Paralelismo como “concern” en Java y su materialización en una herramienta de software

Matías Hirsch Mariano Fernández

UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina.
email: {matias.hirsch;marianoluisf}@gmail.com

Resumen La computación está experimentando una revolución de hardware dada por la creciente disponibilidad de máquinas multinúcleo y ambientes distribuidos como clusters y Grids. Como consecuencia, el poder computacional está al alcance de la mano, pero muchos programadores de hoy en día no están completamente preparados para explotar paralelismo en sus aplicaciones de forma tal de sacar el máximo provecho a este nuevo hardware. En particular, el lenguaje Java ha ayudado a mitigar la heterogeneidad de software inherente a la programación de aplicaciones secuenciales sobre estos ambientes. De todas maneras, persiste aún la necesidad de herramientas para paralelizar aplicaciones de forma fácil y versátil, para que un programador con poca experiencia en programación paralela pueda rápidamente ejecutar una aplicación en paralelo en varios de estos ambientes. Recientemente, una alternativa que se ha propuesto para lograr esto la constituye el concepto de Paralelismo como “Concern” (PcC), el cual se basa en ideas de la programación orientada a aspectos. En este artículo se listan y analizan las herramientas para la programación paralela existentes, acotando a aquellas implementadas en Java, y se presenta una alternativa basada en PcC que apunta a resolver los problemas de las herramientas analizadas.

1. Introducción

La aparición de hardware como las máquinas multinúcleo y de ambientes distribuidos como clusters y Grids [7], ha creado, indudablemente, la necesidad de nuevas herramientas para la programación paralela. Consecuentemente, existe un conjunto de bibliotecas y frameworks que permiten explotar el paralelismo en las aplicaciones de usuario. Sin embargo, muchos de estos esfuerzos son de difícil uso para un programador promedio, y priorizan el rendimiento por sobre otros atributos también deseables como una baja intrusividad de la API paralela en el código de las aplicaciones secuenciales, o la independencia del ambiente de ejecución. Un modelo de programación paralela simple es factible de ser utilizado por usuarios sin conocimientos de programación paralela, lo que a su vez ayuda a los desarrolladores habituados a la programación secuencial a incorporar gradualmente nociones de paralelismo. Por otra parte, una baja intrusividad de código y la “neutralidad” en el ambiente de ejecución soportado por la herramienta son también importantes, dado el alto grado de consenso que existe actualmente sobre lo beneficioso que resulta separar el código que implementa paralelismo del código que implementa la lógica de una aplicación.

En lo referido a lidiar con la diversidad de software y plataformas al momento de implementar aplicaciones, especialmente en entornos distribuidos, Java ha ganado una gran popularidad ya que provee independencia de la plataforma, y ha alcanzado un rendimiento muy competitivo respecto de lenguajes convencionales tales como C y C++. Sin embargo, la mayoría de las bibliotecas de paralelización Java se enfocan en ejecutar aplicaciones paralelas en un ambiente paralelo específico, dado que están apuntadas o bien a la programación para CPUs multinúcleo, clusters o Grids. Por otra parte, las herramientas usualmente ofrecen APIs para coordinar subcómputos en paralelo. Este enfoque requiere entonces conocimientos de programación en paralelo por parte del desarrollador para hacer uso efectivo de dichas APIs. Además, llevan a que el código se vuelva dependiente de la biblioteca (API) usada, haciendo que la tarea de mantenerlo se vuelva compleja y que el esfuerzo de

portarlo a otras bibliotecas y otros ambientes de ejecución se incremente. En otras palabras, no existe una clara separación entre escribir la lógica de la aplicación y paralelizarla.

Este enfoque "intrusivo" de paralelismo es también promovido por varios lenguajes paralelos que se han propuesto recientemente, y que están diseñados para incrementar la productividad de los programadores. Ejemplos son Fortress (Oracle), X10 (IBM), Axum (Microsoft) y Chapel (Cray). Sin embargo, no existe aún evidencia de la potencialidad de adopción de éstos lenguajes. De hecho, muchos investigadores proponen que para lograr un uso masivo de paralelismo se extiendan los lenguajes de programación actuales, antes que construir nuevos lenguajes paralelos desde cero. Un ejemplos de dichos dialectos es Intel® TBB [16]. En conclusión, la programación paralela es hoy en día la regla y no la excepción. Por consiguiente, los investigadores ya han puesto en su agenda la tan esperada meta de herramientas de paralelización versátiles que requieran un esfuerzo mínimo por parte del desarrollador.

2. Paralelismo en Java: Estado del arte

Dentro del contexto hasta aquí descrito, se pueden encontrar varias herramientas basadas en Java que permiten alterar aplicaciones secuenciales a fin de aprovechar una gran cantidad de recursos de CPU para ejecutar en paralelo. A continuación se resumen estos esfuerzos organizándolos de acuerdo al ambiente de ejecución que soportan.

2.1. CPUs multinúcleo

Doug Lea's framework [11] es un paquete adicionado a Java desde su versión 5 que ofrece funcionalidad para el manejo de sincronismo, concurrencia bloqueante y no bloqueante, y operaciones para la administración de colas de tareas. Alternativamente, JCilk [5] provee a Java de las primitivas *spawn* y *sync*. Cada método a paralelizar tiene asociada dos copias, una utilizada en casos comunes donde la semántica secuencial es suficiente, y la otra ejecutada explotando paralelismo cuando se requiere un mejor rendimiento. JCilk obedece la semántica ordinaria de la sentencia *try/catch* de Java en una CPU de un sólo núcleo, pero ocasiona la cancelación de las tareas paralelas en ejecución cuando una excepción ocurre en una CPU multinúcleo. JAC [9] simplifica la programación concurrente separando la lógica de la aplicación del código de declaración de hilos y de manejo de sincronismo, mediante la utilización de anotaciones Java. JAC enfatiza en remover las diferencias entre el código secuencial y el código concurrente. De manera similar, JOMP [3] implementa OpenMP, un estándar popular que incluye un conjunto de directivas y rutinas de bibliotecas para programación en paralelo mediante esquemas de memoria compartida.

2.2. Clusters y Grids

JR [4] provee un modelo de concurrencia que soporta máquinas virtuales remotas y creación de objetos, comunicación y mensajería asíncrona. El código de JR es traducido a código Java común y corriente. JCluster [19] soporta la ejecución de aplicaciones paralelas orientadas a tareas en clusters compuestos de hardware heterogéneo. Las tareas son programadas de acuerdo a un novedoso algoritmo denominado Transitive Random Stealing, el cual mejora el tradicional algoritmo Random Stealing. Satin [18] es una biblioteca para paralelizar código divide y conquista en redes LAN y WAN que obedece el modelo de paralelismo propuesto por JCilk. El programador indica a través de la API y de artefactos de software extra (interfaces) los métodos de la aplicación que deben correr en paralelo. Luego, Satin paraleliza la aplicación modificando directamente el código Java compilado.

JavaSymphony [10] es una biblioteca que provee un modelo de ejecución semi-automático que, de manera transparente, se encarga de la migración, paralelización y balanceo de carga de aplicaciones Grid. También permite que los programadores controlen y personalicen estos aspectos desde el

código de su aplicación, mediante llamados a una API. De manera similar, VCluster [20] ejecuta en clusters aplicaciones basadas en múltiples hilos. Cada hilo puede migrar entre diferentes nodos de un cluster con el propósito de balancear la carga, comunicándose mediante *canales virtuales* que ocultan la ubicación real de los hilos. ProActive [2] permite a los usuarios desarrollar aplicaciones paralelas móviles compuestas de *objetos activos*. Estos objetos dejan visibles métodos a ser invocados por otros objetos activos (posiblemente remotos) u objetos regulares, y viceversa. Las invocaciones se manejan de forma asíncrona utilizando el mecanismo de *wait-by-necessity*, que es equivalente al mecanismo de Futures de Java. La creación de objetos activos, su paralelización y movilidad son manejados en la aplicación mediante código específico provisto por la API de ProActive. Adicionalmente, JGRIM [14] permite gridificar aplicaciones de manera no intrusiva incorporando "concerns" Grid como la búsqueda de recursos Grid, movilidad y paralelismo a través del concepto de inyección de dependencias, el cual es utilizado por muchos frameworks para el desarrollo Web. Finalmente, respecto a las herramientas de código abierto para ejecutar aplicaciones estilo maestro-esclavo, vale la pena mencionar a JPPF (`http://www.jppf.org`) y GridGain (`http://www.gridgain.com`).

2.3. Discusión

Desde la perspectiva de los lenguajes de programación, los enfoques adoptados para tratar el paralelismo son comúnmente clasificados en implícitos y explícitos [8]. El paralelismo implícito permite a los programadores escribir sus aplicaciones sin ningún conocimiento sobre la explotación del paralelismo, el cual se realiza de forma automática a nivel de plataforma. El paralelismo explícito, por otro lado, provee construcciones para que los desarrolladores describan y coordinen computaciones en paralelo. De esta manera, los programadores poseen más control sobre las ejecuciones en paralelo pudiendo implementar aplicaciones más eficientes. Sin embargo, la carga de manejar el paralelismo recae en el programador [8].

Aunque hayan sido diseñados con la simplicidad como objetivo, la mayoría de los esfuerzos mencionados anteriormente se basan mayormente en paralelismo explicito. Luego, paralelizar aplicaciones requiere incorporar primero conceptos fundamentales y estudiar la API de la herramienta a ser usada, lo que puede no ser sencillo para un programador promedio. Además, desde una perspectiva de ingeniería de software, los códigos paralelizados son difíciles de mantener y portar a otras bibliotecas paralelas. Además, el paralelismo explícito produce código fuente que contiene declaraciones para manejar los subcómputos, como así también instrucciones que optimizan la aplicación de acuerdo a las características del ambiente donde ejecuta. Esto produce que la lógica de optimización sea obsoleta al momento de portar la aplicación a un nuevo ambiente, por ejemplo al trasladarse de un cluster a un Grid.

Un enfoque alternativo al paralelismo explícito tradicional es tratar el paralelismo como un "concern" o aspecto en el sentido de programación orientada a aspectos (AOP), evitando así la mezcla de la lógica de la aplicación con el código que implementa su comportamiento paralelo (ver Figura 1). Recientemente, esta idea ha cobrado gran aceptación, como se ve reflejado en herramientas actuales que se basan parcial o completamente en mecanismos para separar dichos aspectos. Ejemplos de esto son las anotaciones de código (JAC, Satin, GridGain), metaobjetos (ProActive) e inyección de dependencias (JGRIM). Además, algunos esfuerzos soportan la misma idea a través de AOP, o esquemas que capturan patrones recurrentes de estructuras de aplicaciones paralelas tales como *pipes* y maestro-esclavo, por citar algunos. Los patrones modelados son instanciados ya sea recubriendo la aplicaciones secuenciales, por ejemplo Muskel [1], o instanciando clases de un framework, como en JaSkel [17] y CO_2P_3S [13]).

Desafortunadamente, los enfoques actuales para el desarrollo paralelo que apuntan en búsqueda de la separación de intereses carecen de un adecuado tratamiento a ciertos atributos como la aplicabilidad, intrusividad y necesidad de conocimiento por parte del programador. Primero, las herramientas diseñadas para explotar el uso de una máquina multinúcleo son, usualmente, no aplicables a ambientes clusters y Grids. Del mismo modo, muchos enfoques diseñados para aprovechar éstos dos últimos

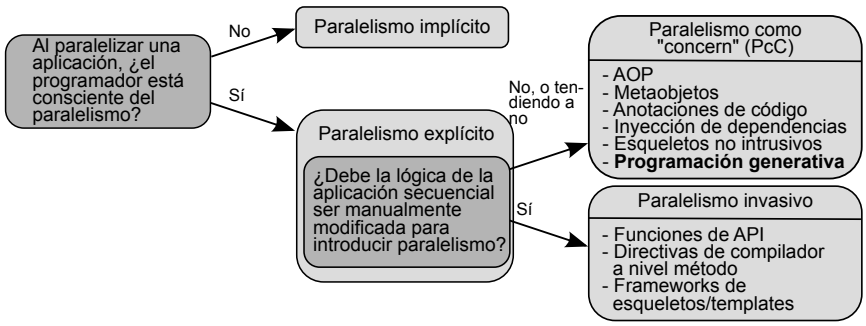

Figura 1: Taxonomía de las formas de paralelizar aplicaciones en Java

ambientes no son eficientes cuando son usados en máquinas multinúcleo debido a su naturaleza distribuida. En segundo lugar, los enfoques basados en anotaciones de código requieren una modificación explícita del código fuente de la aplicación para introducir el paralelismo e incluso optimizaciones específicas acordes a la aplicación, resultando en un código fuente poco claro. Los metaobjetos y especialmente AOP han probado ser de gran ayuda para brindar una solución a este problema, pero incurren en el costo de demandar a los programadores aprender un nuevo paradigma de programación. Por último, los enfoques que brindan soporte para la programación vía patrones de paralelismo han demostrado buena aplicabilidad para una gran variedad de aplicaciones, sin embargo se requieren conocimientos de paralelización por parte de los desarrolladores. Además, cuando es preciso introducir modificaciones a la lógica de la aplicación en la aplicación paralela, tales como resolución de errores, el desarrollador primero debe entender el diseño del patrón o estructura paralela de la aplicación detrás del código obtenido.

Este artículo propone que el concepto de PcC debe ser mejor explotado para ofrecer a los usuarios novatos un enfoque híbrido de paralelismo que permita el desarrollo de aplicaciones en paralelo utilizando la simplicidad del paralelismo implícito, y la flexibilidad y eficiencia del paralelismo explícito. El enfoque debe implícitamente, posibilitar paralelismo en aplicaciones secuenciales, y permitir al usuario que explícitamente ajuste y optimice el código paralelo resultante sin afectar la lógica de la aplicación de entrada. La ejecución de computaciones en paralelo debe ser realizada sin necesidad de "reinventar la rueda", sino aprovechando siempre que sea posible las bibliotecas de paralelización de Java, ya en estado de madurez.

El enfoque debe ofrecer al desarrollador sin experiencia en programación paralela un medio para poder paralelizar de forma sencilla un amplio rango de aplicaciones sin tener que explorar los problemas, típicamente complejos, del desarrollo paralelo. Para ello, se propone adoptar un modelo de programación que provea la oportunidad de explotar formas de paralelización implícitas y versátiles. Los desarrolladores con conocimientos de programación en paralelo podrán también optimizar el código paralelo generado. Esto plantea, sin embargo, varias preguntas importantes. ¿Qué modelo de programación debemos adoptar como base del modelo? ¿Cuáles son los requerimientos involucrados en construir código paralelo que reutilice los mecanismos de ejecución de bibliotecas paralelas existentes? ¿Cómo deben ser manejados los aspectos del paralelismo?. En las siguientes subsecciones se propone un enfoque, y una herramienta de software que lo materializa, que dan respuesta a estos interrogantes.

3. Paralelismo Fork-Join al rescate

El paralelismo Fork-Join (FJP) es una técnica simple pero efectiva que se basa en expresar paralelismo mediante dos primitivas básicas: *fork* y *join*. El uso de *fork* inicia la ejecución de un fragmento

de código (comúnmente un procedimiento o un método) en paralelo, mientras que el uso de *join* bloquea al llamador hasta que la ejecución del fragmento de código finalice. Ciertamente, FJP provee una alternativa al modelo de hilos, el cual ha influenciado fuertemente el desarrollo de bibliotecas en paralelo, pero que ha recibido un sinnúmero de críticas debido a la complejidad de desarrollar aplicaciones que los utilizan [12]. De hecho, Java, que ha sido por años el principal oferente de este modelo de programación, actualmente incluye un framework FJP para explotar CPUs multinúcleo. Los modelos de programación de fácil uso como FJP permiten mejorar significativamente el rendimiento de las aplicaciones secuenciales de hoy día y aprovechar el poder de procesamiento sin la necesidad de una sólida experiencia en programación paralela por parte de los usuarios.

FJP no esta restringido a programación para máquinas multinúcleo, sino que también es útil para ejecuciones en ambientes donde existe la noción de “tarea” y “procesador”. Por ejemplo, las tareas resultantes del uso de *fork* pueden correr en paralelo en los nodos de un cluster alcanzando así un mejor rendimiento y escalabilidad. Más recientemente, Grid Computing ha emergido como un nuevo paradigma para computación distribuida en paralelo. Los Grids organizan recursos de hardware geográficamente dispersos para proveer a las aplicaciones con una gran supercomputadora virtual. De esta manera, las CPUs multinúcleo, los clusters y los Grids pueden ser igualmente usados para ejecutar tareas FJP, ya que estos ambientes están conceptualmente compuestos de nodos de procesamiento interconectados a través de "vínculos" de comunicación. Específicamente, un nodo puede ser una CPU o máquinas individuales, y los vínculos pueden ser el bus del sistema, una red LAN de alta velocidad o una red WAN. Esta uniformidad sugiere que la misma aplicación FJP puede ser corrida en cualquiera de estos ambientes, siempre que el soporte hardware disponga de un planificador de tareas capaz de ejecutar éstas de acuerdo al ambiente subyacente. Luego, un requerimiento de mayor rendimiento de una aplicación FJP diseñada para una máquina multinúcleo se satisface modificándola para usar una biblioteca Grid.

En líneas generales, la existencia de bibliotecas de paralelización Java que recaen en un modelo de ejecución orientado a tareas ofrecen primitivas para lanzar la ejecución en paralelo de una tarea individual o varias al mismo tiempo. Estas tareas son implícita o explícitamente mapeadas a través de llamados a API a unidades de ejecución a nivel de biblioteca. Sin embargo, existen diferencias operacionales entre las diferentes bibliotecas respecto a las primitivas que ofrecen para sincronizar subcómputos. A partir del análisis de la Sección 2, se observa que estas primitivas obedecen uno o dos patrones de sincronización en el contexto de FJP: fork-join simple (SFJ) y fork-join múltiple (MFJ). El primero representa una relación uno a uno entre puntos *fork* y puntos *join* en el código de una aplicación. En otras palabras, el programador debe bloquear la aplicación para esperar por el resultado de cada tarea. Con MFJ, el programador espera por el resultado de las tareas lanzadas a ejecutar en un único punto de sincronización. Por ejemplo, a continuación, en el código de la izquierda, dos llamados de tipo SFJ son necesarios para poder continuar con resultados ya calculados, mientras que el código de la derecha posee el mismo comportamiento pero utilizando un llamado MFJ.

```
void someMethod(){
 ...
 fork(task1);
 fork(task2);
 SFJ(task1);
 ...
 SFJ(task2);
 ...
}
```

```
void someMethod(){
 ...
 fork(task1);
 fork(task2);
 MFJ();
 ...
}
```

Ejemplos de bibliotecas de paralelización de Java y su soporte para patrones de sincronización son GridGain (SFJ), JPPF (SFJ y MFJ), ProActive (SFJ y MFJ) y Satin (MFJ). Las primitivas de *join* de éstas son utilizadas por los desarrolladores mediante llamados de sus APIs. Esto requiere aprender la API en cuestión, y deja ligado el código de la aplicación a una biblioteca específica, comprometiendo

la portabilidad del código generado. También, el manejo manual de sincronización en aplicaciones más complejas es tedioso, insume mucho tiempo, y es muy propenso a errores.

4. FJP como "concern": El proyecto EasyFJP

FJP es adecuado para paralelizar aplicaciones divide y conquista (D&C), un paradigma que representa una manera natural de resolver problemas aplicando una división en subproblemas del mismo tipo, aplicando esta operación de forma descendente con cada subproblema hasta obtener problemas triviales. Luego, las soluciones a los diferentes subproblemas son combinadas ascendentemente hasta hallar la solución al problema original. En terminología de FJP, una aplicación creará repetidamente *forks* para cada subproblema, cuyas soluciones serán combinadas (*join*) para obtener la solución entera. Los subproblemas pequeños y no divisibles son comúnmente resueltos mediante la invocación de un fragmento de código secuencial.

El proyecto EasyFJP [15] apunta a diseñar algoritmos de análisis de código fuente y técnicas de generación de artefactos de código que automaticen la tarea de introducir llamados SFJ y MFJ en código secuencial. Básicamente, los algoritmos explotan la estructura implícita fork-join presente en aplicaciones D&C secuenciales, generando una versión FJP de la aplicación, independiente de la elección de bibliotecas de paralelización que realice el usuario. La generación de código también considera el soporte de sincronización nativo ofrecido por la biblioteca destino, ya sea SFJ o MFJ.

EasyFJP difiere de esfuerzos similares en el hecho de que ofrece una alternativa balanceada en las tres dimensiones discutidas anteriormente: aplicabilidad, intrusividad de código y conocimiento por parte del programador. Primero, una amplia aplicabilidad es alcanzada mediante el uso de Java, basándose en modelos simples y versátiles como FJP y D&C, y proveyendo integración con bibliotecas de paralelización Java existentes para explotar sus primitivas paralelas. Segundo, la baja intrusividad se alcanza utilizando un enfoque basado en programación generativa para traducir de código secuencial a código paralelo, manteniendo además la lógica de optimización separada del código traducido. Precisamente, esta separación, junto con la simplicidad de FJP y D&C, hacen que EasyFJP sea adecuado para posibilitar un movimiento gradual al mundo de la programación paralela. En este sentido, EasyFJP sintetiza ideas interesantes presentes en herramientas contemporáneas pero no simultáneamente explotadas, incluyendo:

- Permitir a desarrolladores paralelizar un amplio rango de aplicaciones secuenciales Java y ejecutarlas en varios ambientes distribuidos y paralelos.
- Usar paralelismo implícito para reutilizar bibliotecas de paralelización basadas en Java existentes.
- Usar sintaxis estándar de Java y modelos de programación intuitivos sin requerir un dialecto de paralelización de Java, o ser especialista en conceptos paralelos.
- Proveer un soporte flexible para la optimización de las aplicaciones resultantes, permitiendo al desarrollador experimentado realizar ajustes al código paralelizado de acuerdo a la naturaleza del mismo y al ambiente donde se ejecutará.

Como se muestra en la Figura 2, el objetivo fundamental de EasyFJP es proveer un proceso semiautomático de paralelización para código D&C secuencial Java. El proceso genera aplicaciones paralelas dependientes de una biblioteca con puntos de configuración (*hooks*) para incorporar optimizaciones provistas por el usuario.

Como primer paso, el código fuente de la aplicación es analizado para detectar los puntos donde los métodos objetivo realizaran los llamados recursivos y los puntos donde acceden a los resultados. Estas dependencias deben ser respetadas una vez paralelizada la aplicación para mantener la correctitud de la misma. Antes de utilizar EasyFJP, el programador debe asignar los resultados de los llamados recursivos a variable locales las cuales deben ser declaradas al comienzo del método. Esta es una convención de código simple de aplicar ya que no involucra uso manual de paralelismo ni depende de la biblioteca paralela objetivo.

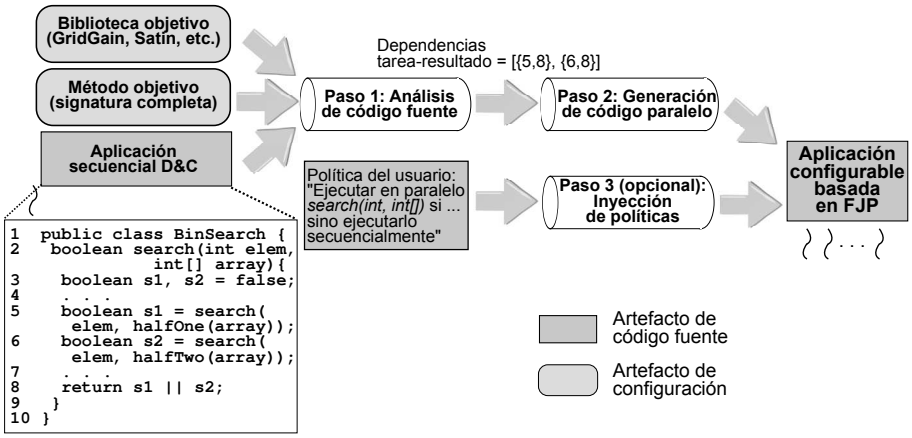

Figura 2: Vista general del proceso de paralelización de EasyFJP

El segundo paso involucra la generación del código paralelo en sí. Esto se realiza mediante componentes llamados *generadores*, que toman ventaja de las primitivas de la biblioteca de paralelización objetivo. Los generadores también son responsables de la inserción de código para evaluar potenciales optimizaciones definidas en el paso tres. Finalmente, los generadores realizan las tareas que sean necesarias de acuerdo a la biblioteca seleccionada, adaptando el código de la aplicación a la estructura dictada por la biblioteca objetivo. Esto incluye la extensión de ciertas clases de la API paralela, la generación de artefactos extras de código y configuración, etc.

Las bibliotecas que soportan SFJ simplifican la tarea de insertar automáticamente código específico de biblioteca para manejar las dependencias tarea-resultado, ya que el acceso al resultado de las tareas puede ser (usualmente) directamente reemplazado por el correspondiente llamado bloqueante de la API. Esta tarea es más compleja al tratarse de MFJ, dado que se debe realizar un análisis de código más inteligente considerando la estructura de las sentencias y el alcance de las variables, asegurándose de respetar las todas dependencias pero minimizar la cantidad de llamados bloqueantes a insertar. Ambos algoritmos de sincronización se comportan mediante una heurística, emulando un desarrollador humano astuto y asegurando a su vez la correctitud del código producido.

Otro aspecto desafiante tener en cuenta es la adaptación del modelo paralelo. Las bibliotecas que soportan conceptos D&C como Satin, comúnmente requieren un simple mecanismo de traducción código a código. En otras palabras, los métodos recursivos en la aplicación de entrada pueden ser bifurcados en la aplicación en paralelo que se obtiene como resultado a través de llamados a la API de la biblioteca paralela. Pero, las bibliotecas que cuentan con modelos convencionales de ejecución como maestro-esclavo, donde no existe una relación jerárquica entre las tareas paralelas, no permiten un mapeo de código directo pues se debe "aplanar" la estructura de tareas de la aplicación de entrada. Ejemplos de estas bibliotecas son GridGain, JPPF y ProActive. Hasta el momento, se han desarrollado generadores para las bibliotecas Satin y GridGain.

Finalmente, en el paso tres, el programador puede adaptar el comportamiento de su aplicación en paralelo para mejorar su eficiencia en ejecución, mediante un mecanismo de ajuste no intrusivo basado en *políticas* [15]. Una política es una clase provista por el usuario que especifica si se bifurca un llamado recursivo o si se ejecuta secuencialmente. Por ejemplo, en una aplicación de búsqueda binaria sobre un arreglo (Figura 2) puede justificarse la bifurcación si los parámetros de entrada se encuentren por encima de un umbral determinado, en este caso, el arreglo y su tamaño, respectivamente:

```
01 public class MyThresholdPolicy implements easyFJP.Policy{
02  public boolean shouldFork(ExecutionContext ctx){
03   int[] array = (int[])ctx.getArg(1); // search(elem, array)
```

```
04   return (array.length > MIN_ARRAY_SIZE);
05  }
06 }
```

ExecutionContext (línea 2) permite al programador acceder por ejemplo a valores de parámetros, o estado a nivel aplicación, como la profundidad actual en el árbol de ejecución de la aplicación. El uso de políticas no intrusivas se asocia a un punto de bifurcación mediante un archivo de configuración. Luego, estas políticas configuradas, las cuales son las reglas que controlan la cantidad de paralelización de la aplicación, pueden ser modificadas o intercambiadas sin alterar el código de la misma. El framework de optimización de EasyFJP permite al desarrollador implementar políticas complejas basadas tanto en la naturaleza de la aplicación como en las características del ambiente de ejecución. Estas pueden incluir, por ejemplo, uso de *memoization* a nivel aplicación, o acotar la cantidad de *forks* para los casos en que las tareas a ejecutar presenten un conjunto de parámetros grande si se cuenta con un ambiente con alta latencia de red. En este sentido, EasyFJP ofrece una API orientada a obtener información en tiempo de ejecución sobre la aplicación en ejecución, y sobre el ambiente donde ésta ejecuta (por ejemplo disponibilidad de CPUs, condiciones de red, etc.).

5. El generador EasyFJP para SFJ

En las subsecciones siguientes se describe concisamente la funcionalidad del generador de aplicaciones paralelas que explota el patrón de sincronización SFJ. Bajo la implementación actual, dicho generador soporta la construcción de código paralelizado para la biblioteca GridGain. La construcción de la nueva aplicación insume tres pasos básicos: el análisis del código secuencial recibido como entrada para detectar oportunidades de paralelismo y sincronización, la inclusión de políticas y la generación del código fuente paralelizado dependiente de la biblioteca propiamente dicho.

5.1. Paso 1: Análisis de código secuencial

Este paso involucra el análisis del código secuencial de la aplicación divide y conquista provista por el usuario con el objetivo de hallar los puntos *fork* y *join* que constituyen la entrada del paso 3 de paralelización del código fuente. A fin de facilitar la tarea, el usuario debe sólo seguir una convención de código muy simple a la hora de escribir su aplicación secuencial, la cual involucra definir, al principio del método a paralelizar, tantas variables locales como llamados recursivos se produzcan. Estas variables reciben el nombre de *variables Grid* y su propósito es alojar el resultado de las diferentes subcómputos. A continuación, se expone un ejemplo de código que respeta tales convenciones, donde x e y constituyen las variables Grid:

```
public long fibonacci ( long n ) { // Método a paralelizar
 if ( n < 2 ) return n;
 long x = fibonacci( n-1 );
 long y = fibonacci( n-2 );
 return x + y;
}
```

El análisis identifica las sentencias *fork* y *join* del método a paralelizar. Las primeras están representadas por los llamados recursivos, que es donde el paralelismo entrará en juego con el fin de ejecutarlos en paralelo. Las segundas, por su parte, estarán representadas por las lecturas a variables Grid, o en otras palabras el uso del resultado de un *fork*. Gracias a esto, el algoritmo de detección para el patrón SFJ se simplifica considerablemente. Por otra parte, para el caso de bibliotecas que emplean MFJ, es necesario otro algoritmo de detección más complejo, el cual se describe en [15].

La detección de los puntos de *fork* y *join* de un método bajo SFJ se ilustra en el Algoritmo 1. La función obtenerPuntosDeBifurcacion retorna una lista con las sentencias que se encuentran del método

a paralelizar que representan llamados recursivos. Por su parte, la función obtenerPuntosDeEncuentro recibe como parámetro la sentencia donde se realiza un llamado paralelo y recorre los ámbitos en busca de todos de la variable que alojará su resultado, devolviendo una lista con las sentencias asociadas. Las funciones auxiliares del algoritmo se detallan en el Cuadro 1 del Apéndice A. Para orientar al analizador de código dónde dirigir el análisis, el usuario debe proveer cierta configuración a través de un archivo XML. En éste se indica la clase a la que pertenece el método a paralelizar mediante su signatura completa, en formato XML.

Algoritmo 1 Pseudocódigo de las funciones que permiten obtener las sentencias que contienen puntos de *fork* (SF) y las que contienen puntos de *join* (SJ) que pertenecen a un método D&C que desea gridificarse, donde L[...] hace referencia a una lista de sentencias, S a una sentencia de código cualesquiera, M al cuerpo de un método recursivo y VG a una variable Grid.

```
L[SF] obtenerPuntosDeFork(M){
  L[...] ← ∅
  for each (S ∈ M)
    if ( esLlamadoRecursivo (S) ) then
      L[...] ←nuevaSentenciaFork(S)
}
L[SJ] obtenerPuntosDeJoin(SF){
  L[...] ← ∅
  VG ← obtenerVariableGrid(SF)
  S1 ← SF
  AMBITO ← true
  while (AMBITO) {
    S2 ← obtenerPrimerUso(VG,S1)
    if ( S2 ≠ ∅ and ámbito(S2) ⊆ ámbito(VG) ) then {
      L[...] ←nuevaSentenciaJoin(S2)
      S1 ← S2
    }
    else
      AMBITO ← false
  }
  return L
}
```

5.2. Paso 2: Inclusión de políticas

Las políticas constituyen un mecanismo de optimización opcional, no intrusivo, a través del cual programadores experimentados pueden adaptar el comportamiento paralelo por defecto de la aplicación paralelizada con el propósito de obtener mayor rendimiento. Una política es una clase provista por el usuario que, por ejemplo, permite controlar el nivel de paralelismo, agregando lógica que permita decidir si un llamado recursivo debe ser ejecutado en paralelo en un nodo remoto o bien ser ejecutado de forma secuencial en el nodo local. Básicamente, la necesidad de controlar el nivel de paralelismo surge a partir de la influencia que ejerce este aspecto sobre el rendimiento de una aplicación a la hora de su ejecución mediante bibliotecas que no cuentan con un mecanismo de creación

y ejecución de tareas que maneje de manera eficiente muchos trabajos de granularidad de tarea fina. Para ejemplificar la idea de cómo controlar el nivel de paralelismo, en el código fuente a continuación, que calcula el número de Fibonacci, se ha introducido una política (la cual es implementada por la clase MyGranularityPolicy) que decide ejecutar en paralelo los llamados recursivos siempre y cuando el parámetro *n* del método sea un número par.

```
01 public long fibonacci ( long n ) { // Método a paralelizar
02  if ( n < 2 )
03   return n;
04
05  long x;
06  long y;
07  if (GranularityPolicy.shouldFork( n )){
08   x = parallelLibrary.fork(fibonacci( n-1 ));
09   y = parallelLibrary.fork(fibonacci( n-2 ));
10  }
11  else { // Ejecución local
12   x = fibonacci( n-1 );
13   y = fibonacci( n-2 );
14  }
15  return x + y;
16 }
17
18 // Política provista por el usuario
19 public class MyGranularityPolicy extends Policy {
20  public static boolean shouldFork ( long n ){
21   return (n mod 2) == 0;
22  }
23 }
```

Por un lado, el código entre las líneas 7-14 es automáticamente derivado a partir del código secuencial normal provisto por el usuario, el cual naturalmente sólo contiene la rama de la bifurcación asociada a ejecutar de forma local. Como se verá en la siguiente sección, este código pasa a formar parte de una nueva clase, denominada *peer*. Por otra parte, las clases que implementan políticas son asociadas a los puntos de bifurcación (para el caso del ejemplo el representado por la línea 7) de manera no invasiva, es decir, a través del archivo de configuración XML mencionado anteriormente, lo que permite cambiarlas sin necesidad de modificar el código de la aplicación.

Por otra parte, la utilización de políticas no sólo es de utilidad para controlar la cantidad de tareas que la ejecución de una aplicación genera dinámicamente, sino que también son útiles para contemplar las características del ambiente de ejecución subyacente evitando, por ejemplo, una excesiva paralelización sobre redes que presentan altas latencias cuando se cuenta con una aplicación cuyos parámetros de entrada poseen gran tamaño. Un caso puntual donde se observa este problema es en códigos recursivos que aplican el mismo algoritmo sobre regiones diferentes de los mismos datos de entrada. Cuando los datos son de gran tamaño, se generan tareas parametrizadas con regiones de dichos datos que potencialmente también tienen gran tamaño. Esto a su vez causa que los beneficios de la paralelización sean inexistentes por los costos de transferencia de las tareas a nodos remotos.

5.3. Paso 3: Generación de código paralelizado

En este paso del proceso es donde se crea la aplicación paralelizada que hace uso de la API de la biblioteca paralela objetivo. Lo que se requiere como entrada es la identificación de los puntos *fork* y *join* como resultado del análisis del código fuente de la aplicación secuencial durante el paso 1, el archivo fuente de la clase Java que contiene los métodos a ser paralelizados, el archivo de configura-

ción en el que se indica la clase que implementa una política (en el caso de hacerse uso de ellas) y la selección de una biblioteca objetivo para la cual se creará código.

Con el propósito de mantener la integridad de la clase convencional, de modo que el usuario pueda modificar fácilmente la lógica de su aplicación en el caso que lo considere necesario, se crea, por cada clase contenedora de métodos D&C a paralelizar, una clase *peer* cuyo código es derivado de la clase secuencial pero se encuentra modificado para explotar las capacidades de la biblioteca de paralelización objetivo. Un primer paso para lograr el nexo entre clases convencionales y sus equivalentes gridificadas es exigir al programador el cumplimiento de otra convención muy simple que es la adhesión a la especificación JavaBeans a la hora de codificar la versión convencional de la clase a gridificar, la cual establece utilizar *getters* y *setters* para acceder a variables de instancia. Esto permite copiar reflexivamente el valor de los atributos de la versión convencional en la versión paralelizada al ejecutar la aplicación. La vinculación propiamente dicha se da en el momento que los métodos D&C pertenecientes a una clase convencional involucrados en el proceso de paralelización son reescritos reemplazando su cuerpo por tres sentencias que corresponden a la creación de la clase *peer*, el seteo de propiedades haciendo uso de los getters y setters, y la invocación al método paralelizado (luego de los pasos 1 y 2) sobre una instancia de la clase *peer* respectivamente. El código que sigue ilustra tal vinculación para el método fibonacci de la clase FibApp:

```
01 public class FibApp {
02  public long fibonacci ( long n ) {
03   FibApp_Peer peer = new FibApp_Peer ();
04   copyProperties ( this, peer );
05   // Se delega la ejecución a la clase peer
06   return peer.fibonacci(n);
07  }
08 }
```

Luego, los puntos de *fork* y *join* producto del paso 1 y el código para invocar políticas detallado en el paso 2 son aplicados sobre el código fuente de la clase *peer* (FibApp) el cual es obtenido inicialmente haciendo una copia exacta del código de la clase convencional (FibApp). La primer transformación, naturalmente, se realiza haciendo uso de la API para creación de tareas y sincronismo de la biblioteca seleccionada por el usuario. En este sentido, el generador realiza transformaciones adicionales a la clase *peer* para que ésta se adapte a la estructura que exige la biblioteca de paralelización objetivo. Por ejemplo, algunas de ellas exigen que la aplicación herede de ciertas clases, implemente ciertas interfaces, defina constructores vacíos, por nombrar algunas. Por ende, cada generador concentra experiencia en el uso de la API de una biblioteca de paralelización particular. Así, se automatiza el empleo de estas APIs, lo que a su vez ayuda a los programadores novatos a evitar invertir tiempo en sortear las dificultades que acarrea el desarrollo paralelo.

Pseudocódigo de generación En la Sección 5.1 se explicó el método para detectar los puntos de *fork* y *join* en una aplicación D&C. En base a dicho algoritmo, se define a continuación la forma en que la herramienta realiza cambios en un código secuencial para obtener su símil paralelizado, el cual se basa en la noción de java.util.concurrent.Future provisto por Java, un objeto especial que "aloja" el resultado de una tarea paralela. El siguiente código ejemplifica el modo en el que opera un Future mediante el método para el cálculo de la sucesión de Fibonacci mostrado anteriormente:

```
01 public long fib( int n ){
02  if ( n < 2 )
03   return n;
04  GridTaskFuture f1 = GridFactory.submit(new GridJob(n-1));
05  GridTaskFuture f2 = GridFactory.submit(new GridJob(n-2));
06  return f1.get() + f2.get();
07 }
```

El código hace uso de la API paralela de la plataforma GridGain, la cual extiende el Future convencional de Java para permitir que los métodos paralelos puedan ser distribuidos en los nodos de un Grid por motivos de balanceo de carga. Básicamente, la clase GridTaskFuture representa un Future cuyo valor será asociado con el resultado una tarea paralela específica. La lectura del valor de uno de estos objetos producirá un bloqueo hasta que el cálculo en cuestión finalice, almacenando el resultado del subcómputo. Además, la ejecución de tareas Grid se inicia mediante el método submit, que recibe como parámetro el cálculo a realizar y retorna el objeto Future extendido que implementa la interface java.util.concurrent.Future. La clase que representa un cálculo a ejecutar en el Grid implementa la interface java.util.concurrent.Callable. Para acceder al valor de un Future, se utiliza su método get.

Este esquema de cálculo y acceso a resultados es utilizado por varias plataformas, entre ellos precisamente GridGain y JPPF. Esto implica que el resultado de cada tarea o trabajo enviado a un Grid podrá accederse de forma independiente. Para establecer una correspondencia entre el código D&C convencional y el mecanismo Future, el *generador* mapea el uso de una variable Grid asociada a un llamado recursivo a la extracción de resultado sobre un Future en el código paralelo generado. Además, los llamados recursivos recursivos se mapean a la creación de las tareas que instanciarán el valor de los Futures. Esta asociación directa se utiliza para generar código paralelizado a partir de los puntos SFJ detectados en el paso 1 de acuerdo al Algoritmo 2. Las funciones auxiliares del algoritmo se describen en el Cuadro 2 del Apéndice A.

Algoritmo 2 Pseudocódigo define la función que permite paralelizar aplicaciones D&C con bibliotecas que están basadas en SFJ y futures, donde M al método recursivo que se desea gridificar, VG hace referencia a una variable Grid, S a una sentencia cualesquiera, L[...] a una lista de sentencias, SF una sentencia fork, SJ una sentencia join y F una variable tipo java.util.concurrent.Future. A su vez, se hace uso de las funciones de obtención de puntos de *fork* y de *join*, y las variables Grid declaradas en el Algoritmo 1.

```
generarCódigo(M){
  LF[...] ←obtenerPuntosDeJoin(M)
  for each (SF ∈ LF) {
    crearTrabajoGrid(SF)
    VG ← obtenerVariableGrid(SF)
    F ←declararFuture(VG)
    LJ[...] ←obtenerPuntosDeJoin(SF)
    for each (SJ ∈ LJ) {
      extraerValorDeFuture(SJ, VG, F)
    }
  }
}
```

6. Resultados experimentales

A grandes rasgos, las implicaciones prácticas de usar EasyFJP respecto al aprovechamiento de los recursos de hardware de un ambiente paralelo está sujeto a dos aspectos cruciales. Por un lado, qué tan competitivo es el soporte de sincronización implícita basado en FJP comparado con el paralelismo explícito. Por otro lado, cuán efectivo es el mecanismo de políticas para optimizar aplicaciones

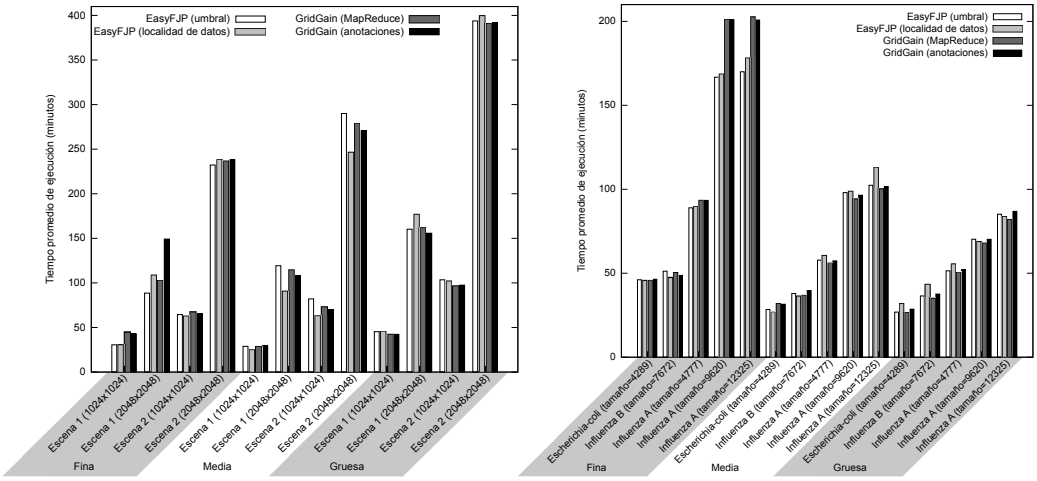

Figura 3: Tiempo promedio de ejecución para las aplicaciones

paralelizadas. Los experimentos preliminares que se han realizado para responder estos interrogantes en base al patrón de sincronización MFJ reportados en [15] han mostrado resultados alentadores.

Aquí, se ha evaluado la viabilidad de utilizar el patrón SFJ mediante las técnicas de paralelización provistas por EasyFJP, con generación de código paralelo para la biblioteca GridGain, en un ambiente Grid. El mismo se compuso de 3 clusters emulados en una red LAN mediante WANem (`http://wanem.sourceforge.net`) bajo condiciones normales de Internet, esto es, ancho de banda de 1.5 Mbps, y latencia de red entre 150-170 ms. Por otra parte, el Grid incluyó 15 nodos con CPUs de un único núcleo de 3 MHz cada uno y con 1.5 GB de RAM repartidas equitativamente entre los 3 clusters. Se utilizaron como aplicaciones de prueba algoritmos de ray tracing y de alineación de secuencias de ADN, cuya versión en paralelo fue obtenida basado en una versión secuencial D&C disponible en el proyecto Satin. Estas aplicaciones poseen una alta complejidad ciclomática, lo que las hizo representativas para evaluar la efectividad de nuestras técnicas de análisis de código.

Se ejecutó, por un lado, la aplicación de ray tracing con varias escenas 3D como entrada, y por otro, la aplicación de alineación de secuencias con bases de datos reales de genes extraídas del NCBI (National Center for Biotechnology Information) (`http://www.ncbi.nlm.nih.gov`). Además, para la aplicación de ray tracing se utilizaron tres granularidades de tarea diferentes: fina, media y gruesa, resultando en 1, 4 y 17 tareas respectivamente en promedio a ejecutar por nodo. Para la alineación de secuencias, también se emplearon tres granularidades de tarea, cada una generando un número de tareas paralelas dependiente del tamaño de la base de datos de entrada con el fin de lograr mayor eficiencia. Para ambas aplicaciones, se implementaron dos variantes EasyFJP utilizando una política de umbral para controlar la cantidad de tareas creadas en tiempo de ejecución y otra política que explota la localidad de los datos, una funcionalidad de EasyFJP para ubicar el procesamiento de regiones cercanas de los datos de entrada de una tarea en el mismo cluster físico. Se desarrolló además una variante utilizando el mecanismo de paralelización de GridGain basado en anotaciones de código, y otra mediante el mecanismo MapReduce de Google [6], el cual es soportado por GridGain. La Figura 3 (a) muestra los tiempos promedios de ejecución obtenidos luego de 40 corridas de la aplicación de ray tracing. La Figura 3 (b) muestra dichos resultados para la aplicación de alineación de secuencias.

Para el caso de ray tracing, los tiempos de ejecución se incrementaron uniformemente cuando la granularidad de tarea se volvía más fina en todos las pruebas, lo que muestra una buena correlación global de las diferentes variantes. Para granularidades de tarea fina y media, EasyFJP logró funcionar mejor que sus competidores ya que la conjunción de SFJ y políticas alcanzó comparativamente

ganancias de rendimiento superiores a un 29 %. Para granularidades de tarea gruesas, sin embargo, la "mejor" variante de EasyFJP introdujo igualmente una penalidad de un 1-9 % respecto a la mejor ejecución de las implementaciones de GridGain. Como era de esperar, la localidad de datos resultó contraproducente. Esto fue debido a que los beneficios en rendimiento producto de ubicar un conjunto de tareas relacionadas (en este caso aquellas que pertenecen a regiones cercanas de la imagen de entrada) en el mismo cluster físico se volvió marginal para granularidad de tarea gruesa, esto es, cuando el procesamiento se dividía en pocas tareas. Las granularidades de tarea que presentaron más eficiencia fueron la fina y la media dado que ofrecían la mejor tasa de comunicación de datos sobre el uso promedio de procesador, esto es, la relación entre los tiempos que una aplicación destina a transmitir resultados de subcómputos y llevar a cabo procesamiento útil.

Para la alineación de secuencias, los tiempos de ejecución fueron menores a medida que la granularidad de tarea crecía. Al igual que el caso de ray tracing, EasyFJP obtuvo mejores resultados para granularidad de tarea fina. Para las variantes que utilizaban granularidades de tarea media y gruesa, EasyFJP se mantuvo competitivo, pero la evaluación confirmó que las variantes implementadas con GridGain fueron levemente más eficientes utilizando tareas de granularidad gruesa. En general, y exceptuando unos pocos casos, la localidad de datos no ayudó a reducir los tiempos de ejecución ya que, a diferencia de ray tracing, las tareas en paralelo poseían un mayor grado de independencia. Esto no implica que las políticas basadas en localidad de datos son poco efectivas, sino que sus beneficios dependen de la naturaleza de la aplicación. Por ende, este aspecto debe ser cuidadosamente tenido en cuenta por el desarrollador de aplicaciones para decidir si es conveniente la utilización de tales políticas.

7. Conclusiones y trabajos futuros

Los resultados expuestos en la sección anterior, en conjunción con los reportados en [15], sugieren que la sincronización implícita utilizando tanto la generación de aplicaciones paralelas utilizando SFJ, como un mapeo MFJ, y los ajustes explícitos basados en políticas, combinados en forma conjunta mediante programación generativa son un enfoque viable al concepto de PcC.

Hasta el momento, se ha mostrado que este enfoque tiene el potencial de ofrecer un mejor balance al *tradeoff* entre la facilidad de uso y el rendimiento, el cual representa un problema inherente a las herramientas para el desarrollo de aplicaciones en paralelo. En cuanto a los trabajos futuros, EasyFJP trata dos aspectos generales de la paralelización, como lo son la sincronización de tareas y la optimización de aplicaciones. Como punto de partida para futuras extensiones se planea incorporar otros aspectos comunes de la programación paralela, como la intercomunicación entre tareas. Además, actualmente se encuentra en desarrollo una herramienta para facilitar la adopción de estas ideas. La misma se está implementando como un plug-in de Eclipse, un entorno de desarrollo que es muy popular entre los desarrolladores Java.

Bibliografía

[1] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: An expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, 2007.

[2] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying on the Grid, pages 205–229. Springer, January 2006.

[3] J. M. Bull and M. E. Kambites. JOMP-an OpenMP-like interface for Java. In *ACM Conference on Java Grande*, pages 44–53. ACM Press, 2000.

[4] Hiu Ning (Angela) Chan, Andrew J. Gallagher, Appu S. Goundan, Yi Lin William Au Yeung, Aaron W. Keen, and Ronald A. Olsson. Generic operations and capabilities in the JR concurrent programming language. *Computer Languages, Systems & Structures*, 35(3):293–305, 2009.

[5] John S. Danaher, I. Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming, Special Issue on Synchronization and Concurrency in Object-Oriented Languages*, 63(2):147–171, December 2006.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design & Implementation*. USENIX, December 2004.

[7] Ian Foster and Carl Kesselman. *Concepts and Architecture*, chapter Concepts and Architecture, pages 37–63. Morgan-Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[8] Vincent W. Freeh. A comparison of implicit and explicit parallel programming. *Journal of Parallel and Distributed Computing*, 34(1):50–65, 1996.

[9] Max Haustein and Klaus-Peter Lohr. JAC: Declarative Java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, April 2006.

[10] Alexandru Jugravu and Thomas Fahringer. JavaSymphony, a programming model for the Grid. *Future Generation Computer Systems*, 21(1):239–246, 2005.

[11] Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, December 2005.

[12] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[13] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[14] Cristian Mateos, Alejandro Zunino, and Marcelo Campo. JGRIM: An approach for easy gridification of applications. *Future Generation Computer Systems*, 24(2):99–118, February 2008.

[15] Cristian Mateos, Alejandro Zunino, and Marcelo Campo. An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. *Computer Languages, Systems & Structures*, 36(3):53–59, 2010.

[16] Chuck Pheatt. Intel®threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[17] J. Sobral and A. Proença. Enabling JaSkel skeletons for clusters and computational Grids. In *IEEE International Conference on Cluster Computing*, pages 365–371. IEEE Computer Society, September 2007.

[18] Gosia Wrzesinska, Rob van Nieuwport, Jason Maassen, Thilo Kielmann, and Henri Bal. Fault-tolerant scheduling of fine-grained tasks in Grid environments. *International Journal of High Performance Computing Applications*, 20(1):103–114, 2006.

[19] Bao-Yin Zhang, Guang-Wen Yang, and Wei-Min Zheng. JCluster: An efficient Java parallel environment on a large-scale heterogeneous cluster. *Concurrency and Computation: Practice and Experience*, 18(12):1541–1557, 2006.

[20] Hua Zhang, Joohan Lee, and Ratan Guha. VCluster: A thread-based Java middleware for SMP and heterogeneous clusters with thread migration support. *Software: Practice and Experience*, 38(10):1049–1071, 2008.

Apéndice A

Signatura	Funcionalidad
obtenerVariableGrid (SF)	Retorna la variable Grid destinada a alojar el resultado de la sentencia *fork* SF.
obtenerPrimerUso (VAR, S)	Retorna la primer sentencia, después de la sentencia S, donde se accede al valor de la variable VAR. La sentencia retornada puede pertenecer a un ámbito incluido dentro del ámbito de S.
ámbito (VAR)	Retorna el ámbito donde se encuentra declarada la sentencia o variable VAR.

Cuadro 1: Funciones auxiliares del algoritmo

Signatura	Funcionalidad
crearTrabajoGrid (SF)	Traduce la sentencia de *fork* SF en la creación de un trabajo Grid independiente. La traducción está ligada a la API de creación de trabajos de la plataforma objetivo.
declararFuture (VG)	Declara y retorna una variable future en el mismo ámbito al que pertenece la variable Grid VG y reemplaza dicha variable por un future en la sentencia donde se produce la asignación del resultado del *fork* asociado a VG.
extraerValorDeFuture (SJ,VG,F)	Reemplaza los usos de la variable VG sobre la sentencia SJ por la extracción de valor del future asociado a dicha variable.

Cuadro 2: Funciones auxiliares del algoritmo de generación de código paralelo