UNIVERSIDAD CARLOS III DE MADRID

DISEÑO Y OPTIMIZACIÓN DE FUNCIONES PARA UNA EJECUCIÓN MULTICORE EN R

Trabajo Fin de Grado

Grado en Ingeniería Informática

15/06/2012

Carlos Villalba Coronado

Tutores:

Daniel Higuero Alonso-Mardones

y

Juan Manuel Tirado Martín

Índice

1 Introducción

Este trabajo consiste en la optimización de funciones computacionalmente pesadas de una librería matemática de grafos, así como de su evaluación posterior. Dicha optimización está realizada aprovechando las características de las computadoras con varios núcleos.

En muchas ocasiones los ordenadores tienen que hacer operaciones que requieren una enorme cantidad de cálculos. Esto hace que gaste mucho tiempo haciendo todas esas operaciones. Este tipo de cálculos son lo que se suelen llamar operaciones pesadas computacionalmente. Dichas operaciones, hacen que los ordenadores estén un determinado tiempo dedicados exclusivamente a resolver un problema.

Por este motivo, es fundamental mejorar el rendimiento de este tipo de cálculos. Ganar tiempo al realizar este tipo de tareas no sólo significa obtener el resultado antes. También significa tener más tiempo de cómputo para hacer otras tareas o ahorrar electricidad.

Optimizar los programas informáticos que consumen tiempo de cálculo es una labor importante, aunque puede no resultar algo sencillo. Hay muchos tipos de optimizaciones y formas de mejorar el rendimiento de un programa, en este trabajo nos vamos a centrar en el aprovechamiento de las capacidades de los ordenadores de varios núcleos.

Desde hace pocos años los procesadores de uso doméstico han dejado de tener un único procesador como lo han tenido tradicionalmente, a tener varios. Al principio, dos o tres y actualmente hasta ocho o más. La razón es que durante años los ordenadores con un solo procesador han ido aumentando el número de instrucciones que podían ejecutar, es decir, su velocidad de cómputo. Podían realizar más trabajo en menos tiempo. Sin embargo, llegó un punto en el que por motivos físicos ese aumento de velocidad cesó. Dado que la tecnología si que permitía hacer cada vez los procesadores más pequeños y baratos, los ordenadores empezaron a tener más de un procesador.

Estos nuevos ordenadores multinúcleo o multicore, cambian completamente la forma de la que se dispone de la potencia. Al tratarse de varios procesadores independientes, si se quiere sacar provecho de este tipo de arquitecturas hay que utilizar cada procesador de forma independiente, pero con un objetivo común. En resumen, si una tarea es pesada se puede repartir entre los distintos procesadores.

Sin embargo, en ocasiones eso no es tan sencillo. Las tareas, tanto en informática como en otros campos a veces se pueden repartir mejor y a veces no. Por ejemplo, al hacer una mudanza, el cargar y descargar el camión es una tarea que se hará más rápido cuantas más personas lo hagan. Por el contrario, da igual cómo se haga que, si

vale con un viaje, en llevar el camión de una casa a la otra no se gana tiempo, por muchas personas y camiones que tengas.

La otra dificultad que tiene aprovechar varios núcleos de un mismo ordenador es la sincronización entre los procesos. El orden en el que se realizan determinadas tareas puede variar el resultado final. Hay que poder garantizar que estas tareas se hagan en un determinado orden. Además, aunque los procesadores sean independientes, si acceden a una memoria común. También hay que evitar que un núcleo lea de la memoria mientras otro la escribe o que escriban a la vez, ya que esto también puede causar ciertos problemas.

Como se puede ver, optimizar un programa para este tipo de máquinas no es demasiado sencillo. Sin embargo, sí puede llegar a ser muy beneficioso. Si se consigue repartir perfectamente la tarea pesada en varias tareas asignadas a cada procesador, el ordenador podría realizar la tarea varias veces más rápido, casi tantas como núcleos tenga. Aunque eso sería en condiciones ideales.

Por otra parte, la librería que se va a optimizar es de grafos. La Teoría de Grafos tiene mucha importancia en la actualidad (ver motivación). Y aunque los grafos pequeños no requieren muchos cálculos. Los grafos de gran tamaño si lo hacen. Asimismo, son cálculos que normalmente se pueden repartir bien entre los distintos núcleos.

1.1 Motivación

En el siglo XXI, vivimos en un mundo rodeados de redes: redes eléctricas, redes de transporte, redes sociales, redes de telefonía, internet (la red de redes),... Para estudiar y optimizar los recursos de estas redes, cobra mucha importancia en la actualidad la llamada teoría de grafos, encargada de estudiar los grafos.

Un grafo es una forma muy simple, pero muy útil de representar información. Básicamente se representan elementos (llamados nodos o vértices) y relaciones entre ellos (llamadas aristas). De esta forma, los grafos son ideales para representar grandes redes u otro tipo de información como relaciones entre personas, o transiciones entre estados.

Probablemente uno de los usos más importante de la teoría de grafos sea la optimización de redes. Una empresa eléctrica, por ejemplo, puede diseñar sus redes mediante grafos antes de construirla y simular diferentes tipos de funcionamientos y fallos con el fin de construir finalmente la red óptima para sus necesidades, tanto en recursos como en flexibilidad. Asimismo, una vez construida podrá seguir usando los grafos para mejorar la red conforme evolucionan sus necesidades, de una forma dinámica.

Otro uso importante es la realización de estudios. Con la gran expansión de las redes

sociales en internet, su uso ha aumentado para realizar todo tipo de estudios sociológicos. Del mismo modo, se usan para realizar estudios en otros campos como la biología dónde se modelan las migraciones de las especies, por ejemplo.

Los autómatas y procesos automáticos utilizados en todo tipo de campos se basan en las transiciones de estados. Una forma de útil y práctica es representarlos por medio de grafos, dónde los vértices son los estados y las aristas las transiciones. Los grafos permiten diseñar y simular estos procesos antes de construirlos. Estos sistemas automáticos cada vez son más grandes, complejos y más extendidos en todo tipo de campos por lo que la teoría de grafos cobra mayor importancia.

En definitiva, las operaciones sobre grafos de gran tamaño cada día son más habituales, ya que cada día todo está más conectado, los sistemas y los estudios son más grandes y complejos y una gran cantidad de campos utilizan la teoría de grafos para su beneficio. Sin embargo, estas operaciones con grandes grafos requieren una gran capacidad computacional. La motivación de este trabajo es mejorar el rendimiento de algunas operaciones de una librería que trabaje con grafos. Para ello se ha seleccionado igraph [1] como la librería indicada.

Igraph no es una librería diseñada para aprovechar las máquinas multinúcleo, realiza sus operaciones de forma secuencial usando un único procesador. Sin embargo, al tratarse de un proyecto de código libre y abierto si se puede modificar con facilidad dicho código para realizar este tipo de mejora. Esta es una de las principales razones por la que se ha elegido esta librería.

Otra de las razones es que igraph es probablemente la principal librería especializada en grafos con interfaz R [2]. Por esta razón, igraph ha mantenido e incluso aumentado sus descargas en los últimos años como se puede ver en el siguiente gráfico.

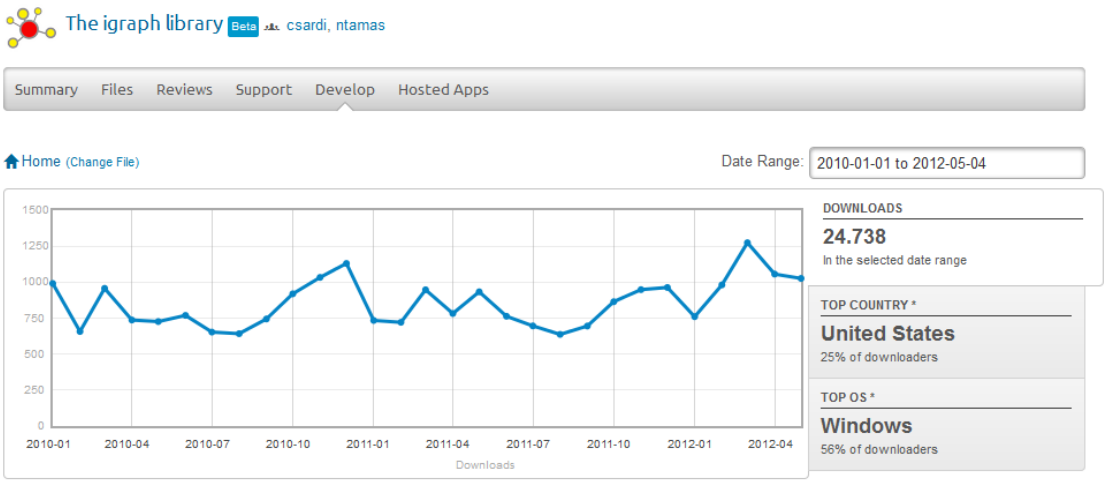


Gráfico 1: *El gráfico muestra las descargas de igraph desde 2010, hasta mayo de 2012.*

Existen otras librerías de grafos como puede ser Lemon [3], que tiene interfaz en Python, pero no en R. Otra alternativa es boost [4], un conjunto de librerías escritas en C++ y que tampoco tiene interfaz en R. Además, boost no tiene los algoritmos implementados los algoritmos que se van a optimizar en este trabajo.

1.2 Objetivos

El principal objetivo del trabajo es desarrollar funciones para grafos que aprovechen las características de las arquitecturas multicore y mejoren su tiempo de cómputo. Para ello habrá que cumplir una serie de objetivos secundarios.

Habrá, en primer lugar, que **desarrollar la librería** en sí misma. Partiendo de una librería ya existente hacer los cambios necesarios para **que funcione en máquinas multicore**.

Después tendrá que poder ser **importada por un paquete matemático**, en el caso particular de este trabajo se ha elegido R por las razones que se exponen posteriormente, en el estado del arte.

El siguiente objetivo será **paralelizar las funciones**, aprovechando las ventajas de las arquitecturas multinúcleo.

Por último, se realizará la **evaluación del rendimiento y la ganancia** obtenida con las nuevas funciones, respecto a las antiguas.

1.3 Secciones

Este documento está estructurado por secciones. Incluyendo esta primera sección en la que se introduce el tema, se comentan la motivación y los objetivos. También se habla, entre otras cosas, de la importancia de este trabajo en la actualidad.

El estado del arte es la segunda sección. En dicha sección, primero se analizan las diferentes alternativas de software matemático existente para este tipo de trabajo con sus pros y sus contras. Se hace con el objetivo de tomar la mejor decisión para las necesidades particulares del proyecto. Después, se estudian diferentes tecnologías que se pueden utilizar para mejorar el rendimiento en máquinas multicore, eligiendo la más adecuada. Y por último se hace un pequeño resumen de lo qué es un Small World, cómo se calcula y los conceptos básicos de teoría de grafos que se necesitarán para entender este trabajo.

La tercera es el desarrollo. Dónde se explican las principales características de igraph y cómo se crea un paquete en R.

La cuarta sección es la implementación. En esta sección se explica el trabajo realizado sobre la librería igraph existente. Incluyendo un análisis del funcionamiento de los

algoritmos implementados y las alternativas de optimización, eligiendo la más beneficiosa.

La evaluación es la quinta. La evaluación contiene las pruebas realizadas, incluyendo gráficos de tiempos y la mejora obtenida. Así como unos comentarios del rendimiento de las funciones con diferentes datos de entrada y número de procesadores asignados.

Las últimas secciones son la planificación (sexta), el presupuesto (séptima), las conclusiones finales y las líneas futuras (octava) y por último están las referencias utilizadas en el trabajo (novena).

2 Estado del arte

En esta sección se va a estudiar el trabajo existente en la materia. Se van a analizar las alternativas más importantes con el fin de elegir la solución que mejor se adapte a las necesidades del proyecto.

Dado que el proyecto es la optimización de algunas funciones pesadas de un paquete matemático para máquinas multi-core. Lo que se va a buscar con el trabajo previo es ver qué paquete puede ser el más adecuado para realizar dicha optimización. Es importante que ese paquete o librería sea compatible o pueda ser adaptado sin demasiadas complicaciones a un software matemático de uso común. Asimismo, se van a estudiar las tecnologías disponibles para realizar la optimización.

Por último, se explicarán los conceptos básicos necesarios sobre la teoría de grafos y se expondrá qué es un Small World y cómo se calcula.

2.1 Software matemático

Si bien una librería de grafos puede escribirse en un lenguaje determinado y ejecutarse directamente, es interesante que pueda utilizar un software matemático a modo de interfaz. Esto aporta varias ventajas, pero principalmente es una cuestión de usabilidad de cara al usuario. La idea es que el usuario usé un único software matemático y que se amplié su funcionalidad mediante librerías. Uno de los objetivos de este trabajo previo es buscar qué software se adapta mejor a nuestros requisitos.

Para ello, se han elegido como muestra cuatro alternativas de paquetes matemáticos ampliamente usados y conocidos especialmente en el mundo de la investigación. Dicha muestra está constituida tanto por programas comerciales, como S-PLUS y Matlab, como por programas con licencia libre y de código abierto, como R y Octave. Además, mientras que Matlab y Octave son programas orientados al cálculo numérico con matrices, S-Plus y R están más orientados al campo estadístico.

2.1.1 R

R es un entorno y un lenguaje de programación diseñado para el análisis estadístico y gráfico, aunque también se puede usar de forma muy eficaz para otros campos como el cálculo numérico.

Fue creado en 1993 por Ross Ihaka y Robert Gentleman del Departamento de Estadística de la Universidad de Auckland, en Nueva Zelanda. En la actualidad, se trata de un proyecto GNU de software libre desarrollado por el R Develpment Core Team y distribuido de forma gratuita bajo licencia GNU General Public License.

R es una implementación del lenguaje de programación S, y está influido también por Scheme, un dialecto de Lisp. Es multiparadigma, mezclando diferentes paradigmas de programación: orientación a objetos, imperativa, funcional, por procedimientos y

reflexiva.

Se pueden desarrollar bibliotecas para ampliar la funcionalidad de R. Estas librerías se cargan dinámicamente y pueden ser escritas en C, C++ y Fortran. Además existen paquetes que permiten usar lenguajes interpretados como Perl o Python, bases de datos e interfaces gráficas. R tiene versiones para los sistemas operativos Windows, GNU/Linux, Unix y Macintosh.

En investigación estadística, R es junto con S-PLUS, el lenguaje más utilizado. Se utiliza en múltiples campos como las matemáticas financieras o la investigación biomédica. En 2010 fue la herramienta más usada en minería de datos [5].

2.1.2 S-Plus

S-Plus [6] es uno de los paquetes de análisis estadístico más importante. Es la principal implementación comercial del lenguaje de programación S, creado por los Laboratorios Bell.

En 1988, la primera versión de S-PLUS salió al mercado desarrollado por la compañía Statistical Sciences Inc. creada R. Douglas Martin, profesor de estadística de la Universidad de Washington en Seattle.

En 2008, TIBCO adquirió la empresa que poseía los derechos de S-PLUS y desde entonces lo distribuye. El precio anual de la licencia varía según las características requeridas. Se conceden tres tipos de licencias: personal a 99$, editor a 1000$ y analista a 4500$. Hay versiones para Windows y para Linux, pero no para Mac OS.

Desde la versión 8, lanzada en 2007, S-PLUS es compatible con los paquetes de R. Esto hace que se beneficie de todos los paquetes ya creados para R y que sea más sencillo extender su funcionalidad. Aunque anteriormente ya se podían realizar extensiones mediante su conexión a otros lenguajes como C, C++, Java y Fortran.

2.1.3 MatLab

MatLab [7] es un programa matemático de cálculo numérico orientado a matrices. Su nombre es la abreviatura de Matrix Laboratory, que en inglés significa laboratorio de matrices. Utiliza M, que es un lenguaje de programación propio y un IDE (Entorno de Desarrollo Integrado).

MatLab lo empezó a desarrollar Cleve Moler a finales de los 70. Cleve Moler era matemático y trabajaba en el Departamento de informática de la Universidad de Nuevo México. Empezó a desarrollar MatLab con el objetivo de proporcionar a sus estudiantes un acceso más sencillo a las librerías matriciales LINPACK [8] y EISPACK [9], evitando el uso de Fortran. Descubriendo su potencial comercial fundó junto con Jack Little una empresa llamada MathWorks en el año 1984. MathWorks es la encargada de

comercializar MatLab.

El coste de la licencia de MatLab es muy variable. Aparte del coste del paquete básico hay paquetes especializados para distintos sectores o cometidos específicos, como pueden ser el paquete financiero o el de procesamiento de imagen. También hay herramientas especializadas que se distribuyen por adicionalmente al paquete básico, como el compilador Matlab que te permite hacer ejecutables de tus programas.

Matlab permite utilizar código C/C++ y Fortran mediante los llamados MEX-Files. Estos archivos actúan de interfaz entre Matlab y las subrutinas escritas en estos lenguajes y permiten cargarlas dinámicamente. Igualmente, Matlab tiene versiones para los sistemas operativos más extendidos: GNU/Linux, Mac OS y Windows.

Tanto en el campo académico y científico, como en el empresarial, Matlab es uno de los programas matemáticos más extendidos. Con más de un millón de usuarios. Los paquetes específicos y las herramientas asociadas siguen creciendo y cada vez tiene más aplicaciones.

2.1.4 Octave

Octave [10] es un software de cálculo numérico. Octave se desarrolló pensando en hacerlo compatible con MatLab, por esta razón se le considera su versión libre. Al igual que éste, ofrece un intérprete de mandatos que permite ejecutar las instrucciones de forma interactiva.

El proyecto Octave empezó en 1988. Fue pensado en un primer momento para usarlo en un curso de diseño de reactores químicos. Sin embargo, en 1992 se decide extenderlo y se pone a cargo del proyecto John W. Eaton. En 1994, tras dos años de trabajo, se lanza la primera versión. Tiene licencia GNU GPL y por lo tanto se distribuye de forma libre y gratuita.

Octave tiene su propio lenguaje de sintaxis similar a Matlab. Además, puede cargar archivos de extensión .m con funciones escritas en Matlab. Para extender su funcionalidad puede cargar y ejecutar funciones y subrutinas escritas en C, C++ y Fortran.

GNU Octave es un programa multiplataforma y posee distribuciones para los principales sistemas operativos de propósito general. Tiene un gran número de usuarios en el mundo científico y de investigación. Así como gran cantidad de paquetes de uso específico para distintos sectores o propósitos.

2.1.5 Comparación

Para evaluar las posibilidades que ofrece cada herramienta se han evaluado cuatro características: los sistemas operativos dónde se pueden ejecutar, los lenguajes en los que se pueden desarrollar, el precio de la licencia de desarrollo y la disponibilidad de

librerías teniendo en cuenta no sólo la cantidad sino también el precio y el tipo de licencia.

Todos los programas tienen un peso importante no sólo en el mundo científico y académico sino también en el empresarial. Tienen también distribuciones en los principales sistemas operativos, a excepción de S-PLUS que no tiene ninguna versión para Mac OS de Apple.

Respecto a las librerías, en las versiones comerciales dan menos soporte al desarrollo. Además, muchas veces son de pago. No obstante también hay paquetes con licencia GPL. S-PLUS además de ser la menos extendida es la que menos librerías adicionales ofrece. En ese aspecto, R sale ganando, no sólo tiene gran cantidad de librerías sino que son de código abierto y de distribución gratuita.

En relación a los lenguajes soportados, todos los programas dan soporte a C/C++ y Fortran que son los principales lenguajes utilizados para el cálculo numérico y la computación científica. Además algunos dan soporte a otros lenguajes como Java o Python.

En el precio de la licencia si hay diferencias significativas. R y Octave tienen licencia GNU General Public License y por lo tanto su distribución es gratuita, además son de código abierto lo que permite realizar las mejoras que se proponen. Por el contrario Matlab y S-Plus son programas propietarios lo que incrementa el coste.

Programa	Sistema Operativo	Lenguajes soportados	Precio licencia comercial	Disponibilidad de librerías
Matlab	Multiplataforma	C/C++, Fortran y Java	2000€	Media
Octave	Multiplataforma	C/C++ y Fortran	Gratuita	Media
R	Multiplataforma	C/C++ y Fortran	Gratuita	Alta
S-Plus	Windows y Linux	C/C++, Fortran y Java	1000$ / año	Baja

Tabla 1: Comparativa software matemático.

Se puede descartar Matlab por el precio de la licencia y porque GNU Octave tiene buena compatibilidad con su lenguaje. S-Plus también se podría descartar ya que además del precio de su licencia, R es un lenguaje más extendido y S-Plus soporta sus paquetes por lo que si se realizase una mejora sobre un paquete en R, S-Plus también se beneficiaría, pero no a la inversa. En definitiva, la decisión dependerá en gran medida del paquete específico y del valor que se le dé, pero normalmente no merecerá la pena pagar una licencia comercial teniendo paquetes de gran calidad gratuitos y de código abierto. R tiene más posibilidades de tener un paquete de estas características.

2.2 Multiprocesamiento

Las funciones de la librería igraph están pensadas para que los pesados cálculos computacionales de los grafos se hagan de forma secuencial utilizando un único procesador. Con el fin de mejorar el tiempo de resolución de dichas operaciones se va aprovechar la arquitectura de las máquinas multinúcleo. Se va a realizar un cambio en la librería de forma que se reparta la carga de trabajo entre los distintos procesadores de la máquina. Esto debería ser muy beneficioso ya que actualmente la mayoría de los ordenadores personales tienen más de un procesador y la tendencia es a aumentar el número de núcleos.

En una máquina con varios procesadores y memoria compartida la forma de mejorar el tiempo de cómputo de una tarea pesada es dividir esa tarea en varias tareas más pequeñas que se puedan realizar de forma simultánea en los distintos procesadores sin alterar el resultado final. A esto se le llama paralelizar una tarea.

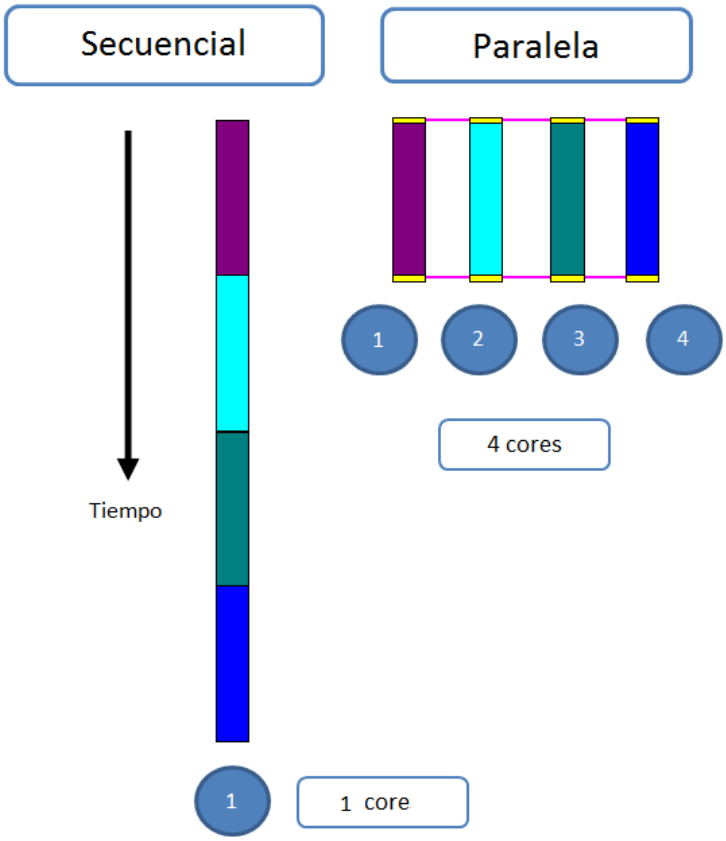


Ilustración 1: Diagrama de ejecución secuencial y de ejecución paralela.

Cuando se paraleliza una tarea es importante tener en cuenta las dependencias y las condiciones de carrera sobre los recursos. En un código secuencial, el orden de ejecución de las instrucciones es predecible y un solo proceso accederá a los recursos. Sin embargo, en un código paralelo el orden de las instrucciones es variable y varios procesos podrían intentar acceder al mismo tiempo a un recurso. Por eso es tan

importante identificar y gestionar debidamente tanto las dependencias como las condiciones de carrera sobre los recursos.

Por tanto, el objetivo es crear y gestionar una serie de procesos ligeros o hilos que se puedan ejecutar simultáneamente en varios núcleos del procesador. De esta forma a cada hilo se le asignará una parte del trabajo original. Se van a estudiar tres alternativas diferentes de cómo implementar la paralelización del código: la librería Intel Threading Building Blocks, la API OpenMP y la creación directa de hilos de ejecución (multithreading).

2.2.1 Intel Threading Building Blocks

Intel Threading Building Blocks (Intel TBB) [11] es una librería desarrollada por Intel para facilitar el procesamiento en paralelo con procesadores multinúcleo. En 2005 se comercializa el Pentium D, el primer procesador multicore de Intel con arquitectura x86. Intel desarrolla esta librería para facilitar el aprovechamiento de este tipo de máquinas por las que Intel estaba apostando. En 2006 sale la primera versión.

TBB es una librería escrita en C++, utilizando sus plantillas (templates). Inicialmente usaba licencia propietaria. Sin embargo, en 2007, con la versión 2.0 se creó un proyecto de código abierto y su código fuente se liberó. Aun así, la versión comercial se sigue comercializando y aunque no añade funcionalidad extra, si que da soporte técnico. Además, TBB está soportada por los principales Sistemas Operativos, incluyendo Windows, Linux y MacOS.

La paralelización con Intel Threading Building Blocks se hace mediante el reparto de tareas entre los procesadores. Esto permite que si un procesador queda libre, se reasignen dinámicamente tareas asignadas a otro procesador. Intel TBB proporciona una serie de componentes para repartir las tareas, sincronizarlas y evitar las dependencias, tales como mútex, operaciones atómicas, barreras o algoritmos que paralelizan los bucles.

En definitiva, TBB facilita a los desarrolladores crear programas escalables, mejorar su rendimiento y hacerlo independiente de las características particulares de la máquina.

2.2.2 OpenMP

OpenMP [12] es una Interfaz de Programación de Aplicaciones (API). Esta API, esta diseñada para facilitar la programación en máquinas multiproceso de memoria compartida. OpenMP Achitucture Review Board es la organización sin ánimo de lucro que desarrolla y gestiona OpenMP. Esta organización está formada por importantes miembros de grandes compañías de Hardware y Software como IBM, Intel, Oracle o Microsoft. En 1997, publican la primera versión de OpenMP que soportaba únicamente el lenguaje Fortran. Las versiones actuales, además del mencionado Fortran, también soportan C/C++.

```
#include <omp.h>
main ()  {
        int A[100];
        int B[100];
        int C[100];
        int i = 0;
        int sum = 0;
         for (i = 0; i < 100; i++){
                A[i] = i;
                B[i] = i;
          }
   #pragma omp parallel for private(i) shared(A,B,C)
   for(i = 0; i < 100; i++){
        C[i] = A[i] + B[i];
    }
}
```

Ejemplo código 1: Un programa en OpenMP que suma dos vectores en un tercero.

OpenMP tiene la característica de ser portable. No sólo es independiente de las características propias de cada máquina, sino también del sistema operativo. Los principales sistemas operativos soportan OpenMP, sólo se necesita un compilador que implemente su API. Muchas compañías implementan esta API en sus compiladores como Microsoft en su Visual Studio C++. O el popular compilador GNU GCC de libre distribución y código abierto que también lo soporta desde la versión 4.2.

El modelo de paralelización de OpenMP se basa en dividir una tarea pesada en varias tareas ligeras mediante hilos, para luego unir los resultados en único proceso. Normalmente se divide en tantos hilos como núcleos tenga el procesador y se asigna cada núcleo a un hilo distinto permitiendo que haya paralelismo real. No obstante OpenMP es muy flexible e incorpora directivas de compilación, funciones y variables de entorno. De esta forma OpenMP permite a los desarrolladores, entre otras cosas, repartir las iteraciones de un bucle entre procesos, seleccionar el número de threads, sincronizar los procesos o asignar tareas diferentes a cada hilo.

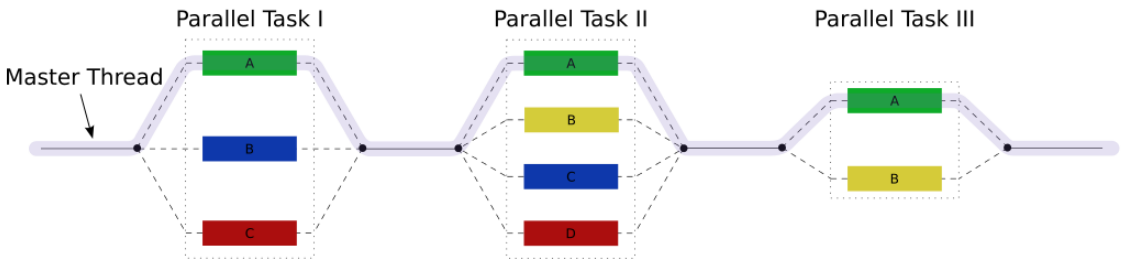


Ilustración 2: Modelo de ejecución OpenMP.

Esta API destaca por su simplicidad y estar basada especialmente en las directivas de compilación o pragmas. También permite la escalabilidad y la portabilidad de los programas. Por último, es destacable que OpenMP puede utilizarse de forma combinada con MPI para aprovechar varias máquinas multiproceso de memoria compartida, pero no común a todas las máquinas.

2.2.3 Multithreading

Sin una API o una librería que ayude a repartir la carga computacional entre los distintos núcleos de la máquina se pueden crear los hilos directamente. Esto se puede hacer mediante un lenguaje de programación que permita manejar hilos como Java. También se puede hacer mediante las llamadas al sistema que proporciona el sistema operativo.

La ventaja principal de utilizar un lenguaje que tenga mecanismos expresamente diseñados para manejar threads es que no tiene depende del sistema operativo en el que se ejecute. Por el contrario, su desventaja es que obliga a utilizar un lenguaje de programación específico.

Por otra parte, si se desarrolla un programa con un lenguaje de programación como C que no posee funciones específicas para manejar hilos, hay que usar llamadas al sistema. La principal desventaja de este método es que disminuye la portabilidad ya que las llamadas al sistema son específicas de cada sistema operativo. Precisamente para solucionar este tipo de problemas muchos sistemas operativos utilizan POSIX, incluyendo algunas ediciones de Microsoft Windows. Sin embargo, la compatibilidad no es total y algunos sistemas operativos tienen ciertas restricciones.

Crear los procesos ligeros mediante llamadas al sistema proporciona mayor flexibilidad que hacerlo mediante otros mecanismos ya que son funciones de más bajo nivel. En este caso, el desarrollador tiene el control absoluto sobre los threads que crea y cómo utilizarlos. Por el contrario, este método es más complejo y requiere más tiempo. También es más difícil hacer un programa que sea escalable.

2.2.4 Comparación

Para elegir la tecnología más adecuada se van a evaluar tres características esenciales: la portabilidad, los lenguajes soportados y la dificultad de desarrollo.

En primer lugar se va a descartar el utilizar un lenguaje específico para el uso de threads. Las librerías de los programas de cálculo suelen estar escritas en C/C++ o Fortran y estos lenguajes no tienen una forma específica de tratar los threads. Habiendo otros mecanismos no merece la pena cambiar el lenguaje de programación por la simple ventaja de estos mecanismos particulares del lenguaje.

En segundo lugar, los inconvenientes de utilizar llamadas al sistema superan a las ventajas. Si bien es cierto que al ser el método de más bajo nivel es el más flexible y con el que se podría obtener el mayor rendimiento, también es cierto que es un método más difícil de escalar, más complejo y costoso de implementar y sobre todo menos portable.

Opción	Portabilidad	Lenguajes soportados	Dificultad de desarrollo
Intel TBB	Alta	C/C++	Fácil-Medio
OpenMP	Alta	C/C++ y Fortran	Fácil-Medio
Multithreading	Media	Todos	Difícil

Tabla 2: Comparativa herramientas multicore.

Por tanto la decisión final está entre Intel TBB y OpenMP. Ambos métodos permiten portabilidad, escalabilidad y facilitan la paralelización. Una de las ventajas de OpenMP es que un estándar abierto y esta más extendido, siendo una tecnología más madura que TBB. Otra diferencia importante es que OpenMP está más orientado a paralelizar bucles, mientras que TBB trabaja mejor con tareas. Ya que las funciones pesadas de un grafo suelen consistir en bucles anidados que recorren los nodos, OpenMP ofrece mayores ventajas.

2.3 Teoría de grafos

La teoría de grafos se encarga de estudiar las propiedades de los grafos. Un grafo está formado por un conjunto de vértices o nodos y un conjunto de aristas que relacionan un vértice con otro, o con él mismo. Normalmente la representación gráfica de un grafo se hace mediante círculos, que representan los nodos y líneas que los interconectan que representan las aristas.

Adicionalmente, hay otras propiedades que permiten representar mayor variedad de situaciones o problemas mediante grafos. Por ejemplo, en una máquina de estados hay transiciones entre estados. En este caso puede haber transiciones que lleven del nodo A al nodo B, pero no a la inversa, es decir, transiciones que tienen una dirección. De esta forma, para representarlo se utilizan grafos dirigidos. Un grafo dirigido es aquel cuyas aristas pueden tener una única dirección y se suelen representar de forma gráfica mediante flechas.

En otras ocasiones se necesita dar valor a los vértices. Esto puede hacerse, entre otras cosas, para asignar un coste a una transición entre estados o para modelar una red eléctrica indicando en cada vértice la capacidad de la línea eléctrica.

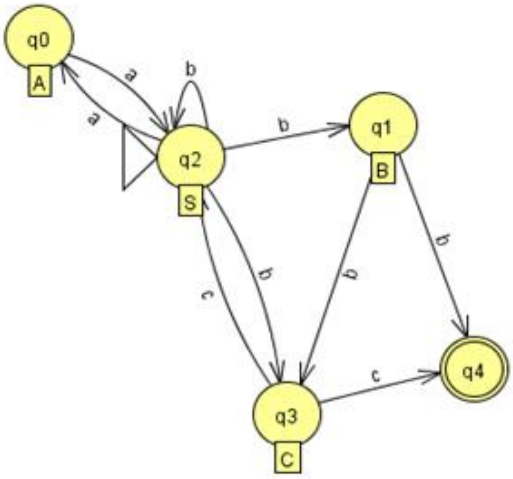


Ilustración 3: Una máquina de estados es una de las múltiples aplicaciones prácticas de los grafos dirigidos.

Como se puede observar, la teoría de grafos tiene múltiples aplicaciones prácticas. Se utiliza en gran cantidad de áreas y problemas. Una de las aplicaciones más típicas es la optimización de redes de distinto tipo, algunos ejemplos son las redes de transporte, eléctricas o de ordenadores. Otro uso habitual es el del estudio de redes sociales, en este caso con la explosión de las redes sociales en internet el campo está actualmente muy de moda.

Una de las propiedades que puede tener un grafo es si se trata de un Small World. Esta propiedad se explicará en las páginas siguientes, pero para entender mejor qué es un grafo de este tipo habrá que entender primero otras dos propiedades de los grafos: el diámetro y el clustering coefficient.

2.3.1 Diámetro

El diámetro de un grafo es una propiedad importante para identificar si un grafo es un Small World. La distancia mínima entre dos vértices es el recorrido posible entre los vértices con menor número de aristas. El diámetro de un grafo es la máxima distancia mínima de entre todas las posibles parejas de vértices del grafo.

Por lo tanto para calcular el diámetro de un grafo, por cada vértice habrá que buscar el vértice que esté a mayor distancia mínima, recorriendo todos los vértices. De esta forma, la complejidad computacional será **O(Vértices*Aristas).** Esto hace que calcular el diámetro de grafos grandes y con muchas conexiones sea computacionalmente muy costoso.

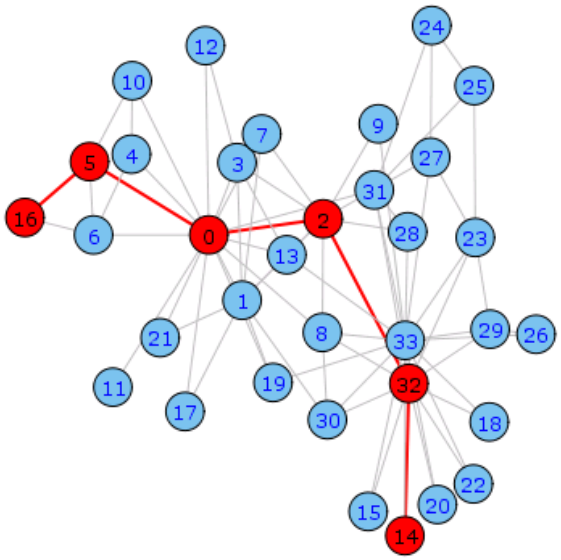


Ilustración 4: Este gráfico tiene diámetro 5, por lo tanto desde cualquier vértice pasando por un máximo de 5 aristas se podrá llegar a todos los vértices.

2.3.2 Clustering Coefficient

El Clustering Coefficient o coeficiente de agrupamiento de un vértice, mide el grado de conexión entre sus vértices vecinos. Este grado de conexión se realiza viendo cuántas conexiones diferentes hay entre los nodos y dividiéndolas entre el total. Por ejemplo, si los vecinos del nodo del que queremos calcular el clustering coefficient son 3 y no hay ningún vértice que los relacione el valor del coeficiente de agrupamiento será 0. Sin embargo, si hay una única arista que une dos de los vértices el valor será de 1/3, si hay dos 2/3 y si hay tres, el clustering coefficient será 1, que es el máximo.

El coeficiente de agrupamiento de un grafo es la media del coeficiente de agrupamiento de cada uno de sus nodos. Por lo tanto, para calcularlo, se necesita calcularlo por cada uno de los grafos. La complejidad por tanto será **O(vértices*media de aristas^2).**

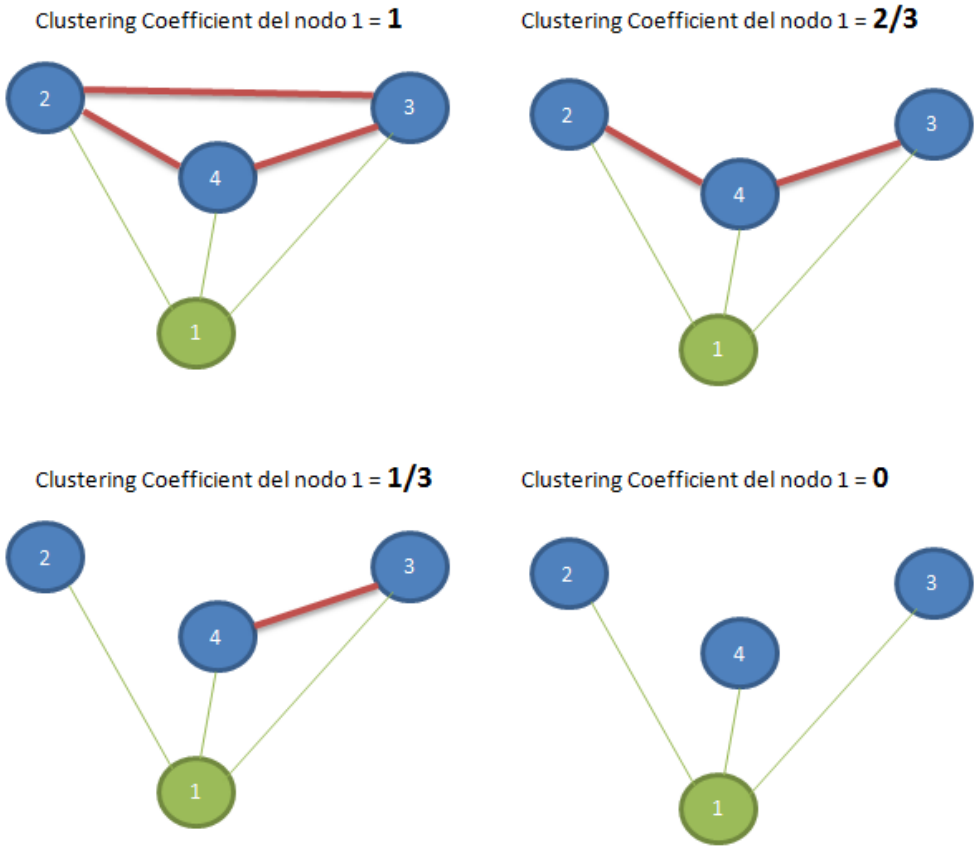


Ilustración 5: El clustering coefficient de un nodo muestra el grado de relación de sus nodos vecinos entre sí.

2.3.3 Small World

Un Small World [13] (Mundo Pequeño) es una característica que poseen las redes que tienen un grado muy alto de interconexión. De esta manera en un Small World se podrá encontrar un camino muy corto para llegar a cualquier nodo.

Ejemplos importantes de small worlds son por ejemplo las carreteras, las influencias sociales o las redes de neuronas. Particularmente en biología hay gran cantidad de situaciones en las que se dan este tipo de redes: en genética o en los procesos de metabolización son tan solo otros ejemplos. En estos campos su estudio es realmente importante.

Otro campo importante, es el de los estudios sociológicos. Estos estudios pueden detectar entre otras cosas los movimientos sociales y cómo se producen las influencias en la sociedad.

Por otra parte, los small worlds tienen algunas propiedades muy beneficiosas. Las redes de este tipo son muy robustas, si un nodo desaparece no hay apenas diferencia. Por lo tanto, muchas redes buscan ser de este tipo con el objetivo de mejorar su tolerancia a los fallos.

Para observar si un grafo es un mundo pequeño se genera un grafo aleatorio con el mismo número de vértices y aristas. Si el grafo es un Small World el diámetro será muy parecido al del grafo aleatorio y el clustering coefficient será mucho mayor ya que estará muy interconectado entre todos sus nodos.

Este cálculo que puede parecer sencillo, calcula dos veces la función diámetro y dos veces el coeficiente de agrupamiento. Este cálculo en grafos con un número moderado de nodos no es ningún problema, pero en grafos de gran tamaño los cálculos pueden resultar muy pesados. Mejorar el rendimiento de estas funciones resulta muy útil y permite ganar mucho tiempo.

3 Desarrollo

En está sección se van a estudiar las principales características de la librería igraph. Además se va a explicar cómo se usan los paquetes de R, viendo su estructura, la forma de crearlos y cómo utilizan código escrito en otros lenguajes.

3.1 igraph

Igraph es una librería desarrollada en C/C++ para el análisis de grafos. Se trata de un proyecto de software libre realizado por Gábor Csárdi y Tamás Nepusz, pertenecientes en el momento del desarrollo, al Departamento de Biofísica del Instituto de Investigación Nuclear y de Partículas de la Academia Húngara de Ciencias. La primera versión de esta librería salió en 2005.

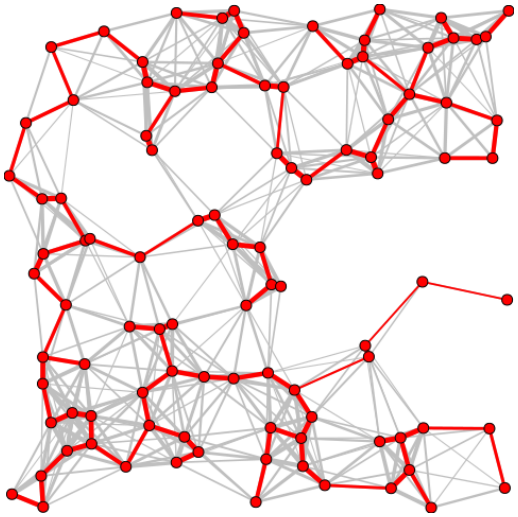


Ilustración 6: El árbol recubridor mínimo calculado con igraph.

La librería está pensada para tratar grandes grafos tanto dirigidos como no dirigidos de miles de vértices y aristas, utilizando para ello una importante cantidad de estructuras de datos propias. Además, la librería permite realizar una gran cantidad de operaciones que van desde las más simples como calcular el grado de un grafo, hasta otras más complejas como calcular el árbol recubridor mínimo. Está librería además permite crear una gran cantidad de grafos con determinadas características, por ejemplo la función igraph_watts_strogatz_game, permite crear grafos small world según el modelo desarollado por Watts y Strogatz [13]. Otra función importante de igraph es que permite leer y escribir grafos en archivos con distintos formatos.

Una característica muy destacable de igraph es que tiene varias interfaces diferentes. Hay interfaces de R, de Ruby y de Python. Además, las interfaces de R y de Python permiten visualizar los grafos. Por supuesto la librería de C/C++ también se distribuye sin interfaz, lo que permite usarla directamente como una librería más. Esta característica es una de las principales por la que se ha elegido igraph para realizar este trabajo. En definitiva, se trata de una librería independiente de la interfaz que

funcionará en R, en Python, en Perl o mediante cualquier interfaz que se pudiese crear en un futuro.

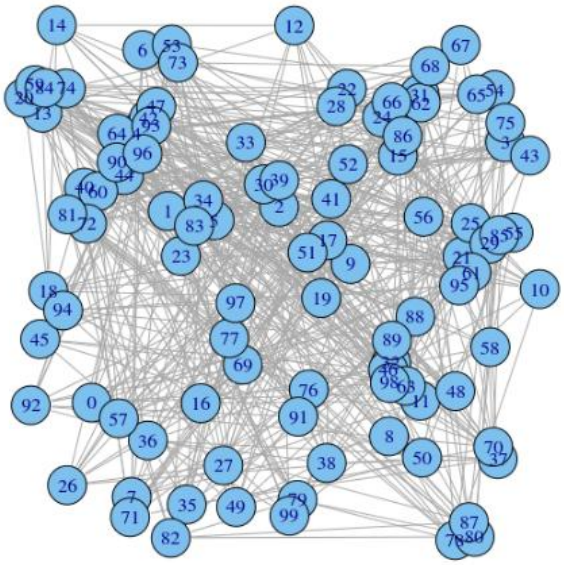


Ilustración 7: Igraph puede crear grafos Small World a partir del modelo Watts-Strogatz.

3.2 Librerías R

Un paquete R contiene una serie de funciones y datos escritos en el lenguaje R. Estas funciones a su vez pueden hacer llamadas a código escrito en C/C++ y Fortran.

Una librería de R tiene la siguiente estructura:

- Un archivo llamado **DESCRIPTION**: este archivo contiene datos del paquete como el nombre, la versión, el autor, descripción, dependencias, licencia y otros datos.
- Un directorio llamado **man**: en este directorio hay archivos .Rd que proporcionan ayuda sobre las funciones de la librería.
- Un directorio llamado **R**: en este directorio están los archivos .R que contienen el código R de las funciones y datos de la librería.
- Un directorio llamado **src**: dónde se encuentra el código escrito en C/C++ y Fortran que utilice la librería.
- Es importante destacar que se puede incluir un archivo especial llamado **Makevars** en este archivo se puede indicar ciertas directivas de compilación especiales mediante flags, por ejemplo, el uso de OpenMP.

3.2.1 Usando C/C++ en R

R proporciona varias funciones para ejecutar código compilado escrito en otros

lenguajes. Las funciones principales para ejecutar código C/C++ son dos: .C y .Call.

La función .C(función, arg1, arg2,…, argn) ejecuta la función transformando los argumentos de tipo R a tipos de C según una tabla de equivalencia. Además si se quieren usar argumentos de complex de R, en el fichero fuente que incluya la función C habrá que incluir la cabecera R.h para poder usar el tipo Rcomplex *.

Tipos R	Tipos C	Tipos FORTRAN
Logical	Int *	INTEGER
Integer	Int *	INTEGER
Double	Double *	DOUBLE PRECISION
Complex	Rcomplex * (R.h)	DOUBLE COMPLEX
Character	Char **	CHARACTER*255
raw	Unsigned char *	ninguno

Tabla 3: Conversión de tipos datos entre lenguajes.

Por otra parte, se puede utilizar la función .Call(función, arg1, arg2,…,argn). Esta función pasa todos los argumentos como objetos del tipo SEXP. Para utilizar estos objetos en C hay que incluir las cabecera R.h y Rinternals.h.

Es importante tener en cuenta que para llamar a una función con .Call, .C u otras funciones que también utilizan código externo como .Fortran o .External, el código tiene que estar cargado, para ello se puede utilizar la función dyn.load que carga dinámicamente las librerías.

3.2.2 Crear y usar una librería

R facilita la forma de crear una librería mediante una función llamada package.skeleton. Esta función permite crear el esqueleto de una librería a partir de tus funciones ya escritas en R. Una vez creado el esqueleto se podrán modificar los ficheros de ayuda y el DESCRIPTION, para tener la librería lista para usarse.

```
> add <-function(x,y) x+y
> mul <-function(x,y) x*y
> package.skeleton(list=c("add","mul"), name="mypkg")
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './mypkg/Read-and-delete-me'.
>
```

Ilustración 8: Ejemplo de uso de la función package.skeleton, creando una librería con funciones para sumar y multiplicar.

Para crear definitivamente la librería se puede ejecutar el comando “R CMD BUILD

librería" desde la línea de comandos. Esto creará el paquete en formato tar.

Para instalar la librería junto en el sitio de instalación de las librerías R por defecto se usa el comando "R CMD INSTALL librería". Por supuesto también se puede instalar en otro sitio cambiando las opciones.

Por último, para utilizar una función de la librería simplemente habrá que cargar en R la librería antes de usar las funciones que contiene. Esto se hace mediante la función library(librería).

4 Implementación

En esta sección se va a explicar cómo se ha desarrollado la implementación de OpenMP sobre las funciones que calculan el diámetro y el clustering coefficient de un grafo. Se va a analizar cómo realizan su cometido las funciones. Así como las distintas alternativas de implementación. Y para finalizar, cómo se ha implementado la solución.

4.1 Diámetro

El diámetro lo calcula en igraph mediante una función llamada **igraph_diameter** que está en el fichero **structural_properties.c**. Esta función se llama al calcular el diámetro de un grafo, en R sería diameter(grafo). La función igraph_diameter también calcula el recorrido del diámetro, devolviendo los nodos extremos de dicho camino. Esto lo usa la función get.diameter(grafo) de R que devuelve el recorrido entero de diámetro del grafo.

4.1.1 Algoritmo original

Entrada:

- Grafo: sobre el que se va a calcular el diámetro.

Salida:

- Diámetro: diámetro del grafo.
- Nodo diámetro desde: un nodo cuya distancia mínima al nodo diámetro hasta sea el propio diámetro.
- Nodo diámetro hasta: un nodo cuya distancia mínima al nodo diámetro desde sea el propio diámetro.

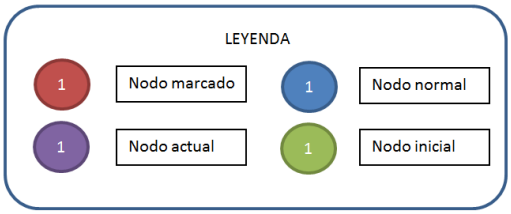
Datos:

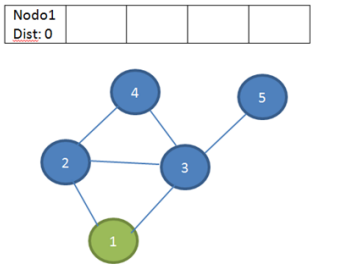
- Nodo de inicio: nodo desde el que se está calculando la distancia al resto de nodos.
- Nodo actual: nodo que se está visitando en ese momento.
- Distancia actual: distancia mínima entre nodo actual y nodo de inicio.
- Nodos visitados: nodos ya visitados o ya introducidos en la cola.
- Cola nodos: Nodos a visitar junto con sus distancias.

Algoritmo:

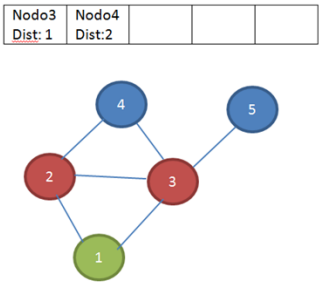
1. Se introduce el nodo de inicio en la cola junto con la distancia 0.
2. Se marca el nodo actual como visitado.
3. Se saca el siguiente nodo y la distancia, asignándolos a nodo y distancia actuales.

4. Si la distancia actual es mayor que el diámetro, se asigna a las variables de salida distancia actual, nodo de inicio y nodo actual.
5. Se sacan los nodos vecinos del nodo actual.
6. Los nodos vecinos que no hayan sido visitados se marcan y se introducen en la cola junto con la distancia actual + 1.
7. Si hay nodos en la cola se vuelve al paso 2.
8. Si el nodo de inicio es el último fin. Sino se vuelve al nodo 1 reiniciando los nodos marcados y pasando al siguiente nodo de inicio.

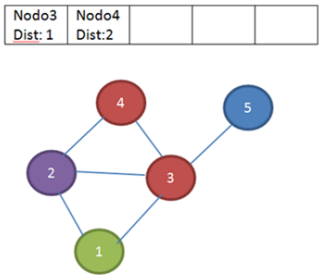




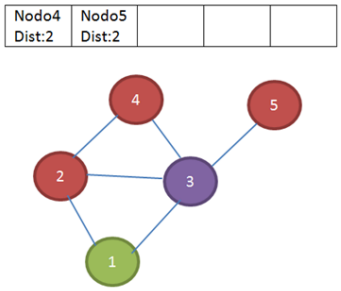
Paso 1: se introduce el nodo inicial en la lista.



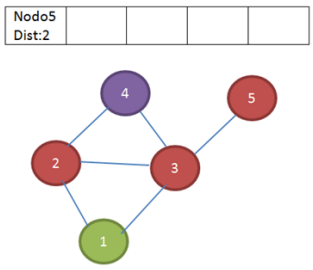
Paso 2: se saca el siguiente nodo de la cola, se marcan los vecinos y los no marcados se introducen en la cola.



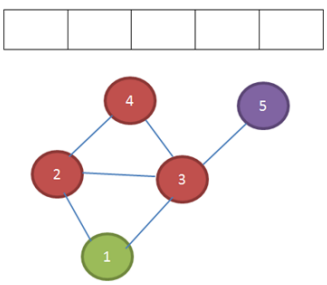
Paso 3: se saca el nodo 2 de la cola. Se marca el nodo 4 y se introduce en la cola, ya que no estaba marcado anteriormente.



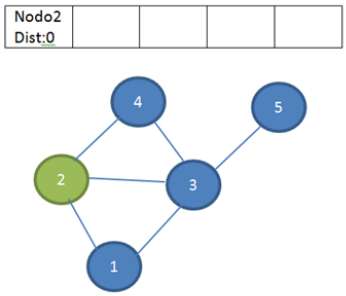
Paso 4: se saca el nodo 3. Todos sus vecinos menos el 5 están marcados. Se marca el 5 y se añade a la cola.



Paso 5: se saca el nodo 4. Como la distancia es mayor se marca el recorrido de los nodos 1 al 4 como posible diámetro. Los nodos vecinos ya están marcados, no se hace nada más.



Paso 6: se saca el nodo 5. Los nodos vecinos ya están marcados, no se hace nada más. La lista queda vacía. Habrá que cambiar, si se puede, el nodo inicial.



Paso 7: el proceso vuelve a empezar con el siguiente nodo inicial. Se hará con todos los nodos.

4.1.2 Método de paralelización

Este algoritmo se puede estructurar de la siguiente manera:

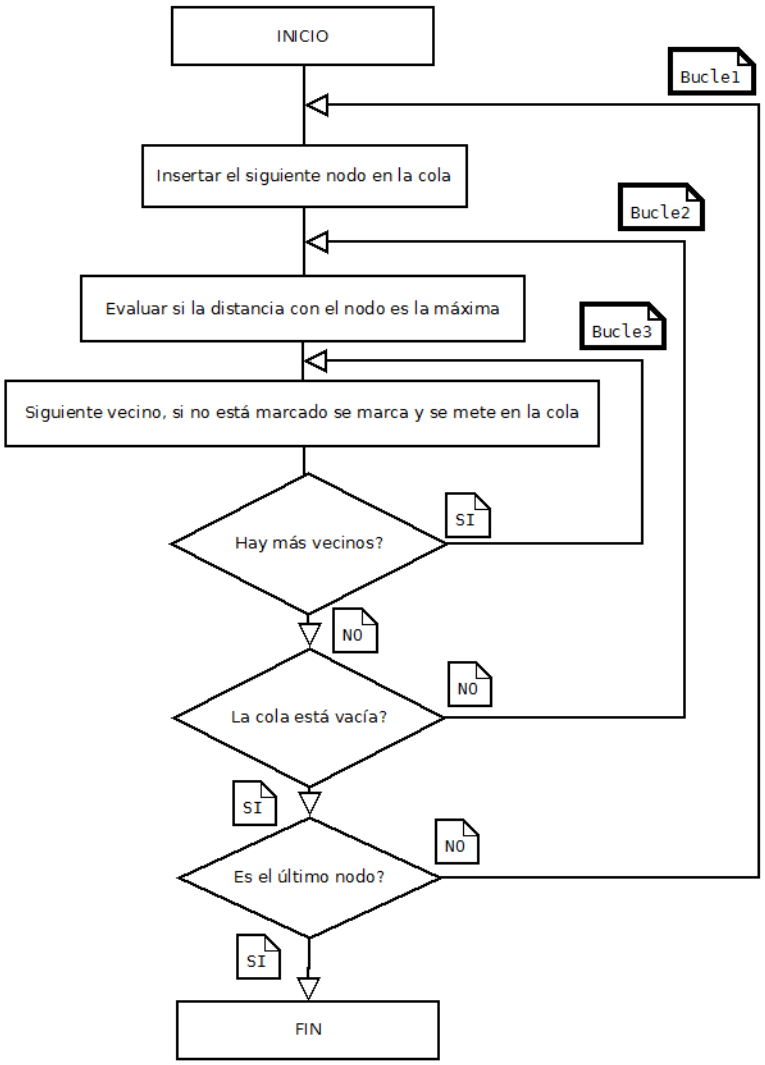


Ilustración 9: Diagrama de flujo de la función diámetro.

OpenMP está pensado especialmente para paralelizar bucles, por lo que usarlo para este algoritmo parece muy apropiado.

Este algoritmo se puede paralelizar de varias formas, aunque habrá que analizar cuál es la más apropiada.

El **bucle3** sólo marca e inserta en la cola los nodos vecinos.

- Se trata de una tarea relativamente ligera ya que es raro que haya muchos nodos vecinos.
- Además, introducir un nodo en la cola es una **tarea atómica**, por lo que la ganancia es todavía menor.
- Por último, hay que tener en cuenta cómo gestionar **la creación de los threads**, si se crean en cada iteración, el tiempo de creación podría ser mayor a la ganancia, por lo que el paralelismo podría llegar a ser contraproducente.

El **bucle2** tiene dependencias complicadas de solucionar. Estas dependencias y recursos compartidos implican que parte del tiempo los procesadores van a estar esperando a que otros procesadores acaben sus tareas, lo cual es ineficiente.

- La cola de nodos y los nodos ya visitados son estructuras que se leen y escriben por varios hilos diferentes de forma simultánea. Tendría que garantizarse que a estos recursos se acceda de forma controlada.
- La condición de salida es que la cola esta vacía, puede darse el caso de que la cola esté vacía y otro thread vaya a meter nodos en la cola. Habría que usar un mecanismo de sincronización para solucionar este caso.
- Por otra parte, el sistema de meter nodos en la cola junto con su distancia está pensado para garantizar la distancia mínima entre el nodo inicial y el nodo que se está evaluando. Si hay paralelismo no se puede garantizar. Habría o que cambiar el algoritmo usando otras estructuras de datos o utilizar métodos de sincronización.

El **bucle1** necesita de sus propias estructuras privadas. Es decir cada thread necesita sus propias variables: una cola propia, sus propios nodos marcados, nodo actual, distancia actual, nodo desde, nodo hasta y diámetro.

La forma de paralelizarlo será repartir los nodos iniciales entre los diferentes threads, cada uno calculará su distancia máxima y el camino. Finalmente habrá una fase de unión en la cuál se compararán los valores del diámetro de cada thread y se elegirá el mayor.

Podemos ver en el siguiente ejemplo, cómo se ejecutaría un grafo de 400 nodos

distribuido entre cuatro procesadores. En este caso, cada procesador calcularía el diámetro parcial de sus 100 nodos correspondientes. Esto se hace mediante la directiva de compilación **pragma omp parallel for**. Del mismo modo, con **shared** y **private** se indican las variables compartidas y las privadas de los threads. Finalmente, se vería entre todos los nodos cuál es el que tiene el diámetro más grande, que será el diámetro del grafo, la salida. Hay que tener en cuenta que OpenMP permite fijar el número de threads a crear, pero que por defecto creará tantos como procesadores tenga el ordenador.

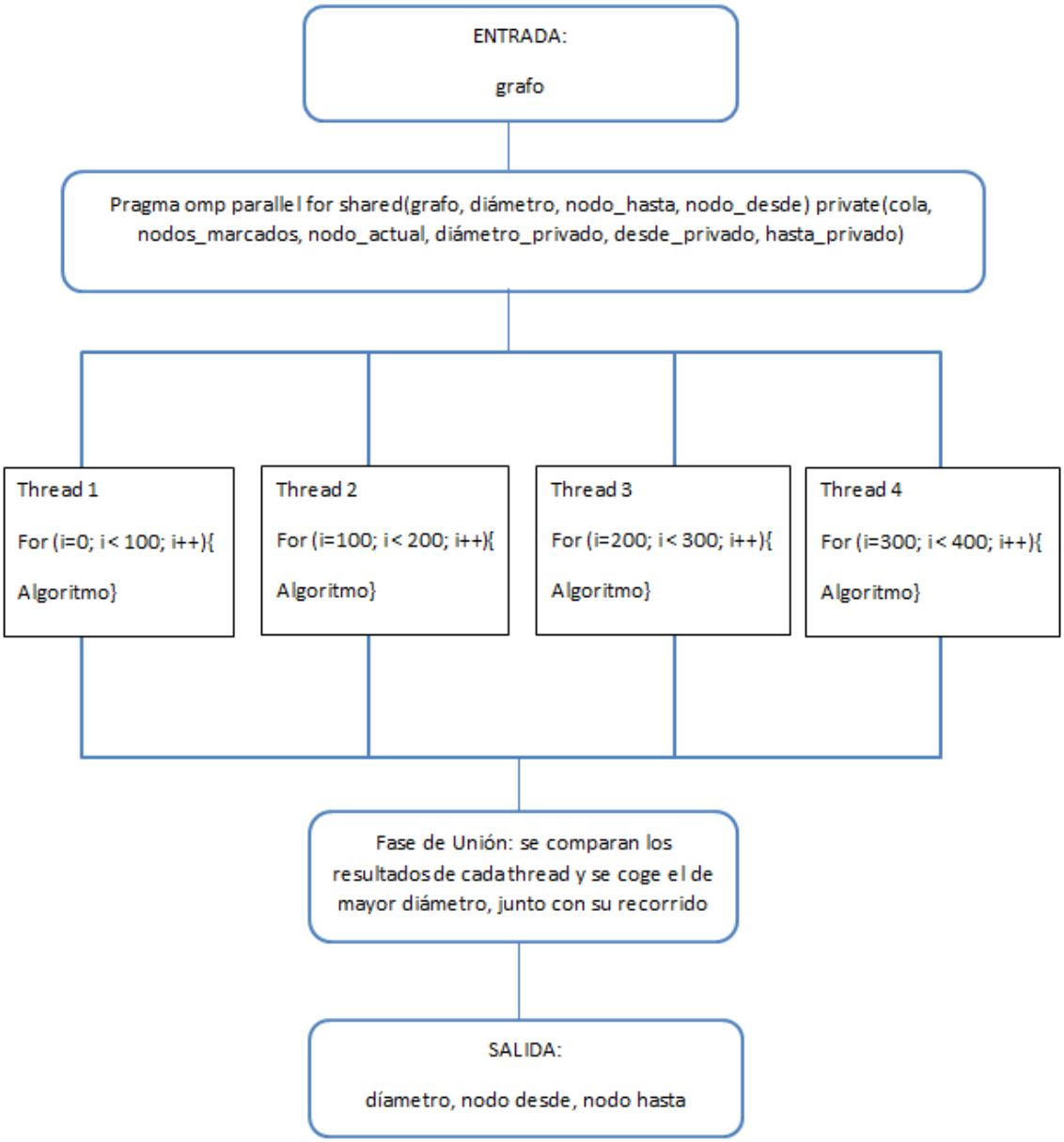


Ilustración 10: Diagrama de ejecución paralela de la función diámetro.

4.2 Clustering Coefficient

El clustering coefficient de la librería igraph se calcula en una función llamada **igraph_transivity_avglocal_undirected** en el fichero **structural_properties.c**, está función se llama en R mediante la función transitivity(grafo, "localaverage"). Esto lo que hace es calcular la media del clustering coefficient de todos los nodos del grafo. Es importante tener en cuenta que esta función está diseñada para grafos no dirigidos. El clustering coefficient para grafos dirigidos es diferente, ya que como las aristas son dirigidas, puede haber el doble de uniones entre dos vértices, contando los de una dirección y la opuesta.

4.2.1 Algoritmo original

Entrada:

- Grafo: sobre el que se va a calcular el clustering coefficient.

Salida:

- Clustering Coefficient: media del clustering coefficient de todos los nodos.

Datos:

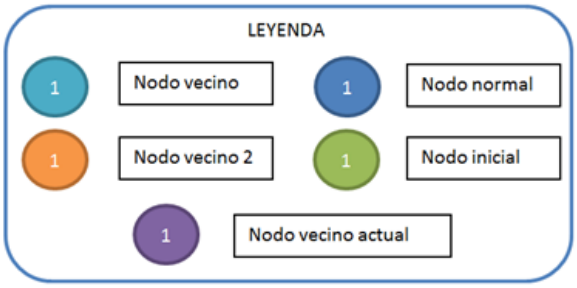
- Nodo de inicio: nodo del que se está calculando el clustering coefficient.
- Nodos vecinos 1: nodos vecinos del nodo de inicio.
- Nodo vecino 1 actual.
- Nodos vecinos 2: nodos vecinos del nodo vecino 1 actual.
- Vector de triángulos: triángulo detectado en una posición concreta.
- Triángulos posibles: número de triángulos posibles dado el número del.
- Suma de CC: variable que acumula los clustering coefficient de cada nodo.

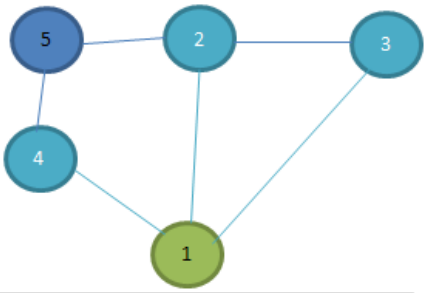
Algoritmo:

1. Se coge el primer nodo de inicio.
2. Se calculan sus vecinos.
3. A partir de los vecinos se calculan los triángulos posibles.
4. Se asigna el primer nodo vecino 1 actual.
5. Se sacan los vecinos 2, que son los vecinos del nodo vecino 1 actual.
6. Se cuentan cuántos nodos vecinos 2 son también nodos vecinos 1 y se apuntan en el vector de triángulos.
7. Se pasa al siguiente nodo vecino 1 actual. Si hay siguiente nodo vecino 1 se pasa va al punto 5.
8. Se calcula el clustering coefficient dividiendo el vector de triángulos correspondiente al nodo con los triángulos posibles y se añade a la suma de

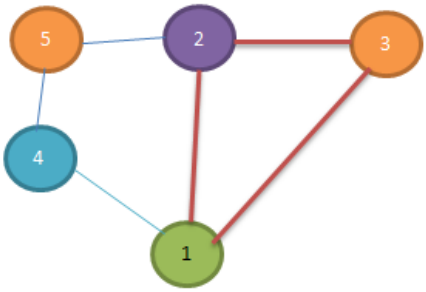
CC.

9. Se pasa al siguiente nodo de inicio si se puede y se va al punto 2.
10. Se hace la media del clustering coefficient dividiendo la suma de CC entre el total de nodos.

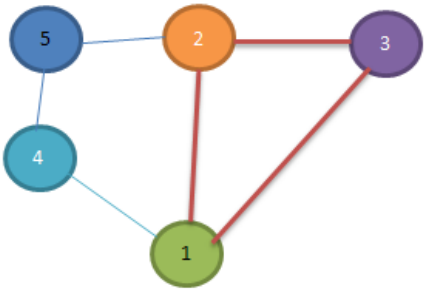




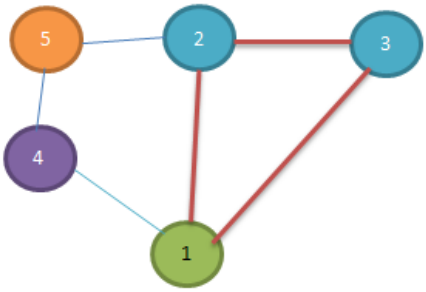
Paso 1: se sacan los vecinos del nodo 1. Hay 3, se podrán formar hasta 3 triángulos.



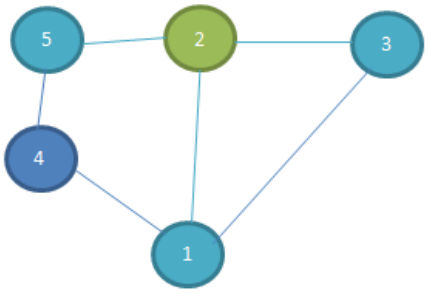
Paso 2: se sacan los vecinos del nodo 2. El nodo 3 es vecino del 1, se forma un triángulo.



Paso 3: se sacan los vecinos del nodo 3. El nodo 2 es vecino del 1, pero ese triángulo ya se había contabilizado.



Paso 4: se sacan los vecinos del nodo 4. Ninguno es también vecino del nodo 1. Al ser el último nodo vecino de 1 se puede determinar que el CC de 1 es 1/3.



Paso 5: se calcula el clustering coefficient del siguiente nodo. Cuando se calcula el de todos se hace la media.

4.2.2 Método de paralelización

Este algoritmo se estructura de la siguiente manera:

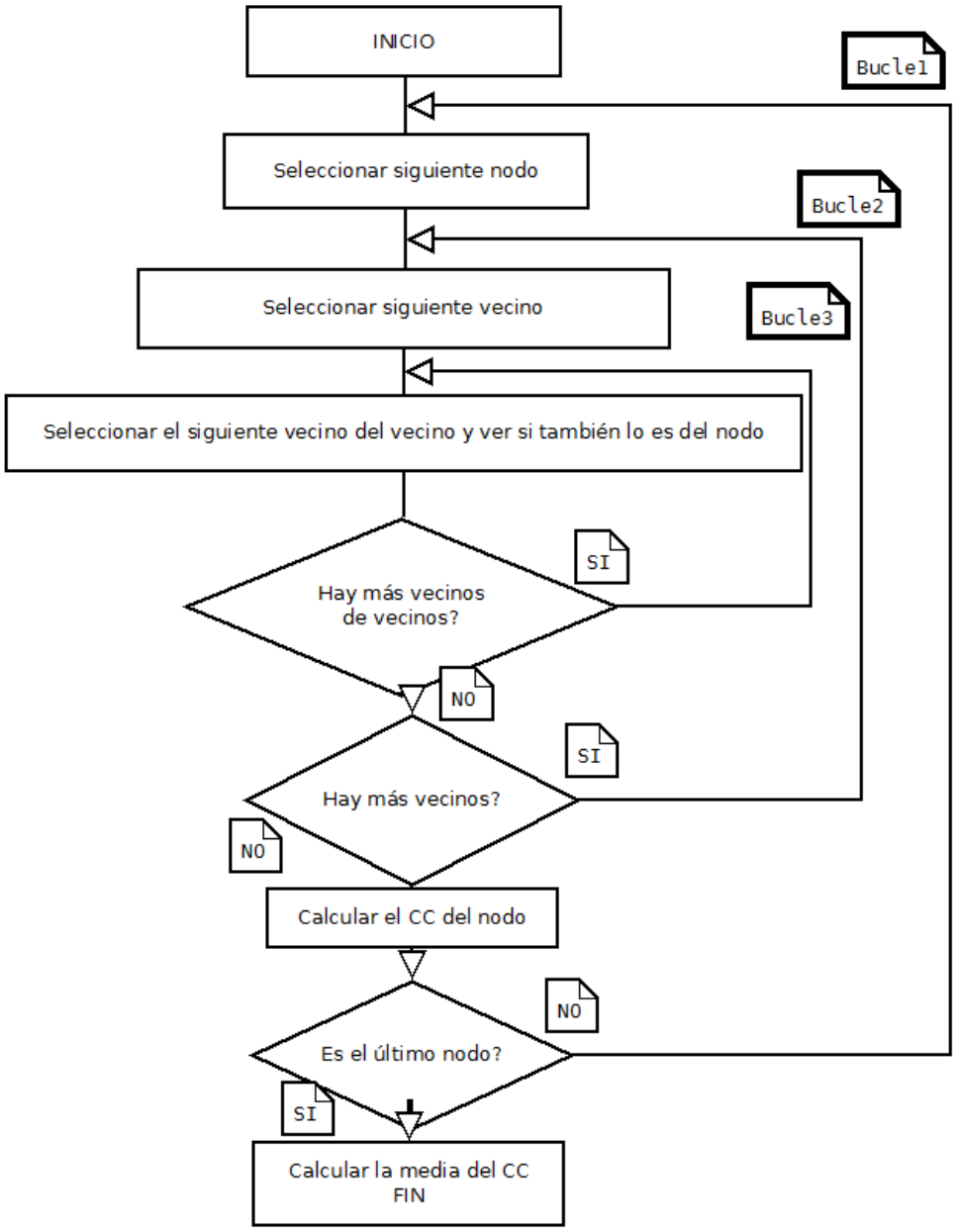


Ilustración 11: Diagrama de flujo de la función clustering coefficient.

Con el **bucle3** no se tendrá una gran ganancia al ejecutarlo en paralelo, salvo en raras ocasiones dónde los grafos tengan un grado muy alto de vecindad.

El **bucle2** será una opción mejor ya que tiene mayor carga computacional $O(media\_vecinos^2)$.

El **bucle1** será la mejor opción en los casos generales ya que no sólo depende de la cantidad de vecinos que tengan los nodos, sino que también depende del número de

nodos del grafo.

Este bucle, sin embargo, tiene dependencias por su implementación. Calcula los triángulos una sola vez y los almacena en un vector. Es decir, cuando calcula el clustering coefficient del nodo1 y detecta un triángulo entre el nodo1, el nodo2 y el nodo3 marca que hay un triángulo en los 3. Esto evita que se recorran los mismos caminos varias veces. El beneficio es que el algoritmo es más eficiente, sin embargo, también hace que el orden de ejecución importe.

Para solucionar esto, existen dos alternativas, una es evitar la dependencia cambiando el algoritmo de forma que el algoritmo recorra todos los nodos siempre y no marque los triángulos de futuras iteraciones. La otra es, en lugar de calcular el clustering coefficient en cada iteración, hacerlo al final en una fase diferente.

Modificar el algoritmo

El nuevo algoritmo es más sencillo, aunque hace más operaciones. Calcula el clustering coefficient de cada nodo y lo suma a una variable global que finalmente se divide por el número total de clusterings coefficients calculados.

Con este nuevo algoritmo, cada iteración es independiente de las demás. Sin embargo, esto hace que haya que visitar todas las posibles uniones entre los nodos vecinos en cada iteración, un cálculo recurrente que se evitaba con el algoritmo anterior.

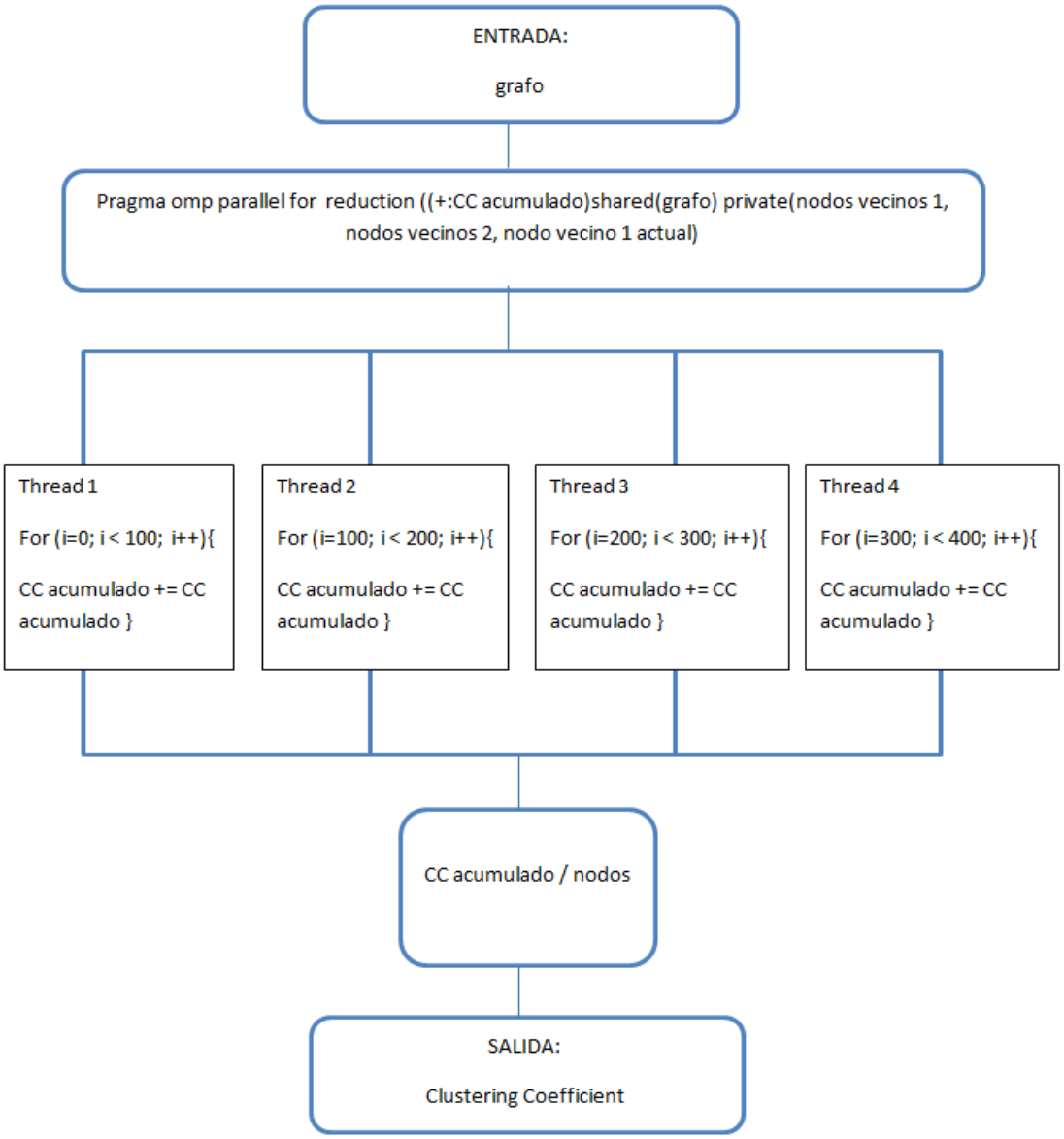


Ilustración 12: Diagrama de ejecución paralela de la función clustering coefficient usando el algoritmo nuevo.

Paralelizar el algoritmo original

Este método, como se ha comentado, utiliza dos fases: se reparte el cálculo de los triángulos y los posibles triángulos utilizando vectores compartidos y posteriormente se calculan los resultados.

Hay que destacar que como el vector de triángulos es compartido por todos los nodos, la operación lectura y suma que se realiza para marcar un triángulo en dicho vector se hará de forma atómica mediante la directiva de compilación **#pragma atomic** de OpenMP.

En la segunda fase además, para calcular el sumatorio de todos los clustering

coefficient de los nodos vamos a utilizar la directiva de compilación #pragma omp for **reduction(+:sumatorio CC).** Esta directiva es muy útil porque hace que la variable compartida del sumatorio del clustering coefficient, actúe como privada dentro del bucle. Sin embargo, al salir del bucle, se suman todos los valores que haya tomado en cada uno de los threads. Esto funciona muy bien para este tipo de casos dónde se quiere hacer un sumatorio.

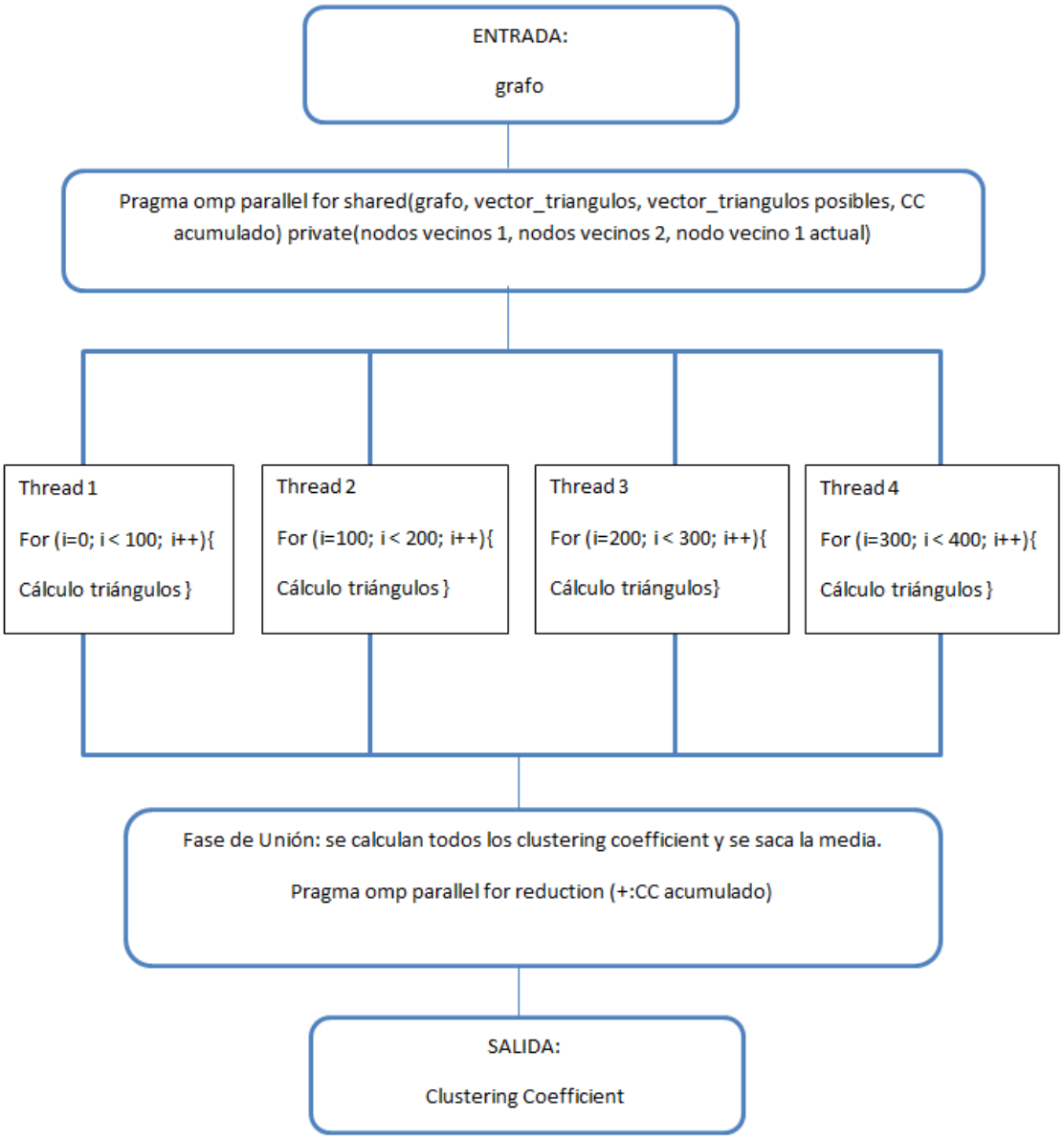


Ilustración 13: Diagrama de ejecución paralela de la función clustering coefficient usando el algoritmo original.

De esta forma la paralelización quedará dividida en dos fases: en la primera se calculan todos los triángulos de todos los nodos y en la segunda se hace la división para calcular

el clustering coefficient de cada nodo y se saca la media.

Comparación

Se ha optado por modificar el algoritmo. Modificar el algoritmo a uno que calcule todas las posibles uniones entre los nodos es más pesado y requiere más cálculos. Sin embargo, no tiene ningún tipo de dependencia y por tanto no se requiere tiempo extra de espera. El nuevo algoritmo es más simétrico ya que no depende de si alguien ha trazado o va a trazar un triángulo, esto hace que se reparta mejor el tiempo de cómputo entre los procesadores. Además, requiere menos memoria ya que no hay dos vectores comunes que almacenen el número de triángulos y los triángulos posibles de cada nodo, ahora es simplemente una única variable. Para concluir, este algoritmo permite calcular el clustering coefficient también de los grafos dirigidos, otra importante ventaja.

4.3 Cómo incluir OpenMP

Hay que tener en cuenta que OpenMP utiliza principalmente directivas de compilación. Para ello los compiladores que soportan OpenMP tiene un flag que hay que añadir al compilar el código. Por ejemplo, si se compila con el compilador gcc se tiene que usar el flag **-fopenmp**. Las librerías R tienen en cuenta esto y para añadir los flags de compilación de OpenMP se utiliza un fichero llamado **Makevars** que está en el directorio src. De esta forma al instalar la librería R el código C/C++ se compilará usando el flag necesario del compilador.

Para incorporar OpenMP también se necesita incluir la cabecera en los archivos fuente que lo usen **omp.h**.

Para finalizar, cuando OpenMP utiliza secciones en paralelo, no permite que haya ni returns ni breaks en el código. En las funciones diámetro y clustering coefficient hay algunos returns. Concretamente, en funciones que comprueban posibles errores o si el usuario quiere interrumpir la ejecución. Para no perder está funcionalidad, una opción es crear funciones alternativas que en lugar de hacer un return saquen a todos los threads de su ejecución paralela y hagan un return.

5 Evaluación

En este apartado se va a realizar la evaluación de las funciones. Para esta tarea se realizarán un conjunto de pruebas significativas con el fin de estudiar el comportamiento de las funciones con distintos conjuntos de datos.

5.1 Presentación del conjunto de datos a evaluar

Las pruebas con diferentes grafos permiten observar el comportamiento particular de las funciones, así como el grado de mejora o speedup existente entre las funciones originales que se ejecutan de forma secuencial y las que se ejecutan de forma paralela.

Aprovechando la capacidad de R de realizar y ejecutar scripts, se ha realizado uno para generar los grafos de las pruebas y guardarlos en el disco. Adicionalmente, se ha realizado otro script para realizar la batería de pruebas, cargando los grafos, ejecutando las funciones y midiendo sus tiempos.

Las pruebas se han realizado generando grafos mediante dos funciones proporcionadas por la librería igraph: barabasi.game y graph.full. La función barabasi.game utiliza el modelo Barabasi-Albert [13] de generación de grafos, mientras que la función graph.full genera grafos completos. Los grafos completos son en los que cada uno de los nodos están conectados con todos los demás nodos.

La batería de pruebas se ha ejecutado en la siguiente máquina:

CPU	Memoria	Sistema Operativo
AMD Opteron(tm) Processor 6168 (2 CPUS de 12 cores por CPU)	60GB	Linux

Tabla 4: Configuración de ejecución de las pruebas

Esta máquina da la posibilidad de utilizar hasta 24 cores. Las pruebas se han hecho lanzando diferente número de threads (2, 4, 8, 16 24 y 48), cambiando el valor de la variable de entorno que proporciona OpenMP NUM_THREADS. La prueba de 48 asigna dos threads por core, puede ser beneficioso para tener más tiempo funcionándolos ya que si se bloquea uno, el otro continua. Además se ha ejecutado el proceso original para comparar.

5.1.1 Grafos según el modelo Barabasi-Albert

El modelo Barabasi-Albert es un modelo de generación de grafos. Este modelo básicamente va introduciendo iterativamente un nodo conectado con el grafo ya generado por un número determinado de aristas.

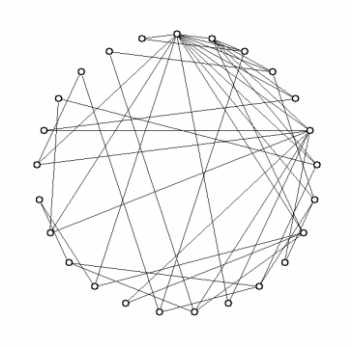


Ilustración 14: Ejemplo de un grafo generado con el modelo Barabasi-Albert, con 25 y 2 aristas generadas por nodo. Algunos nodos tienen más de 2 aristas porque se generaron primero.

Este modelo resulta especialmente útil para realizar la evaluación del diámetro debido a sus dos parámetros principales: el número de nodos y el número de aristas que genera cada nodo. Esto permite de forma fácil aumentar el número de aristas o el número de nodos de un grafo mediante sus parámetros principales.

Los grafos generados son todas las combinaciones de los parámetros de la siguiente tabla:

Nodos	Aristas generadas por nodo
10, 100, 1000, 10000	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048

Tabla 5: Grafos generados mediante la función barabasi.game para la batería de pruebas.

5.1.2 Grafos completos

La función graph.full permite realizar un grafo completo con un número de nodos determinado.

Este modelo resulta especialmente útil para realizar la evaluación del clustering coefficient ya que se tratan de grafos con un grado muy alto de aristas que crecen en gran cantidad por cada nuevo nodo. Este tipo de grafo es el que resulta computacionalmente pesado para calcular el clustering coefficient.

Los grafos generados los siguientes:

Número de nodos del grafo completo
10, 100, 1000, 10000

Tabla 6: Grafos generados mediante la función graph.full para la batería de pruebas.

5.2 Evaluación Diámetro

Tras la ejecución de la batería de pruebas se han elegido los conjuntos de datos que mejor muestran el comportamiento de la función diámetro. Como esta función depende del número de nodos y del número de aristas, se mantendrán fijos uno de esos parámetros y el otro se variará.

5.2.1 Según el número de nodos

Para ver cómo actúa la función según el número de nodos se han mantenido fijo el parámetro de número de aristas generadas por nodo.

Nodos	10	100	1000	10000
Tiempo(s) Original	0,000	0,023	2,497	261,259
Tiempo(s) 2 Threads	0,000	0,022	1,375	147,736
Tiempo(s) 4 Threads	0,009	0,021	0,847	84,817
Tiempo(s) 8 Threads	0,001	0,012	0,563	42,649
Tiempo(s) 16 Threads	0,001	0,020	0,431	29,032
Tiempo(s) 24 Threads	0,003	0,027	0,499	25,044
Tiempo(s) 48 Threads	0,005	0,062	0,808	25,637
Speedup con 2 Threads	0,000	1,045	1,816	1,897
Speedup con 4 Threads	0,000	1,095	2,948	3,080
Speedup con 8 Threads	0,000	1,917	4,435	6,126
Speedup con 16 Threads	0,000	1,150	5,794	8,999
Speedup con 24 Threads	0,000	0,852	5,004	10,432
Speedup con 48 Threads	0,000	0,371	3,090	10,191

Tabla 7: Tiempos y speedup de la función diámetro según el número de nodos y manteniendo en 256 las aristas generadas por nodo.

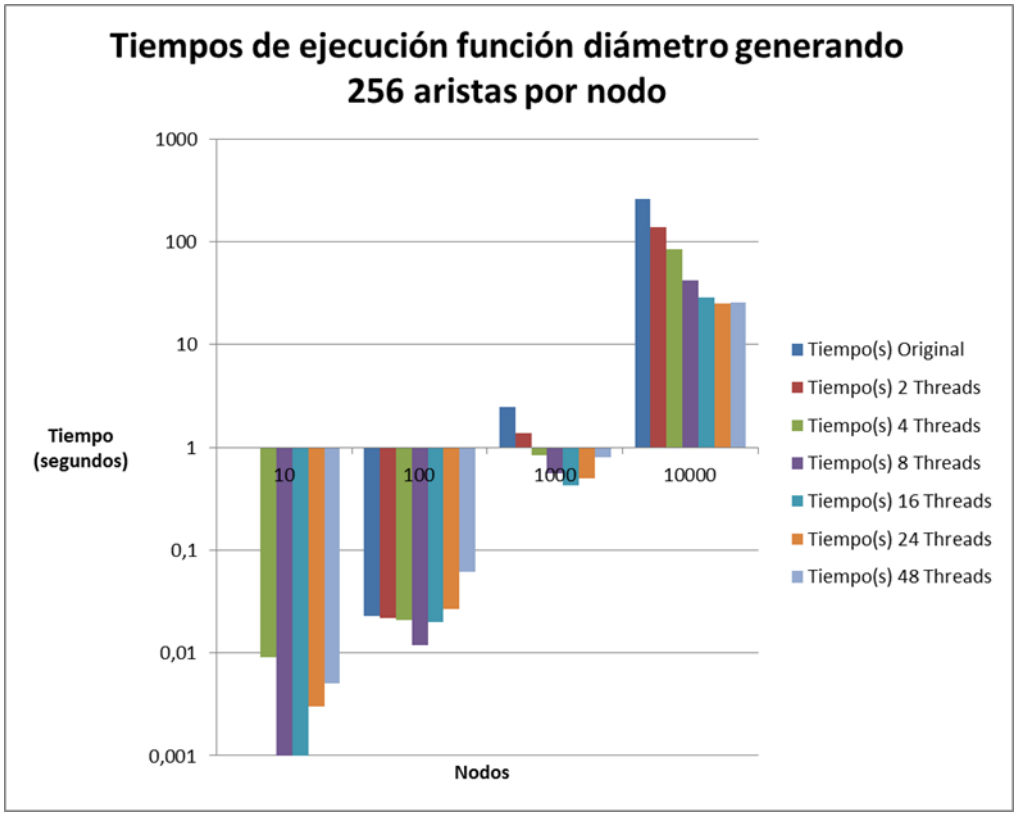


Gráfico 2: Tiempos de ejecución de la función diámetro según el número de nodos y manteniendo en 256 las aristas generadas por nodo.

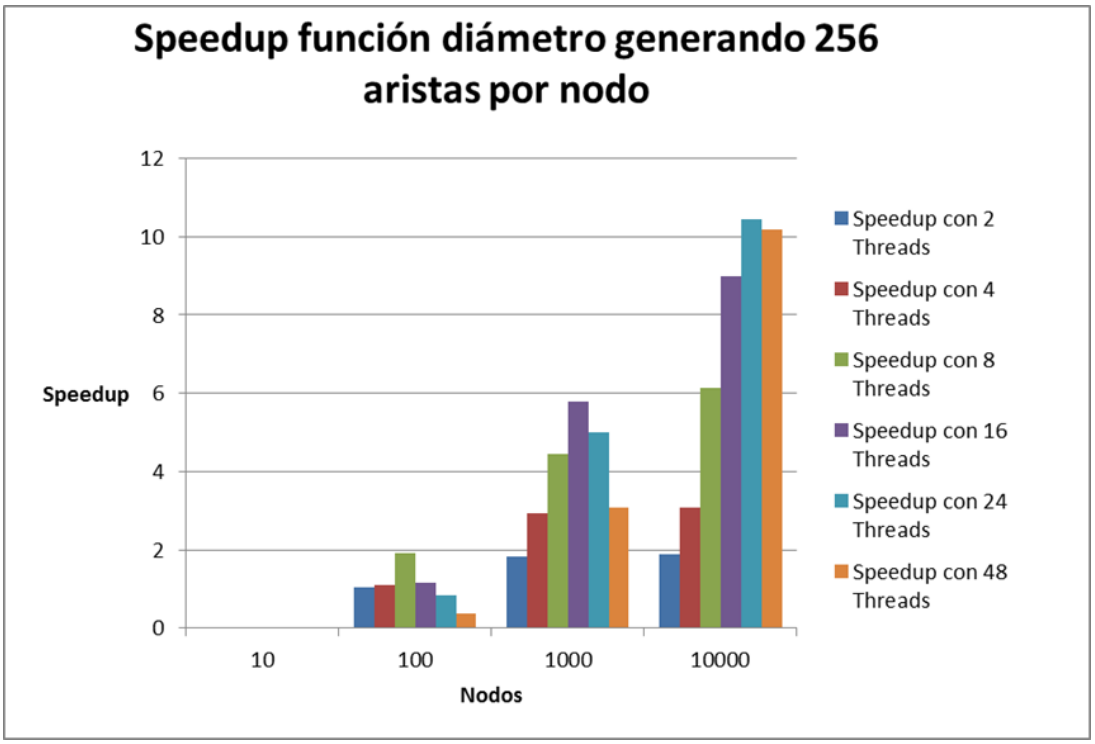


Gráfico 3: Speedup de la función diámetro según el número de nodos y manteniendo el numero de aristas generadas por nodo en 256.

Se puede observar como en el orden de los 1.000 nodos, el tiempo empieza a ser un factor importante. En los 10.000 ya es una tarea que tarda algunos minutos.

También se observa la ganancia al ejecutar el proceso en paralelo. Con pocos nodos esa ganancia es despreciable. Sin embargo, cuando la tarea es más pesada, los procesadores se reparten bien la carga de trabajo obteniendo un buen rendimiento.

El speedup con 10.000 nodos y 24 threads, llega a aumentar en 10 veces la velocidad del proceso original. Sin embargo, el speedup no aumenta en la misma proporción que el número de threads. El aumento del rendimiento va mejorando, pero cada vez menos.

5.2.2 Según el número de aristas generadas por nodo

Para evaluar cómo evoluciona la función diámetro según el número de aristas, se mantienen fijos el número de nodos y se modifican el número de aristas que genera cada nodo.

Nodos	1	2	4	8	16	32	64	128	256	512	1024	2048
Tiempo(s) 1 Original	9,506	12,487	14,042	15,540	23,190	39,937	73,794	136,158	261,259	503,265	995,030	1974,328
Tiempo(s) 2 Threads	5,102	6,830	8,156	8,489	13,103	22,781	43,035	76,62	147,736	285,012	572,667	1141,050
Tiempo(s) 4 Threads	2,780	3,454	3,865	4,313	6,745	11,750	22,298	43,122	84,817	171,508	316,649	607,911
Tiempo(s) 8 Threads	1,387	1,745	1,956	2,212	3,829	7,181	12,201	22,838	42,649	85,251	174,713	357,053
Tiempo(s) 16 Threads	0,748	0,918	1,068	1,266	2,378	4,746	8,311	15,006	29,032	54,121	112,380	205,624
Tiempo(s) 24 Threads	0,557	0,689	0,855	1,065	2,208	4,350	7,699	12,462	25,044	47,189	91,238	182,308
Tiempo(s) 48 Threads	0,628	0,774	0,931	1,123	2,358	3,699	7,130	12,987	25,637	49,029	96,331	185,462
Speedup con 2 Threa	1,863	1,828	1,722	1,831	1,770	1,753	1,715	1,777	1,768	1,766	1,738	1,730
Speedup con 4 Threa	3,419	3,615	3,633	3,603	3,438	3,399	3,309	3,158	3,080	2,934	3,142	3,248
Speedup con 8 Threa	6,854	7,156	7,179	7,025	6,056	5,561	6,048	5,962	6,126	5,903	5,695	5,530
Speedup con 16 Thre	12,709	13,602	13,148	12,275	9,752	8,415	8,879	9,074	8,999	9,299	8,854	9,602
Speedup con 24 Thre	17,066	18,123	16,423	14,592	10,503	9,181	9,585	10,926	10,432	10,665	10,906	10,830
Speedup con 48 Thre	15,137	16,133	15,083	13,838	9,835	10,797	10,350	10,484	10,191	10,265	10,329	10,645

Tabla 8: Tiempos y speedup de la función diámetro según el número de aristas generadas por nodo.

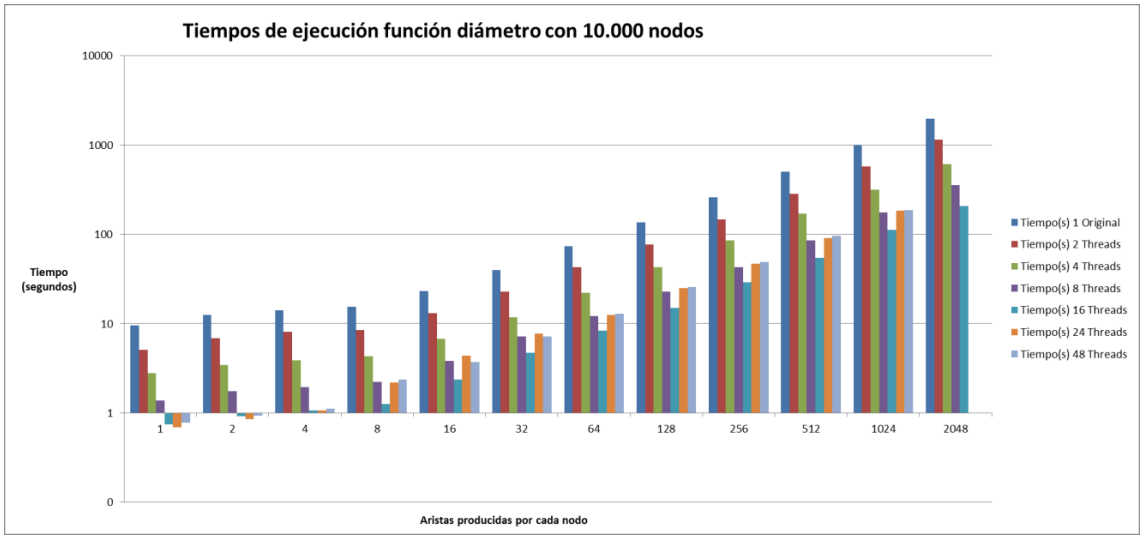


Gráfico 4: Tiempos de ejecución de la función diámetro según el número aristas generadas por nodo.

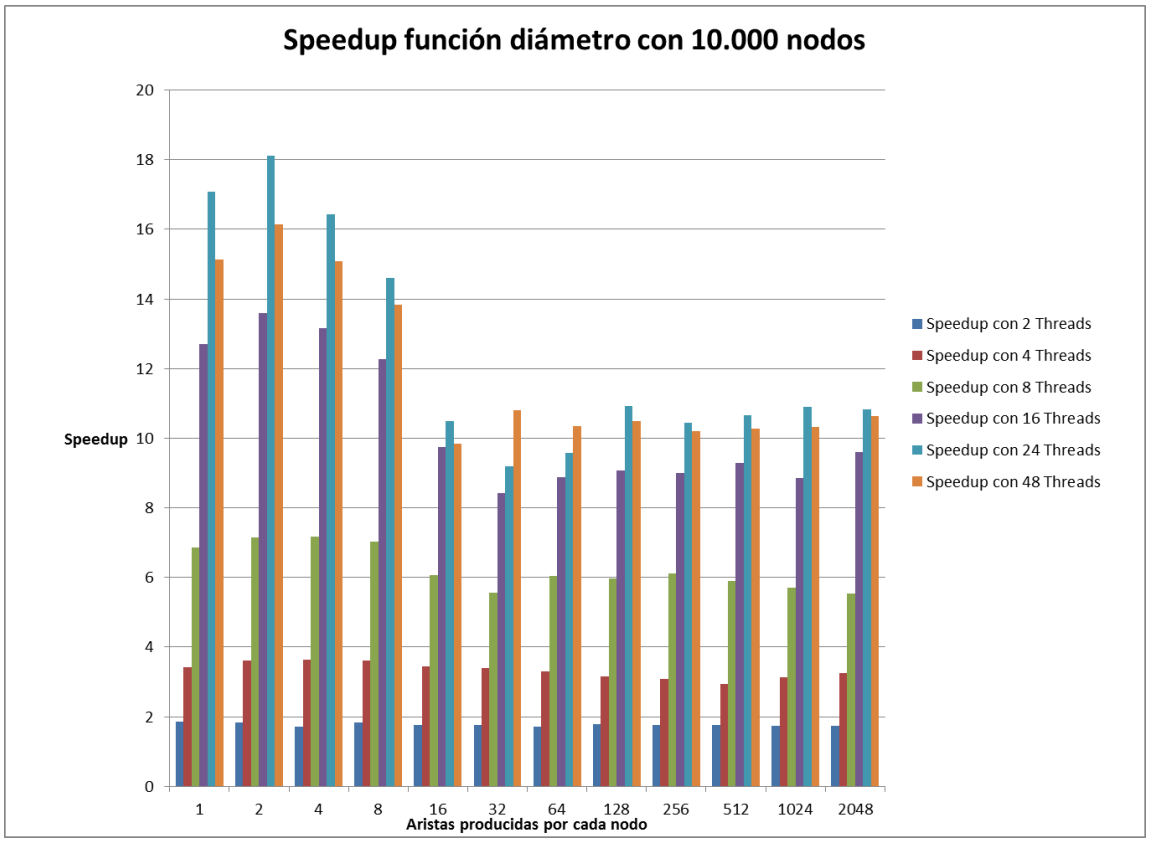


Gráfico 5: Speedup de la función diámetro según el número de aristas generadas por nodo.

Se observa como el tiempo de ejecución crece en el mismo orden que el número de aristas. El doble de aristas significa el doble de tiempo de procesamiento.

Por otro lado, se ve como el speedup es mejor con pocas aristas. El modelo de Barabasi-Albert produce que los primeros nodos insertados tengan muchas más aristas que los que se insertan después. Esto hace que la carga esté peor repartida cuando hay muchos nodos.

5.3 Evaluación Clustering Coefficient

Tras la ejecución de la batería de pruebas se han elegido los conjuntos de datos que mejor muestran el comportamiento de la función del cálculo del Clustering Coefficient. Concretamente los grafos completos.

Nodos	10	100	1000	10000
Tiempo(s) Original	0,001	0,005	4,374	4722,874
Tiempo(s) 2 Threads	0,000	0,003	2,545	2578,876
Tiempo(s) 4 Threads	0,001	0,002	1,321	1558,137
Tiempo(s) 8 Threads	0,002	0,002	0,761	981,771
Tiempo(s) 16 Threads	0,004	0,003	0,946	577,724
Tiempo(s) 24 Threads	0,006	0,004	0,705	666,152
Tiempo(s) 48 Threads	0,005	0,003	0,545	572,633
Speedup con 2 Threads	0,000	1,667	1,719	1,831
Speedup con 4 Threads	1,000	2,500	3,311	3,031
Speedup con 8 Threads	0,500	2,500	5,748	4,811
Speedup con 16 Threads	0,250	1,667	4,624	8,175
Speedup con 24 Threads	0,167	1,250	6,204	7,090
Speedup con 48 Threads	0,200	1,667	8,026	8,248

Tabla 9: Tiempos y speedup de la función clustering coefficient con grafos completos

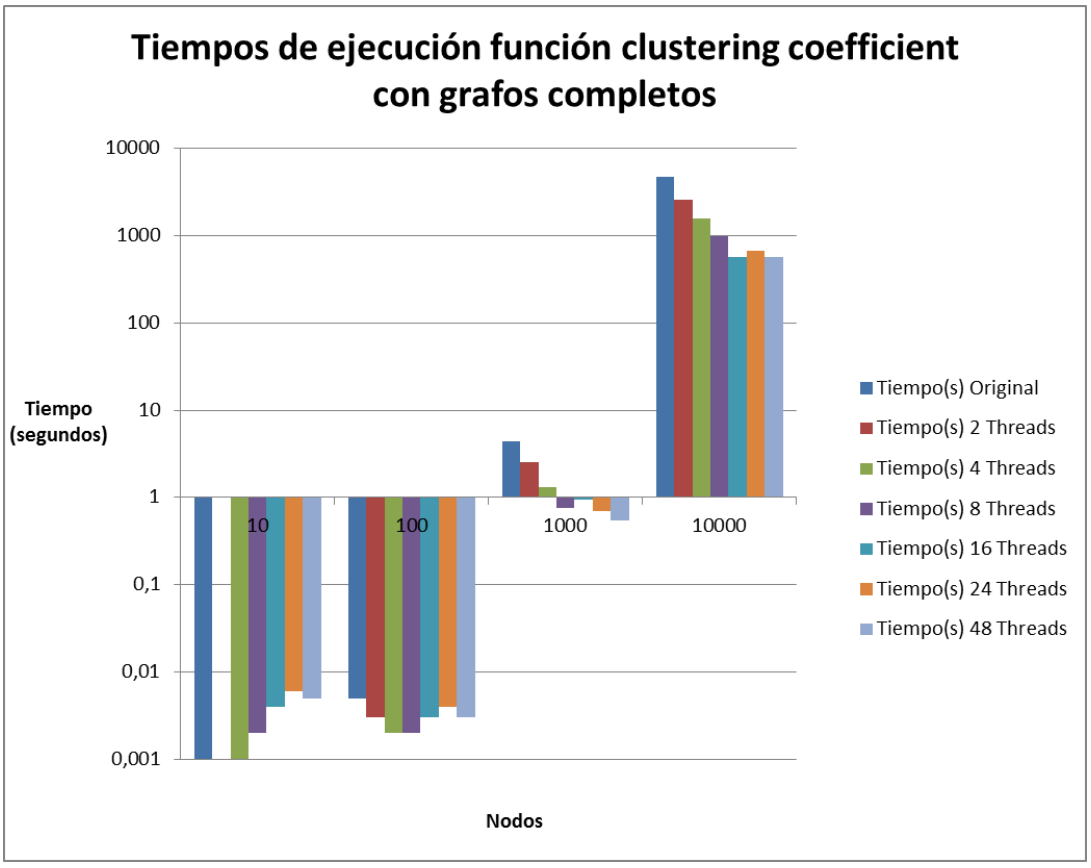


Gráfico 6: Tiempos de ejecución de la función clustering coefficient para grafos completos.

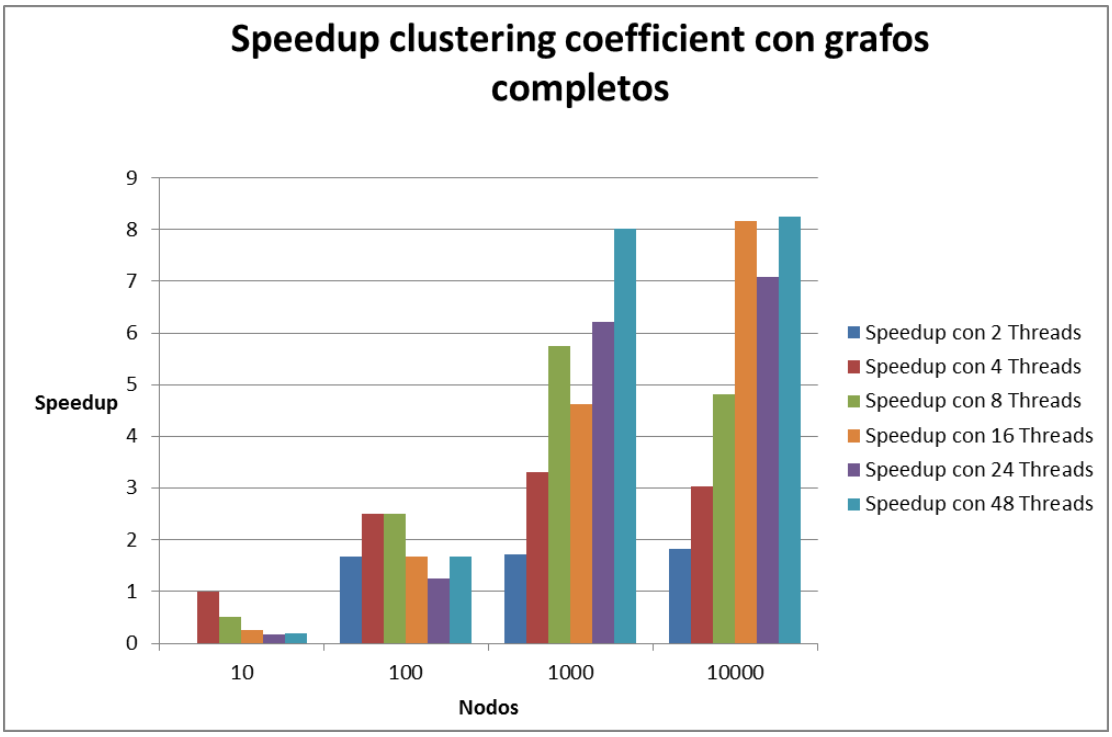


Gráfico 7: Speedup de la función clustering coefficient para grafos completos.

Se observa como el tiempo de ejecución aumenta exponencialmente debido no tanto al número de nodos, como a las aristas que los conectan.

Los tiempos de ejecución son prácticamente despreciables con 10 y con 100 nodos. Empiezan a requerir un coste de varios segundos en torno a los 1.000 nodos. Y con 10.000 se tarda en secuencial en torno a la hora y media.

El speedup no crece linealmente respecto al número de cores. Cuantos más cores menor es el aumento, debido a los costes de sincronización. En ocasiones esos costes adicionales son mayores que el beneficio, por ejemplo en el caso de 10.000 cores el speedup es mejor con 16 threads que con 24.

El mejor caso se produce al lanzar dos threads por core, de forma que si uno queda bloqueado, el siguiente se ejecuta.

6 Planificación

La planificación se ha hecho por semanas y el tiempo de trabajo se ha medido en horas.

El trabajo se ha dividido en 8 partes:

- Estado del arte: investigación del trabajo previo existente y análisis de las alternativas.
- Implementación: diseño y codificación de la solución propuesta.
- Pruebas: pruebas sobre la solución propuesta.
- Diseño/Arquitectura: documentación de la solución propuesta.
- Evaluación: análisis de los resultados obtenidos.
- Introducción: presentación inicial del trabajo.
- Conclusiones: cierre y reflexiones sobre el trabajo.
- Resumen: resumen en inglés del trabajo.

Inicialmente se realizó una planificación inicial tal como podemos ver en la siguiente imagen:

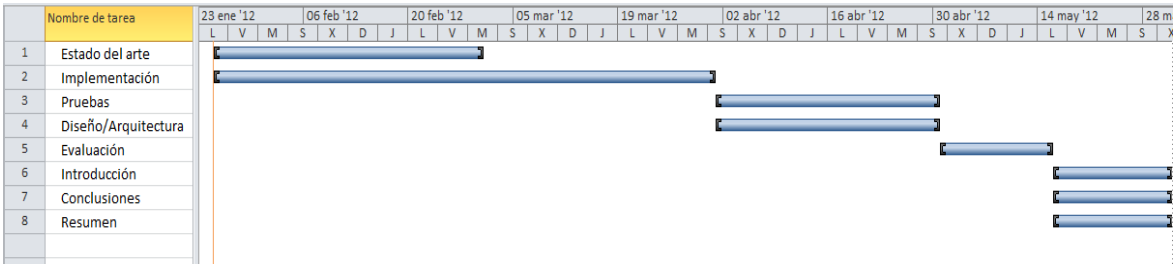


Ilustración 15: Planificación semanal de las distintas partes.

Se han ido contabilizando las horas a lo largo del desarrollo del proyecto. Como podemos ver en la siguiente tabla y gráfico:

Fechas	23-ene	30-ene	06-feb	13-feb	20-feb	27-feb	05-mar	12-mar	19-mar	26-mar	02-abr	09-abr	16-abr	23-abr	30-abr	07-may	14-may	21-may	28-may	04-jun	11-jun
Semana	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
HORAS PLANIFICADAS	10	20	30	40	50	60	70	80	90	100	112	124	136	148	150	162	174	186	198	198	198
Estado del arte	5	10	15	20	25	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
Implementación	5	10	15	20	25	30	40	50	60	70	70	70	70	70	70	70	70	70	70	70	70
Pruebas	0	0	0	0	0	0	0	0	0	0	6	12	18	24	25	25	25	25	25	25	25
Diseño/Arquitectura	0	0	0	0	0	0	0	0	0	0	6	12	18	24	25	25	25	25	25	25	25
Introuducción	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	6	9	12	12	12
Conclusiones	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	8	12	16	16	16
Resumen	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	10	15	20	20	20
HORAS REALIZADAS	6	16	21	24	24	28	35	48	52	54	62	70	75	75	88	102	116	140	153	153	163
Estado del arte (reales)	6	13	14	15	15	15	15	15	19	19	19	22	25	25	32	32	32	32	32	32	32
Implementación (reales)	0	3	7	9	9	13	20	33	33	33	37	42	42	42	48	55	67	69	69	69	69
Pruebas (reales)	0	0	0	0	0	0	0	0	0	2	6	6	8	8	8	10	12	12	16	16	16
Diseño/Arquitectura (reales)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	5	16	18	18	18
Introuducción (reales)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	11	11	11
Conclusiones (reales)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	7	7
Resumen (reales)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10
DESVÍO	4	4	9	16	26	32	35	32	38	46	50	54	61	73	62	60	58	46	45	45	35

Tabla 10: Horas planificadas y realizadas del proyecto.

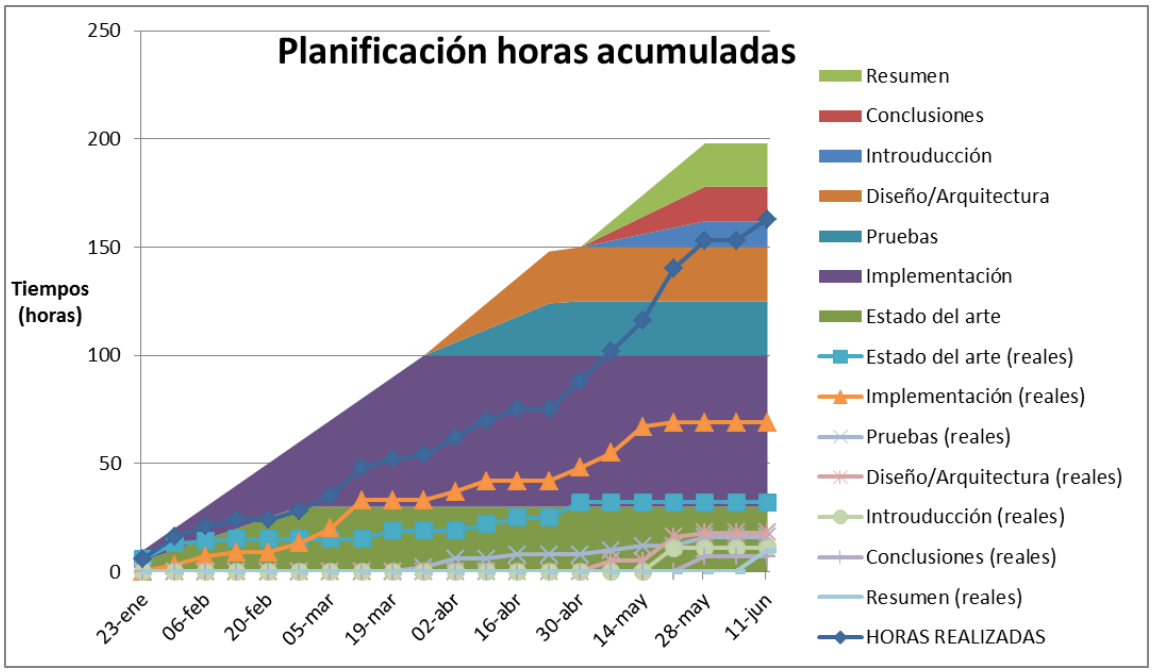


Gráfico 8: Horas planificadas acumuladas y reales.

Como se puede ver las horas planificadas han sido mayores que las reales, aunque el desvío es pequeño, sigue siendo beneficioso para los intereses económicos del proyecto.

7 Presupuesto

Coste personal

Personal	Salario mensual	Nº Pagas	Salario	Salario con seguridad social (35%)	Horas /año	Coste hora	Horas Estimadas
1	1.720,00 €	14	24.080,00 €	33.712,00 €	1750	19,26 €	198
TOTAL COSTE EMPLEADOS							**3.814,27 €**

Tabla 11: Coste empleados.

*Sueldo habitual en un recién graduado en ingenería.

Coste equipos

Artículo	Coste unitario (€)	Unds.	Coste total (€)	Vida útil (años)	Coste al mes (€)	Meses de uso
HP Pavilion p6-2103es	599,00 €	1	599,00 €	5	9,98 €	4
			TOTAL COSTE EQUIPO			**39,93 €**

Tabla 12: Coste equipos.

Coste material fungible

Concepto	Coste (€)	Unds.	Coste total (€)
Paquete 500 DIN A4	5,00 €	1	5,00 €
CDs	0,90 €	5	4,50 €
Encuadernación	10,00 €	1	10,00 €
TOTAL COSTE MATERIAL FUNGIBLE			**19,50 €**

Tabla 13: Coste material fungible.

Coste software y herramientas

Se ha utilizado software libre y gratuito por lo que su uso no ha tenido coste adicional.

Producto	Coste (€)
R	0€
Sistema Operativo Linux	0€
OpenMP	0€
Librería igraph	0€
TOTAL	**0€**

Tabla 14: Coste software y herramientas.

Total costes directos

COSTES DIRECTOS	Coste(€)
Empleados	3.814,27 €
Equipo	39,93 €
Material fungible	19,50 €
TOTAL	**3.873,71 €**

Tabla 15: Total costes directos.

Presupuesto:

Se considera un 10% de gastos indirectos tales como luz, internet o calefacción.

Además se aplica un 5% de margen de riesgo y se espera obtener un 7% de beneficios.

CONCEPTO	COSTE	ACUMULADO
COSTES DIRECTOS	3.873,71 €	3.873,71 €
COSTES INDIRECTOS (10%)	387,37 €	4.261,08 €
RIESGO (5%)	213,05 €	4.474,13 €
BENEFICIO (15%)	671,12 €	**5.145,25 €**

Tabla 16: Presupuesto final.

PRECIO TOTAL: 5.145,25€ (IVA no incluido).

Cinco mil ciento cuarenta y cinco euros y veinticinco céntimos (IVA no incluido).

8 Conclusiones y líneas futuras

Durante este trabajo se ha desarrollado un método para mejorar el tiempo de cálculo necesario para determinar si un grafo de gran tamaño es un Small World. Para ello se ha aprovechado la arquitectura multinúcleo, paralelizando las funciones necesarias de la librería igraph.

Para calcular un Small World se necesita hacer dos veces la función diámetro y dos veces calcular el clustering coefficient, lo que hace un total de cuatro funciones pesadas, cualquier mejora sobre cualquiera de estas funciones permite ganar mucho tiempo total. Tras haber realizado la evaluación, se puede concluir por tanto, que el resultado es un éxito ya que la ganancia de tiempo es muy alta.

Otra cuestión importante es que las funciones de calcular el diámetro y el clustering coefficient no sólo se utilizan para calcular si un grafo es un Small World. La mejora en estas funciones beneficia a aplicaciones más generales. Especialmente el cálculo del diámetro que es una de las funciones más comunes de la teoría de grafos. Además, es más normal que la función resulte costosa ya que un grafo grande tiene un alto coste (mayor o menor según el número de aristas), pero para calcular el clustering coefficient no sólo depende del tamaño sino que mucho de que sea muy conexo.

Respecto a igraph, como se ha visto es una completa y muy utilizada librería para tratar grafos. Sin embargo, tiene dos carencias fundamentales: no aprovecha más de un procesador y no puede trabajar con grafos del orden de millones de vértices y aristas.

La primera carencia es que sus cálculos se hacen de manera secuencial. Dado que muchos de los cálculos son computacionalmente pesados y en la actualidad la mayoría de ordenadores dónde se ejecutan este tipo de programas tienen procesadores de varios núcleos esta carencia resulta importante.

Como se ha podido observar durante este trabajo, la mayoría de las funciones de esta librería utilizan una estructura paralelizable. Esto hace que el rendimiento mejore considerablemente.

Cada día son más habituales las computadoras con varios núcleos y memoria compartida. Los ordenadores domésticos y los portátiles de prácticamente todas las gamas utilizan este tipo de arquitectura y ya se empiezan a imponer en smartphones, tablets y otros dispositivos. Por esta razón es fundamental conseguir sacarle el máximo provecho a estas arquitecturas, especialmente en tareas dónde la diferencia sea grande.

Herramientas como OpenMP son de gran utilidad. OpenMP es una API sencilla y extendida que permite aprovechar las arquitecturas multicore de forma fácil y

transparente a la máquina. La experiencia al utilizarla en igraph ha sido muy satisfactoria ya que es una herramienta perfecta para paralelizar bucles y tiene una serie de mecanismos de control que son sencillos, pero efectivos.

A pesar de la los grandes beneficios del paralelismo en este tipo de tareas, diseñar un programa que aproveche este tipo de procesamiento es más complejo que hacerlo con un programa que funcione de forma secuencial. Un programa secuencial es mucho más previsible ya que en un programa paralelo hay que tener en cuenta que el orden de ejecución puede variar. Habrá que detectar las dependencias y los recursos compartidos y utilizar los mecanismos de sincronización necesarios y las variables privadas de cada proceso para garantizar la eficacia y buscar la máxima eficiencia.

En muchos programas el beneficio del paralelismo es mínimo o nulo porque no requieren cálculos pesados. También hay otros casos de algoritmos complejos y pesados que no se pueden paralelizar o resulta difícil y poco beneficioso. Sin embargo, actualmente todavía hay muchos programas y librerías como igraph que aun teniendo unas características idóneas para aprovechar las máquinas actuales, no lo hacen.

Como se ha podido observar durante las pruebas, el rendimiento no es lineal respecto al número de procesadores. Más procesadores implican más recursos compartidos, más sincronización y mayores tiempos de espera. Por lo que aunque su aumento suele mejorar el tiempo d ejecución, cada nuevo procesador lo mejora menos.

Por otra parte, la librería no está pensada para trabajar con grafos de órdenes de millones de vértices y aristas, al contrario de lo que dice en su página web. Esta limitación se hace más notable al paralelizar las funciones. Ya que lo que con una ejecución secuencial era muy pesado computacionalmente, con una ejecución en paralelo no lo es tanto.

Dada la importancia de la teoría de grafos en la actualidad, dónde además, cada vez más se utilizan redes de todo tipo con mayor tamaño y complejidad. Habría que pensar en que una librería especializada en grafos como igraph debería estar pensada para tratar grafos de mayor tamaño y hacerlo con mayor velocidad. Aprovechando las características actuales de los ordenadores tanto las arquitecturas multinúcleo como el aumento de la memoria.

Yendo un paso más allá, para buscar el máximo rendimiento computacional en una librería de grafos habría que pensar en un modelo de múltiples procesadores con memoria distribuida, o el cada vez más frecuente modelo híbrido que combina los beneficios de ambos modelos.

A modo valoración personal, el trabajo me ha concienciado especialmente de la importancia de la programación paralela. Tradicionalmente las mejoras de las

máquinas eran en la velocidad de procesamiento. Los procesadores mejoraban y los mismos programas iban más rápido con los nuevos procesadores.

Realmente con este trabajo me he dado cuenta que eso ya no es así. Si se quiere sacar el máximo rendimiento de los ordenadores actuales hay que saber realizar programas paralelos. Por supuesto, sin olvidarse de las herramientas que permiten hacerlos portables.

Por otra parte, aunque había usado grafos antes del trabajo para algunas tareas sencillas, tampoco estaba concienciado de su importancia. No imaginaba muchos de sus usos, especialmente los que requieren usar grafos enormes. Por esa razón, no pensaba que pudieran generar problemas por su coste computacional y de memoria. Y sin embargo, probablemente son esos problemas los más importantes, razón por la que solucionarlos cobra mayor importancia.

9 Referencias

[1] Igraph, «The igraph Project,» [En línea]. Available: http://igraph.sourceforge.net/index.html.

[2] GNU R, «The R Project for Statistical Computing,» [En línea]. Available: http://www.r-project.org/.

[3] Lemon, «Graph library,» [En línea]. Available: http://lemon.cs.elte.hu/trac/lemon.

[4] boost, «The Boost Graph Library,» [En línea]. Available: http://www.boost.org/doc/libs/1_49_0/libs/graph/doc/index.html.

[5] Herramientas de minería de datos, 2010, «Rexer Analytics,» [En línea]. Available: http://www.rexeranalytics.com/Data-Miner-Survey-Results-2010.html.

[6] S-PLUS, «Tibco Spotfire S+,» [En línea]. Available: http://spotfire.tibco.com/products/s-plus/statistical-analysis-software.aspx.

[7] MatLab, «MathWorks,» [En línea]. Available: http://www.mathworks.es/products/matlab/.

[8] LINPACK, «LINPACK Fortran library,» [En línea]. Available: http://www.netlib.org/linpack/.

[9] ESIPACK, «EISPACK Fortran library,» [En línea]. Available: http://www.netlib.org/eispack/.

[10] GNU Octave, «GNU Octave,» [En línea]. Available: http://www.gnu.org/software/octave/.

[11] Intel TBB, «Intel Threading Building Blocks,» [En línea]. Available: http://threadingbuildingblocks.org/.

[12] OpenMP, «API specification for parallel programming,» [En línea]. Available: http://openmp.org/wp/.

[13] R. A. a. A.-L. Barabási, «Statistical mechanics of complex networks,» 2002. [En línea]. Available: http://www.nd.edu/~networks/Publication%20Categories/03%20Journal%20Articles/Physics/StatisticalMechanics_Rev%20of%20Modern%20Physics%2074,%2047%20%282002%29.pdf.