Paradigma de Planificación: Work Stealing

César Bragagnini (cesarbrma91@gmail.com)
Elisban Flores (elisban.flores@gmail.com)
Angela Mayhua (amayhuaq@gmail.com)

Maestría en Ciencia de la Computación
Universidad Católica San Pablo

Junio 2016

1. Introducción

Para la ejecución eficiente de tareas en paralelo sobre una arquitectura MIMD, un algoritmo de planificación debe asegurar que los hilos de los procesadores permanezcan activos para mantener a los procesadores ocupados; además debe asegurar que los hilos activos utilicen razonablemente la memoria porque ésta tiene capacidad limitada [1]. El planificador en lo posible debe mantener los hilos de tareas relacionadas en un mismo procesador para minimizar la comunicación entre los procesadores.

Work Stealing es un paradigma de planificación que permite resolver el problema de planificar tareas en paralelo; en este paradigma los procesadores que tienen menos carga de trabajo intentan robar (*steal*) hilos/tareas de otros procesadore [1].

La idea de work stealing se remonta al trabajo realizado por Burton y Sleep en 1981, titulado *"Executing functional programs on a virtual tree of processors"*, sobre ejecución paralela de programas funcionales; y a la implementación de Multilisp por Halstead presentada en 1984 en su trabajo titulado *"Implementation of Multilisp: Lisp on a multiprocessor"*.

2. Algoritmo Work Stealing

A continuación se detalla la lógica de funcionamiento de un planificador basado en Work Stealing [1; 6; 8]:

- Se tiene una cola global y un *deque (double-ended queue)* por cada worker (Figura 1), donde se mantiene las subtareas a ejecutar. Se usa un deque porque permite realizar operaciones LIFO y FIFO.
  - *Push*: worker local agrega las nuevas subtareas.
  - *Pop*: worker local remueve las tareas a ejecutar.
  - *Steal*: worker remoto remueve las tareas.

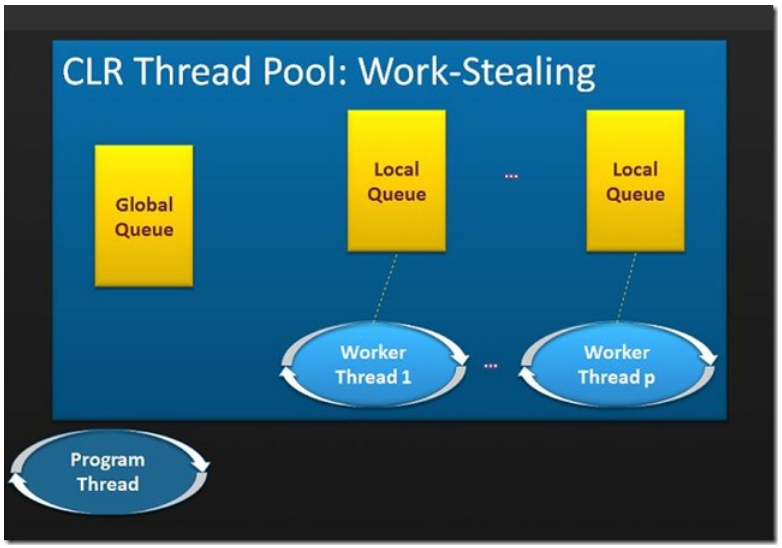

Figura 1: Componentes: una cola global y un deque por cada worker

- Las tareas generadas por el hilo principal del programa son almacenadas en la cola global (Figura 2).

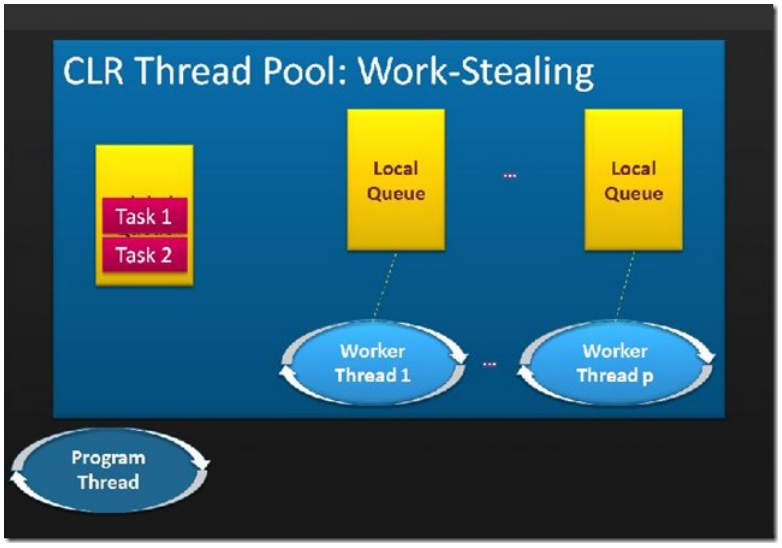

Figura 2: La cola global almacena las tareas del hilo principal

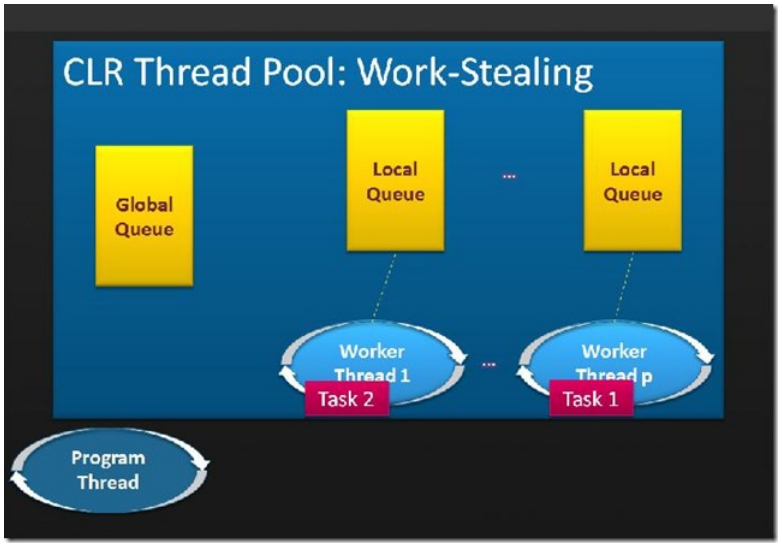

Figura 3: Los workers toman la tarea de la cola principal

- Cada worker puede tomar una tarea de la cola global para procesarla (Figura 3).
- Las subtareas generadas por la ejecución de alguna tarea en un worker son colocadas en su propia cola (Figura 4).

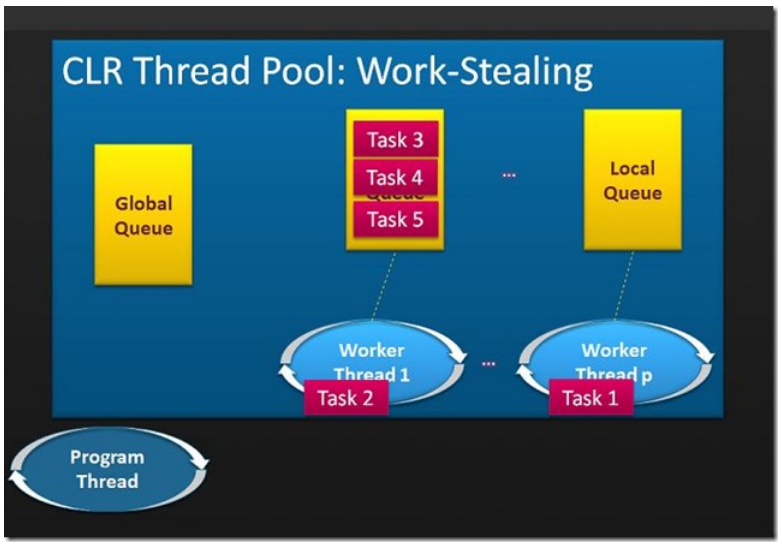

Figura 4: Una tarea puede generar subtareas que son almacenadas en el deque local

- Cuando un worker culmina con su tarea actual, procesa su propia cola y extrae la última tarea ingresada (orden LIFO); lo que mejora la localidad porque esta tarea podría estar aún en caché (Figura 5).
- Cuando un worker no tiene tareas para ejecutar en su propia cola, verifica si hay tareas en la cola global y si no hay tareas intenta tomar (*steal*) una tarea de otro worker elegido aleatoriamente, tomando la tarea más antigua de esa

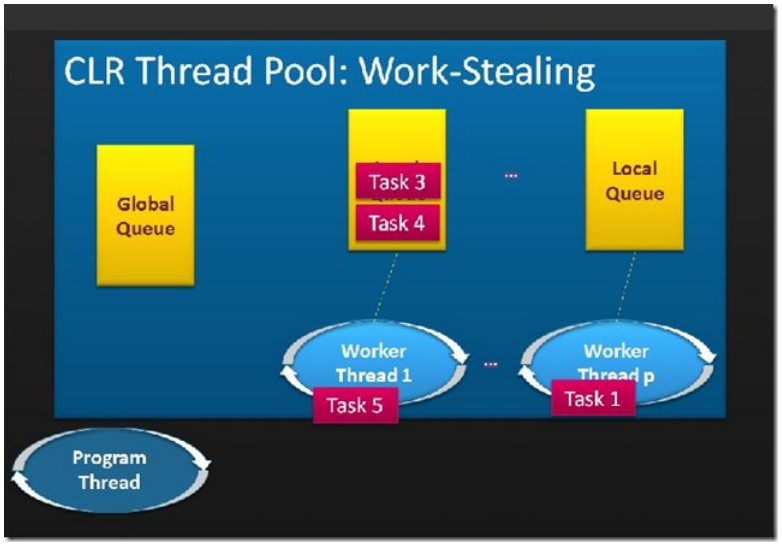

Figura 5: El worker procesa su cola local en orden LIFO

cola (orden FIFO) porque es menos probable que se encuentre en la caché del worker seleccionado (Figura 6).

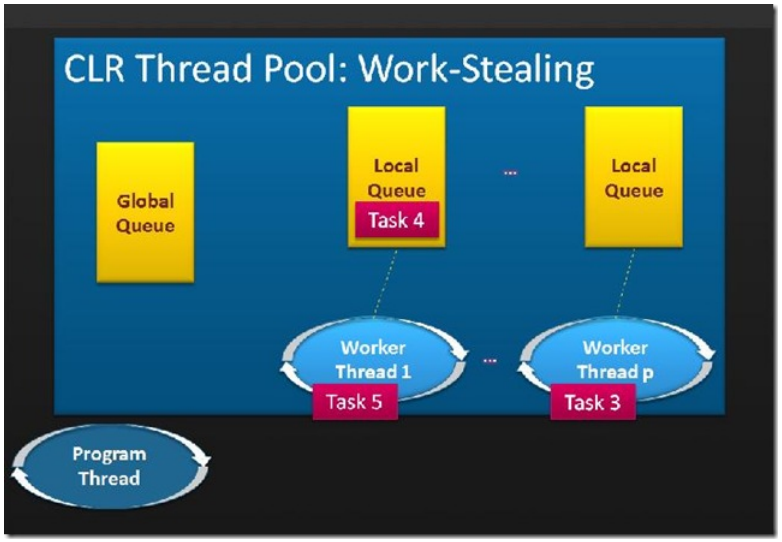

Figura 6: Si no hay tareas en la cola local, el worker toma una tarea de otro worker aleatorio

- Si la tarea tomada genera subtareas, éstas son insertadas en la propia cola del worker (Figura 7).
- Cuando un worker no tiene trabajo o falla al "robar" de otro worker, retrocede e intenta nuevamente, a menos que todos los workers se encuentren en estado de reposo, en este caso todos son bloqueados hasta que otra tarea sea invocada desde el nivel más alto.

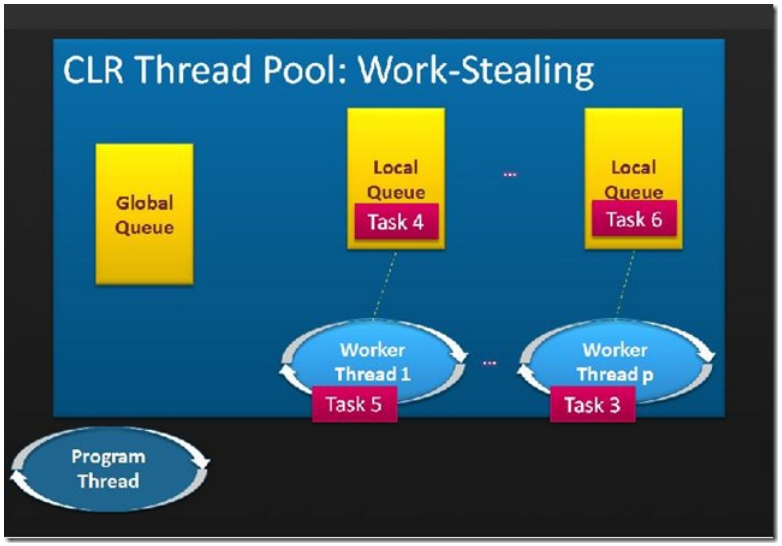

Figura 7: Toda subtarea generada por una tarea es almacenada en la cola local del worker

3. Características principales de Work Stealing

- La comunicación se realiza sólo cuando hay inactividad, es decir que no hay comunicación mientras los procesadores están en su capacidad máxima.
- Cada tarea puede ser robada a lo sumo una vez.
- Este planificador asume que los stealers siempre tendrán éxito.
- Es un algoritmo distribuido escalable.
- El tiempo de ejecución es teóricamente calculable.
- El número de intentos de "robo" es teóricamente calculable.

4. Análisis de desempeño

4.1. Tiempo de ejecución

La version inicial presentada por Blumofe y Leirzon[1], es ejecutado en:

$$T_1/P + \mathcal{O}(T_\infty)$$

- P es el número de procesadores.
- T_1 cantidad de tiempo minimo para una ejecucion serial.

- T_∞ es el span(cantidad de tiempo de ejecución en una maquina ideal con un número infinito de procesadores).

4.2. Memoria

La version inicial presentada por Blumofe y Leirzon[1], estable que la cantidad de espacio es S_p donde se cumpl que

$$S_p < S_1 P$$

- S_1 es la cantidad de memoria usada por el stack de un solo procesador.
- P número de procesadores.

4.3. Tiempo de comunicación

La version inicial presentado por Blumofe y Leirzon[1] es a lo mas:

$$T_\infty S_{max} P$$

- S_{max} es el tamaño del mas largo de registro de activación de algun thread.
- T_∞ es el span(cantidad de tiempo de ejecución en una maquina ideal con un número infinito de procesadores).

5. Implementaciones de Work Stealing

Existen varias librerías y lenguajes que implementan su propio algoritmo de Work Stealing [1], entre ellos tenemos:

5.1. Intel® Cilk™ Plus

Extensiones para simplificar el paralelismo de procesos y datos

Intel® Cilk™ Plus[3] añade extensiones simples a los lenguajes C y C++ para expresar el paralelismo de procesos y datos. Esta extensión del lenguaje es poderosa, y fácil de aplicar y usar en una amplia gama de aplicaciones.

En Cilk++ su balanceamiento de carga dinámico se adapta bien en entornos de mundo real, esto es "si un worker queda como desplanificado (porque otra aplicación comienza a ser ejecutada), otro worker puede robar el trabajo que está ejecutando este worker desplanificado"[7].

Los programas desarrollados con Cilk++ son performance-composable, esto si una librería paralela desarrollado en Cilk++, esta librería no solo puede ser llamada únicamente por un programa serial o porciones de código paralelo puede ser invocada múltiples veces por varios programas dando un buen rendimiento, en contrastes con otra plataformas que limitan sus librerías para ser ejecutado en un número dado de procesadores, si hay varias librerías ejecutándose en simultáneo chocan entre ellas porque compiten con los recursos. Intel Cilk Plus incluye las siguientes características y beneficios [3]:

Características	**Beneficios**
Keywords	Simple, expresion ponderosa de paralelismo de procesos: ▪ *cilk_for* Paralelización de bucles ▪ *cilk_spawn* Especifica que una función puede ejecutar en paralelo con el resto de la función de llamada. ▪ *cilk_sync* Especifica que todas las llamadas generados en una función deben completar antes de que se continúe con la ejecución.
Reducers	Elimina la contención de las variables compartidas entre las tareas mediante la creación automática de vistas según sea necesario y reduciéndolos de una manera libre de bloqueo.
Array Notation	Paralelismo de data para arrays o secciones de arrays.
SIMD-Enabled Functions	Define funciones que pueden ser vectorizadas cuando son llamadas en expresiones con notación de arrays ó una expresión #pragma simd loop.
#pragma simd	Especifica si un bucle se vectoriza.

5.2. Intel Threading Building Blocks

Es una biblioteca que le ayuda a aprovechar el rendimiento de procesadores multinúcleos sin tener que ser un experto en threading. La ventaja de Intel TBB [11] es que funciona a un nivel más alto que la thread primitivos, sin embargo, no requiere lenguajes o compiladores exóticos. La biblioteca se diferencia de otros en las siguientes maneras:

- TBB le permite especificar el paralelismo lógico en lugar de los threads;
- TBB hace uso del threading para un mejor rendimiento;
- TBB es compatible con otros paquetes de threading;
- TBB hace hincapié en la programación en paralelo escalable de los datos;
- TBB se basa en la programación genérica, (por ejemplo .: uso de Standar Template Library en C ++).

TBB [11] implementa Work Stealing para equilibrar la carga de procesos en paralelo a través de los núcleos de procesamiento disponibles con el fin de aumentar la utilización del núcleo y por tanto la escalabilidad. En un principio, la carga de procesos se divide en partes iguales entre los núcleos de procesamiento disponibles. Si un núcleo completa su trabajo, mientras que otros núcleos todavía tienen una cantidad significativa de trabajo en su cola, TBB vuelve a asignar una parte del trabajo de uno de los núcleos que permanece ocupado a uno que está inactivo. Esta capacidad dinámica libera al programador de la máquina, permitiendo que las aplicaciones escritas usando la biblioteca utilicen los núcleos de procesamiento disponibles sin necesidad de realizar ningún cambio en el código fuente o el archivo de programa ejecutable.

5.3. Satin

El modelo de programación Satin [10] es una extensión del modelo Java de un solo Thread. Para lograr la ejecución en paralelo, los programas de Satin no tienen que usar hilos de Java o invocaciones de métodos remotos (RMI). En su lugar, utilizan las primitivas de divide y vencerás mucho más simples

Satin [10] es un sistema para correr programas con algoritmos de divide y vencerás en sistemas de memoria distribuida (y últimamente en un amplio rango sistemas metacomputing). El compilador Satin y el tiempo de ejecución del sistema cooperan para implementar estas primitivas de manera eficiente en un sistema distribuido, utilizando el Work Stealing para distribuir las cargas de trabajo. Satin optimiza la sobrecarga de los trabajo locales utilizando la serialización en demanda, lo que evita la copia y la serialización de los parámetros para los trabajos que no han sido robados. Esta optimización está implementada utilizando la invocación explícita de registros

5.4. Kaapi

Similar a TBB/Cilk pero con dependencias del flujo de datos, es un soporte de ejecución para la planificación de los programas de flujo de datos macro irregulares

en un clúster de multiprocesadores utilizando el algoritmo Work Stealing.

El modelo de programación se basa en un espacio de direcciones global llamada memoria global y permite describir las dependencias de datos entre las tareas que tienen acceso a los objetos de la memoria global. El lenguaje extiende C ++ con dos palabras clave. La palabra clave shared es un tipo clasificador para declarar objetos en la memoria global. La palabra clave fork crea una nueva tarea que pueda ser ejecutado en concurrencia con otras tareas.

La comunicación en KAAPI [5] se basa en mensajes activos. El envío de un mensaje es una operación sin bloqueo. El proceso emisor será notificado con la comunicación de que los datos que componen mensajes han sido enviados de manera efectiva. En su recepción, una función definida por el usuario es llamado por el sistema de Ejecución KAAPI para procesar el mensaje. El no bloqueo de envío es importante y permite a superponerse de manera eficiente a los retardos de comunicación por cálculo. Por otra parte, en KAAPI, los mensajes generados por un Thread KAAPI se ponen en una cola privada (local). Para cada tarjeta de interfaz de red, un thread POSIX llamado demonio, es el encargado de enviar mensajes. La programación de los mensajes enviados se basa en un algoritmo de Work Stealing: el demonio roba indefinidamente mensajes de las colas de Threads KAAPI y los envía. Cada operación de robo trata de obtener la mayor secuencia de mensajes. Esto permite agregar mensajes juntos y disminuye el retardo en el inicio de transferencia por mensaje. Por otra parte, este tipo Work Stealing para la comunicación permite ocultar retrasos en la red debido a la sobrecarga de la esta: mientras que el demonio se bloquea durante una utilización de la red, los threads KAAPI pueden seguir generando mensajes. La próxima vez que el daemon roba secuencias de mensajes más grandes, así agrega más mensajes.

5.5. X10 / XWS

X10 es un lenguaje de programación orientado a objetos basado en java diseñado para programación paralela, es de código abierto desarrollado por IBM. XWS(X10 Work Stealing)[2] framework es de código abierto para tiempo de ejecucion y es usado para X10, implementa concurrencia de grano fino

5.6. PARLANSE

LANguage for Symbolic Expression para Windows en procesadores X86 [4], ofrece un buen paralelismo de grano, logrando planificar bloques de 100 a 200 instrucciones consiguiendo un paralelismo útil. Implementa work-stealing de tal manera que los procesadores cooperativamente pueden consumir y generar menor unidades de paralelismo, al haber montones de trabajo(works) disponible de tamaño modesto, por lo que cada procesador siempre tiene algo que hacer.

5.7. Problemas de implementación en C/C++ puro

En C/C++ esta diseñado de tal manera que realizar el cambio de contexto(ejemplo setear el contexto para que un procesador registre una pila) no puede ser declarado puramente en C/C++, esto puede ser resuelto escribiendo una parte del paquete en el lenguaje de la maquina objetivo, otro problema es el acceso atómico a colas para multiprocesadores no puede hacerse puramente en C/C++, se tendría que implementar el algoritmo de Decker(soluciona el problema de mutua exclusión en programación concurrente) o implementar código semejante a como trabaja el lenguaje ensamblador para sincronización de primitivas.

6. Variantes

Algunas variantes del work stealing son presentadas en las subsiquientes secciones.

6.1. Probabilistic Work Stealing(PWS)

PWS[9] se basa en la probabilidad de elegir un worker objetivo por algún stealer no es uniforme sino es inversamente proporcional a la distancia entre el stealer y el worker. Esto da la ventaja de aumentar la localidad de data y reducir el tiempo de demora de una steal-request.

6.2. Hierarchical Work Stealing(HWS)

HWS[9] su principio es el mismo que es usada en el mecanismo de un cluster. Los procesadores son agrupados de acuerdo a la conexión que mantienen entre ellos (una rápida conexión entre ellos), cada grupo tiene un líder steal, el cual se encarga de robar procesos a otros grupos pero esto incrementa el balance de carga, para sopesarlo el líder roba la mayor cantidad posible de tareas. El líder es el encargado de distribuir las tareas entre su grupo.

6.2.1. Variantes de HWC

En la librería Satin [10] se han incluido tres heurísticas de hierarchical work-stealing:

- **Cluster-aware Hierarchical Stealing(CHS)[9]** cada cluster es representado por un árbol, a la hora de que hay una solicitud de steal, la víctima pregunta a sus hijos. En el caso de que haya algún obstáculo, la solicitud es transferida a su padre. Esta heurística intenta favorecer la localidad del trabajo y disminuir el número de solicitudes de steal remotas.
- **Cluster-aware Load-Based Stealing(CLS)[9]** la plataforma es dividida en muchos clusters, en cada cluster se elige a un líder. Cada computadora envía información acerca de la cantidad de trabajo que tiene a su líder. Cada líder toma en cuenta esa información para decidir robarle a una computadora en otro cluster. Esta heurística reduce el número de transferencias de datos entre clusters.
- **Cluster-aware Random Stealing(CRS)[9]** los worker perciben dos tipos de herencia: workes en el mismo cluster y en otros cluster. Cada worker es capaz de enviar dos tipo de intentos de Steal: asincrono y síncrono, La asíncrona está restringida a un worker en otros clusters al mismo tiempo. En cambio la síncrona está restringida a workes del mismo cluster. Las steal-request son enviadas solo cuando la pila esta vacía, tal como sería en una work stealing clásico.

7. Conclusiones

- Work Stealing es un excelente algoritmo para balanceo de carga.
- Existen varias librerías que implementa su versión de work stealing, aunque la idea principal se mantiene, agregan ciertas funcionalidades para mejorar su rendimiento.
- La idea principal del Work Stealing es trivial, pero su implementación en lenguajes de programación de uso común puede no serla, ya que estos lenguajes presentan limitaciones.
- Work Stealing es mejor en forma aleatoria, porque una búsqueda secuencial global en cada cola agregaria tiempo considerable.

Referencias

[1] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[2] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *2008 37th International Conference on Parallel Processing*, pages 536–545, Sept 2008.

[3] Intel Corporation. Introducing Intel Cilk Plus: Extensions to simplify task and data parallelism. `https://www.cilkplus.org/cilk-plus-tutorial`, 2014.

[4] Semantic Designs. A PARallel LANguage for Symbolic Expression for 80x86 Symmetric Multiprocessors under Windows. `http://www.semdesigns.com/products/parlanse/index.html`.

[5] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA, 2007. ACM.

[6] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[7] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, New York, NY, USA, 2009. ACM.

[8] Jennifer Marsman. Work-Stealing in .NET 4.0. `https://blogs.msdn.microsoft.com/jennifer/2009/06/26/work-stealing-in-net-4-0/`, 2009.

[9] Jean-Noël Quintin and Frédéric Wagner. *Hierarchical Work-Stealing*, pages 217–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[10] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. *Satin: Efficient Parallel Divide-and-Conquer in Java*, pages 690–699. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[11] Jan Verschelde. Introduction to the Intel Threading Building Blocks. `http://homepages.math.uic.edu/~jan/mcs572/lec11.pdf`, 2014.