.

### 6.3.1. Implementación

No fue necesario realizar modificaciones al *framework* para realizar esta versión simplificada. Este permite agregar funciones a una cola para su ejecución y recuperar los resultados en una `ArrayList`.

Solo requirió crear las siguientes clases:

#### MangerFunctional[R]

Esta clase extiende `Manager[()=>R,R,ArrayList[R],ArrayList[Long]]`. Los tipos *template* de `Manager` son:

Tarea es `()=>R`, una función que no recibe parámetros y retorna un elemento del tipo R

Resultado es un tipo *template* R, definido por quien utiliza el *framework*

Acumulador es `ArrayList[R]`, una `ArrayList` de resultados.

Objeto Callback no es utilizado, pero se define como `ArrayList[Long]` por que es obligatorio completar el *template*.

#### 6.3.1.0.1. Métodos y atributos utilizados por el *framework*

**batchSize():Long**
Retorna `1`, las tareas se realizan de a una.

**hasHomeCallback():Boolean**
Retorna `false`, no se utiliza esta funcionalidad.

**runLocaly()**
Retorna `true`, no se realiza distribución.

**distributor():TaskDistributor[()=>R]**
Retorna la instancia de `DistributorQueue` utilizada.

**contextFactory():Context[()=>R,R,ArrayList[R],ArrayList[Long]]**
Retorna una nueva instancia de `ContextFunctional`

#### 6.3.1.0.2. Métodos y atributos utilizados el usuario

**enqueue(task:()=>R):void**
Permite encolar una nueva tarea.

**executeAsync():void**
Comienza con la ejecución del *framework* pero no espera su finalización.

**join():ArrayList[R]**
Indica que no se van a agregar más tareas y bloquea hasta que finalice la ejecución de las existentes. Retorna el resultado de la ejecución.

**this (threads:Long, queueSize:Long)**
Constructor de `MangerFunctional`. Recibe como parámetros la cantidad máxima de `threads` simultaneas a utilizar y `queueSize` indica la cantidad máxima de elementos que soporta la cola.

#### 6.3.1.1. ContextFunctional[R]

Esta clase implementa `Context[()=>R, R, ArrayList[R], ArrayList[Long]]`, donde los tipos *template* son los mismos que en `MangerFunctional[R]`.

##### 6.3.1.1.1. Métodos y atributos principales

**processTask(tarea:()=>R):R**
Recibe como parámetro una función y la ejecuta retornando el resultado de la misma.

### 6.3.2. Utilización

Para utilizar esta versión simplificada del *framework* solo es necesario definir funciones que retornen el mismo tipo de objeto (eventualmente `Any`, que es una interfaz de la cual heredan todos los objetos en x10).

En la figura 6.1 se encuentra un ejemplo de uso del *framework*. En esta se agregan funciones a ejecutar que imprimen y luego retornan un número. `fw.executeAsync()` indica que ya se puede comenzar a ejecutar elementos (en este caso no hay ninguno aún). A medida que se van encolando las funciones estas quedan prontas para ejecutarse. El orden en el que comienzan a ejecutarse las funciones es en el que fueron agregadas (este no es necesariamente el orden en el que terminan de ejecutarse, el cual es no determinista).

El orden en el que se encuentran en en la *array* de resultados es no determinista. `fw.join()` indica que se debe esperar a que todas las tareas ejecuten para continuar y retorna el resultado.

Ejemplo de salida:

```
f2
f1
f3
```

```
f4
f5
f6
f8
f7
f9
f10
2
5
9
1
8
3
7
4
6
10
```

```
1  val fw = new MangerFunctional[Long](4,100);
2
3  fw.executeAsync();
4
5  for (val i in 1..10){
6    fw.enqueue(()=>{
7      Console.OUT.println("f" + i);
8      return i;
9    });
10 }
11
12 val results = fw.join();
13
14 for (val i in results){
15   Console.OUT.println(i);
16 }
17
```

Figura 6.1: Ejemplo de utilización del *framework* simplificado

### 6.3.3. Aplicación a c4.5

Esta versión simplificada del *framework* fue utilizada dentro del cálculo del mejor atributo para filtrar en un nodo. Para esto se encapsuló la llamada que calcula el nodo tentativo (de un atributo) en una función sin parámetros. Luego se recupera el resultado (el cual es ordenado por id de forma de que el resultado sea determinista) y se elije el mejor nodo.

#### 6.3.3.1. Optimizacion del cambio de estrategia

Dado que este nivel de paralelismo no es necesario durante toda la ejecución del algoritmo (una vez pasados los primeros nodos hay trabajo para todos los procesadores), se va a buscar el punto optimo en cual dejar de realizar esta parte en paralelo. Para esto se busca un número de elementos mínimo para el cual esto sea conveniente.

En la figura 6.2 se ven los tiempos para los diferentes puntos de cambio. Se puede ver que menos 5000 elementos es un buen punto de cambio a búsqueda secuencial.

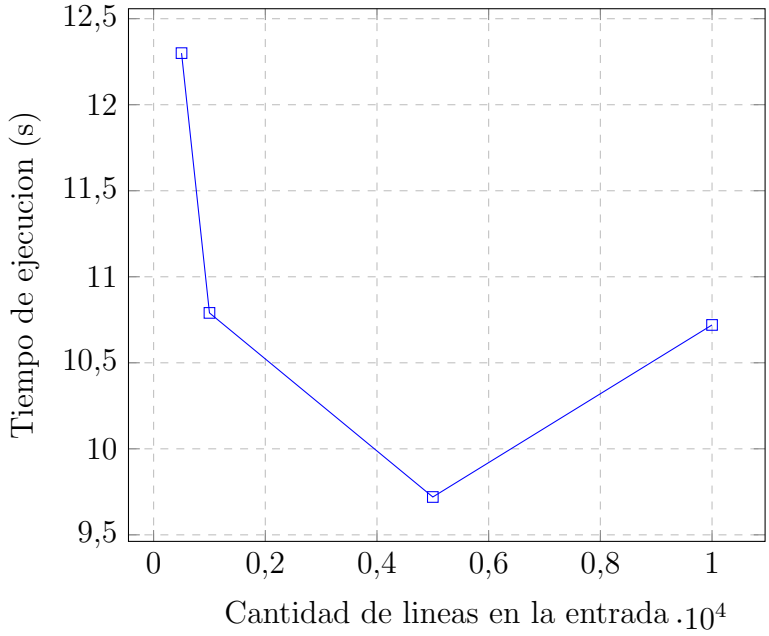


Figura 6.2: Tiempos de ejecución para diferentes valores de punto de cambio entre implementación paralela y secuencial de la búsqueda de mejor atributo

#### 6.3.3.2. Resultados

En la implementación de referencia se agrega el uso del *framework* a nivel de búsqueda dentro de nodo. Se miden los tiempos de ejecución para las diferentes combinaciones, estos se pueden ver en la tabla 6.3.

| | Tiempo promedio (s) | Mejora de tiempo |
|---|---|---|
| Referencia | 14.9 | 2.4 |
| *Framework* local sin `at` | 15.8 | 2.3 |
| **Referencia paralelismo a nivel de nodo** | **9.3** | **3.9** |
| ***Framework* local sin `at` paralelismo a nivel de nodo** | **9.7** | **3.7** |

Tabla 6.3: Tiempos de ejecución con 6 núcleos utilizando dos niveles de paralelismo

# 7 Conclusiones

## 7.1. Lenguaje

A lo largo de las diferentes pruebas realizadas, x10 demostró ser un potente lenguaje y altamente flexible. Siendo tan "joven", tiene deficiencias en cuanto al ciclo de trabajo. Actualmente no tiene *debugger*, por lo que debuggear código complejo y paralelo se vuelve una tarea no trivial. Adicionalmente no cuenta con muchas estructuras (colas, semáforos), las cuales deben ser implementadas por el usuario.

x10 cuenta con dos formas de compilarlo: de forma nativa, en la que pasa por C++ antes de producir un ejecutable y de forma *managed*, en la que pasa por java antes de producir Java *bytecode*, el cual puede ser corrido por la Java *virtual machine*. El *framework* solo corre en la versión que utiliza java, debido a que al compilarlo para C++ esta falla (posiblemente por problemas del compilador de x10 a C++).

Es de esperar que estas deficiencias sean eventualmente corregidas, siendo un lenguaje relativamente joven y con tal potencial.

## 7.2. Framework

Mediante el uso del *framework* fue posible abstraer las primitivas de paralelismo de x10, permitiendo que el usuario solo deba codificar lo correspondiente a su "lógica de negocios".

El *framework* pudo ser utilizado incluso cuando el problema no se presta naturalmente para una estrategia *map-reduce*, como en el caso de c4.5. Incluso en estos casos se logro un bajo nivel de acoplamiento, solo siendo necesario codificar

un *wrapper* a los métodos de mayor nivel de la lógica.

En las diferentes implementaciónes se vio que el costo en performance de utilizar el *framework* es bajo, incluso para casos para los cuales no se adapta perfectamente, obteniéndose resultados similares a las implementaciónes de referencia.

Como beneficio del uso del *framework* se tiene que el usuario programador no debe preocuparse por cuestiones de paralelismo, simplemente debe concentrarse en la implementación de su lógica. De esta forma se logra mejorar la productividad.

# 8 Trabajo Futuro

## 8.1. Localización de las tareas

La principal oportunidad de trabajo futuro se presenta en la sacar provecho de la localización de los nodos teniendo colas de tareas de centralizadas, una en cada uno de ellos. Esto tiene múltiples beneficios:

- Se disminuye el overhead de red y de serialización/deserialización. Al generar nuevas tareas resultado de la ejecucion de otras, no es necesario que estas pasen por el proceso de serialización, red y deserialización, estas serian guardadas en el nodo que las creo. Adicionalmente no seria necesario realizar este proceso con nuevas tareas a ejecutarse dado que estas se encontrarian dentro del mismo nodo.
- Menor requerimiento de memoria en el *place* del `main`. Dado que las tareas serian guardadas de forma distribuida, este *place* ya no cuenta con una mayor exigencia en este aspecto.
- Mayor posibilidad de escalado. Como el *place* del `main` no debe estar asignando constantemente tareas, se baja su carga, permitiendo que coordine a mas nodos sin saturarse.

Esto implicaría agregar mecanismos que permitan a un *place* sin tareas "robar" tareas a otros places.

## 8.2. Agregado y quitado de *places*

En la implementación actual de *framework*, se requiere que todos los *places* estén presentes antes de comenzar y en caso de que un falle, se pierde toda la

ejecución.

Desde hace ya un tiempo, x10 soporta continuar con la ejecución en el caso de que un *place* falle. Adicionalmente, se agrego de forma reciente la posibilidad de agregar nuevos *places* habiendo ya comenzado la ejecución.

Este cambio implicaría que se debe considerar que realizar con las tareas en ejecución en el caso de que este *place* falle. Esto se vuelve aun mas complicado si se implementa la mejora de localización de las tareas, dado que ya no queda tan claro que tareas tenia el *place* caído desde el punto de vista del *place* donde se inicia el *framework*.

# 9 Bibliografía


[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kauffman, 2007.

[2] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: Recording microprocessor history," *ACM Queue*, 2012.

[3] x10 language. [Online]. Available: http://x10-lang.org/

[4] Chapel language. [Online]. Available: http://chapel.cray.com/

[5] Julia language. [Online]. Available: http://julialang.org/

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[7] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 2012.

[8] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification." [Online]. Available: http://x10.sourceforge.net/documentation/languagespec/x10-250.pdf

[9] Licencia de x10, eclipse public license v1.0. [Online]. Available: http://x10-lang.org/articles/24.html

[10] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu, "The asynchronous partitioned global address space model," in *The First Workshop on Advances in Message Passing*. [Online]. Available: http://www.cs.rochester.edu/u/cding/amp/papers/full/The%20Asynchronous%20Partitioned%20Global%20Address%20Space%20Model.pdf

[11] V. A. Saraswat, O. Tardieu, D. Grove, D. Cunningham, M. Takeuchi, and B. Herta. The apgas model. [Online]. Available: http://x10-lang.org/documentation/intro/2.5.0/html/node4.html

[12] HashiCorp. About vagrant. [Online]. Available: https://www.vagrantup.com/about.html

[13] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

# 10 Anexos

## 10.1. Estructuras de datos creadas

Durante las diferentes etapas del proyecto, se crearon múltiples estructuras y utilidades en x10. No todas fueron utilizadas en las versiones finales.

### 10.1.1. Util

La librería `Util` cuenta con elementos de uso general. Esta solo depende de lo incluido con el lenguaje x10.

#### 10.1.1.1. IQueue[T]

Esta Interfaz cuenta con los siguientes métodos:

- `size():Long` retorna el tamaño de la cola.
- `enqueue(e:T):void` encola un elemento.
- `dequeue():T` desencola un elemento y lo retorna.

#### 10.1.1.2. Queue[T]

Esta clase implementa la interfaz `IQueue[T]`. Cuenta con un máximo de elementos (definido al momento de crearla), si se intenta agregar elementos pasado este, tira una excepción. No es *thread-safe*. Internamente utiliza una *array* en forma circular.

#### 10.1.1.3. QueueBloc[T]

Esta clase implementa la interfaz `IQueue[T]`. Algunos de los métodos de esta cola bloquean si se trata de encolar/desencolar cuando la capacidad de la cola no lo permite. Esta es *thread-safe*. Internamente utiliza una `Queue[T]`

Cuenta con los siguientes métodos:

- `size():Long` retorna el tamaño de la cola.
- `enqueue(e:T):void` encola un elemento, si la cola esta completa, bloquea hasta que se pueda insertar.
- `dequeue():T` desencola un elemento y lo retorna, si no hay elementos bloquea hasta que haya.
- `enqueue(es:Array_1[T])` y `enqueue(es:ArrayList[T])` encolan todos los elementos de la *array*, si no hay espacio para todos, tiran una excepción y no encolan ninguno.
- `dequeueMax(n:Long):ArrayList[T]` desencola hasta `n` elementos, si no hay elementos se queda esperando hasta que pueda des encolar por lo menos 1.
- `dequeueMaxTry(n:Long):ArrayList[T]` desencola hasta `n` elementos, si no hay elementos retorna una `ArrayList[T]` vacía.

#### 10.1.1.4. QueueBlocSteal[E,T]

Esta clase contiene múltiples colas, una para cada par de productor/consumidor. Cuando un productor encola, lo hace a su cola, y cuando su productor des encola primero intenta en la cola que le corresponde y luego en cualquiera de las demás. Internamente mantiene múltiples colas en un `HashMap[T,QueueBloc[E]]`, adicionalmente cuenta con una cola extra si productor/consumidor en la que se puede encolar sin *token*. El tamaño de las colas es fijo y una vez que se termina de definir los productores/consumidores no es posible continuar agregando.

El template es:

`E` tipo de los elementos almacenados en la cola.

`T` tipo del *token* utilizado para acceder a las diferentes colas.

Cuenta con los siguientes métodos:

- `this(size:Long)` constructor, define el tamaño que va a tener cada una de las colas creadas.
- `registerProducer(token:T)` registra un *token* de productor/consumidor, creando una cola para este.
- `enqueue(e:E):void` encola en una cola sin productores/consumidores especifico.
- `enqueue(e:E, token:T):void` encola en una cola especifica un elemento.
- `enqueue(es:ArrayList[E], token:T):void` encola los elementos en la cola especifica.
- `dequeue(token:T):E` desencola un elemento, primero intenta en la cola especifica, en caso de que no tenga elementos de cualquier cola.
- `dequeueMax(n:Long,token:T):ArrayList[E]` desencola como máximo `n` elementos. Primero intenta en la cola especifica, si esta no tiene elementos desencola de cualquier otra. Si las colas tienen menos de `n` elementos desencola los elementos que haya.

#### 10.1.1.5. QueueDisc[T]

Esta clase implementa `IQueue[T]`. Esta tiene como particularidad que escribe los elementos a disco. Para serializar y des serializar utiliza los métodos usados por x10 durante los `at`. Los métodos `enqueue` y des `dequeue` no son bloqueantes.

#### 10.1.1.6. QueueHybrid[T]

Esta clase implementa `IQueue[T]`. Esta cola es bloqueante e internamente tiene dos colas: una `QueueDisc[T]` y otra `QueueBloc[T]`. Al encolar se escribe el elemento a disco y al des encolar se hace de `QueueBloc[T]`. Las dos colas se conectan mediante un `while` que mientras se este corriendo des encola elementos del disco y los encola en `QueueBloc[T]`.

#### 10.1.1.7. *Semaphore*

Esta clase implementa un semáforo, utilizando monitores. Adicionalmente a los métodos típicos de semáforo, cuenta con algunos extra:

- `p()` espera.
- `pTry():Boolean` intenta esperar, si lo logra retorna `true`, de lo contrario no espera y retorna `false`.
- `p(max:Long)` decrementa el contador del semáforo entre `1` y `max`, luego retorna la cantidad. Si el semáforo se encuentra en `0` espera.
- `pTry(max:Long):Long` decrementa el contador del semáforo entre `0` y `max`, luego retorna. Si el semáforo se encuentra en `0`, no espera.
- `v()` señala.
- `v(n:Long)` senala `n` veces.

#### 10.1.1.8. Tuple2[A,B]

Una tupla de dos elementos. El constructor recibe como parámetro los elementos y los guarda como `val` (no puede cambiarse la referencia al objeto).

### 10.1.2. Framework

#### 10.1.2.1. DistributorQueueHybrid[T]

Esta clase implementa `TaskDistributor[R]`, cuenta con una cola de tareas, pero a diferencia de `DistributorQueue`, y como el nombre lo sugiere, utiliza `QueueHybrid`, lo que permite utilizar el *framework* cuando la cantidad de trabajo es tan grande que es conveniente guardarla en disco.

## 10.2. Ejemplo de utilización del *framework*: WordCount

En esta sección se detalla como utilizar el *framework* para contar la cantidad de repeticiones de las palabras de un texto. Este caso es frecuentemente utilizado como ejemplo de la utilización de *Map-Reduce*.

### 10.2.1. Elementos de soporte

Se definen los siguientes elementos para completar los *templates* del *framework*.

Task Esta es representada por un String que contiene una linea de texto, con múltiples palabras.

Result Este es representado por una `ArrayList[String]`. Esta contiene las diferentes palabras del String.

Acumulator Este es representado por un `HashMap[String,Long]`, que mapea de una palabra a la cantidad de repeticiones.

Callback Object Este no es utilizado, se completa el *template* con `Any`.

### 10.2.2. Context

Se crea la clase `ContextWordCount` que implementa `Context[String, ArrayList[String], HashMap[String,Long], Any]`. Se puede ver en el codigo 10.1.

Métodos importantes:

`processTask` Recibe como parámetro un string múltiples palabras, lo separa en palabras y retorna una *array* de estas.

reduceR Recibe como parámetro una `ArrayList[String]` con palabras y un `HashMap[String,Long]`, para cada palabra en la *array* incrementa el contador del *hash* (o la agrega con valor 1).

reduceA Recibe dos `HashMap[String,Long]` y los combina.

### 10.2.3. TaskDistributor

Se crea la clase `DistributorWordCount`, esta implementa `TaskDistributor[String]`. Tiene como responsabilidad leer de un archivo lineas. Se puede ver en el código 10.2.

Métodos importantes:

`nextN` Recibe como parámetro la cantidad de lineas a retornar. Y devuelve una `ArrayList[String]`, con estas. Cada elemento de la *array* es una linea con múltiples palabras.

`taskToDO()` Retorna `true` si hay lineas por leer del archivo, `false` sino.

### 10.2.4. ManagerWordCount

Se crea la clase `ManagerWordCount`, esta extiende `Manager[String, ArrayList[String], HashMap[String,Long], Any]`. Tiene como responsabilidad leer de un archivo lineas. Esta se puede ver en el código 10.3.

Métodos y propiedades importantes:

`distributor()` Esta propiedad devuelve la instancia de `DistributorWordCount` utilizada.

`contextFactory()` Retorna una nueva instancia de `ContextWordCount`.

### 10.2.5. Codigo

A continuación se incluyen los códigos de la implementación de *WordCount*. Estos pueden ser copiados manteniendo la intendentacion con Adobe Reader.

```
package wordCount;

import frameworkMR.Context;
import x10.util.HashMap;
import util.ArrayListCustom;
import frameworkMR.PlaceThread;
import x10.util.ArrayList;

public class ContextWordCount implements Context[String, ArrayList[String],
    HashMap[String,Long], Any] {
```

```
public def processTask(task:String):ArrayList[String] {
  val s = task.toLowerCase();
  val ret = new ArrayList[String]();

  var offSet:Long = 0;
  while (offSet<s.length()){
    while (offSet<s.length()&&isSplitChar(s,offSet)){
      offSet++;//the space before
    }
    val start = offSet;

    while (offSet<s.length()&&!isSplitChar(s,offSet)){
      offSet++;
    }
    ret.add(s.substring(start as Int,offSet as Int));

    while (offSet<s.length()&&isSplitChar(s,offSet)){
      offSet++;//the space after
    }
  }
  return ret;
}

private def isSplitChar(s:String, i:Long){
  val char = s.charAt(i as Int);
  return !(('a'<=char&&char<='z')||('A'<=char&&char<='Z'));
}

public def reduceR(result:ArrayList[String], acumulator:HashMap[String,Long])
  :HashMap[String, Long] {
  for (s in result){
    acumulator.put(s, acumulator.getOrElse(s,0)+1);
  }
  return acumulator;
}

public def reduceA(acumulator1:HashMap[String, Long], acumulator2:HashMap[
  String, Long])
:HashMap[String, Long] {

  val merge = (bigger:HashMap[String, Long],smaller:HashMap[String, Long])=>{
    for (val entrie in smaller.entries()){
      bigger.put(entrie.getKey(),entrie.getValue() + bigger.getOrElse(entrie.
  getKey(),0));
    }
    return bigger;
  };

  if (acumulator1.size()<acumulator2.size()){
```

```
      return merge(acumulator2, acumulator1);
    }else{

      return merge(acumulator1, acumulator2);
    }
  }

  public def callbackValidsHome(results:ArrayListCustom[ArrayList[String]],
    callObjetc:Any, token:PlaceThread):void {
  }

  public def factoryAcumulator():HashMap[String, Long] {
    return new HashMap[String,Long]();
  }


  public def init():void {
  }

  public def isValid(task:String, resultado:ArrayList[String]):Boolean {
    return true;
  }
}
```

Código 10.1: Implementación de `ContextWordCount`

```
package wordCount;

import frameworkMR.TaskDistributor;
import x10.util.ArrayList;
import frameworkMR.PlaceThread;
import x10.io.FileReader;
import x10.io.File;
import x10.io.ReaderIterator;

public class DistributorWordCount implements TaskDistributor[String] {
  val pahtToFile:String;
  var input:File;
  var reader:FileReader;

  var closed:Boolean = false;

  var iterator:ReaderIterator[String];

  public def this(path:String){
    pahtToFile= path;
```

```
  }

  public def nextN(n:Long, pt:PlaceThread):ArrayList[String] {
    val ret = new ArrayList[String](n);
    while (ret.size()<n&&taskToDO()){
      val i = iterator.next();
      if (i!=null){
        ret.add(i);
      }
    }
    return ret;
  }

  public def init(var pts:ArrayList[PlaceThread]):void {
    input = new File(pahtToFile);
    reader = new FileReader(input);
    iterator = reader.lines();
  }

  public def statusMonitor():String {
    var ret:String =  "";
    ret+="iterator.hasNext(): ";

    if (iterator!=null){
      ret+= iterator.hasNext();
    }else{
      ret+="null";
    }
    ret+="\n";
    return ret;
  }

  public def taskToDO():Boolean {
    val hn = iterator.hasNext();
    if (!hn&&!closed){
      reader.close();
      closed=true;
    }
      return hn;
  }
}
```

Código 10.2: Implementación de `DistributorWordCount`

```
package wordCount;

import frameworkMR.Manager;
import frameworkMR.Context;
```

```
import x10.util.HashMap;
import frameworkMR.TaskDistributor;
import frameworkMR.PlacesSpec;
import x10.util.ArrayList;

public class ManagerWordCount extends Manager[String, ArrayList[String],
    HashMap[String,Long], Any] {

  private val distributorWC:TaskDistributor[String];
  property objetoCallback():Any= 1;
  property batchSize():Long = 1000;
  property runLocaly() = true;
  public property hasHomeCallback():Boolean = false ;
  public property monitorUpdateInterval():Long = 1000;

  public def this (path:String, places:PlacesSpec){
    super(places);
    distributorWC = new DistributorWordCount(path);
  }

  property distributor():TaskDistributor[String] = (distributorWC) as
    TaskDistributor[String];

  protected def contextFactory():Context[String, ArrayList[String], HashMap[
    String,Long], Any] {
    return new ContextWordCount();
  }
}
```

Código 10.3: Implementación de `ManagerWordCount`