UNIVERSIDAD DE EXTREMADURA

CENTRO UNIVERSITARIO DE MÉRIDA

INGENIERO EN INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

PROYECTO FIN DE GRADO

INTEGRACIÓN DE UNA HERRAMIENTA DE CÓMPUTO EVOLUTIVO CON UNA DE PROCESAMIENTO MASIVO DE INFORMACIÓN

Autor: Daniel Lanza García

Mérida - Junio 2015

UNIVERSIDAD DE EXTREMADURA

CENTRO UNIVERSITARIO DE MÉRIDA

INGENIERO EN INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

PROYECTO FIN DE GRADO

INTEGRACIÓN DE UNA HERRAMIENTA DE CÓMPUTO EVOLUTIVO CON UNA DE PROCESAMIENTO MASIVO DE INFORMACIÓN

Autor: Daniel Lanza García

Director: Francisco Fernández de Vega

Codirector: Francisco Chávez de la O

Mérida - Junio 2015

Prólogo

En las primeras lineas que describen este proyecto fin de carrera, ...

Agradecimientos

Quiero dar mi mas sincero agradecimiento a A todos muchas gracias.

Índice general

Índice de figuras

Índice de tablas

1. INTRODUCCIÓN

1.1 Motivaciones

Muchos problemas computacionales de diferente naturaleza se intentan afrontar haciendo uso de modelos computacionales tradicionales obteniendo no muy buenos resultados, la computación evolutiva aporta un enfoque bioinspirado que en muchos casos consigue proporcionar resultados más que aceptables. Es por esto que su popularidad está en aumento, siendo aplicada esta metodología a problemas de muy diversa índole.

Con el paso de los años y la evolución de los sistemas computacionales, la computación evolutiva se ha intentado aplicar a la resolución de problemas cada vez más complejos, lo cual en la mayoría de los casos, suele conllevar el uso de más recursos como capacidad de cómputo, memoria y tiempo. Esto hace que se busquen soluciones para mejorar el uso de estos recursos.

Numerosas investigaciones se han llevado a cabo con el fin de minimizar el uso de uno de los recursos más importantes mencionados anteriormente, el tiempo. Se han diseñado algoritmos en este modelo cuya ejecución puede tomar años y que por lo tanto su tiempo de ejecución es impracticable. Varias metodologías se han aplicado para reducir el tiempo de ejecución, una de ellas es la paralelización de parte del proceso, esto puede llevarse a cabo con los procesadores de última generación los cuales poseen varios núcleos de procesamiento o también se puede conseguir con el uso de varios computadores conectados en red.

Este trabajo plantea una solución para utilizar los recursos disponibles de forma eficiente y así reducir los tiempos de ejecución, la solución que se describirá hace uso de una herramienta que explota ambas metodologías, ejecución paralela en procesadores multi-núcleo y uso de numerosas computadoras.

1.2 Objetivos

1.3 Recursos empleados

1.4 Organización del documento

2. ANÁLISIS DEL SISTEMA

En este capítulo nos sumergiremos en los dos campos de conocimiento que este proyecto contempla, la computación evolutiva y el procesamiento masivo de información. Ambos están adquiriendo cada vez más importancia en el mundo de la computación y la resolución de problemas no convencionales, a los que en la teoría de la complejidad computacional se les conoce como problemas de tiempo polinomial no determinista, o también conocidos como NP. La importancia de esta clase de problemas de decisión es que contiene muchos problemas de búsqueda y de optimización para los que se desea saber si existe una cierta solución o si existe una mejor solución que las conocidas.

Se cubrirán temas como porqué surgen estás metodologías, sus propósitos y de que manera intentan resolver el problema para el cual han sido desarrolladas.

2.1 Algoritmos evolutivos

Existen problemas computacionales que no pueden ser resueltos con técnicas tradicionales, o porque no existe una que pueda proporcionar un resultado aceptable o porque la técnica aplicada necesita de un tiempo o recursos de los que no se disponen. Así es el caso en problemas NP donde una búsqueda exhaustiva encontraría la mejor solución pero el tiempo necesario para su ejecución se hace impracticable. Para este tipo de problemas se buscan técnicas que no proporcionan siempre la mejor solución pero que intentar acercarse lo máximo posible haciendo uso de recursos razonables, a estos problemas se les conoce como problemas de optimización.

Se han desarrollado diferentes formas de afrontar estos problemas de optimización y una de ellas es la computación evolutiva, este modelo se basa en las teoría de la evolución que Charles Darwin postuló. Esta idea de aplicar la teoría Darwiniana de la evolución surgió en los años 50 y desde entonces han surgidos diferentes corrientes de investigación:

- Algoritmos genéticos, donde los individuos de la población son representados por cadenas de bits o números.
- Programación evolutiva, una variación de los algoritmos genéticos, donde lo que cambia es la representación de los individuos. En el caso de la Programación evolutiva los individuos son ternas cuyos valores representan estados de un autómata finito.

- Estrategias evolutivas, se diferencia de las demás en que la representación de cada individuo de la población consta de dos tipos de variables: las variables objeto, posibles valores que hacen que la función objetivo alcance el óptimo, y las variables estratégicas, las cuales indican de qué manera las variables objeto son afectadas por la mutación..

Este modelo por tanto, se basa generalmente en la evolución de una población y la lucha por la supervivencia. En su teoría, Darwin dictaminó que durante muchas generaciones, la variación, la selección natural y la herencia dan forma a las especies con el fin de satisfacer las demandas del entorno, con la misma idea pero con el fin de satisfacer una buena solución al problema que se plantee, surge la computación evolutiva. Podemos observar entonces, algunos elementos importantes como son:

- La población de individuos, donde cada una de ellos representa directa o indirectamente una solución al problema.
- Aptitud de los individuos, atributo que describe cuanto de cerca está este individuo (solución) de la solución óptima.
- Procedimientos de sección, es la estrategía a seguir para elegir los progenitores de la siguiente genración. Ésta normalmente elige a los individuos mas apto pero existen otras muchas técnicas.
- Procedimiento de transformación, se lleva a cabo sobre los individuos seleccionados y puede consistir en la combinación de varios individuos o en la mutación (cambios normalmente aleatorios en el individuo).

Para llevar a cabo la implementación en computadoras, se ha dividido el problema en diferentes fases y procedimientos, los cuales se ejecutan con un orden determinado, describimos a continuación de forma general como se lleva a cabo la resolución de problemas utilizando este modelo.

1. Inicialización, en esta primera fase se genera la población inicial, normalmente se genera una cantidad de individuos que es configurada y cada uno de ellos se genera de manera aleatoria, siempre respetando las restricciones que el problema imponga a la solución.
2. Evaluación, se calcula la aptitud de cada uno de los individuos de la población para poder determinar posteriormente cuales son más aptos.
3. Las fases que siguen a continuación se repiten hasta que se cumpla una de las siguientes dos condiciones: que se encuentre la solución óptima o que se alcance un límite impuesto por el programador como un número de generaciones máximo o un tiempo máximo.

    a) Selección, siguiendo la estrategia de selección de progenitores elegida, se eligen individuos de la población. Normalmente los que sean más aptos.

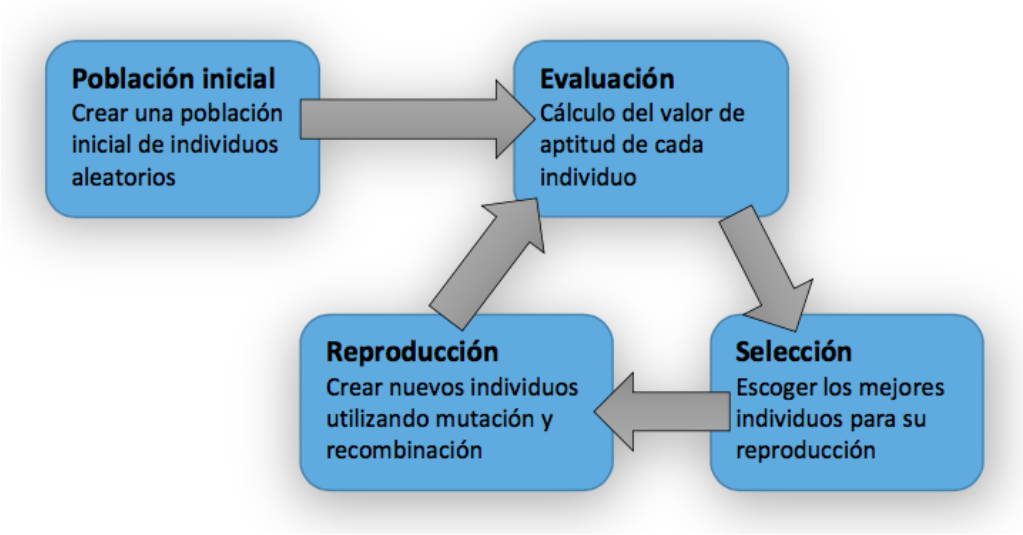

Figura 2.1: Fases del proceso evolutivo

b) Procreación, utilizando los individuos seleccionados, se combinan para generar nuevos individuos, y con ellos una nueva población (generación).

c) Mutación, a un porcentaje de los individuos recién generados se les aplica una modificación aleatoria.

d) Evaluación, todos los individuos de la nueva población son evaluados.

Con el fin de entender mejor este proceso, se muestra una imagen (ver figura 2.1) donde se observan cada una de las etapas descritas anteriormente.

Varias herramientas han surgido en la comunidad para ayudar a la investigación de este modelo, en diferentes lenguajes de programación y plataformas. En nuestro caso hemos elegido ECJ que es un framework bien conocido por la comunidad, implementado en el lenguaje de programación Java y que posee una flexibilidad importante para la ejecución de problemas de muy distinta naturaleza. Más adelante (ver apartado 3.1) se describe con más detalle esta herramienta, explicando su funcionamiento e implementación.

2.1.1 Paralelización

Como hemos comentado anteriormente (ver apartado 1.1) este proyecto surge de la necesidad de optimizar el uso de recursos cuando se intentan resolver problemas complejos con este modelo. Con este fin, y con la posibilidad de paralizar el procesamiento de algunas de las partes del proceso, surge la viabilidad de este proyecto.

Varias partes del proceso evolutivo pueden ser paralizadas, pero no todas merecen el esfuerzo ya que el coste en algunas de ellas es mínimo. Una de las fases que suele

conllevar un coste computacional alto y que su paralelización en la mayoría de problemas es sencilla, es la fase de evaluación de individuos.

Se han utilizado diferentes técnicas para este propósito [**?**][**?**][**?**][**?**], una de ellas es la ejecución de la evaluación de individuos haciendo uso de procesadores multinúcleo/multihilo. Esta técnica consigue buenos resultados pero se limita a las capacidades del procesador que posea la computadora donde se ejecute. Otro intento para llevar a cabo la paralelización del proceso ha sido la ejecución en diferentes máquinas, las cuales se conectan haciendo uso de una red. Este planteamiento requiere de una implementación más compleja y no suele explotar todos los recursos de forma eficiente, ademaás de que algunas soluciones planteadas carecen de la escalabilidad deseada. También han surgido implementaciones que hacen uso de ambas técnicas, esta solución suele ser la más apropiada ya que hace un uso más eficiente del hardware disponible, aunque requiere de una implementación aún más compleja.

Con el fin de llevar a cabo la paralelización, han surgido diferentes modelos paralelos, algunos de los cuales se describen a continuación:

- Modelo maestro-esclavo, se mantiene una población donde la evaluación del fitness y/o la aplicación de los operadores genéticos se hace en paralelo. Se implementan procesos maestro-esclavo, donde el maestro almacena la población y los esclavos solicitan parte de la población al proceso maesto para evaluarla.

- Modelo de Islas, la población de individuos se divide en subpoblaciones que evolucionan independientemente. Ocasionalmente, se producen migraciones entre ellas, permitiéndoles intercambiar información genética. Con la utilización de la migración, este modelo puede explotar las diferencias en las subpoblaciones; esta variación representa una fuente de diversidad genética. Sin embargo, si un número de individuos emigran en cada generación, ocurre una mezcla global y se eliminan las diferencias locales, y si la migración es infrecuente, es probable que se produzca convergencia prematura en las subpoblaciones.

- Modelo celular, utilizado entre otros por [**?**], consiste en colocar cada individuo en una matriz, donde cada uno sólo podrá buscar reproducirse con los individuos que tenga a su alrededor escogiendo al azar o al mejor adaptado. El descendiente pasará a ocupar una posición cercana. No hay islas en este modelo, pero hay efectos potenciales similares. Asumiendo que el cruce esta restringido a individuos adyacentes, dos individuos separados por 20 espacios están tan aislados como si estuvieran en dos islas, este tipo de separación es conocido como aislamiento por distancia. Luego de la primera evaluación, los individuos están todavía distribuidos al azar sobre la matriz. Posteriormente, empiezan a emerger zonas como cromosomas y adaptaciones semejantes. La reproducción y selección local crea tendencias evolutivas aisladas, luego de varias generaciones, la competencia local resultara en grupos más grandes de individuos semejantes.

La implementación del paralelismo de estos y otros modelos han hecho uso de tecnologías que permitan la comunicación y distribución de los diferentes procesos que llevan a cabo el proceso evolutivo, algunas de ellas se describen a continuación:

- MPI, utilizada en [**?**] y [**?**], es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

- PVM, utilizada en [**?**] y [**?**], es una biblioteca para el cómputo paralelo en un sistema distribuido de computadoras. Está diseñado para permitir que una red de computadoras heterogénea comparta sus recursos de cómputo (como el procesador y la memoria RAM) con el fin de aprovechar esto para disminuir el tiempo de ejecución de un programa al distribuir la carga de trabajo en varias computadoras.

- BOINC, utilizado en [**?**], una plataforma de computación voluntaria de propósito general para proyectos de computación distribuida, que permite compartir los recursos de las computadoras de sus contribuyentes con otros proyectos.

El interés por explorar el uso de otras tecnologías para la paralelización de los procesos evolutivos nos ha llevado a la integración de una herramienta de cómputo evolutivo con una herramienta que tiene como principal propósito el procesamiento masivo de información, algo que conlleva la distribución y paralelización de multiples tareas a lo largo de amplios clusters de computadores.

2.2 Procesamiento masivo de información

Vivimos en el momento más álgido en la genración de información, nunca antes había existido plataformas que generaran la cantidad de información que se genera hoy en día. El hecho de que del análisis de grandes cantidades de información se puedan extraer valiosos datos como estrategias de negocio, hacen que numerosas empresas y organizaciones almacenen cantidades ingentes de información para poder sacarle el máximo partido.

La generación de información se está produciendo en ámbitos muy dispares, estos pueden ser redes sociales que mantienen millones de usuarios, grandes empresas con muchos clientes, laboratorios de física con redes de millones sensores y muchos otros ejemplos que podríamos mencionar. Todos ellos queriendo extraer el máximo valor a la información que recaban.

■ **Computación distribuida**

Cuando hablamos de cantidades de informaciónón del orden de terabytes o petabytes no podemos pensar en otra cosa que no sea computación distribuida. Para una sola máquina, el tiempo que conllevaría procesar esas cantidades de datos podrían ser del orden de años.

Por tanto, la computación distribuida es la única solución con el hardware que hoy en día manejamos, que permitá analizar y manejar esas cantidades de información. Esta solución implica el uso de múltiples computadores conectados a una red común, cada uno de las cuales tiene su propio procesador, arquitectura, memoria, etc, con lo que pueden ser una composición de maquinas con diferentes configuraciones (heterogénea). En contraposición a este modelo encontramos la computación paralela en una sola máquina, que consiste en utilizar más de un hilo de procesamiento simultáneamente en un mismo procesador. Idealmente, el procesamiento paralelo permite que un programa se ejecute más rápido, en la práctica, suele ser difiícil dividir un programa de forma que CPUs separadas ejecuten diferentes porciones del programa sin ninguna interacción que deteriore esa mejora de rendimiento.

2.2.1 Modelo computacional: Map/Reduce

En este ámbito, surge la necesidad de diseñar herramientas que puedan no solo mantener esta información, si no también que tengan la capacidad de analizarla y extraer el valor que se desea de una forma distribuida y con un modelo sencillo.

Google, el buscador de internet más utilizado en el mundo, ha hecho frente ha este problema antes que nadie ya que desde hace años maneja cantidades de información realmente grandes, es por esto que sus investigaciones y experiencia son avanzadas. Varios años atrás hicieron pública [**?**] una solución para el análisis de grandes cantidades de datos de forma distribuida y con un modelo que da solución a la complejidad de dividir el problema para poder paralelizarlo, ha este modelo se le conoce como Map/Reduce. Esta publicación ha dado pie a que se implemente una herramienta que se conoce con el nombre de Hadoop [**?**], la cual se describe con más detalle en un capítulo posterior (ver apartado 3.2). La creación de esta herramienta y el hecho de que se hayan obtenidos buenos resultados de ella, ha provocado que surjan otras muchas herramientas a su alrededor, las cuales ayudan a diferentes tareas como el volcado de información (Sqoop), bases de datos para consulta (Impala), gestion de flujos de datos (Flume) y otras muchas.

La propuesta de computación distribuida que se hace en esta publicación, hace uso de un modelo computacional anteriormente conocido en la programación funcional. Este modelo se basa en aplicar dos funciones básicas conocidas como Map (mapeo) y Reduce (reducción). La función de mapeo consiste en aplicar una transformación o procedimiento a todos los datos, obteniendo así la entrada de la siguiente fase, reducción, la cual consiste en "resumir", aplicando la misma función a diferentes

partes de la salida de la fase de mapeo, obteniendo finalmente la salida deseada. Numerosas fases de mapeo y reducción pueden ser concatenadas en el orden que se quiera con el fin de producir el resultado esperado. Este modelo es el implementado en Hadoop pero con algunas peculiaridades (ver apartado **??**).

3. INTEGRANDO ECJ CON HADOOP

En te capitulo se describirá el desarrollo del proyecto, el cual ha consistido en primer lugar en analizar las dos herramientas que se integraran, ECJ y Hadoop, con el objetivo de conocerlas en profundidad y así poder integrarlas de la mejor manera posible. Una vez analizadas ambas, se explicará el procedimiento llevado a cabo para su integración y finalmente los resultados obtenidos al ejecutar la evaluación de los individuos haciendo uso de Hadoop.

3.1 Estudio de ECJ como herramienta de cómputo evolutivo

Las personas que se dedican a la investigación han apreciado desde siempre disponer de herramientas que faciliten la tarea de la investigación e implementen los algoritmos que necesitan utilizar, en este sentido surgen numerosas herramientas en el campo de la computación evolutiva, una bien conocida por la comunidad es ECJ. Esta herramienta ha sido llevada a cabo por el departamento de ciencias de la computación de la universidad de George Mason, US y ahora mismo se encuentra en su versión 22.

Esta herramienta ha sido concebida para proporcionar una amplia flexibilidad que permita albergar numerosos tipos de problemas. Además, persiguiendo este mismo objetivo, ha sido desarrollada en el lenguaje de programación Java, lo que permite que pueda ser ejecutada en cualquier sistema operativo tanto Windows como Linux.

Algunas de sus características más importantes y por las cuales ha adquirido la popularidad que posee son las siguientes:

- Facilidad de analizar la ejecución con una util implementación de logging.
- Posibilidad de generar puntos de restauración, los cuales permiten detener la ejecución y reiniciarla en otro momento.
- Ficheros de parámetros anidados, lo cual permite jerarquizar la configuración y mantener mas clara su configuración.
- Ejecución multihilo de diferentes partes del proceso, esto da la posibilidad de

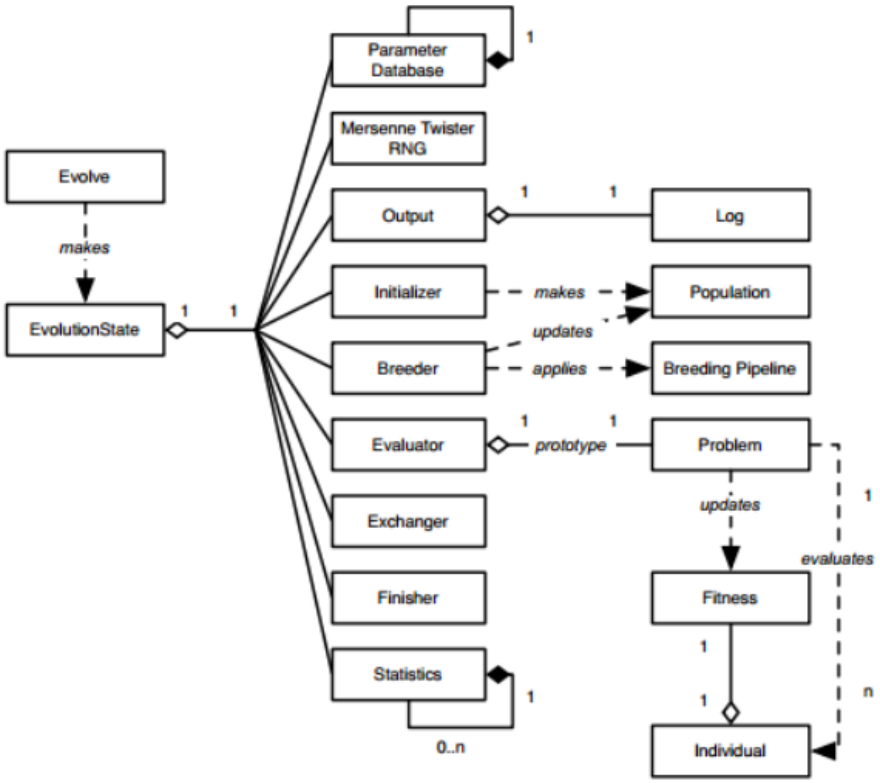

Figura 3.1: Clases que representan el proceso evolutivo en ECJ

paralizar el proceso en procesadores que posean varias unidades de procesamiento.

- Generación de numeros pseudoaleatorios de manera que se permite reproducir los resultados generados en anteriores ejecuciones.
- Soporte para diferentes técnicas evolutivas.
- Proceso de reproducción muy flexible representado por una jerarquía de operaciones.

Estas y otras características han hecho que ECJ se posicione como una de las herramientas favoritas para la investigación de la computación evolutiva. El hecho de su amplia utilización en la comunidad ha motivado el uso de ECJ en este trabajo ya que de este modo el público al que puede ir dirigido es más amplio y más grupos de investigación puedan beneficiarse de los resultados que de este se obtengan.

La ejecución de cualquier algoritmo en ECJ esta guiada a través de los ficheros de configuración que deben ser anteriormente establecidos. Estos ficheros de configuración poseen una estructura jerárquica lo que permite que un fichero de configuración pueda incluir a otro/s y sobreescribir parámetros que hayan sido establecidos. Esta estructura permite que se pueda describir un problema con pocos parámetros ya

que los que se utilicen de manera general estarán contenidos en otros que serán simplemente incluidos. En estos ficheros de configuración se incluyen numerosos parámetros que describen el comportamiento del proceso evolutivo y existen algunos de ellos que son obligatorios especificar, como por ejemplo los que determinan que clase implementa cada una de las partes de la evolución.

3.1.1 Proceso evolutivo

En la documentación de ECJ se facilita un diagrama (ver figura 3.1) que describe de forma clara como se implementa el proceso evolutivo en esta herramienta. Podemos observar que la clase que inicia el proceso evolutivo es Evolve, la cual genera un EvolutionState que es el contiene cada una de las clases/etapas del proceso. La mayor parte de estas clases deben ser especificadas en los ficheros de configuración, de esta manera ECJ sabe que clase implementa cada una de las partes.

Como se ha comentado anteriormente (ver apartado 2.1.1), la parte del proceso que suele ser más costosa es la evaluación de individuos, representada (ver figura 3.1) en ECJ por la clase Evaluator. En esta herramienta, ésta es la clase encargada de la evaluación de cada uno de los individuos de la población, lo cual suele ser lo más costoso computacionalmente y por consiguiente será la parte de ECJ donde se centre la implementación de la integración con Hadoop.

3.2 Estudio de Hadoop como herramienta de procesamiento masivo de información

Apache Hadoop es un framework de software que soporta aplicaciones distribuidas con capacidades de procesamiento de cantidades masivas de información. Permite a las aplicaciones trabajar con miles de nodos y petabytes de datos. Hadoop se inspiró en los documentos de Google para MapReduce [**?**] y Google File System (GFS) [**?**].

Hadoop es un proyecto de alto nivel Apache que está siendo construido y usado por una comunidad global de contribuyentes bastante amplia y activa, mediante el lenguaje de programación Java. Yahoo! ha sido el mayor contribuyente al proyecto, y usa Hadoop extensivamente en su negocio.

Hadoop requiere tener instalados en los nodos del clúster la versión 1.6 o superior del entorno de ejecución de Java (JRE) y SSH. Un clúster típico incluye un nodo maestro y múltiples nodos esclavos. El nodo maestro consiste en un proceso jobtracker (rastreador de trabajo), tasktracker (rastreador de tareas), namenode (nodo de nombres), y datanode (nodo de datos). Un esclavo o compute node (nodo de cómputo) consisten en un nodo de datos y un rastreador de tareas (ver figura 3.2).

Como se puede apreciar, Hadoop puede ser claramente dividido en dos partes, el

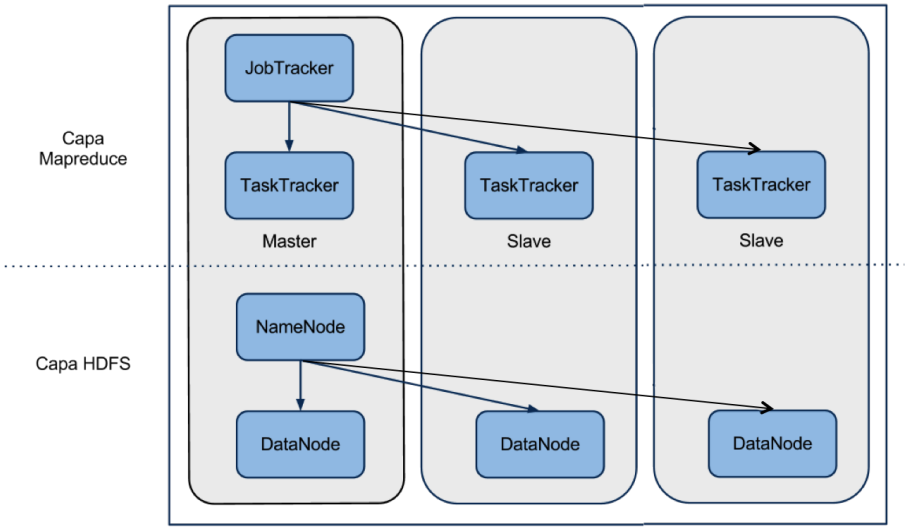

Figura 3.2: Procesos en los diferentes nodos de un cluster Hadoop

sistema de ficheros, HDFS, y la parte que implementa la parte de procesamiento de información, la capa MapReduce.

3.2.1 El sistema de ficheros

El Hadoop Distributed File System (HDFS) es un sistema de archivos distribuido, escalable y portátil escrito en Java para el framework Hadoop. Cada nodo en una instancia Hadoop típicamente tiene un único nodo de datos; un clúster de datos forma el clúster HDFS. La situación es típica porque cada nodo no requiere un nodo de datos para estar presente.

Cada nodo sirve bloques de datos sobre la red usando un protocolo de bloqueo específico para HDFS. El sistema de archivos usa la capa TCP/IP para la comunicación; los clientes usan RPC para comunicarse entre ellos. El HDFS almacena archivos grandes, a través de múltiples máquinas. Consigue fiabilidad mediante replicado de datos a través de múltiples hosts, y no requiere almacenamiento RAID en ellos. Con el valor de replicación por defecto, 3, los datos se almacenan en 3 nodos: dos en el mismo rack, y otro en un rack distinto. Los nodos de datos pueden hablar entre ellos para reequilibrar datos, mover copias, y conservar alta la replicación de datos. HDFS no cumple totalmente con POSIX porque los requerimientos de un sistema de archivos POSIX difieren de los objetivos de una aplicación Hadoop, porque el objetivo no es tanto cumplir los estándares POSIX sino la máxima eficacia y rendimiento de datos. HDFS fue diseñado para gestionar archivos muy grandes. Algo que debe ser tenido en cuenta es que no proporciona alta disponibilidad, ya que la caída de su nodo maestro supone la caída del sistema de ficheros.

Su escalabilidad y su rendimiento se pueden llevar a cabo gracias a la falta de

una de las características más comunes en un sistema de ficheros, la capacidad de modificar el contenido de los ficheros que contiene, una característica no necesaria para el propósito para el que esta diseñado.

Aunque Hadoop tiene su propio sistema de ficheros, HDFS, no esta cerrado al uso de tan solo ese sistema, otros sistemas de ficheros son también compatibles como Amazon S3, CloudStore o FTP.

3.2.2 La capa MapReduce

Aparte del sistema de archivos, está el motor MapReduce, que consiste en un Job Tracker (rastreador de trabajos), para el cual las aplicaciones cliente envían trabajos MapReduce.

El rastreador de trabajos (Job Tracker) impulsa el trabajo fuera a los nodos Task Tracker disponibles en el clúster, intentando mantener el trabajo tan cerca de los datos como sea posible. Con un sistema de archivos consciente del rack en el que se encuentran los datos, el Job Tracker sabe qué nodo contiene la información, y cuáles otras máquinas están cerca. Si el trabajo no puede ser almacenado en el nodo actual donde residen los datos, se da la prioridad a los nodos del mismo rack. Esto reduce el tráfico de red en la red principal backbone. Si un Task Tracker (rastreador de tareas) falla o no llega a tiempo, la parte de trabajo se reprograma. El TaskTracker en cada nodo genera un proceso separado JVM para evitar que el propio TaskTracker mismo falle si el trabajo en cuestión tiene problemas. Se envía información desde el TaskTracker al JobTracker cada pocos minutos para comprobar su estado. El estado del Job Tracker y el TaskTracker y la información obtenida se pueden ver desde un navegador web proporcionado por Jetty.

■ Un trabajo MapReduce

Un trabajo MapReduce aplica dos funciones, como su nombre indica, una es una función Map (una operación a cada uno de los registros de entrada) y una función Reduce (una operación que resuma la salida de la función Map) para aplicar este modelo, Hadoop divide el proceso en diferentes fases, podemos resumirlas de la siguiente manera:

1. Se divide los ficheros de entrada en tantas partes como número de tareas Map vayan a componer el trabajo.

2. Cada tarea Map lee una de las partes y aplica a cada registro de entrada una función que define el usuario. La salida de esta fase son tuplas de clave-valor.

3. La salida de cada tarea Map se organiza de forma local en el nodo que la ha ejecutado formando grupos, cada uno de ellos contiene todas las tuplas con la misma clave.

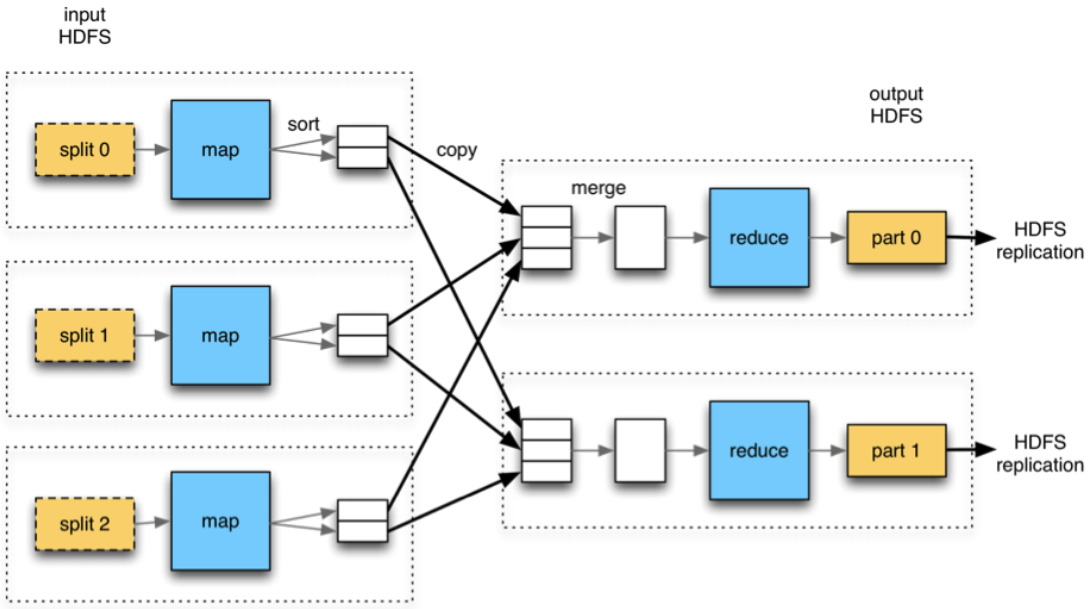

Figura 3.3: Fases de un trabajo MapReduce

4. Se transfieren todos los grupos con la misma clave al nodo que vaya a ejecutar la tarea de Reduce, de forma que las claves se entregan de manera ordenada.

5. Se unen todos los grupos recibidos con la misma clave creando así la entrada de la tarea de Reduce.

6. Se aplica la función de Reduce la cual recibe todas las tuplas con la misma clave y produce también tuplas de clave-valor.

7. La salida de la operación de Reduce se almacena en el sistema de ficheros.

Todo este proceso se ve resumido en el diagrama (ver figura 3.3) extraido de [**?**], donde se muestran cada una de estas fases.

La idea de la integración es que la entrada del trabajo de MapReduce este compuesta por todos los individuos a evaluar, de manera que la función Map sea la que evalúe cada individuo, prescindiendo de la etapa de Reduce, de manera que se escriba en el sistema de ficheros directamente la salida de la fase de Map, el resultado de la evaluación de los individuos.

3.3 Diagramas de diseño

//TODO: escribir introducción

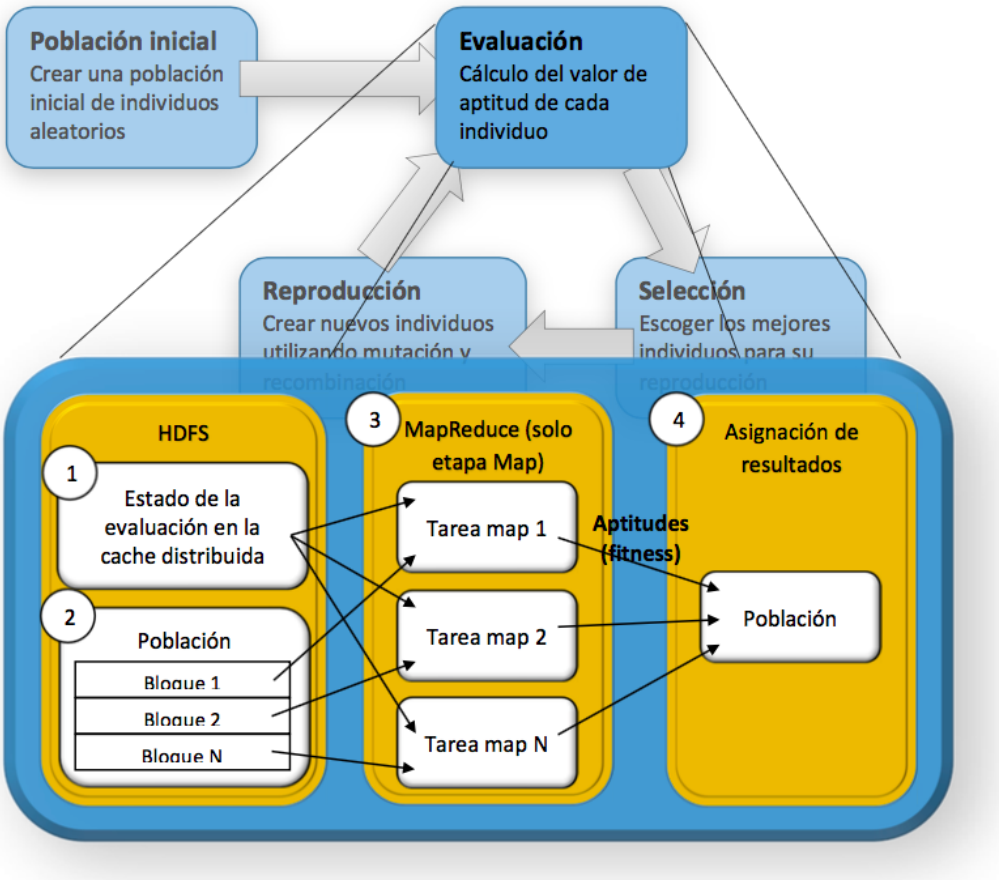

Figura 3.4: Flujo de información cuando se utiliza un trabajo para la evaluación

3.3.1 Diagrama de contexto

//TODO: incluir diagrama

3.3.2 Diagrama de flujo de datos

Con el fin de entender bien los flujos de datos que se producen en la implementación realizada se han elaborado dos diagramas de flujo que representan los dos enfoques seguidos en la integración de ECJ y Hadoop. El primero de ellos (ver figura 3.4) representa la implementación que consiste en un trabajo que evalúa toda la población, esta implementación se explica con mayor detalle más adelante (ver apartado 3.4). El segundo diagrama representa la implementación realizada para el problema de reconocimiento facial, la cual también se describe con detalle más adelante (ver apartado 4.2.1).

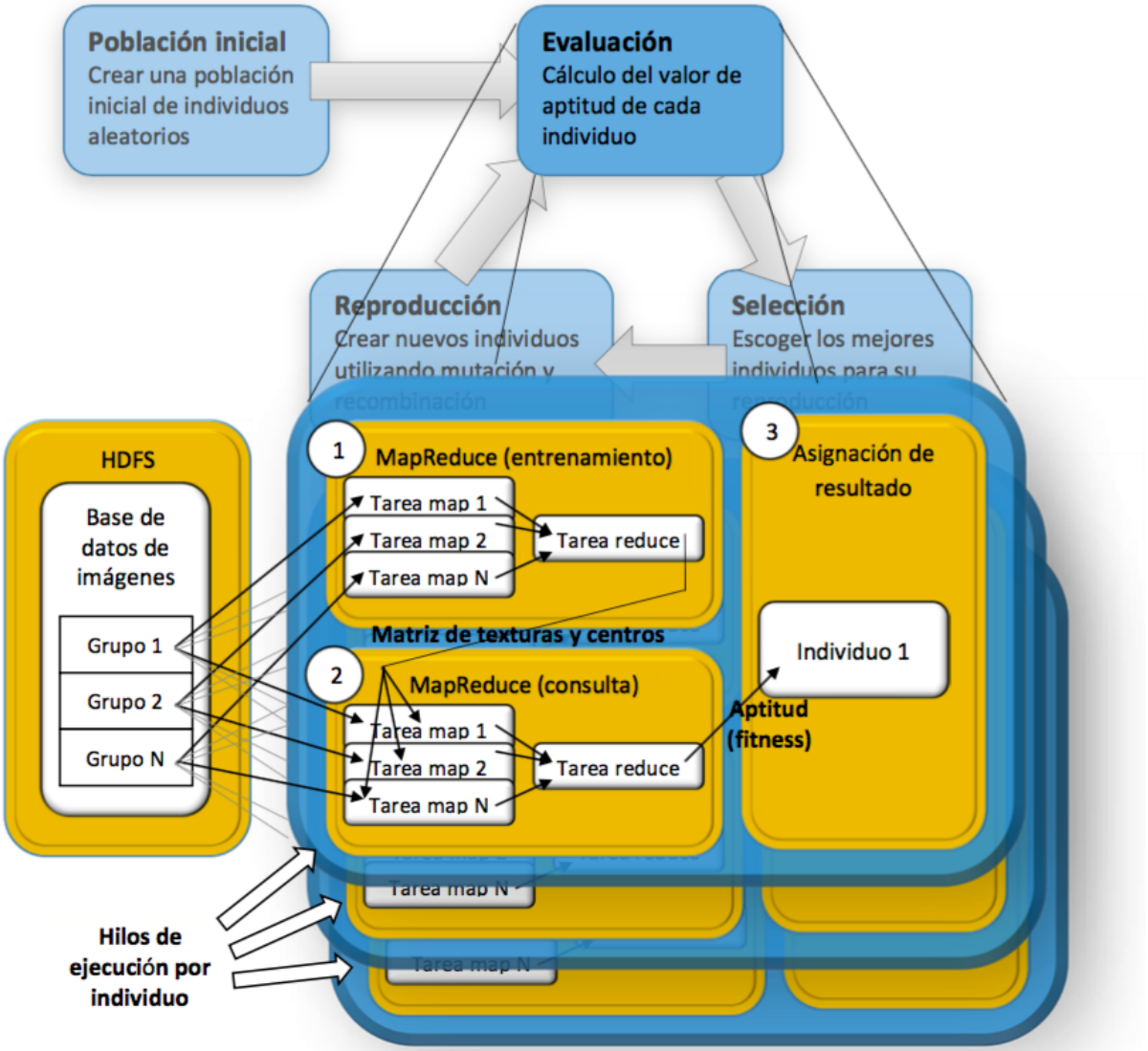

Figura 3.5: Flujo de información cuando se utiliza un trabajo para cada individuo

3.4 Implementación

La herramienta de cómputo evolutivo ECJ es conocida entre otras cosas por su flexibilidad a la hora de poder desarrollar problemas en este ámbito, es por esto que la implementación que se plantea sigue la misma idea con el objetivo de que gran parte de los problemas que pueden ser implementados con esta herramienta, puedan hacer uso de la característica que se desarrolla en este trabajo de una forma sencilla.

El propósito del trabajo es distribuir el costo computacional de la fase de evaluación de un algoritmo de computación evolutiva, es por esto que lo que debemos desarrollar es un Evaluator (ver figura 3.1) el cual lleve a cabo la evaluación de los individuos.

Para poder hacer esto y que pueda ser este evaluador establecido en cualquier problema que se plantee en la herramienta, se debe crear una clase que extienda de la clase abstracta ec.Evaluator. Al heredar de una clase abstracta debemos implementar los métodos marcados como abstractos, los de la clase ec.Evaluator se muestran a continuación.

```
1    /** Evalua el fitness de una poblacion entera. */
2    public abstract void evaluatePopulation(final EvolutionState state);
3
4    /** Debe retornar true si la ejecucion del algoritmo debe detenerse
         por algun motivo.
5    Un ejemplo es cuando se encuentra al individuo ideal */
6    public abstract boolean runComplete(final EvolutionState state);
```

En ambos métodos recibimos un objeto EvaluationState el cual contiene el estado de la evaluación, eso engloba también a la población o subpoblaciones que es en lo que nosotros estamos interesados ya que debemos obtener cada individuo para evaluarlo.

Abordaremos en primer lugar el método runComplete por su simple implementación, la cual se puede observar a continuación:

```
1    public boolean runComplete(final EvolutionState state) {
2        for (int x = 0; x < state.population.subpops.length; x++)
3            for (int y = 0; y < state.population.subpops[x].individuals.
                 length; y++)
4                if (state.population.subpops[x].individuals[y].fitness.
                     isIdealFitness())
5                    return true;
6
7        return false;
8    }
```

La implementación realizada se encarga únicamente de recorrer todos los indivi-

duos de la población (lo que conlleva recorrer cada subpoblación) y comprobar si el fitness de cada uno de los individuos es ideal haciendo uso del método .isIdealFitness(). En este caso, si alguno de los individuos posee un fitness ideal, se retorna true y si al recorrer todos los individuos ninguno es ideal, retornamos false.

3.4.1 Aclaraciones previas

Antes de entrar en detalle de la implementación del método evaluatePopulation, debemos realizar algunas consideraciones. Para realizar la evaluación de los individuos, no es solo necesario las características que definen a cada individuo (genotipo), si no también aspectos como que función de evaluación utilizar o parámetros que determinan la forma de evaluar el individuo, estos y otros aspectos necesarios están contenidos en el objeto que representa el estado de la evaluación (EvaluationState). Es por esto que en cada uno de los procesos que queramos evaluar los individuos, se debe tener acceso a este objeto, para tal propósito haremos uso de dos funciones clave que poseen ECJ y el sistema de ficheros de Hadoop (HDFS).

Para que todos los procesos tengan acceso al estado de la evaluación, haremos uso de la característica "puntos de restauración"que posee ECJ, esta característica permite serializar todo el estado del proceso evolutivo y enviarlo a través de la red o guardarlo en algún fichero, en principio esta característica se ideó con el fin de que se pueda detener el proceso y reanudarlo por donde iba cuando se desee pero en esta implementación se utilizará para distribuir el estado del problema entre las diferentes máquinas que llevarán a cabo la evaluación.

Por otro lado, utilizaremos una característica de HDFS denominada cache distribuida, esta característica permite almacenar ficheros de forma que se distribuyan por todos y cada uno de los nodos que conforman el cluster, lo cual es ideal en este caso ya que si almacenamos aquí el estado de la evolución, todos los procesos que se distribuyan podrán acceder a el.

Debemos tener en cuenta que el estado de la evaluación contiene todos los individuos de la población, lo cual no es necesario distribuir por dos motivos principales: en primer lugar porque cada nodo evaluará solo una parte de la población con lo que no necesita contener toda la población, y en segundo lugar porque la población será la entrada del trabajo de Hadoop la cual estará contenida en el sistema de ficheros (HDFS). Sabiendo ésto, se distribuirá el estado de la evaluación sin la población, esta será proporcionada a través de los ficheros de entrada al trabajo, los cuales Hadoop se encargá automáticamente de dividir y distribuir.

■ Dos modelos para dos situaciones

La implementación que se describirá a continuación aborda el problema ejecutando un trabajo de Hadoop que evalúa la población completa. Este trabajo tiene como

entrada la población de individuos, ésta es dividida y distribuida para su evaluación y finalmente se recaban estos resultados para asignar las aptitudes (fitness) a los individuos. Este modelo debe ser utilizado para problemas donde se utilizan amplias poblaciones o con individuos cuya evaluación es costosa, aunque si la evaluación es notablemente costosa y la evaluación de un solo individuo puede ser paralizada quizás debamos tener en cuenta el modelo que se explica a continuación.

Existe otro enfoque en el que podemos utilizar un trabajo de Hadoop para evaluar cada uno de los individuos. Este modelo debe ser considerado cuando la evaluación de cada individuo toma varios minutos ya que tan solo la creación de cada trabajo conlleva varios segundos (5 segundos). La idea que persigue este modelo es lanzar de forma paralela tantos trabajos como individuos haya en la población, procesando cada uno de ellos la aptitud del individuo que ha evaluado, finalmente como en el modelo anterior, se recaban los resultados para asignarselos a los individuos. Este enfoque se ha utilizado para dar solución al problema de reconocimiento facial que se estudia en el capitulo siguiente (ver apartado 4).

3.4.2 Paso 1: Distribución del estado de la evaluación

Una vez descrito el proceso necesario para la evaluación distribuida de los individuos, pasamos ahora a describir la implementación realizada del método evaluatePopulation, la cual establece varias etapas que describimos a continuación. Cada una de estas etapas se ven representadas en el diagrama de flujo de datos visto en la sección anterior (ver figura 3.4).

Con el objetivo de simplificar el desarrollo se ha implementado un cliente que lleva a cabo todas las tareas relacionadas con Hadoop:

```
1  HadoopClient hadoopClient = new HadoopClient(
2                                      hdfs_address,
3                                      hdfs_port,
4                                      jobtracker_address,
5                                      jobtracker_port);
6
7  hadoopClient.setWorkFolder(work_folder);
```

El cliente implementado se ve caracterizado por cinco aspectos, las direcciones y puertos de los procesos principales de Hadoop y el directorio sobre el que se trabajará en el sistema de ficheros (HDFS).

El primer paso que se lleva a cabo para la evaluación, es la creación y distribución del punto de restauración el cual almacena el estado de la evolución, esto se realiza en el siguiente fragmento de código:

```
1  //Extrae poblacion del estado de la evolucion
2  LinkedList<Individual[]> individuals_tmp = new LinkedList<Individual
       []>();
```

```
3     for (Subpopulation subp : state.population.subpops){
4         individuals_tmp.add(subp.individuals);
5         subp.individuals = null;
6     }
7
8     //Crear punto de restauracion
9     Checkpoint.setCheckpoint(state);
10
11    //Restaurar poblacion en el estado de la evaluacion
12    int i = 0;
13    for (Individual[] individuals : individuals_tmp) {
14        state.population.subpops[i++].individuals = individuals;
15    }
16
17    //Distribuir punto de restauracion
18    hadoopClient.addCacheFile(new File("" + state.checkpointPrefix + "." +
          state.generation + ".gz"), true, true);
```

Como se mencionaba anteriormente, la población es eliminada del punto de restauración, para ello la extraemos del estado de la evaluación, creamos el punto de restauración y posteriormente la introducimos de nuevo. Por último se distribuye el punto de restauración haciendo uso del cliente de Hadoop que utiliza la cache distribuida de HDFS para tal fin.

3.4.3 Paso 2: Creación de la entrada del trabajo

La población de individuos a evaluar conformará la entrada del trabajo de MapReduce, por lo que debemos escribir cada uno de los individuos en HDFS para que después el trabajo puede leerlos desde ahi. Para ello, desde el evaluador llamamos al método createInput del cliente de Hadoop implementado. Este método consiste en lo siguiente:

```
1     Path input_file = new Path(work_folder.concat("/input/population.seq")
          );
2     Writer writer = getSequenceFileWriter(input_file,
          IndividualIndexWritable.class, IndividualWritable.class);
3
4     Subpopulation[] subpops = state.population.subpops;
5     int len = subpops.length;
6     for (int pop = 0; pop < len; pop++) {
7         for (int indiv = 0; indiv < subpops[pop].individuals.length; indiv
              ++) {
8             if (!subpops[pop].individuals[x].evaluated){
9                 writer.append(new IndividualIndexWritable(pop, indiv),
10                                new IndividualWritable(state, subpops[pop].
                                      individuals[indiv]));
11            }
```

```
12          }
13      }
14
15      writer.close();
```

Creamos un fichero secuencial en el directorio de trabajo con el nombre "population.sql", el cual contendrá la población entera. Debemos tener en cuenta que la entrada de un trabajo de MapRedice está compuesta por tuplas de clave-valor por lo que ese será el contenido de este fichero, donde la clave está compuesta por dos números que identifican de forma inequívoca a cada individuo, lo población (pop) y el número de individuo dentro de ella (indiv), y el valor será el propio individuo que estará compuesto entre otras cosas por el genotipo.

Una vez creado el fichero se recorren todas las subpoblaciones e individuos y se añaden al fichero que contiene la población. Se debe comprobar si el individuo no está evaluado, ya que pudiera provenir de una generación anterior y eso supone que ya está evaluado y no es necesario volver a evaluarlo.

3.4.4 Paso 3: Creación y ejecución del trabajo de MapReduce (evaluación)

Una vez distribuido el estado de la evolución y generada la entrada del problema, podemos definir el trabajo de MapReduce para lanzarlo en Hadoop. Como se ha comentado anteriormente (ver apartado 3.2.2), se usará la fase de Map y no la de reduce para la evaluación de individuos, para ello debemos definir la función que se llevará a cabo en esta fase. Se dede implementar un mapper que será una clase que extienda de la clase org.apache.hadoop.mapreduce.Mapper y debe implementar el método map. En nuestro caso, el propósito es la evaluación de los individuos por lo que la definimos de la siguiente manera:

```
1   @Override
2   protected void map(IndividualIndexWritable key, IndividualWritable
        value, Context context)
3           throws IOException, InterruptedException {
4
5       Individual ind = value.getIndividual();
6
7       //Evaluar individuo
8       SimpleProblemForm problem = ((SimpleProblemForm) state.evaluator.
            p_problem);
9       problem.evaluate(state, ind, key.getSubpopulation(), 0);
10
11      //Escribir fitness
12      context.write(key, new FitnessWritable(state, ind.fitness));
13  }
```

Como podemos observar, la clave y el valor que recibimos son los mismos que contienen nuestro fichero de entrada (IndividualIndexWritable y IndividualWritable), este método map será llamado por Hadoop con cada uno de los registros de los ficheros de entrada, en nuestro caso por cada individuo. Lo que hacemos es, en primer lugar, obtener el individuo desde el valor recibido, acceder al problema que estará definido por el usuario (problem) desde el estado de la evaluación (state) y una vez adquirido el problema, evaluamos al individuo. El resultado de la evaluación (el fitness) compondrá la salida de la función map junto al identificador del individuo, que si recordamos es la clave de entrada a la función map.

Definida la función, estamos en condiciones de crear el trabajo de MapReduce e iniciarlo. Mostramos en primer lugar el procedimiento a llevar a cabo y a continuación lo explicamos.

```
1   //Creacion del trabajo
2   Job job = new Job(conf);
3   job.setJarByClass(EvaluationMapper.class);
4
5   //Configurar entrada
6   job.setInputFormatClass(SequenceFileInputFormat.class);
7   Path input_directory = new Path(work_folder.concat("/input"));
8   SequenceFileInputFormat.addInputPath(job, input_directory);
9
10  //Configurar fase Map
11  job.setMapperClass(EvaluationMapper.class);
12  job.setMapOutputKeyClass(IndividualIndexWritable.class);
13  job.setMapOutputValueClass(FitnessWritable.class);
14
15  //Configurar fase Reduce
16  job.setNumReduceTasks(0);
17
18  //Configurar salida
19  job.setOutputFormatClass(SequenceFileOutputFormat.class);
20  job.setOutputKeyClass(IndividualIndexWritable.class);
21  job.setOutputValueClass(FitnessWritable.class);
22  Path output_directory = new Path(work_folder.concat("/output"));
23  hdfs.delete(output_directory, true);
24  SequenceFileOutputFormat.setOutputPath(job, output_directory);
```

En primer lugar creamos el objeto que representa el trabajo de MapReduce, posteriormente configuramos la entrada indicando que es un fichero secuencial y el directorio de entrada. Ahora debemos definir las fases del trabajo, en primer lugar la fase Map, indicando la clase que implementa el método mostrado anteriormente (map) y en segundo lugar la fase de reduce de la cual vamos a prescindir por lo que tan solo debemos indicar que no habra ninguna tarea reduce. Por último, configuramos la salida indicando de que tipos está compuesta, los cuales coinciden con los tipos de salida del Mapper implementado, borramos el directorio de salida que contendría el resultado de la evaluación de la generación anterior y indicamos

el directorio de salida.

Una vez configurado lo inicioamos de la siguiente manera:

```
1    job.waitForCompletion(true);
```

Esto enviará el trabajo al cluster que hayamos indicado en el cliente de Hadoop y evaluará todos los individuos contenidos en los ficheros de entrada.

3.4.5 Paso 4: Asignación de resultados

El último paso es asignarle a los individuos los resultados obtenidos de forma distribuida con el trabajo en MapReduce para continuar así de forma normal el proceso de la evolución. El trabajo ha generado en el directorio de salida un conjunto de ficheros (uno por cada proceso Mapper) que contienen la salida de la función map la cual está compuesta por tuplas del identificador del usuario y el fitness calculado.

El procedimiento para recabar los resultados se lleva a cabo en el cliente de Hadoop implementado llamando al metodo readFitness y su implementación es la siguiente:

```
1    Path output_directory = new Path(work_folder.concat("/output"));
2
3    // Obtenemos todos los ficheros que produjo el trabajo
4    FileStatus[] output_files = hdfs.listStatus(output_directory);
5
6    // Establecemos los fitness calculados
7    IndividualIndexWritable key;
8    SequenceFile.Reader reader;
9    for (FileStatus output_file : output_files) {
10       reader = new SequenceFile.Reader(conf, SequenceFile.Reader.file(
             output_file.getPath()));
11
12       key = new IndividualIndexWritable();
13       while (reader.next(key)) {
14           Individual individual = state.population.subpops[key.
                 getSubpopulation()].individuals[key.getIndividual()];
15
16           //Establecemos el fitness
17           reader.getCurrentValue(new FitnessWritable(state, individual.
                 fitness));
18
19           //Marcamos individuo como evaluado
20           individual.evaluated = true;
21       }
22
23       reader.close();
24   }
```

En primer lugar, obtenemos una lista con todos los ficheros de salida producidos y a continuación recorremos cada uno de ellos. Leemos cada registro del fichero y asignamos al individuo correspondiente (obtenemos subpoblacion y numero de individuo desde la clave leída) el fitness extraído y finalmente marcamos cada individuo como evaluado.

3.5 Mejoras introducidas

Antes de llegar a la implementación descrita, se realizó una implementación menos perfeccionada. Ésta fue testada y se obtuvieron los tiempos que se pueden observar en la primera parte de la sección de resultados (ver apartado **??**). Tras estas pruebas se refinó la implementación introduciendo algunas mejoras y se realizaron configuraciones que consiguieron que la fase de evaluación se ejecutara en menos tiempo (ver apartado 3.7.1). Estas modificaciones a la primera implementación se listan a continuación:

- **Uso de la cache distribuida.** En principio el checkpoint se almacenaba en el sistema de ficheros (HDFS) desde donde se leía por todos los nodos, pero de esta manera el fichero estaba solo almacenado en algunos de los nodos del cluster (en tantos nodos como replicas de los bloques, por defecto 3). Los nodos que no contenían el fichero necesitaban mas tiempo para leerlo, ya que conllevaba transferirlo en primer lugar a través de la red. Gracias a almacenar el checkpoint en la cache distribuida, el fichero se distribuye a lo largo de todos los nodos del cluster y por consiguiente la lectura es mucho mas rápida.

- **Checkpoint sin la población.** La implementación realizada establece dos entradas para el trabajo de Hadoop, una es la población almacenada en HDFS y otra es el fichero de checkpoint el cual contiene el estado de la evolución (incluida la población). Siendo así, se ha eliminado la población del checkpoint ya que se lee desde el sistema de ficheros y de esta manera se disminuye considerablemente el tamaño del fichero, acelerando así su escritura y posterior lectura.

- **Leer checkpoint solo una vez.** El checkpoint debe ser leído por cada tarea que va a evaluar a los individuos, es por esto que se implementó su lectura en un método llamado setup() que se ejecuta una sola vez al principio de la tarea. Esto conlleva la lectura por cada tarea lanzada, si en vez de en el método setup(), implementamos la lectura en la inicialización del lector de la entrada del trabajo de Hadoop (InputReader), conseguiremos que solo sea leído una vez por cada una de las máquinas en cada trabajo.

- **Evaluar los individuos necesarios.** Debido a distintos fenómenos como el del elitismo, se puede dar el caso de que algunos de los individuos de la población ya estén evaluados (en caso del elitismo porque son individuos que provienen directamente de la generación anterior), esto hace que el fitness ya

haya sido calculado y que no sea necesario volverlos a evaluar. Los individuos que se encuentren en esta circunstancias no se incluirán en el fichero de entrada del trabajo de Hadoop.

- **Compresión de ficheros.** En el trabajo que se lanza en Hadoop, hay ficheros tanto de entrada (la población) como de salida (resultados de la evaluación) y ambos pueden tener un tamaño considerable si contemplamos poblaciones de millones de individuos. Estos ficheros pueden ser comprimidos con el objetivo de ser transferidos mas rápidamente a través de la red. Los ficheros de entrada deben ser transferidos desde el cliente (en este caso ECJ) hasta el sistema de ficheros, HDFS, y los ficheros de salida desde HDFS hasta el cliente para asignar el resultado, esto hace que el uso de la compresión acelere los tiempos de transferencia y por consiguiente mejore el tiempo de evaluación.

- **Buffers aumentados.** Como se mencionaba en el punto anterior, los ficheros pueden llegar a ser de tamaños considerables por lo que es conveniente que los buffers encargados de las transferencias a través de la red tengan tamaños grandes, acordes al tamaño del fichero a transferir. Es por esto que el buffer para crear la entrada del trabajo como el encargado de recibir los resultados han sido aumentados.

- **Reuso de las máquinas virtuales de Java.** Hadoop está implementado en Java y por consiguiente, hace uso de maquinas virtuales de Java (JVM por su siglas en inglés) para su ejecución. Cada una de las tareas que se lanzan en los nodos que componen el cluster conlleva la creación y destruciión de la JVM. Para tareas donde la ejecución conlleva varios minutos, el tiempo necesario para crear y destruir la JVM puede ser despreciable, pero si las tareas toman pocos segundos puede tener importancia. Hadoop puede ser configurado para que no se cree una JVM por cada tarea, de manera que son reutilizadas para otras tareas tantas veces como se configure.

- **Serialización de los ficheros de entrada y salida.** En la primera implementación realizada, los ficheros tanto de la población (la entrada) como los que contienen los resultados de fitness (salida), eran escritos en formato texto. Los ficheros en formato texto requieren para su lectura y escritura que la información sea parseada a sus correspondientes tipos y además suelen ocupar mas que los ficheros binarios. Es por esto que se decidió realizar la implementación para que ambos ficheros sean binarios, facilitando así la escritura y lectura de los datos y además consiguiendo que los mismos reduzcan su tamaño.

3.6 Despliegue de problemas utilizando la implementación

Describimos en esta sección cual es el procedimiento a seguir para desplegar en ECJ dos problemas bien conocidos en el mundo de la computación evolutiva. En

primer lugar describimos como ejecutar el algoritmo genético MaxOne, y en segundo lugar el algoritmo de programación genética Parity. En ambos se hace inicialmente un despliegue que no hace uso de la integración con Hadoop y finalmente se explica como de una manera sencilla puede ser configurado para que la evaluación se haga de forma distribuida en un cluster Hadoop.

3.6.1 MaxOne

Para probar la implementación realizada, se ha configurado un problema sencillo que hace uso de la evaluación en un cluster Hadoop. El problema configurado se conoce con el nombre MaxOne y tiene como objetivo la construcción de una cadena de 1s, cada individuo esta representado por un conjunto de 1s y 0s teniendo todos los individuos una cadena de la misma longitud. La función de fitness es tan sencilla como contar el número de 1s en la cadena, el individuo ideal será aquel que su cadena contenga solo 1s.

Si quisiéramos ejecutar este problema con ECJ, sin hacer uso de la integración con Hadoop, debemos definir dos cosas. En primer lugar una clase que representa el problema, la cual en este caso tan solo implementa la función de evaluación, y en segundo lugar el fichero de configuración de ECJ.

A continuación mostramos una posible implementación de la clase que representa el problema:

```
1  public class MaxOnes extends Problem implements SimpleProblemForm {
2      public void evaluate(final EvolutionState state, final Individual ind,
           final int subpopulation, final int threadnum) {
3          int sum = 0;
4
5          BitVectorIndividual bv_ind = (BitVectorIndividual) ind;
6
7          //Contamos numero de 1s
8          for (int x = 0; x < bv_ind.genome.length; x++)
9              sum += (bv_ind.genome[x] ? 1 : 0);
10
11         //Establecemos el fitness
12         ((SimpleFitness) ind2.fitness).setFitness(state,
13                 (float) (((double) sum) / bv_ind.genome.length),
14                 sum == bv_ind.genome.length); // es el individuo ideal?
15
16         //Indicamos que ha sido evaluado
17         bv_ind.evaluated = true;
18     }
19 }
```

La imlementacion es tan básica como convertir el individuo al tipo que le corresponde, BitVectorIndividual, contar el número de bits a true, establecer el fitness

indicando si es el individuo ideal o no y por último marcarlo como evaluado.

Una vez hecho esto debemos definir el fichero de configuración (con un nombre como config.params), el cual puede contener algo como lo siguiente:

```
1  //Numero de hilos a usar para evaluacion y reproduccion
2  breedthreads   = 1
3  evalthreads = 1
4
5  //Semilla de aleatoriedad
6  seed.0     = 4357
7
8  //Clases a utilizar para cada fase del proceso de evolucion
9  state      = ec.simple.SimpleEvolutionState
10 pop    = ec.Population
11 init       = ec.simple.SimpleInitializer
12 finish     = ec.simple.SimpleFinisher
13 breed  = ec.simple.SimpleBreeder
14 eval       = ec.simple.SimpleEvaluator
15 stat       = ec.simple.SimpleStatistics
16 exch       = ec.simple.SimpleExchanger
17
18 //Numero de generaciones maximo
19 generations    = 200
20 quit-on-run-complete  = true
21 checkpoint     = false
22
23 //Definimos una subpoblacion de BitVectorIndividuals de 200 bits y una
       reproduccion con seleccion por torneo de 2 individuos
24 pop.subpops    = 1
25 pop.subpop.0      = ec.Subpopulation
26 pop.subpop.0.size     = 10
27 pop.subpop.0.duplicate-retries  = 0
28 pop.subpop.0.species     = ec.vector.BitVectorSpecies
29 pop.subpop.0.species.fitness  = ec.simple.SimpleFitness
30 pop.subpop.0.species.ind = ec.vector.BitVectorIndividual
31 pop.subpop.0.species.genome-size = 200
32 pop.subpop.0.species.crossover-type = one
33 pop.subpop.0.species.crossover-prob = 1.0
34 pop.subpop.0.species.mutation-prob = 0.01
35 pop.subpop.0.species.pipe       = ec.vector.breed.VectorMutationPipeline
36 pop.subpop.0.species.pipe.source.0  = ec.vector.breed.
       VectorCrossoverPipeline
37 pop.subpop.0.species.pipe.source.0.source.0 = ec.select.
       TournamentSelection
38 pop.subpop.0.species.pipe.source.0.source.1 = ec.select.
       TournamentSelection
39 select.tournament.size    = 2
40
41 //Indicamos la clase que define el problema (contiene la funcion de
```

```
    evaluacion)
42 eval.problem      = ec.app.tutorial1.MaxOnes
```

Una vez definidos estamos ficheros, podríamos ejecutarlo de forma local compilando el código fuente y ejecutando el siguiente comando:

```
[usu@host src]$ java ec.Evolve -file config.params
```

De este modo ejecutaríamos el un algoritmo evolutivo de forma local el cual terminará su ejecución en cuanto encuentre a un individuo idea o alcance la generación 200.

Para este problema la ejecución es rápida, quizás un minuto o dos, ya que la función de evaluación es sencilla, y son pocos individuos, pero y si la evaluación fuera costosa o tuviéramos millones de individuos en la población? pues si disponemos de acceso a un cluster Hadoop podemos cambiar una linea y la evaluación se ejecutará de forma distribuida haciendo uso de todos los recursos disponibles del cluster de manera que el tiempo de ejecución se reduzca notablemente. Esto podríamos conseguirlo cambiando el evaluador en el fichero de configuración, anteriormente lo establecimos a ec.simple.SimpleEvaluator (linea 14), si lo establecemos a ec.hadoop.HadoopEvaluator se ejecutará haciendo uso de la implementación realizada, por lo que los individuos serán evaluados en todos los nodos del cluster Hadoop. Esta configuración ha sido probada obteniendo los mismos resultados que con la ejecución local.

3.6.2 Parity

En esta sección vamos a configurar un sencillo problema de programación genética conocido por la comunidad por el nombre de Parity, el objetivo de este problema de programación genética es encontrar un programa que produzca un valor de la paridad par booleana dadas n entradas booleanas independientes. Por lo general, se utilizan 6 entradas booleanas (Parity-6), y el objetivo es que coincida con el valor del bit de paridad para cada una de las entradas de 2 elevado a $6 = 64$ posibles.

Si queremos ejecutar este problema en ECJ debemos definir 3 cosas, una clase que represente el problema y la cual pueda evaluar los individuos, otra clase que represente la paridad y ayude a la transferencia de esta entre individuos de la población y por último el fichero de parámetros de ECJ.

Comentamos en primer lugar la implementación de la clase que representa la paridad, utilizada en este caso para transferir un 0 o un 1 dependiendo de la paridad calculada.

```
public class ParityData extends GPData {
    // Valor de retorno, debe ser siempre 1 o 0
    public int x;

    public void copyTo(final GPData gpd) {
        ((ParityData) gpd).x = x;
    }
}
```

La clase que representa el problema, la cual lleva a cabo la evaluación de los individuos, se muestra a continuación.

```
public class Parity extends GPProblem implements SimpleProblemForm {
    public int numBits;
    public int totalSize;
    public int bits; // data bits

    public void setup(final EvolutionState state, final Parameter base) {

        //Obtnemos el numero de bits a utilizar desde el fichero de
            parametros
        numBits = state.parameters.getIntWithMax(base.push(P_NUMBITS), null
            , 2, 31);

        //Calculamos la combinacion ma\'xima
        totalSize = 1;
        for (int x = 0; x < numBits; x++)
            totalSize *= 2;
    }

    public void evaluate(final EvolutionState state, final Individual ind,
            final int subpopulation, final int threadnum) {

        //Aqui se almacenara el valor de paridad retornado por el programa
            gen\'etico
        ParityData input = (ParityData) (this.input);

        int sum = 0;
        //Recorremos cada combinacion a probar
        for (bits = 0; bits < totalSize; bits++) {
            //Comprobamos si es par o impar
            int tb = 0;
            for (int b = 0; b < numBits; b++)
                tb += (bits >>> b) & 1;
            tb &= 1; // now tb is 1 if we're odd, 0 if we're even

            //Ejecutamos el programa genetico que representa al individuo
            ((GPIndividual) ind).trees[0].child.eval(state, threadnum,
                input, stack, ((GPIndividual) ind), this);
```

```
34
35              //Si coinciden sumamos 1
36              if ((input.x & 1) == tb)
37                  sum++;
38          }
39
40          //Establecemos el fitness en funcion del numero de aciertos
41          KozaFitness f = ((KozaFitness) ind.fitness);
42          f.setStandardizedFitness(state, (float) (totalSize - sum));
43          f.hits = sum;
44      }
45  }
```

En primer lugar en esta implementación se ejecuta el método setup el cual obtiene el parámetro del número de bits a utilizar en el problema y posteriormente se calcula lel número de combinaciones con ese número de bits. Respecto a la evaluación de los individuos en el método evaluate, en primer lugar se crea un objeto que representa el valor devuelto por la ejecución del programa genético, posteriormente se recorren todos los valores posibles y por cada uno se calcula la paridad del valor y se ejecuta el programa genético, si ambos coinciden en el valor devuelto se incrementa el contador (sum). Finalmente se establece el fitness del individuo en función del número de aciertos.

Tras definir las dos clases anteriores, lo único que queda es el fichero de parámetros, el cual podemos nombrar parity.params y cuyo contenido puede ser el siguiente:

```
1  # Heredamos parametros basicos desde otro fichero
2  parent.0 = ../../gp/koza/koza.params
3
4  # Definimos todas las funciones a utilizar
5  gp.fs.size = 1
6  gp.fs.0.name = f0
7  gp.fs.0.func.0 = ec.app.parity.func.And
8  gp.fs.0.func.0.nc = nc2
9  gp.fs.0.func.1 = ec.app.parity.func.Or
10 gp.fs.0.func.1.nc = nc2
11 gp.fs.0.func.2 = ec.app.parity.func.Nand
12 gp.fs.0.func.2.nc = nc2
13 gp.fs.0.func.3 = ec.app.parity.func.Nor
14 gp.fs.0.func.3.nc = nc2
15 # Definimos tantas como queramos
16 ...
17
18 # Definimos el problema
19 eval.problem = ec.app.parity.Parity
20 eval.problem.data = ec.app.parity.ParityData
21
22 # Numero de bits a utilizar
23 eval.problem.bits = 12
```

```
24 gp.fs.0.size = 16 # = eval.problem.bits + 4 para este problema
```

Con el objetivo de no repetir parámetros que suelen utilizarse con mucha frecuencia, heredamos desde un fichero de pararemos que contiene los más usuales y sobreescribimos los que especifican el problema concreto, esto lo hacemos en la primera linea del fichero de parámetros con el parámetro parent y la ruta al fichero de parámetros que queremos heredar. A continuación se definen las funciones que pueden componer los programas genéticos producidos, indicamos el problema con las clases que hemos generado anteriormente y por último definimos el número de bits a utilizar en el problema.

Una vez definidos los ficheros necesarios, al igual que en el problema MaxOne, podemos ejecutarlo si compilamos el código y ejecutamos el siguiente comando:

```
[usu@host src]$ java ec.Evolve -file parity.params
```

La configuración actual hace uso del evaluador definido en el fichero "../../gp/koza/koza.params"del cual hereda y corresponde a ec.simple.SimpleEvaluator, este evaluador realizara una evaluación secuencial de los individuos pero si lo establecemos a ec.hadoop.HadoopEvaluator se ejecutará en el cluster Hadoop. Para hacer esto podemos añadir la siguiente linea al fichero de parámetros:

```
1 eval = ec.hadoop.HadoopEvaluator
```

Esto sobreescribirá el valor del evaluador establecido en el fichero que heredamos y por consiguiente se hará uso del evaluador de Hadoop.

Mas adelante (ver apartado 3.7.1) se muestran los beneficios de utilizar esta solución, donde vemos que los tiempos se reducen considerablemente si lo comparamos con una ejecución secuencial.

3.7 Resultados

Analizamos ahora los resultados de la integración realizada centrándonos en uno de los problemas anteriores, Parity. Este problema engloba las circunstancias de ambos ya que el coste computacional de la evaluación en estos problemas es despreciable. Analizamos los tiempos obtenidos en la ejecución de este problema de programación genética, donde se evoluciona un programa hasta que se consigue generar una función de paridad que retorne el bit de paridad correcto en todas las combinaciones posibles.

3.7.1 Millones de individuos: Parity

Para los problemas donde el coste de la evaluación es despreciable, la paralelización de esta fase cobra sentido cuando la cantidad de individuos a evaluar es importante. ECJ posee un evaluador que permite paralelizar la evaluación de los individuos de manera que se use los diferentes núcleos de procesamiento que posea la máquina en la que se ejecute, de manera que podemos comparar los tiempos obtenidos con una ejecución secuencial y la multihilo. En la tabla que se muestra a continuación, mostramos tiempos en segundos que toma la etapa de evaluación en diferentes ejecuciones con diferente número de individuos y utilizando ejecuciones multihilo, estos tiempos corresponden a ejecuciones del problema Parity y la máquina donde se ejecutó disponía de 8 núcleos de procesamiento.

Número de individuos	Secuencial	2 hilos	4 hilos	8 hilos
30.000	19	10	5	3
50.000	31	16	8	4
100.000	65	32	16	8
300.000			47	25
500.000			82	41
1.000.000				87
1.500.000				130

Tabla 3.1: Tiempos de ejecución secuencial y mulhilo

Observamos claramente como los tiempos son directamente proporcionales al número de individuos e indirectamente proporcionales al nḿuero de hilos que utilicemos para la evaluación. Como se puede apreciar con ejecuciones de millones de individuos, incluso utilizando 8 hilos de procesamiento, nos acercamos a tiempos del orden de minutos, teniendo en cuenta que esto debemos hacerlo por generación, la evaluación de los individuos empieza a ser un problema.

■ Resultados sin mejoras

Abordamos ahora una ejecución utilizando la implementación realizada, de manera que los individuos se evalúen a lo largo de un cluster utilizando cada una de las máquinas disponibles y los núcleos de procesamiento de cada una. Las ejecuciones realizadas hicieron uso de un cluster de 7 máquinas interconectadas donde se encuentra desplegada la herramienta Hadoop.

Como se mencionó anteriormente (ver apartado 3.5), se hizo una implementación inicial y posteriormente se incluyeron mejoras. Los tiempos que se muestran (ver tabla 3.2) son los obtenidos con la implementación inicial.

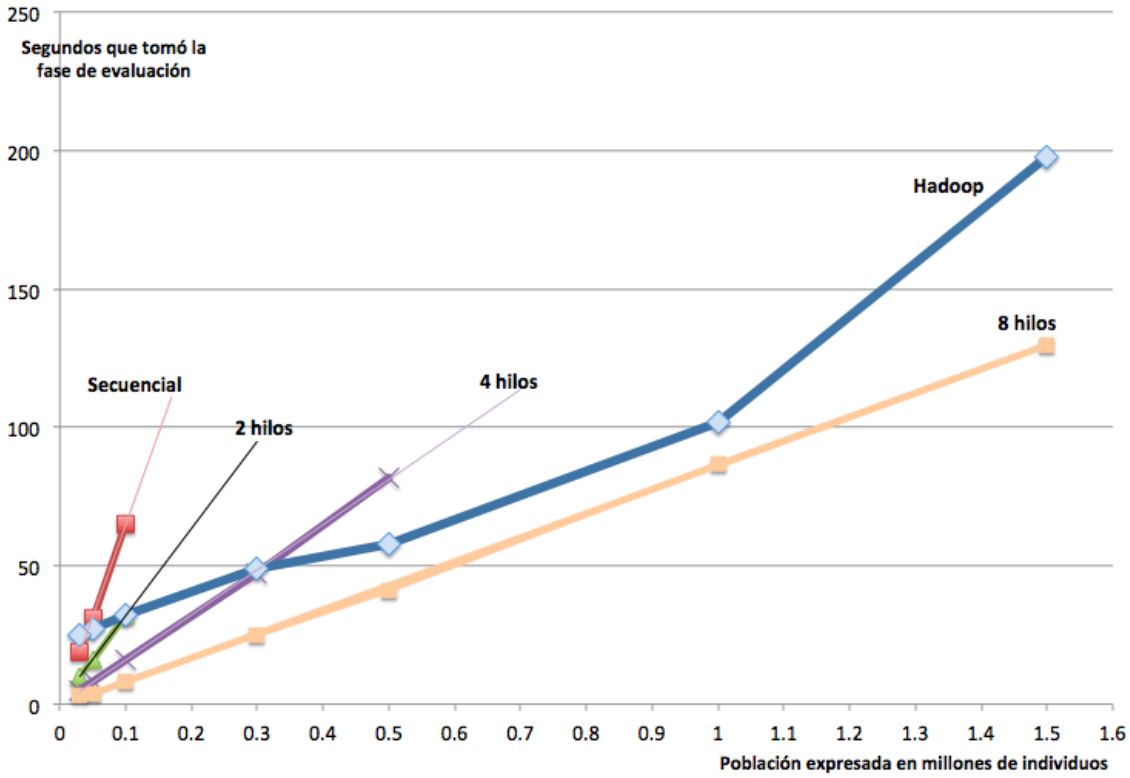

Figura 3.6: Comparación de tiempos de evaluación sin y con la integración (sin mejoras)

Número de individuos	Tiempo de evaluación (segundos)
30.000	25
50.000	27
100.000	32
300.000	49
500.000	58
1.000.000	102
1.500.000	198

Tabla 3.2: Tiempos de ejecución utilizando la integración con Hadoop sin los mejoras

Se muestra una gráfica (ver figura 3.6) donde se observan los tiempos de la tabla anterior comparados con los tiempos de las diferentes ejecuciones sin la integración con Hadoop.

Observamos que si lo comparamos con una ejecución secuencial (línea roja), el uso de la implementación realizada mejora los tiempos con pocos miles de individuos, sin embargo, necesita alcanzar los 100.000 individuos para mejorar los tiempos de la ejecución con 2 hilos y hasta los 300.000 individuos para mejorar los de la ejecución con 4 hilos. Por otro lado observamos que los tiempos de la ejecución con 8 hilos nunca son alcanzados, llegando incluso a empeorar notablemente con una ejecución

de un millón y medio de individuos.

■ Resultados con mejoras

Analizamos ahora los tiempos tras la introducción de las mejoras descritas (ver apartado 3.5), estos tiempos se muestran a continuación (ver tabla 3.3).

Número de individuos	Tiempo de evaluación (segundos)
30.000	19
50.000	20
100.000	24
300.000	31
1.000.000	65
1.500.000	92
2.000.000	108
2.500.000	130
3.000.000	141

Tabla 3.3: Tiempos de ejecución utilizando la integración con Hadoop con los mejoras

Al igual que lo hicimos con los resultados sin las mejoras, comparamos en una gráfica estos tiempos con la ejecución secuencial y con hilos (ver figura 3.7).

Ahora observamos como los tiempos obtenidos ejecutando la fase de evaluación con Hadoop han mejorado notablemente, disminuyendo los tiempos de la ejecución secuencial, 2 y 4 hilos con un tamaño de la población no muy elevado, llegando a mejorar los tiempos de la ejecución con 8 hilos con una población no superior a los 500.000 individuos.

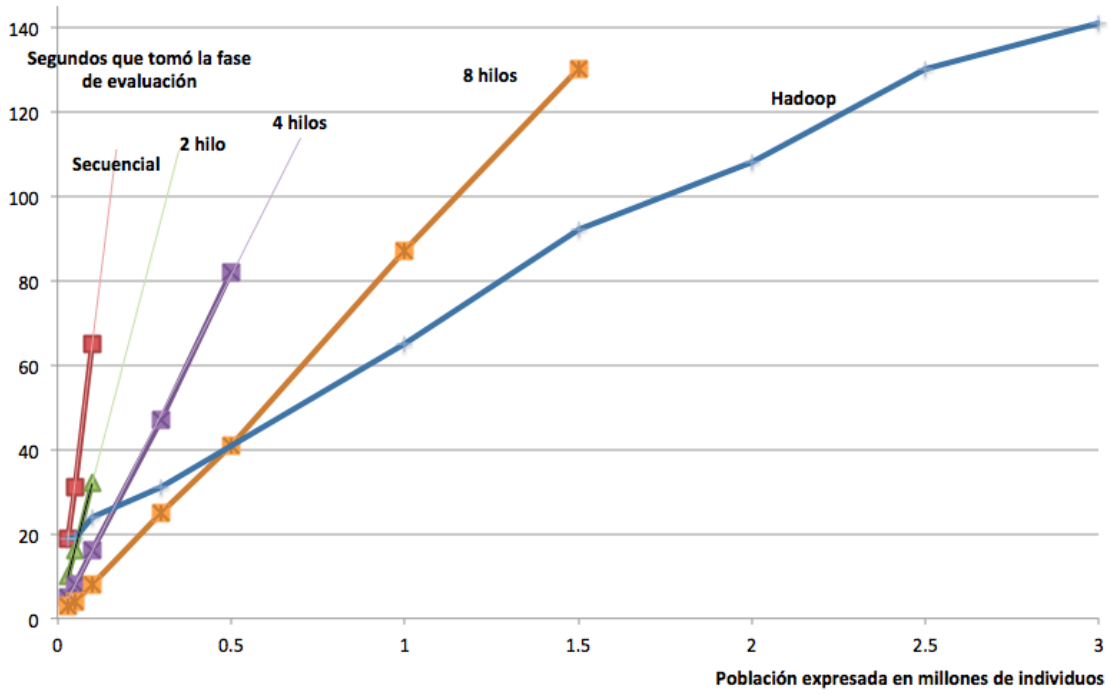

Figura 3.7: Comparación de tiempos de evaluación sin y con la integración (con mejoras)

4. RECONOCIMIENTO FACIAL MEDIANTE ALGORITMOS EVOLUTIVOS MASIVAMENTE PARALELOS

El reconocimiento facial se utiliza en numerosas aplicaciones hoy en día tales como sistemas de seguridad, sistemas de identificación de personas, localización, etc. Las soluciones planteadas conllevan un coste computacional alto ya que el tratamiento de imágenes y la extracción de características son tareas costosas para un computador. En este capítulo describimos la aplicación a este problema real y costoso la integración entre ECJ y Hadoop, esta integración reducirá los tiempos de procesamiento notablemente mediante la paralelización utilizando Hadoop y proporcionará un enfoque evolutivo con el cual mejorar la efectividad del algoritmo de reconocimiento facial.

Como se ha comentado en varias ocasiones en este documento, la integración de estas dos herramientas cobra sentido cuando el coste computacional es alto y esto se puede dar en dos situaciones, una en la que el tamaño de la población sea elevado, lo cual no es usual en las poblaciones utilizadas con algoritmos evolutivos y otra en la que la evaluación de los individuos sea realmente costosa, este es el caso en el problema de reconocimiento facial.

4.1 Descripción del problema

Surgen numerosas investigaciones que afrontan el problema de reconocimiento facial, un intento por resolver el problema se describe en [**?**] donde se presenta un sistema de clasificación de rostros haciendo uso de técnicas de clasificación no supervisadas, apoyándose en el análisis de textura local con una técnica CBIR (Content Image Based Retrieval), por medio de la extracción de la media, la desviación estándar y la homogeneidad sobre puntos de interés de la imagen.

En este sistema se pueden diferenciar claramente dos fases, una es la fase de entrenamiento donde el sistema adquiere el conocimiento necesario para que posteriormente en la fase siguiente, recuperación, se pueda clasificar cada imagen (indicar a que persona pertenece) de forma correcta. El tiempo que toma ambas fases puede

ser del orden de 4 minutos, estos tiempos son ya bastante reducidos gracias a que la implementación realizada hace uso de una librería de procesamiento de imágenes llamada OpenCV [**?**]. La librería OpenCV consigue realizar procesamiento sobre matrices (las imágenes pueden ser tratadas como matrices) con una eficiencia notable.

■ Base de datos de imágenes

Antes de describir en detalle el proceso, explicamos en que consiste la base de datos de imágenes que utiliza este algoritmo como entrada. La base de datos está formada por imágenes en condiciones ideales y cada una de ellas contiene información de diferentes puntos de interés situados en la imagen, todas tienen el mismo número de puntos de interés y corresponden cada uno de ellos a la misma parte del rostro. Así, por ejemplo, el punto número 67 de cada imagen corresponde a la punta de la nariz. Un ejemplo de localización de los puntos se puede observar en la imagen del rostro que contiene cada uno de los puntos de interés (ver figura 4.2).

4.1.1 Fase de entrenamiento

En la primera fase llevada a cabo por el sistema de reconocimiento facial, se siguen diferentes pasos para obtener el conocimiento suficiente para poder posteriormente realizar la fase de consulta. Estos pasos se ven resumidos en el diagrama (ver figura 4.1) y son más detalladamente explicados a continuación.

1. **Lectura de la imagen**. Se encarga de extraer las imágenes de la base de datos que serán procesadas. La lectura de la imagen da como resultado una imagen en tres capas, correspondientes al espacio de color RGB.

2. **Acondicionamiento.** Realiza el procesamiento previo de la imagen en cada una de las capas, ya que no se puede trabajar directamente con una imagen recién adquirida, esta debe pasar por la etapa de acondicionamiento para que sea eliminada cualquier impureza generada, tanto por el dispositivo de adquisición, la mala iluminación, el fondo, etc. Una vez eliminadas todas las impurezas, la imagen está lista para su correcto análisis.

3. **Pasar al espacio de color HSI.** El espacio de color RGB no nos proporciona la información necesaria o al menos no lo suficientemente clara para la extracción de características, por lo que la imagen RGB se debe pasar al espacio de color HSI. Este espacio de color nos proporcionará la información que necesitamos saber acerca de la imagen, como es la textura, luminosidad, saturación de color, etc. información mas útil que la que nos proporciona el espacio de color RGB, al final de este proceso tendremos nuevamente la imagen dividida en tres capas pero ahora en el espacio de color HSI.

4. **Extracción de características.** Este bloque es el mas importante ya que es

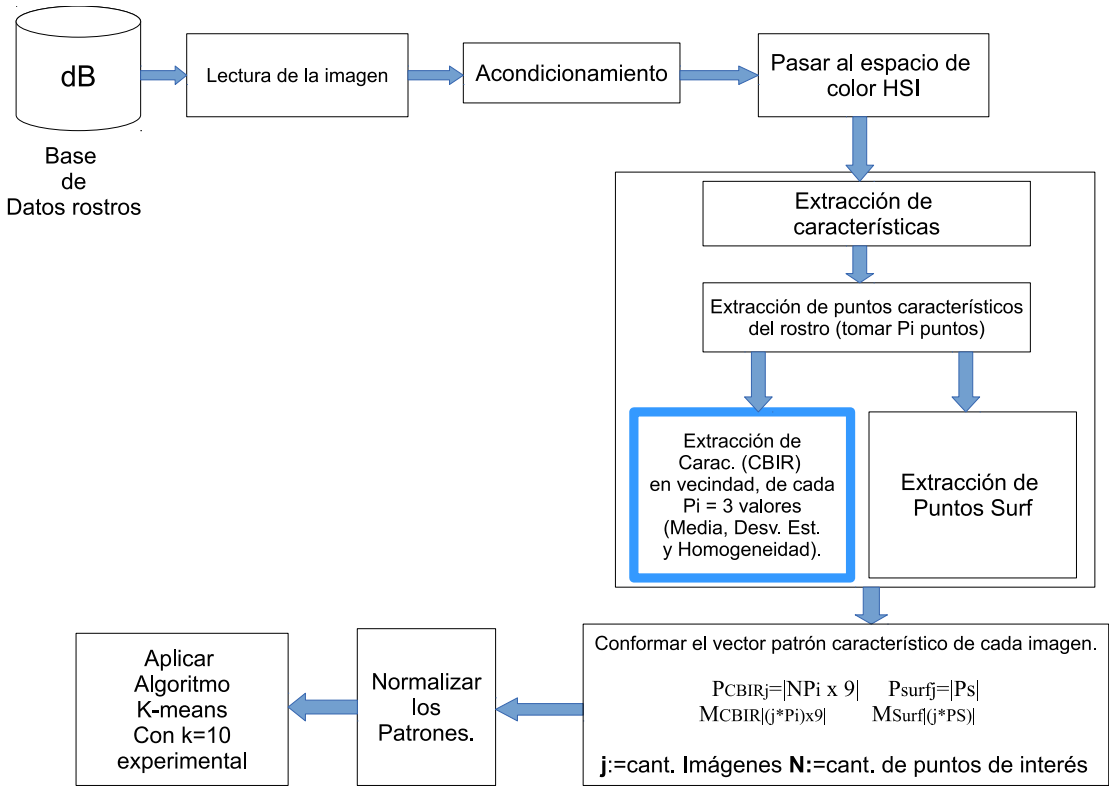

Figura 4.1: Arquitectura de la fase de entrenamiento

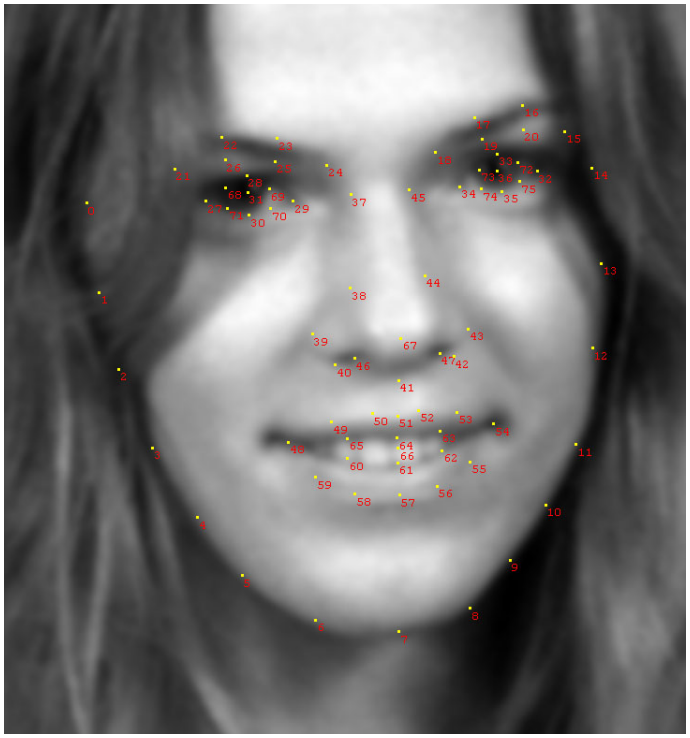

Figura 4.2: Ejemplo de puntos de interés sobre un rostro

donde implementaremos la mayor parte del trabajo. Consta de tres etapas:

a) **Extracción de los puntos de interés.** Las bases de datos con las que se trabaja tienen los puntos de interés marcados de manera manual, la localización de los puntos de interés está contenida dentro de un documento de texto en el que cada línea describe los puntos de interés para cada una de las imágenes, por lo que nosotros debemos extraer los puntos de inteés con los que trabajará nuestro sistema.

b) **Extracción de los puntos Surf (Speeded-Up Robust Features).** Este procedimiento viene implementado en la biblioteca de procesamiento digital de imágenes de OpenCV. Es un algoritmo de visión por computador, capaz de obtener una representación visual de una imagen y extraer una información detallada y específica del contenido. Esta información es tratada para realizar operaciones como por ejemplo la localización y reconocimiento de determinados objetos, personas o caras, realización de de escenas 3D, seguimiento de objetos y extracción de puntos de interés. Este algoritmo forma parte de la mencionada inteligencia artificial, capaz de entrenar un sistema para que interprete imágenes y determine el contenido. El Algoritmo SURF se presentó por primera vez por Herbert Bay en ECCV 9.ª conferencia internacional de visión por computador celebrada en Austria en Mayo de 2006 [**?**].

c) **Extracción de los rasgos estadísticos.** Los rasgos estadísticos son generados para cada uno de los puntos de interés, de cada una de las capas del espacio de color HSI. A partir de un punto de interés determinado en un pixel se genera una ventana p de tamaño Pi + p x Pi + p píxeles con el punto de interés en el centro, dentro de esta ventana (vecindad) se extraerán tres valores de información estadística: media, desviación estándar y homogeneidad, por lo que al final obtendremos N puntos de interés, 3 valores estadísticos para cada vecindad del punto de interés y 3 capas por imagen, tendremos N x 3 x 3 valores estadísticos de la imagen en cuestión.

5. **Conformar el vector patrón característico de la imagen.** Los rasgos geométricos SURF, se calculan de manera independiente, a los rasgos estadísticos CBIR.

   - **Rasgos estadísticos CBIR.** Estos se calculan para cada uno de los puntos de interés seleccionados de la base de datos, para cada una de las imágenes se seleccionan N puntos que se localicen dentro del rostro (ojos, nariz y boca), a cada uno de estos puntos se le extrae la información estadística (media, desviación estándar y homogeneidad), alrededor de una vecindad de tamaño Pi + p x Pi + p

   Al final de este proceso se obtendrá una matriz Mj x 97, con j = Número de imágenes.

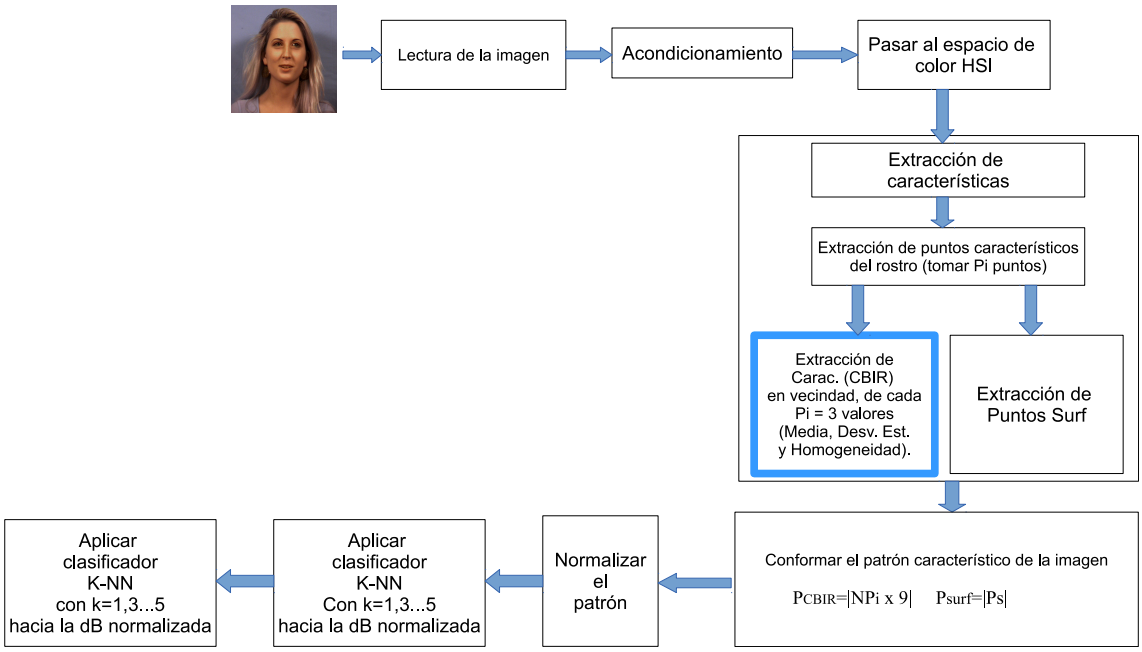

Figura 4.3: Arquitectura de la fase de consulta

6. **Normalizar los patrones.** A partir de los valores de la matriz M se toman los valores máximos y mínimos de cada componente para normalizar los valores de la matriz entre 0 y 1, así se obtendrá una nueva base de datos con la información del contenido de las imágenes en valores normalizados.

7. **Aplicar algoritmo k-Means con k = 10 experimental.** Se generan 10 clases (grupos), a partir de los valores de la base de datos normalizada, el valor de 10 es tomado como referencia ya que este valor es aplicado en la literatura y funciona muy bien, por lo que para comenzar con las pruebas tomaremos este valor de k = 10.

4.1.2 Fase de consulta

Obtenidos los resultados de la fase de entrenamiento, la fase de consulta puede llevarse a cabo. Al igual que la anterior, ésta se divide en diferentes pasos que se explican a continuación, estos se ven resumidos en la siguiente imagen (ver figura 4.3).

Los pasos que se llevan acabo son similares a los de la fase de entrenamiento, exceptuando los tres últimos: la normalización, la apliación del algoritmo k-NN y la obtención de las classes.

1. **Lectura de la imagen.** Se encarga de extraer la imagen de la base de datos que será procesada. La lectura de la imagen da como resultado una imagen en tres capas, correspondientes al espacio de color RGB.

2. **Acondicionamiento.** Realiza el procesamiento previo de la imagen en cada una de las capas. Una vez eliminadas todas las impurezas, la imagen está lista para su correcto análisis.

3. **Pasar al espacio de color HSI.** El espacio de color RGB no nos proporciona la información necesaria, por lo que la imagen RGB se debe pasar al espacio de color HSI.

4. **Extracción de características.** Este bloque es el mas importante ya que es donde implementaremos la mayor parte del trabajo. Consta de tres etapas:

   a) **Extracción de los puntos de interés.** Se consulta el fichero que contiene los puntos de interés, leyendo la línea que corresponde a la imagen a procesar, obteniendo de esta los puntos de interés.

   b) **Extracción de los puntos Surf (Speeded-Up Robust Features).** Es un algoritmo de visión por computador, capaz de obtener una representación visual de una imagen y extraer una información detallada y específica del contenido.

   c) **Extracción de los rasgos estadísticos.** Los rasgos estadísticos son generados para cada uno de los puntos de interés, de cada una de las capas del espacio de color HSI. A partir de un punto de interés determinado se genera una ventana con el punto de interés en el centro, dentro de esta ventana (vecindad) se extraerán tres valores de información estadística: media, desviación estándar y homogeneidad.

5. **Conformar el vector patrón característico de la imagen.** Los rasgos obtenidos de la imagen forman en esta etapa un vector que caracteriza la imagen.

6. **Normalizar los patrones.** A partir de los valores de la matriz M, se toman como referencia los valores máximos y mínimos de cada componente (obtenidos en la fase de entrenamiento) para normalizar los valores del patrón entre 0 y 1, de la imagen en cuestión.

7. **Aplicar algoritmo k-NN con N = 5.** El valor del patrón es comparado contra los valores de las clases generadas en la etapa de entrenamiento, esta operación no es mas que una simple distancia euclidiana entre dos puntos. Al terminar las operaciones se obtienen las 5 imágenes mas parecidas a la imagen de consulta.

8. **Asignación de classes.** En las 5 imágenes obtenidas se comprueba que la primera es ella misma (el patron debe ser el mismo) y que las 4 restantes corresponden a la misma persona. Con este metodo comprobamos la eficacia del algoritmo.

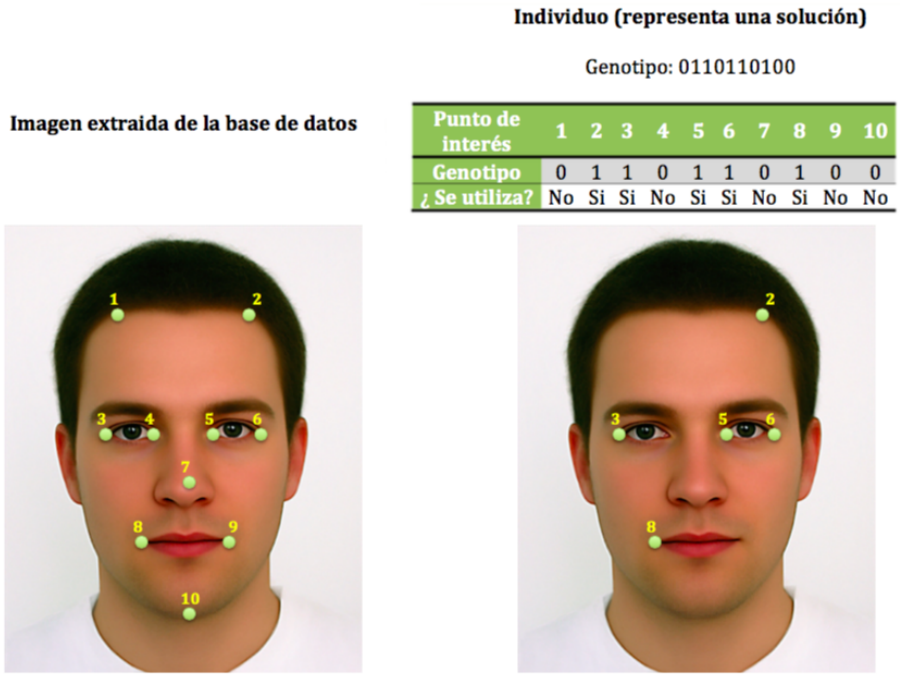

Figura 4.4: Ejemplo de individuo de la población

4.2 Modificación planteada: introducir evolución y paralelismo

El sistema debe ser entrenado haciendo uso de una base de datos de imágenes en la que cada imagen está caracterizada por unos puntos de interés. Todos estos puntos son analizados en la implementación descrita anteriormente. La modificación que se propone y es por esto que se plantea aquí, es la evolución de los puntos de interés a utilizar por el algoritmo, ya que puede ser que algunos de ellos caractericen las imagenes de mejor manera que otros o que algunos sirvan incluso para equivocar al sistema.

La configuración de este problema en ECJ guarda mucha relación con el problema de MaxOne (ver apartado 3.6.1), ya que los individuos también estarán representados por una cadena de 0s y 1s que indican en este caso si se utiliza o no el punto de interés, por lo que prácticamente lo único que cambia es la evaluación de individuos la cual es radicalmente más compleja y costosa. De este modo, si nuestra base de datos asigna a cada imagen 10 puntos de interés, el genotipo de nuestros individuos consistirá en una cadena de 0s y 1s de una longitud de 10 caracteres, indicando cada uno de ellos si el punto de interés correspondiente a esa posición se utiliza (tendrá valor 1) o no (valor 0). Lo anteriormente explicado se expresa en un ejemplo en la imagen del rostro donde se seleccionan algunos puntos de interés (ver figura 4.4).

La evaluación de cada individuo consistirá en ejecutar el sistema de reconocimiento facial solo con los puntos que el genotipo del individuo indique que se deban utilizar,

el fitness corresponderá al porcentaje de imágenes correctamente clasificadas en la fase de consulta.

Si tenemos en cuenta el tiempo que se tarda en la ejecución secuencial del algoritmo (4 minutos) y hacemos unas cuentas sencillas podemos ver que si tenemos una pequeña población de 10 individuos, cada generación tardará en ser evaluada una media hora, si queremos evolucionarlo necesitaremos quizás decenas o centenas de generaciones lo cual conllevaría un tiempo impracticable. Es por esto que además de la integración con ECJ, haremos uso de la integración con Hadoop para distribuir y paralizar el proceso y así poder hacer que la evaluación de cada generación sea cuestión de pocos minutos. Los resultados obtenidos pueden ser consultados más adelante en este documento (ver apartado 4.3).

4.2.1 Implementación

Como se ha comentado en la sección sobre la implementación de los problemas anteriores (ver apartado 3.4.1), existe otro modelo que se puede llevar a cabo cuando la evaluación de cada uno de los individuos toma varios minutos y ésta puede ser paralizada. Éste es el caso del problema de reconocimiento facial que abordamos en este captítulo en el que la evaluación de cada individuo puede tomar aproximadamente 4 minutos y consiste en aplicar un procesamiento a cada una de las imágenes de la base de datos de entrada, por lo que su paralelización no se plantea muy complicada. Teniendo en cuenta esto, haremos uso de este modelo para la integración de este problema con Hadoop. La paralelización se producirá a dos niveles ya que los trabajos se ejecutarán de manera simultánea y las tareas dentro de cada trabajo también. Anteriormente (ver figura 3.5) se ha mostrado el flujo de información que se produce con esta implementación, ese diagrama puede ayudar a entender mejor las etapas que se llevan a cabo.

La entrada del trabajo de Hadoop estará almacenada en el sistema de ficheros y estará formada por la base de datos de imagenes. Ésta será dividida y distribuida a lo targo del cluster para que las diferentes tareas que se ejecuten puedan acceder a ella.

Para hacer esto, el método encargado de evaluar los individuos en el Evaluator (evaluatePopulation) genera un hilo de ejecución de forma local para cada individuo de la población, de manera que cada uno de ellos se encarga de la evaluación de cada individuo, el propósito de estos hilos es tan solo lanzar los trabajos en Hadoop por lo que la mayoría del tiempo tan solo se dedican a esperar que la ejecución de los trabajos finalicen para proseguir con la ejecución normal del proceso evolutivo. La implementación de lo anteriormente explicado se puede observar en el siguiente fragmente de código:

```
1    Individual[] inds = subpops[0].individuals;
2
3    //Creamos un thread por cada individuo
```

```
4     EvaluateIndividual[] threads = new EvaluateIndividual[inds.length];
5     for (int ind = 0; ind < inds.length; ind++)
6         threads[ind] = new EvaluateIndividual(state,
7                                 conf,
8                                 state.generation,
9                                 ind,
10                                (BitVectorIndividual) inds[ind]);
11
12    //Iniciamos los threads
13    for (int ind = 0; ind < inds.length; ind++)
14        threads[ind].start();
15
16    //Esperamos a la finilizacion de todos
17    for (int ind = 0; ind < inds.length; ind++)
18        threads[ind].join();
```

Pasamos ahora a describir que es lo que hace cada uno de los hilos. Al ser hilos, deben implementar un método llamado run(), este método se encarga de lanzar en primer lugar el trabajo que realiza la fase de entrenamiento del sistema de reconocimiento facial y una vez que acaba correctamente este trabajo, lanza otro encargado de la fase de consulta, estos trabajos no pueden ser paralizados ya que la fase de consulta requiere de los resultados de la fase de entrenamiento. La implementación se puede observar a continuación:

```
1     //Lanzamos trabajo de entrenamiento, en caso de no terminar
          exitosamente lanzamos una excepcion
2     if(traningJob != null && !traningJob.run())
3         throw new RuntimeException("Individual " + ind + ": there was a
              problem during the training phase");
4
5     //Lanzamos trabajo de consulta y obtenemos el fitness
6     Float fitness = queryJob.run();
7     if(fitness == null)
8         throw new RuntimeException("Individual " + ind + ": there was a
              problem during the query phase");
9
10    //Asignamos el fitness calculado y marcamos el individuo como evaluado
11    ((SimpleFitness) individual.fitness).setFitness(state, fitness,
          fitness >= 1F);
12    individual.evaluated = true;
```

■ Trabajo de la fase de entrenamiento

Este trabajo implementa las dos fases de un trabajo de Hadoop, la fase de Map y la de Reduce por lo que describiremos en que consiste cada una. La primera fase, la de Map, tiene como implementación el siguiente fragmento de código:

```
    @Override
    protected void map(NullWritable key, ImageWritable image, Context
        context)
            throws IOException, InterruptedException {

        MatE parameters = image.getParameters(windows_size);

        context.write(NullWritable.get(), new MatEWithIDWritable(image.
            getId(), parameters));
    }
```

Como se puede observar, lo que recibe nuestra tarea map son tuplas de NullWritable y ImageWritable, en este caso la clave no la utilizamos por que la establecemos a NullWritable y lo que recibimos en el valor es una imagen a procesar. De lo que se encarga esta función es de extraer los parámetros de cada uno de los puntos de interés (solo los que el genotipo del individuo indique a 1) y almacenarlos en una matriz de OpenCV (MatE), la cual junto con el identificador de la imagen, formaran la salida de la función map. El cálculo de estos parámetros conlleva numerosas operaciones con cada imagen las cuales no se describen ya que se considera no es el propósito de este trabajo, sin embargo estas pueden ser consultados en el código fuente que se proporciona.

Una vez finalizada la fase de Map tenemos todos los parámetros de cada imagen, de modo que la fase de Reduce puede comenzar, en nuestro caso la fase de Reduce se lleva a cabo en un solo nodo (no en uno en concreto, si no en alguno de los que componen el cluster) ya que este necesita toda la información producida por la fase de Map para obtener sus resultados. Procedemos ahora a mostrar la implementación de la fase de Reduce.

```
@Override
    protected void reduce(
            NullWritable key,
            Iterable<MatEWithIDWritable> values,
            Context context)
            throws IOException, InterruptedException {

        //Unimos todos los parametros recibidos por orden en una sola
            matriz
        MatE matRef = new MatE();
        List<MatEWithIDWritable> mats = new LinkedList<MatEWithIDWritable
            >();
        for (MatEWithIDWritable mat : values) {
            MatEWithIDWritable tmp = new MatEWithIDWritable();
            mat.copyTo(tmp);
            mat.release();
            mats.add(tmp);

            number_of_images++;
        }
```

```
19          Collections.sort(mats);
20          Core.vconcat((List<Mat>)(List<?>) mats, matRef);
21
22          //Obtenemos valores maximos por columnas y la normalizamos
23          MatE max_per_column = matRef.getMaxPerColumn();
24          matRef = matRef.normalize(max_per_column);
25
26          //Calculamos Kmeans
27          MatE centers = new MatE();
28          MatE labels = new MatE();
29          TermCriteria criteria = new TermCriteria(TermCriteria.EPS +
                TermCriteria.MAX_ITER, 10000, 0.0001);
30          Core.kmeans(matRef, num_centers , labels, criteria, 1, Core.
                KMEANS_RANDOM_CENTERS, centers);
31
32          //Generamos matriz de indices de texturas
33          MatE textureIndexMatriz = new MatE(Mat.zeros(number_of_images,
                num_centers, CvType.CV_32F));
34          int pos;
35          for (int i = 0; i < number_of_images; i++) {
36              pos = number_of_poi * i;
37              for (int j = pos; j < pos + number_of_poi; j++) {
38                  double[] valor = textureIndexMatriz.get(i, (int) labels.get
                        (j, 0)[0]);
39                  valor[0] = valor[0] + 1;
40                  textureIndexMatriz.put(i, (int)labels.get(j,0)[0], valor);
41              }
42          }
43
44          //Producimos la salida
45          context.write(NullWritable.get(), new TrainingResultsWritable(
                max_per_column, centers, textureIndexMatriz));
46      }
```

Podemos observar como el método reduce recibe como entrada una lista (values) la cual contiene todas matrices producidas por la fase de Map. Lo que hacemos en el reduce es unir todas esas matrices de parámetros en una sola, calcular los máximos por columna, normalizarla, calcular Kmeans y generar una matriz de índices de textura. Finalmente escribimos todos los resultados los cuales necesitaremos para el proximo trabajo, el de consulta.

■ Trabajo de la fase de consulta

De la etapa de consulta del sistema de reconocimiento faciel es de lo que se encarga el trabajo de Hadoop que ahora vamos ha describir. Al igual que el anterior, este utiliza la fase de Map y la de Reduce (una sola tarea de reduce) para obtener finalmente el fitness del individuo. Mostramos en primer lugar la implementación de

la fase de Map.

```
1   private TrainingResultsWritable trainingResults;
2
3   @Override
4   protected void setup(Context context) throws IOException,
        InterruptedException {
5
6       //Obtenemos los resultados del entrenamiento
7       String file = context.getConfiguration().get(EvaluateIndividual.
            INDIVIDUAL_DIR_PARAM).concat("training/part-r-00000");
8       Reader reader = new Reader(fs.getConf(), Reader.file(file));
9       reader.next(key, trainingResults);
10  }
11
12  @Override
13  protected void map(NullWritable key, ImageWritable image, Context
        context)
14          throws IOException, InterruptedException {;
15
16      //Normalizamos los parametros de la imagen
17      MatE normalized_params = image.getParameters(windows_size).
            normalize(trainingResults.getMaxPerCol());
18
19      //Obtenemos centroides con Knn
20      MatE idCenters = knn(trainingResults.getCenters(),
            normalized_params);
21
22      //Obtenemos vector de consulta
23      MatE queryVector = MatE.zeros(1, trainingResults.getCenters().rows
            (), CvType.CV_64F);
24      for (int poi_index = 0; poi_index < idCenters.rows(); poi_index++)
            {
25          int x = (int) idCenters.get(poi_index, 0)[0];
26          double y = queryVector.get(0, x)[0];
27          queryVector.put(0, x, y + 1);
28      }
29
30      context.write(new MatEWritable(trainingResults.
            getTextureIndexMatrix()),
31                  new MatEWithIDWritable(image.getId(), queryVector));
32  }
```

Al igual que el anterior trabajo, el de entrenamiento, la fase de Map recibe las imágenes como entrada, pero este difiere del anterior en que implementa el método setup() el cual se ejecuta antes del iniciar la ejecución de la tarea y se encarga de obtener los resultados producidos por el trabajo anterior los cuales están en el directorio del individuo. Respecto al método map, en primer lugar obtiene los parámetros de la imagen recibida y los normaliza con respecto los máximos obtenidos

del trabajo anterior, una vez normalizados, obtiene los centroides haciendo uso de un algoritmo Knn y por último genera un vector de consulta que junto a la matriz de índices de textura del trabajo anterior compondrán la salida de la fase de Map.

Abordamos ahora la fase de Reduce del trabajo, cuya implementación se muestra a continuación.

```
1   //Relacion de imagen-clase (persona)
2   HashMap<Integer, Integer> img_class;
3
4   @Override
5   protected void setup(Context context) throws IOException,
        InterruptedException {
6       img_class = getImagesClass(conf);
7   }
8
9   @Override
10  protected void reduce(MatEWritable textureIndexMatrix, Iterable<
        MatEWithIDWritable> queryVectors, Context context)
11          throws IOException, InterruptedException {
12
13      //Unimos todos los vectores de consulta
14      MatE query_mat = new MatE();
15      List<MatEWithIDWritable> vectors = new LinkedList<
            MatEWithIDWritable>();
16      for (MatEWithIDWritable queryVector : queryVectors) {
17          MatEWithIDWritable tmp = new MatEWithIDWritable();
18          queryVector.copyTo(tmp);
19          queryVector.release();
20
21          vectors.add(tmp);
22      }
23      Collections.sort(vectors);
24      Core.vconcat((List<Mat>)(List<?>) vectors, query_mat);
25
26      //Obtenmos los ids mas cercanos
27      MatE nearestIds = knn(textureIndexMatrix, query_mat, num_nearest);
28
29      //Calculamos matriz de confusion
30      MatE confusionMatrix = generateConfusionMatrix(nearestIds,
            num_nearest);
31
32      //Obtenemos porcentaje de acierto
33      float suma=0;
34      for (int i=0;i<confusionMatrix.rows();i++)
35          suma = suma + (float)confusionMatrix.get(i,i)[0];
36
37      float percentage = suma / (float)vectors.size() / (float)(
            num_nearest - 1);
38
```

```
39          context.write(NullWritable.get(), new FloatWritable(percentage));
40      }
```

En esta fase, al igual que la anterior, implementamos el método setup() encargado en este caso de obtener la clase (persona) que corresponde a cada una de las imágenes. Respecto al método reduce, une en primer lugar por orden todos los vectores de consulta recibidos, obtiene los ids de las imagenes que más se le asemejan, calcula la matriz de confusión y por último obtiene el porcentaje de aciertos. Este porcentaje al fitness del individuo y por consiguiente será la salida del trabajo de consulta.

4.3 Resultados

Analizamos los resultados obtenidos al ejecutar un problema en el que la evaluación de cada individuo toma varios minutos. En una ejecución secuencial, los tiempos de la evaluación de cada generaración puede llevar horas dependiendo del tamaño de la población.

La implementación secuencial del algoritmo de reconocimiento facial utiliza la librería OpenCV para el tratamiento de las imágenes. El uso de esta librería ayuda a reducir notablemente el tiempo de procesamiento, así la fase de entrenamiento en la ejecución secuencial necesita sobre unos 127 segundos y la fase de consulta unos 140 segundos.

En primer lugar se realizó la integración del problema de reconocimiento facial con Hadoop para paralelizar el procesamiento de las imágenes haciendo uso de esta herramienta. El nivel de paralelización depende de como se divida la entrada del problema ya que se ejecutarán tantas tareas en paralelo como divisiones de la entrada existan (ver figura 3.5). Esta idea nos podría llevar al convencimiento de que obtendremos mejores tiempos mientras mas tareas tengamos, pero la creación de la tarea, distribución de la entrada al trabajo y la posterior destrucción de la tarea conllevan tiempos que deben ser considerados. Es por esto que se han realizado ejecuciones para encontrar el mejor número de tareas que se ejecuten de forma paralela. Los tiempos obtenidos en segundos se muestran en una tabla (ver tabla 4.1).

Grupos	10	15	20	25	30	40
Entrenamiento (s)	63	42	39	41	45	30
Consulta (s)	57	46	43	46	50	54

Tabla 4.1: Tiempos de ejecución en Hadoop en función del número de grupos

Se graficamos estos resultados (ver figura 4.5) comparándolos con los tiempos de la ejecución secuencial podemos observar facilmente dos cosas. En primer lugar, el hecho de que el número de grupos mas apropiado en el que se debe dividir la

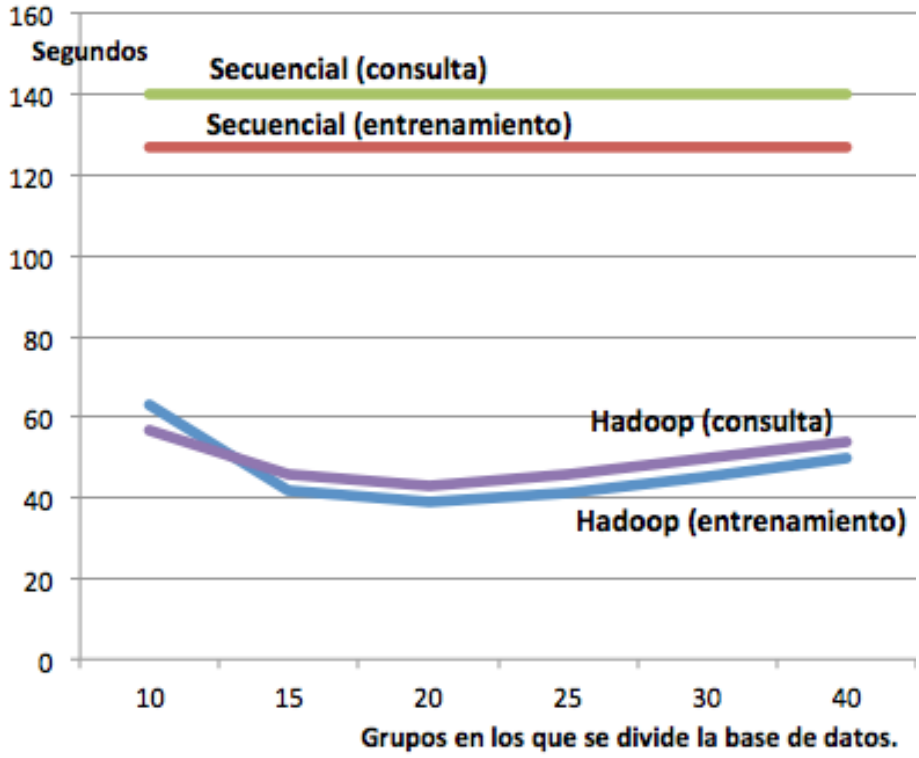

Figura 4.5: Comparación de tiempos en función del numero de grupos de la entrada

entrada son 20. Por otro lado vemos como la ganancia de tiempo con respecto a la ejecución secuencial es significativa. Si sumamos los tiempos de ambas fases en ambas implementaciones (tiempos con 20 grupos) y los comparamos, vemos como la integración ha conseguido que se ejecute el algoritmo en un 30 % del tiempo de la ejecución secuencial.

Tras la integración con Hadoop, se procedió con la adaptación del algoritmo de reconocimiento facial a un algoritmo evolutivo utilizando ECJ. Esta integración provoca que cada individuo de la población ejecute dos trabajos, el de entrenamiento y consulta, y todos ellos de forma paralela por lo que se produce otro nivel de paralelización.

Con el fin de probar la escalabilidad del sistema se ha lanzado ejecuciones con diferentes número de nodos y el mismo número de individuos (10) registrando los tiempos obtenidos (ver tabla 4.2).

Número de nodos	2	4	6
Evaluación (min)	25	11.5	6.4

Tabla 4.2: Tiempos de evaluación de la población en Hadoop con diferente número de nodos para 10 individuos

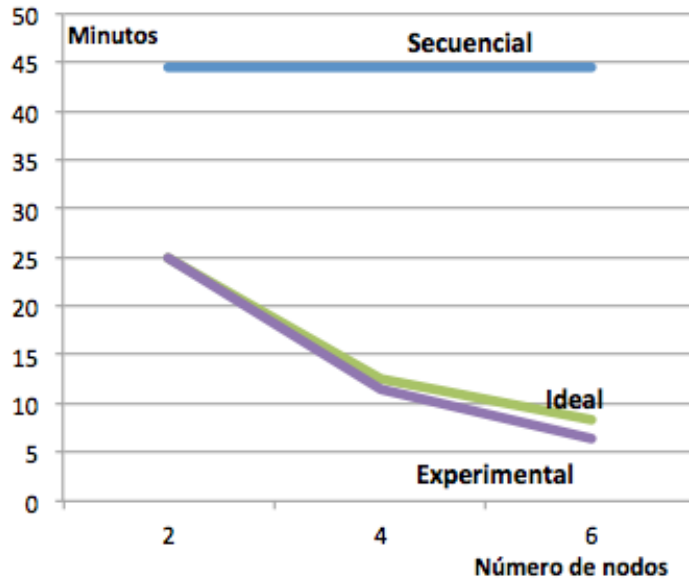

Figura 4.6: Comparación de tiempos en función del número de nodos

Graficados estos datos (ver figura 4.6) observamos como la escalabilidad del sistema mejora incluso los tiempos ideales, esto nos indica que la inclusión de nuevos nodos en el cluster en el que lo corramos (fácil tarea) provocará que los tiempos se reducirán acorde al número de nodos que añadamos. Para observar mejor la escalabilidad del sistema se ha añadido una línea de tiempo ideal correspondiente a la mitad del tiempo de 2 nodos para 4 nodos y a la tercera parte del tiempo de 2 nodos para 6 nodos. Además, se ha añadido al gráfico el tiempo que tomaría esta ejecución de forma secuencial (evaluar un individuo = 127 + 140 = 267 s, evaluar 10 = 10 x 267 / 60 = 44.5 min) para observar lo que supone el uso de Hadoop.

El hecho de que se mejoren los tiempos ideales es de extrañar, pero debemos tener en cuenta que Hadoop hace uso de numerosos servicios que se pueden ver sobrecargados si todo se ejecuta en un número reducido de nodos. Por esta razón es fácil que se produzcan este tipo de anomalías en los tiempos analizados.

Analizamos ahora ejecuciones con diferentes tamaños de población y el mismo número de nodos (6). Se espera que los tiempos de evaluación sean directamente proporcionales al tamaño de la población.

Número de individuos	10	20	30
Evaluación (min)	6.4	14.5	22.1

Tabla 4.3: Tiempos de evaluación de la población en Hadoop con diferente número de individuos en 6 nodos

Al igual que hemos hecho con los resultados anteriores, lo comparamos con los

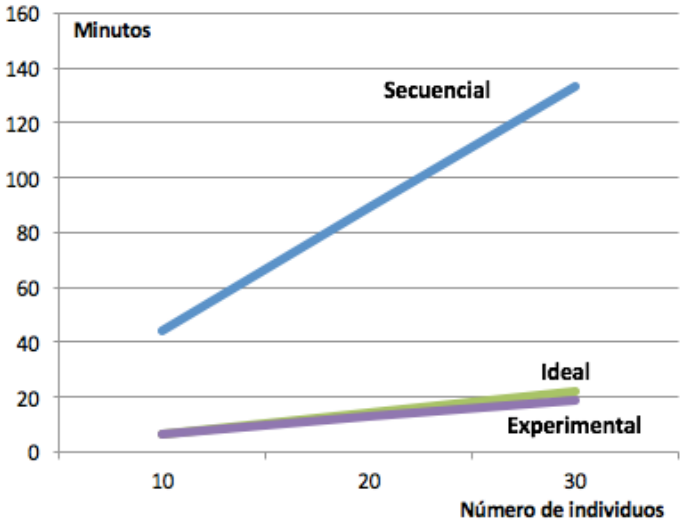

Figura 4.7: Comparación de tiempos en función del número de individuos

tiempos de ejecuciones secuenciales y tiempos ideales (ver figura 4.7). Vemos como los beneficios del uso de la integración con Hadoop son significativos, además se obtienen resultados similares a los ideales.

5. MANUAL DE USUARIO

5.1 Obtención

El primer paso para la utilización de la solución implementada, no puede ser otro que obtenerla. Con el fin de que la comunidad pueda conseguirla sin problema, se ha creado un repositorio público desde donde se puede descargar.

El repositorio está localizado en la conocida página por los desarrolladores GitHub (`https://github.com`), en ella se encuentran numerosos proyectos de código abierto. Todos los proyectos almacenados en esta página hacen uso de una herramienta de control de versionado llamada Git, esta herramienta ha sido utilizada en este proyecto y su utilización se explica mas adelante.

¿Qué es Git?

Es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux. Algunas de sus características mas destacadas son:

- Fuerte apoyo al desarrollo no lineal, por ende rapidez en la gestión de ramas y mezclado de diferentes versiones.
- Gestión distribuida. Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales.
- Los almacenes de información pueden publicarse por HTTP, FTP, rsync o mediante un protocolo nativo, ya sea a través de una conexión TCP/IP simple o a través de cifrado SSH.
- Los repositorios Subversion y svk se pueden usar directamente con git-svn.
- Gestión eficiente de proyectos grandes.

El proyecto se puede encontrar a través del buscador del sitio escribiendo "ecj_hadoop", o accediendo directamente al repositorio siguiendo esta dirección: `https://github.com/dlanza1/ecj_hadoop` . Una vez en el repositorio, el proyecto se puede obtener de dos formas, una es pulsando en el botón "Download ZIP"(situado en la columna de la derecha), y una vez descargado descomprimir el fichero, y la otra forma es clonar el repositorio con Git (crear un repositorio igual de forma local).

Para clonar el repositorio con Git se debe tener instalada esta herramienta en el sistema [**?**] o utilizar un entorno de desarrollo como Eclipse el cual la trae incluida.

Para clonarlo desde un entorno de desarrollo como Eclipse [**?**] el procedimiento suele ser el de importar un proyecto pero con la diferencia de que se debe indicar que la fuente es un repositorio Git, nos solicitará el repositorio que queremos clonar y es ahí donde debemos escribir la URL del repositorio. Algo que debe ser aclarado es que esta operación creará un repositorio local, no genera el proyecto en el entorno de desarrollo, lo cual se explica en la sección siguiente.

Si de otro modo, tenemos la herramienta instalada en el sistema, debemos en primer lugar abrir una consola, posteriormente dirigirnos al directorio donde queremos crear el repositorio local y por último clonar el repositorio con el siguiente comando:

```
[usu@host repo]$ git clone https://github.com/dlanza1/ecj_hadoop
```

Una vez clonado tendremos una copia idéntica del repositorio la cual contiene la última versión del proyecto además de toda la información necesaria para que funcione el sistema de versionado utilizada por Git.

5.2 Importar a entorno de desarrollo

Esta sección tiene como objetivo explicar el procedimiento para importar el proyecto en un entorno de desarrollo, esto no es necesario para su ejecución por lo que no estén interesados en modificar/ampliar el proyecto pueden continuar la lectura en la sección siguiente (Compilación).

En el apartado anterior se ha explicado como obtener el proyecto, pero lo obtenido es básicamente los ficheros de código fuente, no se incluyen proyectos de entornos de desarrollo ni ejecutables.

Si acabamos de descargar/clonar el proyecto, no lo tendremos importado en nuestro entorno, para llevar a cabo esta operación tan solo debemos dirigirnos al menú de importación de proyectos, indicar que es un proyecto Maven y seleccionar el directorio que contiene el proyecto.

Otra opción, teniendo la herramienta Maven instalada en el sistema [**?**], es generar primero el proyecto del entorno de desarrollo y posteriormente importarlo como un proyecto normal. Para generar el proyecto del entorno de desarrollo debemos abrir una consola, dirigirnos al directorio donde se sitúa el proyecto y ejecutar el comando Maven correspondiente a nuestro entorno, por ejemplo para Eclipse debemos ejecutar el siguiente comando:

```
[usu@host repo]$ mvn eclipse:eclipse
```

Esto nos generará el proyecto Eclipse y podremos importarlo como cualquier otro proyecto.

5.3 Compilación

Como se mencionaba anteriormente, los ficheros ejecutables no son proporcionados, por lo que deben ser generados. Con el uso de la herramienta de construcción de proyectos, Maven, explicaremos como construir el proyecto para que pueda ser ejecutado, no obstante, si tenemos el proyecto importado en un IDE, no es necesario utilizar Maven para compilarlo, podemos utilizar los usuales procedimientos para compilarlo.

¿Qué es Maven?

Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Tiene un modelo de configuración de construcción simple, basado en un formato XML. Es un proyecto de nivel superior de la Apache Software Foundation.

Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado.

El motor incluido en su núcleo puede dinámicamente descargar plugins de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos. Este repositorio pugnan por ser el mecanismo de facto de distribución de aplicaciones en Java. Una caché local de artefactos actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

Maven se ejecuta en unas circunstancias similares a las que explicábamos con Git. Existen dos opciones, o tener Maven instalado en el sistema [**?**], o utilizar un entorno de desarrollo como Eclipse [**?**], el cual lo trae incluido.

Si tenemos importado el proyecto en un IDE, para compilarlo (o en nomenclatura de Maven, construirlo) es necesario hacer clic derecho en el fichero incluido en el proyecto pom.xml, y seleccionar la opción: Run as (Ejecutar como) y en el submenú, Maven build, si hacemos esto por primera vez nos aparecerá un cuadro de dialogo donde se nos solicita los objetivos (goals), ahí debemos indicar "install"(sin las comillas) y aceptar/ejecutar.

En caso de que no estemos utilizando un IDE, necesitaremos tener instalado Maven. Para compilarlo de esto modo debemos abrir una consola, dirigirnos al directorio donde se encuentra el proyecto y ejecutar el siguiente comando:

```
[usu@host repo]$ mvn install
```

Al compilarlo con Maven, tanto desde el IDE como desde la consola, se generará un fichero .jar el cual contiene todas las clases compiladas.

5.4 Ejecucion

Para la ejecución de cualquiera de los problemas incluidos en ECJ, debemos iniciar la ejecución en la clase *ec.Evolve*. Es en esta clase donde se encuentra el método main() el cual inicia todo el proceso evolutivo.

Al igual que en los casos anteriores, tenemos dos escenarios posibles, uno es utilizando un entorno de desarrollo y otro directamente desde la consola de comandos. Para ejecutarlo desde un entorno de desarrollo debemos encontrar en el código fuente la clase *Evolve* situada en el paquete *ec*, una vez localizada debemos iniciar la ejecución desde esa clase enviándole como argumentos el fichero de parámetros que configura la ejecución del algoritmo. Los argumentos que se deben indicar son "-file ruta_fichero_parametros", para establecerlos en un entorno de desarrollo como Eclipse debemos hacer clic con el botón derecho en la clase *Evolve* y en el menú seleccionar "Run as" y a countinuación "Run configuration". Se nos abrirá una ventana emergente donde debemos seleccionar la pestaña "Arguments" y en el campo "Program arguments" establecemos los argumentos. Una vez establecidos pulsamos sobre el botón "Run" y se iniciará la ejecución.

Para ejecutar uno de los tutoriales que incluye la distribucion de ECJ debemos indicar en los argumentos: "-file src/main/java/ec/app/tutorial1/tutorial1.params", pulsamos sobre el botón "Run" y se iniciará la ejecución del tutorial número 1, el problema MaxOne.

Sin embargo, el uso de un entorno de desarrollo para la ejecuión de los problemas incluidos no es necesario, podemos ejecutarlo directamente desde la consola teniendo instalado Java en el sistema (versión 1.6 o superior). Para ello abrimos una consola y nos dirigimos al directorio donde se encuentra el proyecto, una vez hecho esto podemos ejecutarlo con el siguiente comando:

```
$java ec.Evolve -file src/main/java/ec/app/tutorial1/tutorial1.params
```

Con esto se iniciará la ejecución mostrando algo como:

```
1
2  | ECJ
3  | An evolutionary computation system (version 21)
4  | By Sean Luke
5  | Contributors: L. Panait, G. Balan, S. Paus, Z. Skolicki, R. Kicinger,
6  |               E. Popovici, K. Sullivan, J. Harrison, J. Bassett, R.
       Hubley,
7  |               A. Desai, A. Chircop, J. Compton, W. Haddon, S. Donnelly,
8  |               B. Jamil, J. Zelibor, E. Kangas, F. Abidi, H. Mooers,
9  |               J. OBeirne, L. Manzoni, K. Talukder, and J. McDermott
10 | URL: http://cs.gmu.edu/~eclab/projects/ecj/
11 | Mail: ecj-help@cs.gmu.edu
12 |       (better: join ECJ-INTEREST at URL above)
13 | Date: May 1, 2013
14 | Current Java: 1.7.0_72 / Java HotSpot(TM) 64-Bit Server VM-24.72-b04
15 | Required Minimum Java: 1.5
16
17
18 Threads: breed/1 eval/1
19 Seed: 4357
20 Job: 0
21 Setting up
22 Initializing Generation 0
23 PARAMETER: pop.subpop.0.species.crossover-prob
24      ALSO: vector.species.crossover-prob
25 Subpop 0 best fitness of generation Fitness: 0.62
26 Generation 1
27 Subpop 0 best fitness of generation Fitness: 0.65
28 Generation 2
29 ...
30 (salida suprimida)
31 ...
32 Generation 43
33 Subpop 0 best fitness of generation Fitness: 1.0
34 Found Ideal Individual
35 Subpop 0 best fitness of run: Fitness: 1.0
```

Observamos la cabecera del problema y como progresa el proceso evolutivo generación por generación hasta alcanzar la generación máxima o encontrar al individuo ideal (como es el caso que se muestra en la ejecución anterior).

5.5 Ejecutar un problema de ECJ en Hadoop

Lo que se ha explicado anteriormente (ver apartado 5.4), representa una ejecución normal de un problema de Hadoop, explicamos ahora como realizar esa ejecución pero utilizando la integración con Hadoop. Este proceso se ha llevado a cabo ya en este documento con dos problemas: MaxOne (ver apartado 3.6.1) y Parity (ver apartado 3.6.2), ahora lo explicamos de forma mas general prestando detalles a los diferentes contextos donde puede ser ejecutado.

Nos podemos encontrar en dos situciones diferentes dependiendo del acceso al cluster Hadoop que dispongamos. Si podemos acceder a alguna de las máquinas del cluster, podemos hacer un ejecución local desde ella, por otro lado si no tenemos acceso debemos realizar la ejecución de forma remota. Ambos cosas requieren que en fichero de parámetros se indique que el evaluador a utilizar debe ser Hadoop, para ello debemos modificar el valor del parámetro *eval* y establecerlo de la siguiente manera:

```
1 eval       = ec.hadoop.HadoopEvaluator
```

Si la ejecución la vamos a realizar en alguno de los nodos del cluster, esto es todo lo que debemos modificar. Para ejecutarlo, debemos seguir el mismo procedimiento que cuando lo ejecutamos sin Hadoop(ver apartado 5.4).

■ Directorio en HDFS

La ejecución en Hadoop se lleva a cabo creando ficheros en HDFS, para ello se crea un directorio donde se van almacenando las poblaciones y los resultados de la evaluación para cada generación. El directorio a utilizar es por defecto *ecj_work_folder_hc*, pero su valor puede ser modificado utilizando el siguiente parámetro:

```
1 eval.hdfs-prefix  = <ruta_directorio_de_trabajo>
```

El directorio indicado se creará en el directorio Home del usuario y será el utilizado por el evaluador.

■ Uso de puertos diferentes a los de por defecto

El evaluador debe conectarse a los procesos NameNode y JobTracker los cuales pueden escuchar por puertos diferentes de los de por defecto. La implementación

realizada permite indicar al evaluador cuales son los que se utilizan, para ello se han creado los siguientes parámetros:

- **hdfs-port**: puerto por el que el NameNode escucha las peticiones de los clientes, su valor por defecto es *8020*.
- **jobtracker-port**: puerto por el que el JobTracker escucha las peticiones de los clientes, su valor por defecto es *8021*.

En el fichero de parámetros deben establecerse de la siguiente manera:

```
1 eval.hdfs-port    = <port_namenode>
2 eval.jobtracker-port     = <port_jobtracker>
```

5.5.1 Ejecución remota

Puede ser que no tengamos acceso a las máquinas del cluster por lo que debemos ejecutar el algoritmo en nuestra máquina indicando las direcciones del cluster remoto. Para ello se deben establecer los siguientes parámetros:

- **hdfs-address**: dirección IP o nombre de host donde se ejecuta el NameNode (proceso principal de HDFS), su valor por defecto es *localhost*.
- **jobtracker-address**: dirección IP o nombre de host donde se ejecuta el JobTracker (coordinador de los trabajos de Hadoop), su valor por defecto es *localhost*.

Si establecemos estos parámetros debemos añadir al fichero de configuración las siguientes líneas:

```
1 eval.hdfs-address = <host_namenode>
2 eval.jobtracker-address = <host_jobtracker>
```

Para ejecutarlo, debemos seguir el mismo procedimiento que cuando lo ejecutamos sin la integración con Hadoop (ver apartado 5.4).

6. TRABAJO FUTURO

Tras el trabajo realizado surgen numerosas ideas para poder mejorar y ampliar la integración entre estas herramientas, algunas de estas ideas se introducen a continuación.

//TODO

7. CONCLUSIONES

Esta implementación multihilo, se ve limitada a la cantidad de núcleos de procesamiento de una solo mńaquina, pero si utilizamos la integración con Hadoop

Aunque con las mejoras no tenia mucho sentido la integración por no mejorar a 8 hilos, tras las mejoras tiene sentido utilizarla cuando tenemos poblaciones grandes con individuos cuya evaluación no es costosa //TODO

BIBLIOGRAFÍA

[1] La verdad sobre el rendimiento de MapReduce en discos SSD [online]. URL: http://blog.cloudera.com/blog/2014/03/the-truth-about-mapreduce-performance-on-ssds/.

[2] Página de la libreria de procesamiento de imágenes OpenCV [online]. URL: http://opencv.org/.

[3] David Andre and John R. Koza. Advances in genetic programming. chapter Parallel Genetic Programming: A Scalable Implementation Using the Transputer Network Architecture, pages 317–337. MIT Press, Cambridge, MA, USA, 1996. URL: http://dl.acm.org/citation.cfm?id=270195.270224.

[4] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer, 2006.

[5] Francisco Fernández de Vega, Juan Manuel Sánchez-Pérez, Marco Tomassini, and Juan Antonio Gómez Pulido. A parallel genetic programming tool based on PVM. In Dongarra et al. [**?**], pages 241–248. URL: http://dx.doi.org/10.1007/3-540-48158-3_30.

[6] Francisco Fernández de Vega, Marco Tomassini, and Leonardo Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–51, 2003. URL: http://dx.doi.org/10.1023/A:1021873026259.

[7] Francisco Fernández de Vega, Marco Tomassini, Leonardo Vanneschi, and Laurent Bucher. A distributed computing environment for genetic programming using MPI. In Dongarra et al. [**?**], pages 322–329. URL: http://dx.doi.org/10.1007/3-540-45255-9_44.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Google, Inc.*, 2004.

[9] Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users'Group Meeting, Balatonfüred, Hungary, September 2000, Proceedings*, volume 1908 of *Lecture Notes in Computer Science*. Springer, 2000.

[10] Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors. *Recent Advances*

in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users'Group Meeting, Barcelona, Spain, September 26-29, 1999, Proceedings, volume 1697 of *Lecture Notes in Computer Science*. Springer, 1999.

[11] Entorno de desarrollo Eclipse [online]. URL: `https://eclipse.org/`.

[12] Malek Smaoui Feki, Viet Huy Nguyen, and Marc Garbey. Parallel genetic algorithm implementation for boinc. In *PARCO*, volume 19, pages 212–219, 2009.

[13] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. A scalable cellular implementation of parallel genetic programming. *IEEE Trans. Evolutionary Computation*, 7(1):37–53, 2003. URL: `http://dx.doi.org/10.1109/TEVC.2002.806168`.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *Google, Inc.*, 2003.

[15] Apache Hadoop [online]. URL: `http://hadoop.apache.org/`.

[16] Procedimientos para la instalación de Git [online]. URL: `http://symfony.es/documentacion/guia-de-instalacion-de-git/`.

[17] Procedimientos para la instalación de Maven [online]. URL: `https://eljaviador.wordpress.com/2013/05/21/guia-de-instalacion-de-maven/`.

[18] P. Martin. "a hardware implementation of a genetic programming system using fpgas and handle-c. *Genetic Programming and Evolvable Machines, 2, 317-343*, 2001.

[19] M. Tomassini. Parallel and distributed evolutionary algorithms: A review. *Engineering and Computer Science*, 1999.

[20] Articulo de la Wikipedia sobre Git [online]. URL: `http://es.wikipedia.org/wiki/Git`.

[21] Articulo de la Wikipedia sobre Maven [online]. URL: `http://es.wikipedia.org/wiki/Maven`.

[22] Cesar Benavides-Alvarez y Juan Villegas-Cortez y Graciela Román-Alonso y Carlos Avilés-Cruz. Reconocimiento de rostros a partir de la propia imagen usando técnica cbir. *Universidad Autonoma Metropolitana, Mexico*, 2015.