Proyecto fin de carrera de Ingeniería en Informática

Facultad de Informática

Universidad de Murcia

Herramientas para el estudio de prestaciones en clusters de computación científica, aplicación en el Laboratorio de Computación Paralela

Autor: Virginio García López
Directores: Javier Cuenca Muñoz
Domingo Giménez Cánovas

Febrero de 2009

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

La proliferación de clusters informáticos en la actualidad plantea una serie de retos que el ingeniero en informática debe afrontar. Los clusters o agrupaciones de ordenadores se utilizan con diversos propósitos: para la investigación, para servicios web, en el modelo cloud computing, o en el cálculo científico entre otros. Es en este último ámbito, el cálculo científico, donde vamos a centrar la atención en este proyecto. Este proyecto afronta el reto de diseñar y construir herramientas que permitan medir el rendimiento de tecnologías habituales en los clusters, y de esa forma responder cuestiones como las que veremos a continuación.

La primera de las tecnologías que podemos encontrar en un cluster de computación científica es OpenMP [1]: se trata de un estándar para la programación en memoria compartida. OpenMP permite escribir programas con directivas que el compilador interpreta como código paralelizable. Para hacer uso de esta tecnología podemos usar el compilador de Intel `icc` o el compilador GNU/Linux `gcc`, que en versiones recientes ha incorporado el soporte para OpenMP. Al empezar a trabajar en un cluster podemos preguntarnos cuál de los dos compiladores ofrece mejor rendimiento con OpenMP, y qué conjunto de optimizaciones del compilador es la adecuada. La herramienta que desarrollaremos permitirá responder a estas cuestiones de forma objetiva mediante la realización de un conjunto de mediciones.

La segunda de las tecnologías también trabaja con programas paralelos y se trata del estándar de paso de mensajes MPI [2]. MPI funciona sobre una red de interconexión que normalmente se basa en los protocolos TCP/IP. Resulta interesante la cuestión del rendimiento en esa red al usar las primitivas de MPI. Además, un aspecto que nos interesa saber es qué ocurre con el rendimiento al trabajar con arquitecturas heterogéneas, tanto el velocidad como en tamaño de palabra: 64 bits frente a 32 bits. Las herramientas desarrolladas expondrán con claridad el comportamiento de estos casos y darán

respuesta a estas dos cuestiones.

La tercera de las tecnologías es el núcleo computacional BLAS [3], que es ampliamente utilizado en álgebra lineal. De este núcleo computacional es frecuente que encontremos distintas implementaciones en el mismo cluster. La cuestión que surge en este caso es ¿cuál de las distintas implementaciones ofrece el mejor rendimiento en cada uno de los tres niveles de rutinas[1] en que se divide BLAS?

La cuarta y última de las tecnologías con la que vamos a trabajar es LAPACK [4]. Se trata de un paquete de álgebra lineal que resuelve problemas de mayor nivel que BLAS. LAPACK utiliza las rutinas de BLAS, por lo que también existen diversas implementaciones de LAPACK asociadas a las distintas implementaciones de BLAS. Cabe entonces preguntarse si las implementaciones que ofrecen mejor rendimiento en BLAS son también las que ofrecen mejor rendimiento en LAPACK y, claro está, averiguar el rendimiento de LAPACK.

De las cuatro tecnologías mencionadas, en otros proyectos fin de carrera[2] y en apuntes de asignaturas de master [5] [6] se detalla la programación con estas herramientas, por lo que este proyecto no tratará sobre la programación en C con OpenMP, MPI, BLAS ni LAPACK.

1.1. Metodología

En la Facultad de Informática de la Universidad de Murcia, en el laboratorio de computación científica se encuentra el cluster SOL, formado por cinco nodos multicore con Intel Xeon de 64 bits y un sexto nodo formado por un Pentium III biprocesador de 32 bits. Este proyecto se ha escrito desde una doble perspectiva: por un lado interesaba medir el rendimiento de las cuatro tecnologías antes comentadas en el cluster SOL, y por otro lado desarrollar una herramienta que permita reproducir todas las mediciones en otros clusters. Las mediciones en el cluster SOL permitirá a los alumnos de quinto curso, a los alumnos de master y doctorado en Ingeniería en Informática y a los componentes del grupo de Computación Paralela de la Universidad de Murcia [7] conocer las prestaciones de un cluster con el que pueden trabajar habitualmente. Puesto que las herramientas de medición se han desarrollado en el cluster SOL, este cluster ha servido de referencia en el desarrollo de las herramientas, sirviendo como guía para el estudio. De esta

[1] Las rutinas de BLAS se clasifican en tres niveles, según el tipo de problemas que resuelven. Los niveles de BLAS se explicarán en el capítulo 6.

[2] En el proyecto [8] se desarrollan seminarios de BLAS y LAPACK. En el proyecto [9] se desarrolla de MPI

forma se desarrollaban herramientas, se experimentaba en el cluster SOL, y se ajustaba la herramienta desarrollada o se añadían nuevas características hasta que la herramienta realmente fuera un instrumento de medición útil. En el conjunto de herramientas, no sólo se llevan a cabo mediciones, sino que también se elaboran gráficos y tablas de resultados. Al trabajar en el cluster SOL, también se han ido creando las herramientas gráficas según se obtenían resultados en el cluster. En este caso también la experimentación ha sido la guía para decidir qué gráficos son más interesantes, o cómo se visualizan mejor los resultados, de forma que según se experimentaba, la herramienta sufría constantes mejoras. Al finalizar la experimentación, ha quedado un conjunto de herramientas de medición y herramientas que generan gráficos para la interpretación de resultados que podemos considerar satisfactoria. Se ha analizado también las posibles diferencias en otros clusters, para intentar que la herramienta se adecue a otros cluster de computación científica.

1.2. Contenido

Este documento está estructurado en ocho capítulos y tres anexos. En el capítulo 2 haremos una introducción al cluster SOL desde el punto de vista del usuario, tanto para explicar el funcionamiento general del cluster, como para explicar el funcionamiento básico de OpenMP, MPI, BLAS y LAPACK en dicho cluster. El seguimiento de los pasos que se explican en el capítulo 2 permite hacerse una idea del funcionamiento interno de las herramientas desarrolladas, ya que las herramientas ejecutarán los pasos habituales que un usuario suele hacer, tal como configuración, compilación y ejecución. Este capítulo se complementa con el anexo C que da una introducción al cluster, pero esta vez desde el punto de vista del administrador.

El capítulo 3 describe el diseño de las herramientas desarrolladas: se trata de diversos *scripts* que compilan código C, lo ejecutan varias veces con distintos parámetros, obtienen mediciones, y generan gráficos e incluso tablas a partir de las mediciones. En este capítulo veremos tanto el diseño como la configuración y ejecución de cada herramienta. No obstante, como se comentó anteriormente, las herramientas se han desarrollado a la vez que se llevan a cabo experimentaciones en el cluster SOL, por lo que realmente no se llega a comprender el potencial de las herramientas hasta los siguientes capítulos. El anexo B complementa a este capítulo 3 mostrando los comandos de forma ordenada: dicho anexo sirve como guía de referencia rápida a la hora de lanzar los comandos que constituyen la herramienta.

Los capítulos 4, 5, 6 y 7 constituyen la parte de estudio de prestaciones de cada una de las cuatro tecnologías: OpenMP, MPI, BLAS y LAPACK.

En estos capítulos se aplican las herramientas desarrolladas al cluster SOL. Es también en estos capítulos donde se explican los experimentos llevados a cabo, así como aspectos más teóricos que requieren consideración para comprender las mediciones realizadas. Estos capítulos permiten conocer en profundidad las prestaciones del cluster SOL y, a la vez, comprender mejor las pruebas que realizan la herramientas, al ser una aplicación directa de dichas herramientas. Los distintos gráficos mostrados en estos capítulos también son producto de la ejecución de la herramienta que desarrollamos para este proyecto. En general, las herramientas generan gran cantidad de gráficos para el estudio de las prestaciones, y en este documento se muestran sólo aquellos cuyos resultados son relevantes.

Las conclusiones y vías futuras se exponen en el capítulo 8. Las conclusiones serán el resultado del estudio de las prestaciones en el cluster SOL como consecuencia del uso de la herramienta diseñada y las vías futuras abren el camino a proyectos más ambiciosos.

Este documento finaliza con los anexos. En el anexo de las mediciones se detallan los resultados obtenidos con la herramienta en el cluster SOL. Las tablas mostradas también son fruto directo de la herramienta, ya que esta es capaz de generar tablas en LaTeX, que es el procesador de documentos en el que se escribe este proyecto fin de carrera. De esta manera, con el uso de esta herramienta en cualquier plataforma obtenemos automáticamente una remesa de gráficos y tablas directamente introducibles en un documento LaTeXque trate sobre las prestaciones en dicha plataforma.

1.3. Medidas de prestaciones

A partir del capítulo 4 iniciamos el estudio de prestaciones partiendo de las mediciones y gráficos obtenidos con la herramienta desarrollada. Frecuentemente nos referiremos a conceptos de medidas de rendimiento que explicaremos a continuación. En cuanto a los algoritmos paralelos (capítulos 4 y 5) definimos los siguientes conceptos [10]:

Speedup: Se llama *speedup* a la ganancia de velocidad que se consigue con un algoritmo paralelo al resolver un problema con respecto a un algoritmo secuencial para el mismo problema. La fórmula del *speedup* es:

$$S(n,p) = \frac{t(n)}{t(n,p)}$$

donde $t(n)$ es el tiempo correspondiente a un algoritmo secuencial que resuelve el problema de tamaño n, y $t(n,p)$ es el tiempo correspondiente al algoritmo paralelo para p hilos o procesos. El valor de $S(n,p)$ está en el rango entre cero y p, siendo mejor cuanto mayor sea.

Eficiencia: La eficiencia da idea de la porción de tiempo que los procesadores se dedican a trabajo útil. Normaliza el valor del *speedup* dividiendo por el número de procesadores:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{t(n)}{pt(n,p)}$$

Tendrá un valor entre cero y uno, siendo lo ideal que sea lo más cercano a uno posible.

Coste: El coste representa el tiempo o trabajo realizado por todo el sistema en la resolución del problema. Es el tiempo de ejecución multiplicado por el número de procesadores usados:

$$C(n,p) = pt(n,p)$$

El coste será mejor cuanto más se acerque al valor del tiempo secuencial $t(n)$.

En cuanto a la elección del tiempo secuencial $t(n)$, escogeremos el tiempo de ejecución en un core del algoritmo paralelo. Esta opción es razonable desde el punto de vista de la herramienta, puesto que es fácil calcularlo experimentalmente. La elección de otras opciones, como el tiempo del mejor algoritmo secuencial conocido, no es posible al trabajar con multiplicación de matrices, al no conocerse el orden del mejor algoritmo secuencial que lo resuelve.

Tanto para trabajar con algoritmos paralelos como secuenciales (capítulos 6 y 7), utilizaremos la medida de los megaflops. Los megaflops se definen como los millones de operaciones en punto flotante que se llevan a cabo en un segundo. Esta medida la calcularemos a partir de la complejidad del algoritmo, despreciando operaciones como saltos y otras con enteros. En todos los cálculos de medidas de rendimiento, ya sean paralelos o secuenciales, tomaremos para este cálculo de los megaflops el tamaño de problema mayor con el que hayamos experimentado.

Capítulo 2

Guía de usuario del cluster SOL

Este capítulo introduce a los usuarios del cluster SOL (alumnos de distintos cursos y el grupo de computación paralela de la Universidad de Murcia), al manejo del cluster de computación científica que pueden utilizar habitualmente. Veremos la topología del cluster, la forma de acceso y la forma de trabajar con el software de computación científica. Comprender la forma de trabajar con el software nos permite hacernos una idea del funcionamiento interno de los *scripts* que conforman la herramienta desarrollada ya que, internamente, estos *scripts* automatizan la forma de trabajar de un usuario. Al finalizar el capítulo se describe el sistema gestor de colas y algunas utilidades que el usuario puede usar para trabajar en el cluster con mayor comodidad.

2.1. Configuración básica de la plataforma

La plataforma de computación paralela está basada en el diseño de clusters Beowulf [11]. El hardware está formado por seis nodos conectados por un conmutador ethernet. Cinco de los nodos tienen un procesador Intel Xeon 3 GHz con 1.5 GB de RAM y arquitectura de 64 bits. Estos nodos tienen en total 16 cores: tres de estos nodos disponen de cuatro cores, y los otros dos nodos tienen dos cores. Entre los nodos de Intel Xeon con cuatro cores se encuentra el nodo SOL, que hace de frontal del cluster. El sexto nodo consta de dos procesadores Intel Pentium III a 1 GHz con 384 MB de RAM, y su propósito es servir para la computación heterogénea (además de ser un nodo más lento que los anteriores, es el único nodo de 32 bits). Contando los procesadores del Pentium III; el cluster dispone de un total de 18 cores de computación.

La arquitectura básica del cluster se ilustra en la figura 2.1. Tanto el conjunto de nodos como el nodo frontal se denominan SOL: distinguiremos

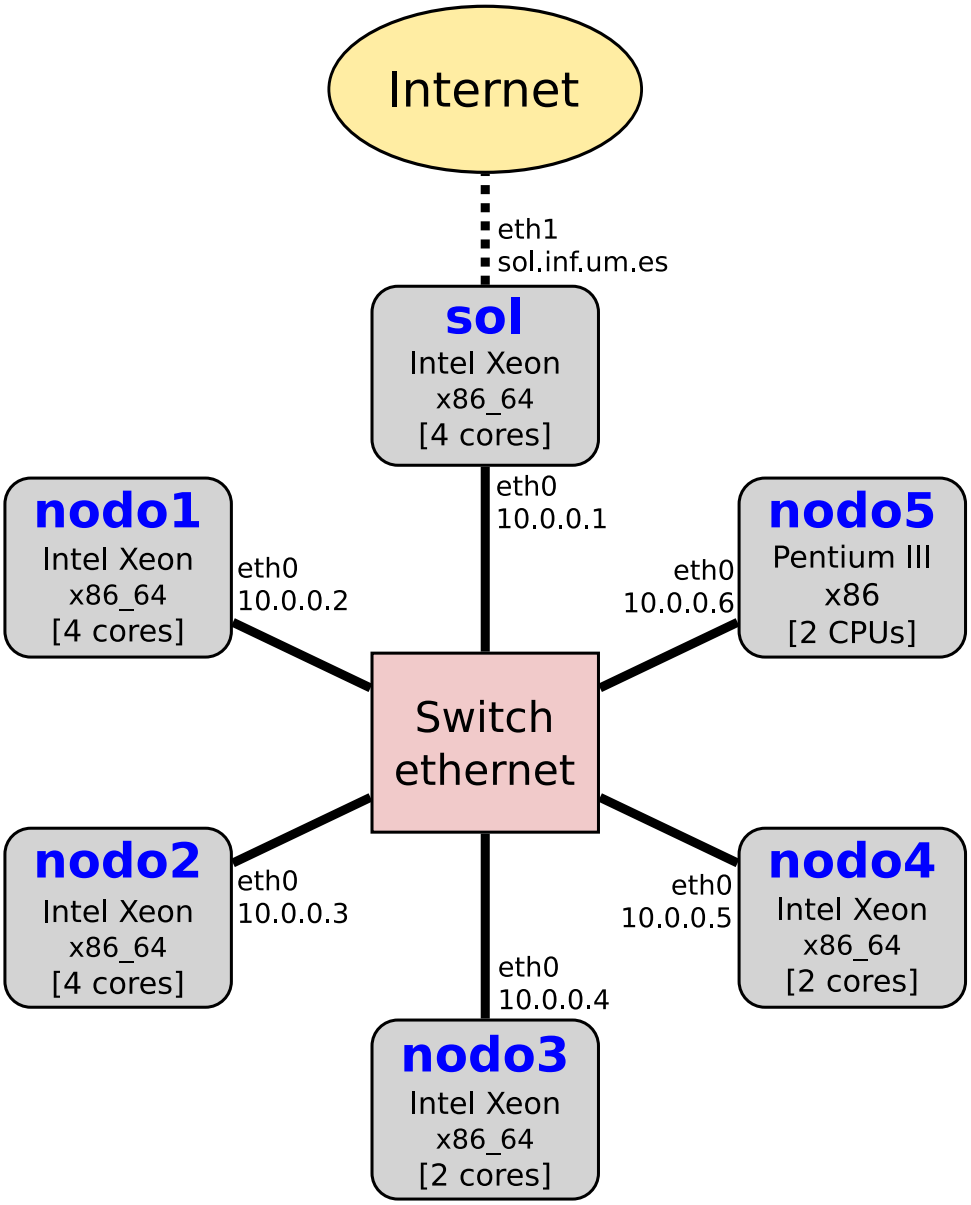

Figura 2.1: Arquitectura básica del cluster SOL

uno de otro según nos refiramos a un sólo nodo o al cluster completo. Para acceder al cluster es necesario entrar primero en el nodo SOL. Este nodo es el frontal (frontend) para el resto del cluster, y es el único que tiene una IP pública. Todos los nodos tienen una IP privada de clase A, que sirve para comunicarse con los demás nodos, y que permite salir a Internet a través del nodo SOL[1]. El nodo SOL tiene dos interfaces, una de la red pública, y otra de la red privada.

Cada nodo tiene instalado su propio sistema operativo Gentoo Linux en sus discos locales. Se aprovecha al máximo la arquitectura de cada máquina compilando para el ordenador específico todas las aplicaciones. Las versiones de los paquetes de computación científica más importantes son iguales en todos los nodos, salvando la diferencia de que los nodos con Intel Xeon son de 64 bits y el Pentium III de 32 bits.

El directorio `/home`, donde residen las cuentas de los usuarios, se encuentra en el nodo SOL y se comparte con el resto de los nodos a través de NFS [12]. Esto permite conectarse a cualquier nodo y seguir trabajando con los mismos datos de usuario.

2.2. Guía de usuario

2.2.1. Acceso al cluster

Para empezar a trabajar en el cluster, es necesario acceder al frontend. La única forma de acceso permitida al frontend es a través del protocolo SSH. Si tenemos una cuenta en el cluster SOL, basta con acceder de forma habitual:

```
1 ssh usuario@sol.inf.um.es
```

Una vez introducida la contraseña, podemos trabajar en el directorio del usuario con el interprete `bash`. Para ir a otro nodo usaremos de nuevo `ssh`, indicando sólo el nombre del nodo al que queremos acceder:

nodo1: Intel Xeon (4 cores)

nodo2: Intel Xeon (4 cores)

nodo3: Intel Xeon (2 cores)

nodo4: Intel Xeon (2 cores)

nodo5: Intel Pentium III (2 procesadores)

[1] Usando NAT, en particular el mecanismo de Linux conocido como Masquerading

En el acceso a cualquiera de los nodos, se preguntará de nuevo la contraseña. Podemos evitar esto estableciendo una relación de confianza entre los nodos para nuestro usuario particular. Para hacer esto, en cualquiera de los nodos ejecutamos:

```
ssh-keygen -t dsa
cp ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys
```

Los comandos anteriores establecen la relación de confianza de la siguiente manera: en primer lugar se genera un par de claves DSA de criptografía asimétrica. Para ello pulsaremos *enter* para aceptar todos los valores por defecto. En segundo lugar se copia la clave pública a las claves autorizadas para la autenticación. Este último comando asume que no hay ninguna otra clave autorizada para la autenticación del usuario actual, y que todos los nodos ven el mismo directorio de usuario. Estas dos suposiciones son válidas para el cluster de computación SOL.

La autenticación basada en claves no es obligatoria, aunque para trabajar con MPI se vuelve casi indispensable. Por ello se recomienda que si se va a trabajar con MPI se establezca la autenticación basada en claves a fin de evitar al usuario ser constantemente interrogado sobre su password cada vez que LAM/MPI use internamente `ssh`.

2.2.2. Trabajando con OpenMP

El usuario que desee trabajar con OpenMP tiene a su disposición dos compiladores: el compilador `icc` de Intel, y el compilador GNU `gcc`. En ambos compiladores podemos ver la versión actualmente instalada invocando al compilador con la única opción `-v`. Un ejemplo de compilación con ambos compiladores sería:

```
gcc -march=native -O2 --openmp programa.c -o programa_gcc
icc -m64 -O2 -openmp programa.c -o programa_icc
```

En el nodo5 (Pentium III), se debe omitir la opción `-m64` del compilador `icc`.

2.2.3. Trabajando con MPI

Para trabajar con MPI hay que utilizar la autenticación basada en claves, tal y como se detalló en la sección 2.2.1. Al entrar en el nodo SOL, arrancamos el demonio de LAM/MPI:

```
lamboot
```

Al arrancar este demonio se mostrará la versión de LAM/MPI. El trabajo con MPI se basa en dos pasos: compilar con el comando `mpicc`, y lanzar la ejecución con `mpirun`. El primer comando invoca al compilador `gcc`, y el segundo usa la red ethernet para ejecutar el programa compilado en varios nodos. Es posible ver el comando de compilación que invoca `mpirun` usando el parámetro `-showme`. De esta forma se puede comprobar que se va a compilar de la forma correcta. Un ejemplo sencillo de trabajo con LAM/MPI es el siguiente:

```
mpicc -march=native -O2 -lblas programa.c -o programa
mpirun n0-2 programa argumento1 argumento2
```

En este ejemplo se compila un programa que usa las bibliotecas de BLAS, y luego se ejecuta en los tres primeros nodos, es decir, los Intel Xeon de cuatro cores, pasando al programa un par de parámetros. Nótese que el nodo `n0` corresponde al nodo SOL. Una vez concluido el trabajo con LAM/MPI, cerraremos el demonio con:

```
lamhalt
```

2.2.4. Trabajando con BLAS

La forma de trabajar con BLAS en el cluster se reduce a tres pasos: seleccionar la implementación de BLAS, compilar el código fuente y ejecutar el programa generado.

Para seleccionar la implementación podemos usar la variable de entorno `LD_LIBRARY_PATH`. Esta variable contendrá la ruta a las bibliotecas que tienen prioridad a la hora de compilar y ejecutar. Las distintas implementaciones de BLAS se encuentran separadas en el directorio `/usr/lib64/blas/`. Este directorio contiene un subdirectorio por cada implementación, de forma que tenemos las siguientes rutas absolutas:

```
/usr/lib64/blas/goto
/usr/lib64/blas/reference
/usr/lib64/blas/atlas
/usr/lib64/blas/threaded-atlas
```

Con el comando `export` exportamos una variable con el valor asignado. La forma de utilizar este comando para seleccionar, por ejemplo, la implementación ATLAS de BLAS es la siguiente:

```
export LD_LIBRARY_PATH=/usr/lib64/blas/atlas
```

Una vez exportada esta variable de entorno, compilamos de forma habitual el programa y lo ejecutamos. Un ejemplo de la secuencia completa de comandos para trabajar con BLAS Goto es la siguiente:

```
export LD_LIBRARY_PATH=/usr/lib64/blas/goto
gcc -march=native -O2 -lblas programa.c -o programa
./programa
```

En este ejemplo se dan los tres pasos. Si en cualquier momento queremos comprobar con qué implementación de BLAS se va a ejecutar un programa, podemos usar el comando `ldd` con ese programa como argumento. Este comando muestra las bibliotecas a las que está vinculado el programa. Por ejemplo:

```
ldd programa | grep libblas
```

Cuando trabajamos con el nodo con Pentium III, al ser de 32 bits, la forma de trabajar es idéntica, pero cambiamos la ruta de los directorios de `/usr/lib64/blas/` a `/usr/lib/blas/`.

2.2.5. Trabajando con LAPACK

La forma de trabajar con LAPACK es muy parecida a la de BLAS. Puesto que LAPACK usa BLAS, configuraremos BLAS y LAPACK en los mismos pasos. En el cluster SOL disponemos de las implementaciones de referencia (Reference) y ATLAS para LAPACK. Para configurar BLAS y LAPACK de referencia una secuencia típica es la siguiente:

```
export LD_LIBRARY_PATH=/usr/lib64/blas/reference:/usr/lib64/lapack/
    reference
gcc -march=native -O2 -llapack -lblas programa.c -o programa
./programa
```

En el caso de seleccionar la implementación de ATLAS, la primera línea sería:

```
export LD_LIBRARY_PATH=/usr/lib64/blas/atlas:/usr/lib64/lapack/
    atlas
```

Para comprobar las bibliotecas a las que está vinculado un programa, usaremos el comando `ldd` de la siguiente forma:

```
ldd programa | egrep "(blas|lapack)"
```

Al trabajar con el nodo de 32 bits, cambiaremos todas las rutas de `/usr/lib64` a `/usr/lib`.

2.2.6. El sistema gestor de colas

En caso de que hayan varios usuarios que necesiten trabajar con la potencia computacional del cluster SOL en exclusiva (por ejemplo, para mediciones

de tiempo), se puede hacer uso del sistema gestor de colas Torque [13]. Con el sistema gestor de colas los usuarios envían trabajos al gestor, indicando el tiempo que quieren reservar para su computación, y el sistema introduce los trabajos en una cola. La cola de trabajos se ejecuta según un planificador de colas. En el caso del cluster SOL, el planificador instalado es Maui [14].

Un ejemplo sencillo de envío de trabajos sería el siguiente:

```
qsub -l vmem=512MB,cput=02:30:00 script
```

Donde además de indicar que se van a usar dos horas y media, se reserva medio giga de memoria RAM. El *script* al final del comando es un *script* en `bash`. En este ejemplo se usa el comando `qsub` para enviar trabajos a la cola por defecto. Actualmente se usa una única cola para todos los trabajos, por lo que no es necesario indicar el nombre de la cola de ejecución.

En lugar de usar parámetros, podemos indicar en los comentarios del *script* la reserva de recursos que vamos a hacer, por ejemplo:

```
#!/bin/bash
#PBS -q secuen
#PBS -l vmem=512MB,cput=02:30:00
./comando
```

De esta forma lanzamos la ejecución de los comandos simplificadamente:

```
qsub script
```

Como hemos visto, el comando `qsub` permite enviar trabajos al sistema gestor de colas; otros comandos de Torque que pueden ser de interés son los siguientes:

qdel: Elimina un trabajo enviado del sistema gestor de colas

qalter: Cambia los parámetros de un trabajo enviado, tales como el tiempo o la memoria

qnodes: Muestra información de los nodos que están en el sistema gestor de colas

qstat: Muestra información de los trabajos enviados por el usuario

Para una descripción detallada de cada comando, se remite a las respectivas páginas `man` y al manual de Torque [15].

2.2.7. Utilidades de interés

En SOL, además de los comandos habituales de Linux, se han instalado herramientas que facilitan el trabajo:

screen: Arranca un *shell* y permite desvincular la salida con la combinación `CTRL+A-D`[2]. Es útil cuando un programa tarda horas en ejecutarse y envía resultados por la pantalla. Al conectarse por `ssh`, es posible salir de la sesión mientra se ejecuta el comando, conectarse más tarde y recuperar la salida en pantalla con `screen -r` [16].

rsync: Permite sincronizar una copia de trabajo en el nodo SOL y en el ordenador del usuario [17].

cssh: Al conectarse por `ssh` con redirección X11, este comando abre ventanas `xterm`. Puede usarse para manejar simultáneamente los distintos nodos del cluster [18].

cexec: Lanza comandos en varios nodos del cluster y muestra la salida de todos ellos. Es útil para lanzar comandos cuya salida sea unas pocas líneas [19].

gnuplot: Herramienta de línea de comandos que permite generar gráficos en distintos formatos, incluido pdf, a partir de datos en ficheros [20].

Para una descripción detallada de cada comando, se remite a las respectivas páginas `man`.

[2]CTRL+A simultáneamente, se suelta y se pulsa D

Capítulo 3

Herramientas para el estudio de prestaciones en clusters

En este capítulo explicaremos el diseño, la estructura y el funcionamiento, desde el punto de vista del usuario, de las herramientas desarrolladas para el estudio de prestaciones en clusters de computación científica. La aplicación de estas herramientas en el cluster SOL y la interpretación de los resultados que con ella se generan constituirán los capítulos siguientes.

3.1. Diseño de las herramientas

El conjunto de herramientas está formado por el código fuente de programas escritos en *C* y *scripts* en *shell bash*. Los *scripts* siguen un esquema de nombrado fijo, lo que facilita la curva de aprendizaje del conjunto de herramientas[1]. Este esquema de nombrado, como veremos, está relacionado con el programa que se va a ejecutar. Las herramientas se proporcionan en un paquete `tar.bz2` que presenta la siguiente jerarquía:

- **OpenMP**
  - **primitivas**
    - **generate**: Primitivas de generación de hilos
    - **pfor**: Primitiva para paralelizar bucles `for`
    - **sections**: Primitiva `sections` y `section`

[1]Nos referimos a una herramienta como un *script* relacionado con un programa en C particular. Por conjunto de herramientas o herramientas (en plural) nos referimos a varias herramientas relacionadas con un software de computación científica o más frecuentemente a todos los *scripts* y ficheros que constituyen este proyecto.

        - **barriers**: Primitiva `barrier`
    - **alto_nivel**
        - **matrizvector**: Multiplicación matriz por vector
        - **matrizmatriz**: Multiplicación matrices almacenadas por filas
        - **matrizmatriz2**: Multiplicación de matrices $A \times B$, con B almacenada por columnas
        - **jacobi**: Relajación de Jacobi
- **MPI**
    - **primitivas**: Primitivas de MPI
    - **multmatriz**: Multiplicación de matrices
- **Blas**
    - **directa**: Multiplicación directa de matrices (sin BLAS)
    - **bloques**: Multiplicación de matrices por bloques (sin BLAS)
    - **mb1**: Multiplicación de matrices llamando a BLAS de nivel 1
    - **mb2**: Multiplicación de matrices llamando a BLAS de nivel 2
    - **mb3**: Multiplicación de matrices llamando a BLAS de nivel 3
- **Lapack**
    - **lu**: Factorización LU

Esta jerarquía se refleja en la estructura de los directorios. En los directorios de mayor profundidad cada nombre tiene asociado un código fuente y un *script*. Vamos a explicar esto con el ejemplo de *generate* (en OpenMP, primitivas). Dentro del directorio `generate`, se encuentra el programa `generate.c` y el *script* `generate.sh`. Si ejecutamos el *script* `generate.sh`, se compilará el fuente `generate.c`. En este ejemplo particular se crearán varios ejecutables compilados con `gcc` e `icc` y usando distintas opciones de compilación. En otros casos se crearán distintos ejecutables según otros criterios. Tras este paso se ejecutarán todos los binarios generados, y se repetirá la ejecución varias veces. De la ejecución repetida se saca una media de tiempo, se escribe en una línea en un fichero de resultados y se cambian los parámetros del programa para volver a ejecutar el programa. Se sigue este ciclo hasta que el *script* ha realizado todas las pruebas y finaliza. El resultado del *script* se guarda ficheros con la extensión `.resultados`.

Estas herramientas también contienen *scripts* que generan documentos `.pdf` con gráficos de las mediciones. A fin de clarificar el funcionamiento de las herramientas, vamos a recurrir a explicarlo con otro ejemplo. Tomamos en este caso la multiplicación de matrices con BLAS de nivel 3. El usuario comienza entrando en el directorio `Blas/mb3` y ejecutando el *script* `mb3.sh`. El estudio de la multiplicación de matrices con BLAS de nivel 3 intenta determinar qué implementación es la más eficiente. Por ello compilará el programa `mb3.c` por cada implementación disponible, creando cuatro binarios: `mb3_goto`, `mb3_reference`, `mb3_atlas` y `mb3_threaded-atlas`. El *script* ejecutará los binarios con cierto tamaño de matriz y repetirá la ejecución varias veces para tomar medias de tiempo. Posteriormente cambiará los parámetros del programa, en este caso cambia el tamaño de la matriz cuadrada, y repetirá la ejecución. Finalmente los resultados de las medias estarán en los ficheros `mb3_goto.resultados`, `mb3_reference.resultados`, `mb3_atlas.resultados` y `mb3_threaded-atlas.resultados`. El formato de cada uno de estos ficheros sigue un patrón determinado, donde cada media se escribe con sus parámetros en una línea. En particular para este ejemplo el formato de una línea es:

```
1 [nombre_programa] N=<tamaño>, S=<segundos>
```

Es decir, se muestra siempre el nombre del programa entre corchetes, seguido por los parámetros del programa, en este caso `N` (tamaño de la matriz cuadrada) y `S` (segundos). En otros programas los parámetros varían ligeramente, por ejemplo en OpenMP se añade el parámetro `P` para indicar el número de hilos.

En la figura 3.1 se ilustra el ejemplo del programa `mb3`. Una vez que las pruebas se han llevado a cabo y el *script* ha generado los ficheros de resultados, el usuario puede ejecutar el *script* `grafico.sh`. En cada directorio donde se realizan las pruebas hay un *script* para generar gráficos, siempre con el nombre `grafico.sh`. Este *script* tratará los datos de los ficheros de resultados y creará ficheros para `gnuplot`. Posteriormente invocará a `gnuplot` para que cree los ficheros `.pdf` y borrará los ficheros intermedios. En este ejemplo un fichero de resultados contiene datos de una implementación particular de BLAS, y dentro del fichero se encuentran mediciones para tamaños de matriz cuadrada de $N = 1000$, 2000 y 3000. El *script* `grafico.sh` buscará en cada fichero cada tamaño y mezclará los resultados para generar un gráfico comparativo. Dicho de otra manera: aunque los ficheros de resultados agrupen los datos con un criterio (la implementación), al generar los gráficos en `pdf` el *script* cambia el criterio (el tamaño de matriz) de forma que se obtenga un gráfico del que se puedan extraer conclusiones para el estudio de prestaciones.

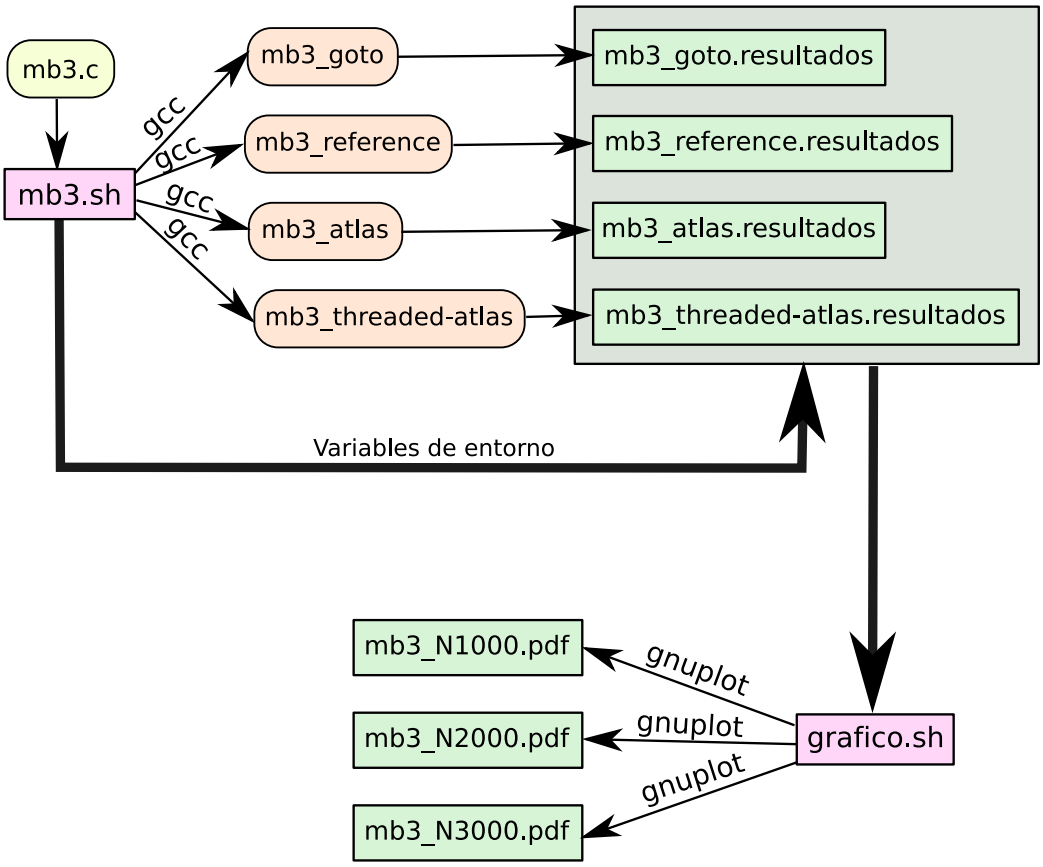

Figura 3.1: Ejemplo de funcionamiento de la herramienta para multiplicación de matrices con BLAS 3

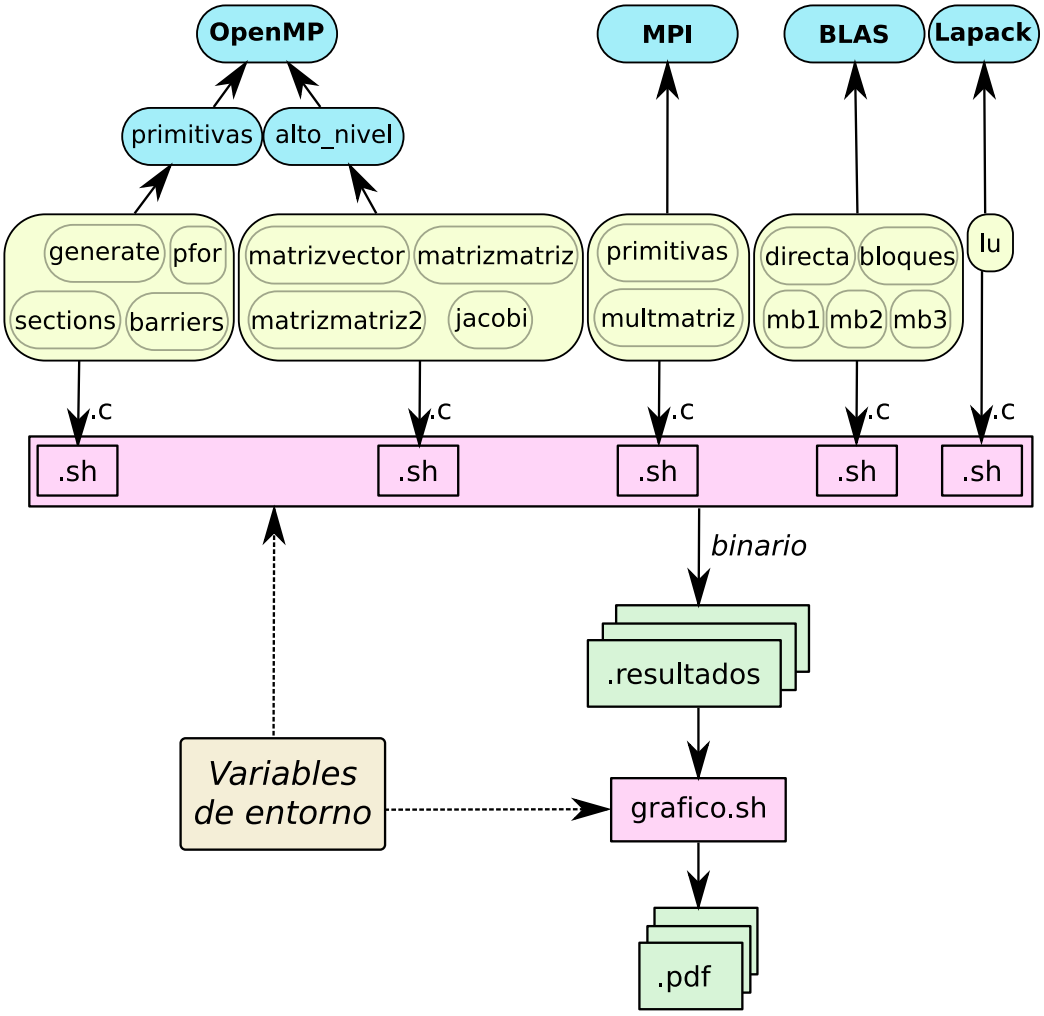

Figura 3.2: Diagrama de estructura y funcionamiento de las herramientas para el estudio de prestaciones en clusters

Los *scripts* no necesitan ser configurados puesto que toman los valores de configuración por defecto que son adecuados en el cluster SOL, y presumiblemente adecuados en otros clusters de computación actuales. No obstante es posible cambiar la configuración mediante variables de entorno. La ventaja de cambiar la configuración mediante variables de entorno es que podemos establecer unas variables comunes para todas las pruebas en un cluster determinado, y de esa forma no hay necesidad de pasar como parámetro al *script* la configuración de forma repetitiva para cada prueba. La figura 3.2 esquematiza el funcionamiento de la herramienta para todas las pruebas. Como se ve en esa figura (que es una generalización de la figura 3.1 anteriormente explicada) ambos *scripts* toman las variables de entorno para configurarse. En el cluster SOL, no es necesario usar ninguna variable de entorno.

3.2. Ejecución de las herramientas

A continuación enumeraremos cada uno de los *scripts*, indicando entre otros detalles su configuración, los ficheros que genera y la función que realiza.

3.2.1. Herramientas para el estudio de OpenMP

El objetivo de esta herramienta es evaluar las prestaciones de OpenMP con los compiladores `gcc` e `icc`, así como determinar el conjunto de optimizaciones adecuado para trabajar con OpenMP. El estudio realizado se divide en una primera parte sobre el rendimiento de las primitivas de OpenMP, y una segunda parte que utiliza las primitivas en rutinas de alto nivel.

Las variables de entorno que podemos configurar son:

`N`: Tamaños del problema separado por un espacio. Para las primitivas toma el valor de `"1 50 100 1000 5000"`, para la multiplicación matriz por vector el valor de `"2000 4000 8000"`, para las multiplicaciones de matrices `"500 1000 1500"` y para `jacobi "1000 2000 4000"`.

`TMP_DIR`: Directorio temporal, por defecto `/dev/shm`

`IT`: Iteraciones: número de veces que se repite cada prueba individual antes de obtener la media, por defecto 10

`P_MAX`: Número de threads máximo. Por defecto 10 (ejecutará desde uno a diez hilos)

`LOG_SCALE`: Sólo si toma el valor 1 genera los gráficos con la escala de tiempo logarítmica. Por defecto la variable no está definida.

Los *scripts* se encuentran en el directorio `OpenMP/primitivas`, que contiene cuatro subdirectorios: `generate`, `pfor`, `sections` y `barriers`. Cada uno de estos directorios corresponde al estudio de una primitiva, y en cada directorio se encuentra un *script* con el mismo nombre y extensión `.sh`. Este *script* es el *script* principal y es el primero que ha de ejecutarse. Adicionalmente en cada directorio se encuentran otros *scripts* que han de ejecutase ordenadamente y que realizan el siguiente trabajo:

`grafico.sh`: Crea los gráficos en `pdf` de todas las pruebas. Acepta la variable `LOG_SCALE`.

`cflag.sh`: Calcula la media de los distintos tamaños de problema y lo escribe en ficheros de resultados. Esto permitirá posteriormente hacer el estudio de los flags de optimizaciones en C.

`grafico_compilador.sh`: Toma los ficheros de resultados de medias obtenidos con `cflags.sh` y crea gráficos comparativos en `pdf` que muestran el rendimiento con distintos flags de optimizaciones.

`latex.sh`: Genera ficheros `.tex` con fragmentos LaTeX con los resultados de las mediciones en una tabla. Es útil para escribir documentación donde tengan que aparecer todas las experimentaciones.

En el directorio `OpenMP/alto_nivel` se encuentran los siguientes subdirectorios: `matrizvector`, `matrizmatriz`, `matrizmatriz2` y `jacobi`. Al igual que con las primitivas, el *script* principal tiene el mismo nombre que el directorio y extensión `.sh`. Los *scripts* descritos para las primitivas también existen en el estudio de alto nivel y tienen el mismo propósito. Además de estos *scripts* se añaden los siguientes:

`mflops.sh`: Calcula los megaflops teóricos del programa. Recibe como parámetro el nombre de un fichero de resultados. La formula de la complejidad está codificada en el *script*.

`speedup.sh`: Genera ficheros `.tex` con fragmentos LaTeX para crear una tabla con el *speedup*, la *eficiencia* y el *coste* de una medición. Se pasa un fichero de resultados como parámetro

En los directorio de `OpenMP/primitivas` y `OpenMP/alto_nivel` se encuentran otros dos *scripts* adicionales llamados `todos_graficos.sh` que sirven para generar todos los gráficos de todas las pruebas, tanto con escala logarítmica como sin ella. Téngase en cuenta que este *script* invoca a los *scripts* de generación de gráficos que hay en cada subdirectorio, por lo que

para funcionar correctamente es necesario que se hayan ejecutado todos los *scripts* principales y el *scripts* `cflags.sh` en todos los subdirectorios. Este *script* es útil cuando se ha repetido las pruebas y el usuario quiere tener la certeza de que todos los gráficos estén actualizados.

En el caso de que se quiera inspeccionar los ficheros de resultados, vamos a ver un ejemplo de extracto del fichero `jacobi_gcc-O2.resultados`:

```
[jacobi_gcc-O2] P=1, N=2000, S=0.3298004
[jacobi_gcc-O2] P=2, N=2000, S=0.1722254
```

El formato de cada línea es el siguiente: el nombre del programa entre corchetes, seguido de `P` que indica el número de threads, `N` el tamaño del problema y `S` los segundos de ejecución de las pruebas. Este tiempo es la media de las 10 pruebas que se realizan por defecto (configurable con la variable de entorno `IT`).

3.2.2. Herramientas para el estudio de MPI

El objetivo de la herramienta de MPI es evaluar las prestaciones de distintas primitivas de paso de mensajes y evaluar las prestaciones de MPI en computación de alto nivel, reflejado en la multiplicación de matrices. Uno de los aspectos más relevantes de la experimentación con MPI es que vamos a trabajar con plataformas heterogéneas tanto en arquitectura (32 y 64 bits) como en velocidad. En el capítulo 5 se tratará la problemática de mezclar arquitecturas heterogéneas y cómo se resuelve a través de esquemas[2]. En los *scripts* desarrollados, los esquemas se generan automáticamente, por lo que tan sólo tendremos que configurar apropiadamente el *script*. Para que los *scripts* de medición ofrezcan resultados fiables es necesario que la fecha y hora de todos los nodos del cluster estén sincronizadas. Antes de lanzar los *scripts* es necesario arrancar `lamboot`.

Los *scripts* de medición se encuentran en los directorios `MPI/primitivas` y `MPI/multmatriz`. Ambos *scripts* siguen la estructura ya explicada: en cada uno existe un *script* `.sh` con el nombre del directorio. De esta forma, los *scripts* de medición son `primitivas.sh` y `multmatriz.sh`. Para el correcto funcionamiento de estos *scripts* es necesario escribir un fichero `.conf` (`primitivas.conf` o `multmatriz.conf`) con lineas en el siguiente formato:

```
nX nombre b
```

donde `nX` es en número de nodo, empezando por cero, nombre es el nombre del nodo (en DNS o en `/etc/hosts`) y `b` es `32` ó `64` dependiendo de si la

[2]Los esquemas son ficheros que especifican los programas que se van a lanzar en cada nodo, permitiendo lanzar programas a medida para cada arquitectura.

arquitectura es de 32 o de 64 bits.

Por defecto se suministran los ficheros de configuración de SOL, que tienen el siguiente contenido:

```
n0 sol 64
n1 nodo1 64
n2 nodo2 64
n3 nodo3 64
n4 nodo4 64
n5 nodo5 32
```

Además se permiten líneas en blanco y líneas que comienzen por almohadilla (#) para indicar comentarios. Una vez que esté listo este fichero, se pueden lanzar los *scripts* de medición. Estos *scripts* usarán la configuración suministrada para compilar el código fuente en C en las máquinas nativas (conectándose por `ssh`), crear los esquemas apropiados, lanzar los programas con `mpirun` (un proceso por cada nodo) y operar con la salida para obtener resultados medios.

Las variables de entorno de los *scripts* de experimentación coinciden en nombre y significado con las de OpenMP: `TMP_DIR` para el directorio temporal e `IT` para el número de repeticiones antes de obtener la media. El valor por defecto de `IT` es 10 para las primitivas y 5 para la multiplicación de matrices. También es posible cambiar los tamaños de problema con la variable de entorno N, que para las primitivas toma el valor por defecto de `"10000000 20000000 40000000"` y para la multiplicación de matrices el valor de `"1000 2000 3000"`. Para generar los ficheros `.pdf` se usará el *script* `grafico.sh` en el directorio correspondiente. Adicionalmente se encuentran otros *scripts* que ayudan a calcular los megaflops de la multiplicación de matrices, que calculan el *speedup*, la eficiencia y el coste, y que generan las tablas con las mediciones para documentación en LaTeX.

El formato de los ficheros de resultados muestra un aspecto similar al de OpenMP. Veamos este ejemplo:

```
[multmatriz] P=1, N=3000, S=369.5808868
[multmatriz] P=2, N=3000, S=184.8165944
```

donde se muestra el tamaño del problema en `N` y los segundos en `S`. El parámetro `P` indican el número de nodos que han intervenido en la ejecución de las pruebas. Este valor coincide con el orden del fichero `.conf`, por lo que, retomando el ejemplo anterior del cluster SOL, el 1 significa que sólo ha intervenido SOL, el 2 que ha intervenido SOL y el nodo1, 3 que ha intervenido SOL, el nodo1 y el nodo2, y así hasta 6, que indica que han intervenido todos los nodos, incluyendo el nodo de 32 bits.

3.2.3. Herramientas para el estudio de BLAS

Las herramientas para el estudio de BLAS pretenden averiguar cuál de las implementaciones disponibles es la de mayor rendimiento en un cluster particular. Todas las experimentaciones con BLAS se harán sobre la multiplicación de matrices. En primer lugar se hará la multiplicación directa sin BLAS, la multiplicación por bloques sin BLAS y la multiplicación llamando a las rutinas de nivel 1, 2 y 3 de BLAS.

El conjunto de herramientas de BLAS se encuentra en el directorio BLAS y contiene cinco subdirectorios: `directa`, `bloques`, `mb1`, `mb2` y `mb3`. Cada uno de estos directorios contiene un fichero con el mismo nombre y extensión `.sh`. También encontramos un fichero `grafico.sh` para generar los gráficos, así como los ya conocidos ficheros `mflops.sh` y `latex.sh` en los directorios correspondientes.

Las variables de entorno `TMP_DIR`, `IT` y `N` tienen el mismo significado y uso explicado en apartados anteriores. En este caso `IT` toma el valor 5 por defecto y N el valor de `"1000 2000 3000"`. A estos *scripts* se añade la variable de entorno `LIB_PATH` que toma el valor por defecto `/usr/lib64`. Esta variable sirve para encontrar las distintas implementaciones de BLAS. Se asume que en esta ruta se encuentra un directorio `blas` y un subdirectorio con el nombre de cada implementación. En el caso de SOL las rutas encontradas son:

```
1 $LIB_PATH/blas/goto
2 $LIB_PATH/blas/reference
3 $LIB_PATH/blas/atlas
4 $LIB_PATH/blas/threaded-atlas
```

En el caso de que la herramienta se utilizase en un cluster que no siga esta estructura de directorios, es posible reproducirla manualmente, bien copiando bibliotecas, o bien creando los directorios y haciendo que los subdirectorios sean enlaces simbólicos.

En la multiplicación por bloques, para encontrar el tamaño óptimo de bloque, se utiliza un *script* llamado `bloques_busca_tam.sh`. Análogamente para generar los gráficos de este *script* se utiliza el *script* `grafico_busca_tam.sh`. Puesto que los distintos *scripts* `grafico.sh` de la multiplicación con BLAS (mb1, mb2 y mb3) comparan los resultados con la multiplicación directa y sin bloques, es imprescindible hacer las pruebas de la multiplicación directa y por bloques antes que las que trabajan con BLAS. Tras averiguar el tamaño óptimo de bloque, puede establecerse en la variable de entorno `BLOQUE` que será usada para las mediciones de la multiplicación directa por bloques (*script* `bloques.sh`). El tamaño de bloque por defecto es de `50`.

Una vez finalizadas todas las pruebas, se puede ejecutar el *script* `grafico.sh` del directorio de BLAS. Este *script* tomará todos los resulta-

dos de todas las pruebas y generará un fichero `Blas.pdf` que muestra una comparativa de todas las pruebas. Este gráfico permite resumir toda la experimentación con BLAS y da una visión de conjunto que facilita la extracción de conclusiones.

El formato de los ficheros de resultados es sencillo, puesto que al tratarse estas pruebas de computación secuencial, sólo se encuentran los parámetros `N` (tamaño de matriz cuadrada), `S` (segundos), y `B` (tamaño del bloque, sólo para la multiplicación directa por bloques) por lo que no hay problema en su interpretación.

3.2.4. Herramientas para el estudio de LAPACK

El objetivo de la herramienta de LAPACK es estudiar el comportamiento de distintas implementaciones. Puesto que LAPACK usa el núcleo computacional de BLAS-3, se espera que los resultados sean coherentes con los obtenidos en la experimentación con BLAS. La experimentación con LAPACK se basa en el algoritmo de factorización LU.

Los *scripts* se encuentran en el directorio `Lapack/lu`. Encontramos `lu.sh` y `gráfico.sh`. La configuración es la ya conocida a través de variables de entorno. Cada prueba se repite cinco veces antes de obtener una media (salvo que se cambie la variable `IT`). Al igual que con BLAS, se utiliza la variable `LIB_PATH` para indicar las rutas a las bibliotecas de LAPACK. En el caso de SOL las rutas son:

```
1 $LIB_PATH/lapack/reference
2 $LIB_PATH/lapack/atlas
```

Donde el valor por defecto de `LIB_PATH` es `/usr/lib64`. Si se utilizara esta herramienta en un cluster que no siga las rutas por defecto, puede reproducirse dicha estructura igual que se hizo con BLAS. Tras la ejecución del *script* `lu.sh` se pueden encontrar los ficheros de resultados, que tendrán como nombre de fichero: `lu_*-dgetf2.resultados`, `lu_*-dgetrf.resultados`, siendo el asterisco cada una de las implementaciones encontradas (en el caso de SOL será ATLAS y Reference). El *script* de gráficos generará `.pdf` comparativos de cada implementación, agrupadas por llamada (`dgetf2` / `dgetrf`), y un gráfico `lu.pdf` que compara todas las implementaciones con las dos llamadas. También podemos encontrar en el conjunto de herramientas de LAPACK el correspondiente *script* `latex.sh`.

3.2.5. Ampliación de la herramienta

El conjunto de herramientas puede ser fácilmente ampliado con nuevas herramientas particulares. Esto permite medir las prestaciones de nuevos algoritmos o de otro software de computación científica.

En el caso de añadir un nuevo algoritmo, basta con añadir un subdirectorio en el directorio apropiado. Por ejemplo, si queremos añadir un nuevo algoritmo de BLAS, crearemos un subdirectorio bajo el directorio `Blas` con el nombre del algoritmo. El programa en C tendrá el mismo nombre que el directorio y extensión `.c`. El *script* de la mediciones también tendrá el nombre del directorio con extensión `.sh`. Los distintos *scripts* pueden copiarse de los ya existentes en la herramienta y adaptarlos ligeramente. La adaptación es sencilla: puesto que los *scripts* usan el nombre del directorio para averiguar el programa a compilar, las modificaciones son pocas: normalmente cambiar los parámetros con los que se invoca al programa compilado y otros retoques sencillos.

En el caso de añadir herramientas para medir las prestaciones de software científico, estas se crearán a partir de un directorio nuevo ubicado en el directorio raíz del conjunto de herramientas. Cada subdirectorio corresponderá a un algoritmo. Siguiendo el diseño de las herramientas creadas, la extensión de nuevo software de computación científica debe ser sencillo, ya que se dispone del código de los distintos *scripts* como guía.

Capítulo 4

Prestaciones de OpenMP

OpenMP es un estándar para la programación en memoria compartida. Se trata de un API (Application Program Interface) que, basándose en directivas de compilación, permite al programador indicar explícitamente las partes paralelizables en un programa secuencial. El compilador que soporte el estándar OpenMP creará hilos en las secciones paralelizables indicadas por el programador siguiendo el modelo *fork-join*.

Puesto que OpenMP es el primer grupo de mediciones llevados a cabo en SOL, servirá para determinar el conjunto de optimizaciones adecuado en las mediciones de los siguientes capítulos. De igual manera, sentará las bases sobre la metodología de medición seguida en el resto del documento.

4.1. Prestaciones de las primitivas

Con el estudio de las primitivas de OpenMP pretendemos averiguar qué compilador ofrece el mejor rendimiento para las operaciones básicas de manejo de hilos. No se trata de hacer complejos cálculos, sino más bien de lo contrario: cálculos triviales que expongan con claridad el tiempo invertido principalmente en manejar los hilos, sin ninguna otra sobrecarga.

Otro aspecto que se quiere determinar con este estudio es el comportamiento de los programas de los distintos compiladores cuando el número de hilos supera al número de cores del procesador[1]. En las mediciones de rendimiento trabajaremos con el rango que cubre desde un único hilo hasta varios hilos por core: en particular, para un nodo con cuatro cores, lanzaremos has-

[1]Estos casos son interesantes cuando ejecutamos aplicaciones que fueron optimizadas para más cores de los que podemos proporcionar, o bien cuando, aún habiendo diseñado la aplicación para el ordenador en el que se ejecuta, hay otros procesos compitiendo por los cores.

ta diez hilos, de forma que podamos ver la tendencia que siguen los tiempos de ejecución.

Para realizar estas pruebas utilizaremos dos compiladores de C que soportan las directivas de OpenMP: el compilador `gcc 4.2.3` [21], con licencia GNU, y el compilador de Intel `icc 10.1.012` [22] con licencia propietaria[2]. Ejecutaremos las pruebas en el nodo SOL del cluster, que tiene cuatro cores, y arquitectura de 64 bits. Para que las comparaciones de ambos compiladores sean en igualdad de condiciones, compilaremos todos los programas con opciones equivalentes. La elección de las opciones de compilación no es fácil: no podemos establecer unas buenas optimizaciones para todos los programas posibles, puesto que cada programa funcionará mejor con unas u otras optimizaciones. No obstante, sí sabemos que ciertas optimizaciones no alteran o mejoran el tiempo de ejecución, tales como aquellas que permiten borrar instrucciones innecesarias o simplificar el código ensamblador usando menos registros. También sabemos que optimizando para el procesador particular mejoramos el tiempo de ejecución, y compilando para la arquitectura del cluster, aprovechamos mejor el potencial de los procesadores. Por estos motivos compilaremos siempre para arquitecturas de 64 bits, y para el procesador específico del nodo SOL. Queda determinar el conjunto de optimizaciones a aplicar.

Para determinar las mejores optimizaciones haremos pruebas con tres conjuntos de optimizaciones: `O2`, `O3` y `Os`. Las optimizaciones `O2` en `gcc` incluyen a las optimizaciones de `O1`[3].

Los conjuntos de optimizaciones `O1`, `O2` y `O3` se comportan de manera incremental; es decir: el conjunto `O1` añade varias optimizaciones, el `O2` añade las de `O1` más algunas propias, y `O3` añade sus propias optimizaciones más las de `O2` y en consecuencia también las de `O1`. Sin embargo `Os` consiste en las optimizaciones de `O2` desactivando algunas que aumentan el tamaño del ejecutable [23]. El enfoque de `Os` se basa en suponer que el procesador hace un buen trabajo optimizando la ejecución de las instrucciones. En este caso la mejora por parte del compilador se centra en ejecutables pequeños, con menos instrucciones, que hagan un mejor uso de las cachés de los procesadores. Los tres conjuntos de optimizaciones tienen en cuenta el paralelismo hardware que existe tradicionalmente en los monoprocesadores, tales como la segmentación

[2]Se ha usado una versión de evaluación no comercial, con licencia de un año, disponible en `http://www.intel.com/cd/software/products/asmo-na/eng/219690.htm`

[3]Una particularidad de `O1` es que sólo añade la optimización `-fomit-frame-pointer` cuando no dificulta la depuración. Esta optimización simplifica las instrucciones de entrada y salida de un procedimiento. En las pruebas, siempre añadiremos esta optimización explícitamente, puesto que nuestro objetivo es medir el rendimiento, sin importarnos la facilidad para la depuración.

(*pipelining*) y réplica de unidades funcionales.

Para indicar el tipo del procesador, en `gcc` se utiliza la opción `-march=cpu`, en este caso para el procesador de SOL lo adecuado sería `-march=nocona`, correspondiente a Intel Xeon. No obstante, en la versión `gcc` con la que trabajamos el propio compilador es capaz de detectar las características del procesador y ajustar esta opción. Para activar esta detección automática pondremos la opción `-march=native`, que indica que se ha de compilar optimizado para la máquina actual. Lógicamente esta opción impide la compilación cruzada (no debemos compilar en otro ordenador y luego copiar el ejecutable a SOL). Estas características de detección del procesador son imprescindibles para que el paquete de medida de prestaciones sea portable a otros clusters de computación paralela. En los dos compiladores hay que indicar que se va a usar OpenMP con las opciones `--openmp` o `-fopenmp` de `gcc` y `-openmp` de `icc`.

Respecto a la metodología de las pruebas, todas se realizan diez veces, se examinan las mediciones para comprobar que ninguna muestra un resultado notablemente alejado de la media, en cuyo caso se repite la prueba. Una vez realizadas las diez mediciones válidas, se obtiene la media. Las diez mediciones no son realmente necesarias: con menos podríamos haber tenido resultados casi idénticos, no obstante, puesto que las pruebas con primitivas tardan poco tiempo, se prefiere que hayan mediciones de más para asegurar resultados fiables. Con los resultados obtenidos elaboramos gráficos que nos permiten determinar con claridad cuál es el compilador y el conjunto de optimizaciones que genera mejor código para trabajar con OpenMP, o bien, determinar cuál es el mejor compilador según los parámetros estudiados de tamaño de entrada y número de hilos.

4.1.1. Rutina generate

Esta rutina consiste en generar un conjunto de hilos con poco trabajo con el objetivo de medir el tiempo que se invierte en crear y manejar hilos. Cada hilo hará un bucle de comparaciones. El tamaño del problema N será el número de iteraciones de dicho bucle[4]. Haremos mediciones para distintos tamaños de bucle: 1, 50, 100, 1000 y 5000. El tamaño de 1 representa una pasada por el bucle, es decir, el tamaño mínimo posible. El esquema de la rutina `generate` se muestra[5] en la figura 4.1.

[4]Los valores de `N` son configurables en la herramienta desarrollada, lo que permite que sea fácil probar con nuevos tamaños sin cambiar código

[5]El código se ha simplificado, poniendo nombres de variables más sencillas. En la comparación `i==x` se ha introducido código adicional para evitar que el compilador elimine la comparación por medio de optimizaciones

```
1  tiempo_inicio = omp_get_wtime();
2  #pragma omp parallel private(actual,P)
3  {
4    P = omp_get_num_threads();
5    actual = omp_get_thread_num();
6    for(i=0; i<N; i++)
7    {
8        { i == x }
9    }
10 }
11 tiempo_fin = omp_get_wtime();
```

Figura 4.1: Código simplificado de la rutina `generate` escrita en C

La metodología del estudio experimental se esquematiza en los siguientes pasos:

- Repetir para cada conjunto de optimizaciones `O2`, `O3` y `Os`:
  - Compilar con `gcc`
  - Compilar con `icc`
    - Repetir para cada tamaño del problema $N = 1, 50, 100, 1000, 5000$, y para cada número de hilos P desde 1 hasta 10
      - ◇ Ejecutar 10 veces el programa `generate`
      - ◇ Calcular la media de tiempo para cada N
- Repetir para `gcc` e `icc`, y para cada conjunto de optimizaciones `O2`, `O3` y `Os`:
  - Calcular la media ponderada de tiempo para todos los tamaños de N

Estas pruebas lanzarán el programa un número elevado de veces, generando gran cantidad de datos, los cuales analizaremos a partir de la media de cada prueba individual[6]. En primer lugar determinamos qué opciones de compilación ofrecen mejores prestaciones. Como se ve en la figura 4.2, el re-

[6]En particular para esta prueba supone 3000 ejecuciones, de las cuales 300 son distintas, es decir, cada prueba individual se repite 10 veces, y tomamos la media, tal y como se ve en el esquema de la metodología de estudio experimental

sultado promedio[7] de todos los tamaños del problema revela que en `gcc` el conjunto de optimizaciones `O2` da el mejor tiempo de ejecución.

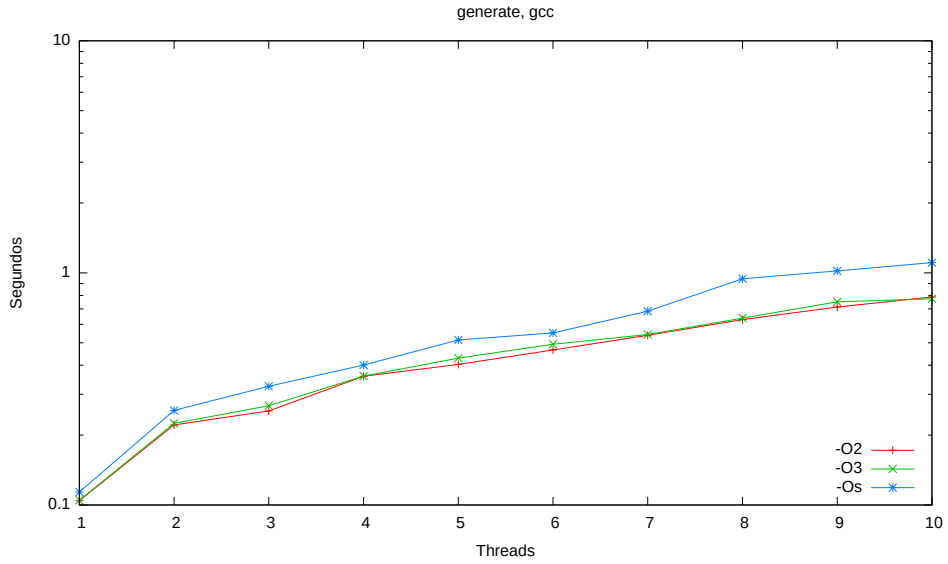

Figura 4.2: Comparación de optimizaciones de `gcc` para el programa `generate` variando el número de hilos. El tiempo se muestra en escala logarítmica

Este resultado es el que esperamos, puesto que el código que medimos es corto y sencillo, sin llamadas a funciones; por lo que no se dan las circunstancias donde los conjuntos de optimizaciones `Os` y `O3` pueden destacar.

Para el compilador `icc`, no hay ningúna optimización que destaque sobre las demás. En la figura 4.3 se ve cómo, cuando el número de hilos es menor o igual que el número de cores de SOL (es decir, cuatro), los resultados son muy similares y, cuando es mayor, se experimenta un aumento en el tiempo de ejecución en los tres conjuntos de optimizaciones. El efecto del conjunto de optimizaciones es claramente distinto para el compilador de GNU que para el de Intel, por lo que se deduce que cada uno lleva a cabo la generación de código de forma muy distinta.

Hemos estudiado cuál es el conjunto de optimizaciones más adecuado para cada compilador, tomando como resultados las medias de ejecutar diez veces cada prueba, para cada uno de los tamaños de problema 1, 50, 100, 1000 y

[7] La media de los distintos tamaños es una media ponderada: si representamos el tiempo en segundos del tamaño i como S_i, la media se calcula como: $\frac{S_1+50S_{50}+100S_{100}+1000S_{1000}+5000S_{5000}}{5}$

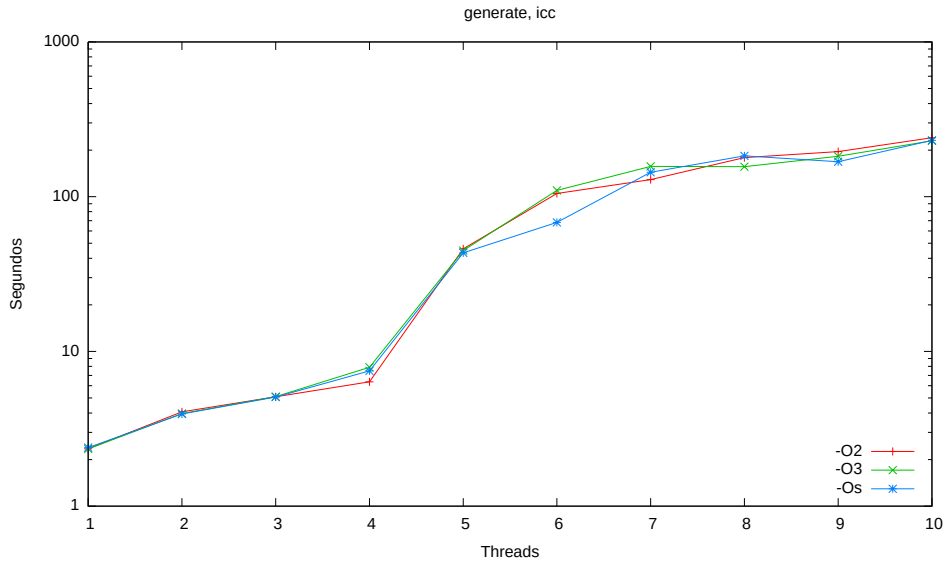

Figura 4.3: Comparación de optimizaciones de `icc` para el programa `generate` variando el número de hilos. El tiempo se muestra en escala logarítmica

5000 (figuras 4.2 y 4.3). Tenemos que averiguar de estas mediciones cuál es el mejor compilador para manejar hilos con OpenMP. Para ello comparamos ambos con los distintos tamaños del problema.

En la figura 4.4 puede verse la comparación de `gcc` e `icc` con `O2` cuando se realizan 5000 iteraciones del bucle principal. Se muestra únicamente este gráfico por ser representativo de la prueba: es decir, para los demás tamaños de problema y otras optimizaciones la gráfica presenta el mismo aspecto (en el anexo se pueden consultar todas las mediciones en detalle).

En dicha figura se ve claramente que `icc` genera un código con un buen rendimiento cuando el número de hilos es menor o igual al número de cores (cuatro), pero el rendimiento es muy pobre cuando el número de hilos supera este umbral. Comparando con `gcc`, el compilador de Intel tiene siempre desventaja. El compilador GNU presenta un excelente comportamiento cuando se ejecuta con más hilos que cores. Como cabe esperar, cuanto mayor es el número de hilos, más tiempo tarda, aunque la diferencia sea muy pequeña en comparación con los grandes saltos de tiempo del compilador de Intel.

Como conclusión de esta prueba podemos afirmar que el compilador `gcc` con las optimizaciones `O2` son las que mejor rendimiento ofrecen para gestionar hilos, destacando especialmente por un buen comportamiento cuando el

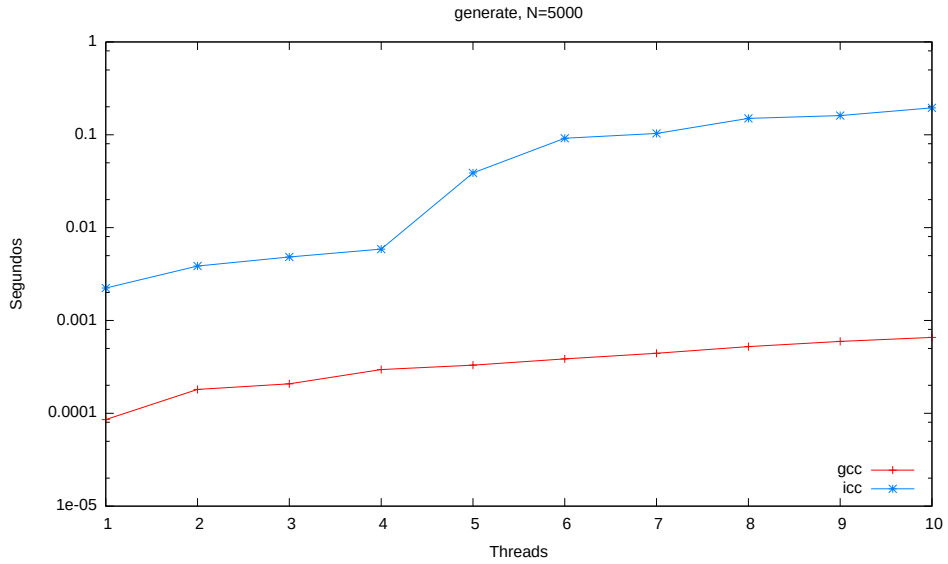

Figura 4.4: Comparación con el tiempo en escala logarítmica de los compiladores `gcc` e `icc` para el programa `generate` con $N = 5000$ y variando el número de hilos

número de hilos supera al número de cores.

4.1.2. Rutina pfor

Esta rutina consiste en paralelizar un bucle `for` en el que cada iteración lleva a cabo cierto trabajo. La carga de trabajo consiste en otro bucle que realiza asignaciones de su índice a un array. Esta carga de trabajo se puede cambiar por parámetros a la hora de invocar al programa. En nuestro caso este será el valor de N, que tomará los mismos valores que en la rutina anterior, es decir 1, 50, 100, 1000 y 5000. Para que el tiempo sea mayor y se pueda medir con precisión, se repetirá el proceso una cantidad `IT` de veces que se puede configurar, y tomaremos como 10. El programa `pfor` se puede resumir en el código de la figura 4.5.

Como en la rutina `generate`, en ambos programas el cuerpo de procesamiento son bucles, pero mientras antes paralelizábamos el código, sin indicar a OpenMP que era un bucle, en la rutina `pfor`, paralelizamos explícitamente un bucle `for`, informando a OpenMP de ello.

La metodología de las pruebas es la misma que con el programa `generate`, compararemos el rendimiento con los compiladores `gcc` e `icc`, cada uno con sus tres conjuntos de optimizaciones, realizando cada ejecución diez veces

```
omp_set_num_threads(P);
tiempo_inicio = omp_get_wtime();
for (i=0; i<10; i++ ) {
  #pragma omp parallel for private(j,k) firstprivate(N)
  for (j=0; j<N; j++ )
  {
    for(k=0; k<1000; k++) {
      r[j] = k;
    }
  }
}
tiempo_fin = omp_get_wtime();
```

Figura 4.5: Código simplificado de la rutina `pfor` escrita en C

para calcular cada media.

Los resultados obtenidos revelan que en el compilador de licencia GNU no hay diferencia alguna entre las optimizaciones de `O2` y `O3` (véase figura 4.6). Este hecho indica que las optimizaciones adicionales que `O3` tiene sobre `O2` no se aplican en este programa, y por tanto resultan en idénticos resultados. Sin embargo las optimizaciones de `Os` se diferencian notablemente de las de `O2` y `O3` ofreciendo un resultado peor.

Para el caso de `gcc`, nos quedamos con las optimizaciones de `O2`, ya que `O3` no aporta mejora. En el caso de `icc`, la figura 4.7 muestra que las optimizaciones dan resultados muy parecidos, notándose claramente el aumento de tiempo al utilizar más de un hilo por core. Para el compilador de Intel, no podemos destacar con objetividad un conjunto de optimizaciones sobre otro.

De estos gráficos ya podemos tener una idea inicial de cómo se comporta cada compilador según aumente el número de hilos. Para obtener una visión más clara, en la figura 4.8 puede verse que en el compilador de Intel, mejoramos según tengamos hilos y cores disponibles, pero cuando el número de hilos supera al de cores (a partir de cinco), el rendimiento empeora sustancialmente. Se ha seleccionado esta gráfica con $N = 5000$ y optimizaciones `O2` por ser representativa en cuanto al tamaño del problema (tamaños menores dan una gráfica similar).

Como conclusión de estas pruebas destacamos la superioridad de `gcc` sobre `icc` en todos los casos, y la idoneidad de las optimizaciones `O2` de este compilador.

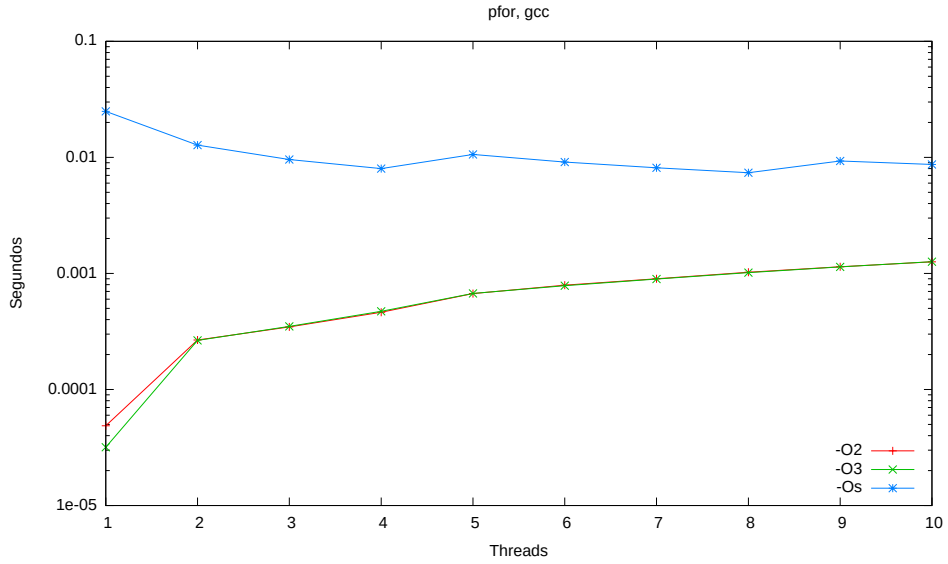

Figura 4.6: Comparación de optimizaciones de `gcc` para el programa `pfor` variando el número de hilos. El tiempo se muestra en escala logarítmica

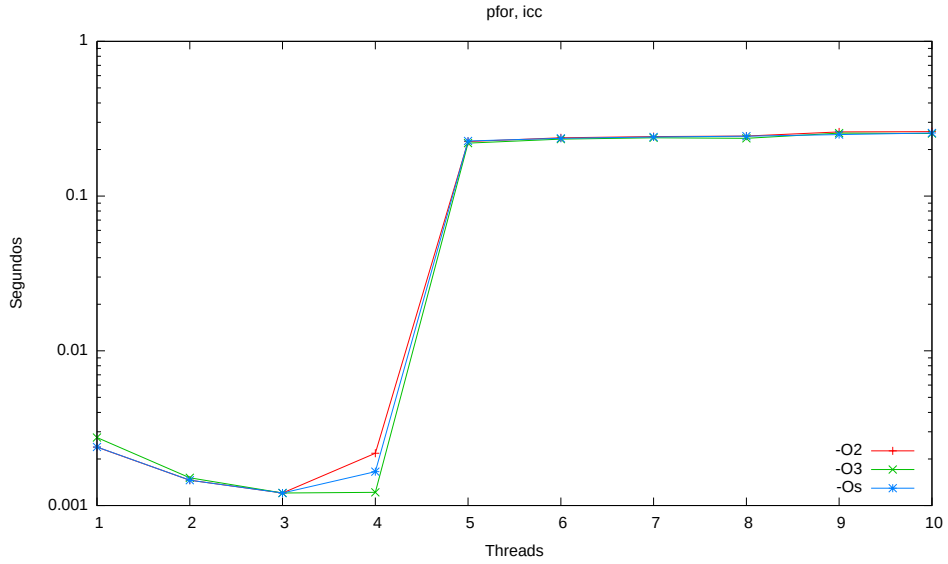

Figura 4.7: Comparación de optimizaciones de `icc` para el programa `pfor` variando el número de hilos. El tiempo se muestra en escala logarítmica

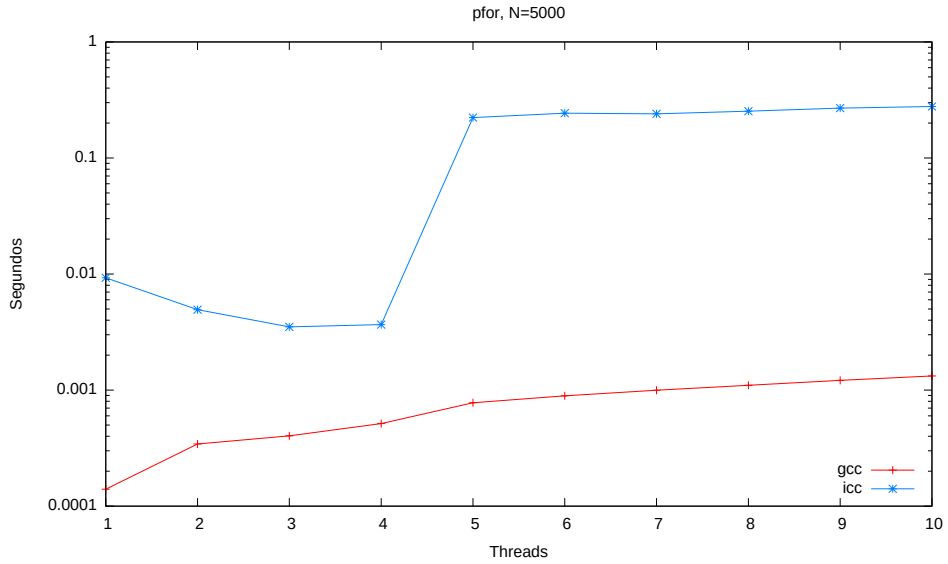

Figura 4.8: Rutina `pfor`, comparativa `gcc` e `icc` con `O2` con tamaño $N = 5000$ y variando el número de hilos. El tiempo se muestra en escala logarítmica

4.1.3. Rutina sections

Esta rutina consiste en un conjunto de secciones ejecutadas por los distintos hilos creados. Con esta rutina queremos medir el tiempo de acceso a distintas regiones exclusivas.

Para las mediciones seguiremos la misma metodología que con las anteriores primitivas, sin embargo esta rutina presenta ciertas particularidades: cada una de las secciones está codificada en el código fuente a través de una directiva del compilador, por lo tanto no es posible cambiar el número de secciones usando parámetros en la invocación del programa. Puesto que queremos probar desde un hilo hasta diez, es obligatorio crear diez ficheros C cada uno con un número distinto de secciones, y compilar los diez programas distintos (sin tener en cuenta las variaciones de compilador y distintos conjuntos de optimizaciones que ya conocemos). La creación de los ficheros en C, para cualquier número de secciones, se lleva a cabo automáticamente por la herramienta desarrollada.

El esquema del programa simplificado se muestra en la figura 4.9. En el lugar donde aparece el comentario habrán otros bloques de sección, según el número de hilos con el que estamos trabajando. Para que las asignaciones (a un array) se repitan varias veces y tomemos valores de tiempos mayores, tomamos 50 como cantidad de repeticiones para el bucle interior.

```
omp_set_num_threads(P);
tiempo_inicio = omp_get_wtime();
for(i=0; i<N; i++) {
  #pragma omp parallel sections private(j,k)
  {
    #pragma omp section
    {
      for(k=0; k<50; k++) {
        r[0] = j+k;
      }
    }
    /* Otras P-1 secciones idénticas *
     * con la asignación a r[P-1] */
  }
}
tiempo_fin = omp_get_wtime();
```

Figura 4.9: Código simplificado de la rutina `sections` escrita en C

En estas pruebas, los conjuntos de optimizaciones dan resultados muy similares, por lo que se deja la gráfica de ambos compiladores en el anexo. Por otro lado, sí resulta interesante la comparación de ambos compiladores. Cuando se realiza una sola iteración (tamaño $N = 1$) el compilador `gcc` muestra una ligera ventaja sobre `icc` (figura 4.10), sin embargo, esta prueba de tamaño tan pequeño no resulta significativa. Si nos centramos en tamaños mayores, es decir, en las pruebas en las que N toma valores de 50, 100, 1000 y 5000, vemos que se repite la misma tendencia según aumenta el tamaño: cuando hay hasta un hilo por core, el compilador de Intel consigue menor tiempo de ejecución. Se produce un cambio de tendencia al haber más de un hilo por core, puesto que este caso es el compilador `gcc` el que consigue mayor rendimiento. Según aumenta el tamaño de problema N, el tiempo de ejecución de ambos compiladores tiende a igualarse. Este comportamiento puede observarse en las figuras 4.11 y 4.12 correspondientes a los tamaños 100 y 5000 (en el anexo se encuentran las figuras correspondientes a los tamaños 50 y 1000).

Como resumen de esta prueba podemos afirmar que el compilador de Intel ofrece el mejor rendimiento cuando hay hasta un hilo por core, y el compilador GNU destaca cuando hay más hilos que cores, aunque en este último caso ambos compiladores tienden a igualarse con valores de N muy grandes.

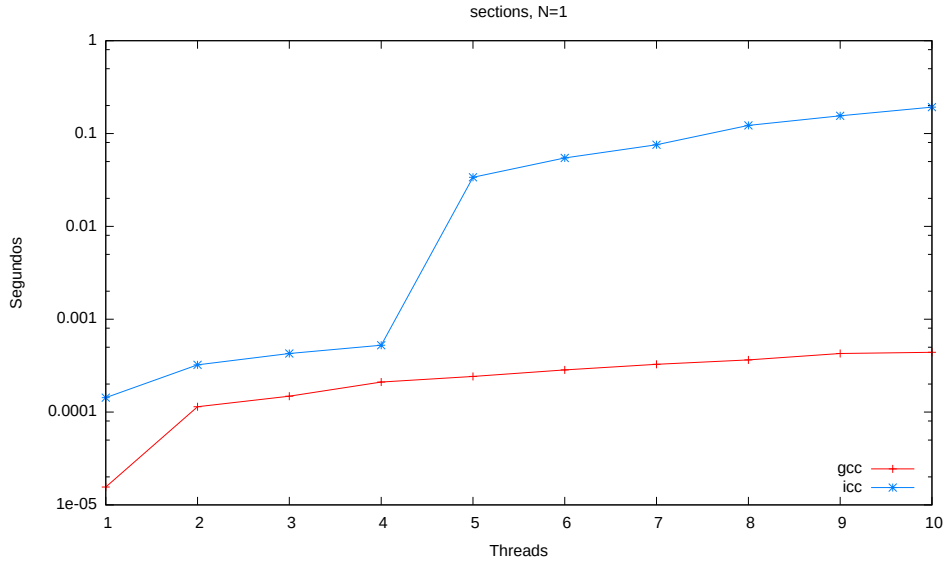

Figura 4.10: Rutina `sections`, comparativa `gcc` e `icc` con `O2` y una iteración del bucle. El tiempo se muestra en escala logarítmica

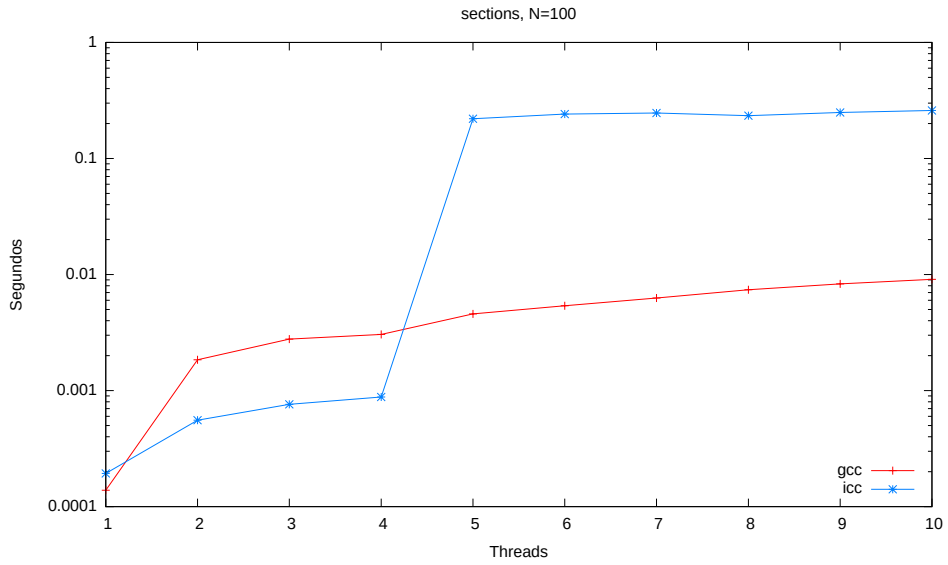

Figura 4.11: Rutina `sections`, comparativa `gcc` e `icc` con `O2`, variando el número de hilos para el tamaño $N = 100$. El tiempo se muestra en escala logarítmica

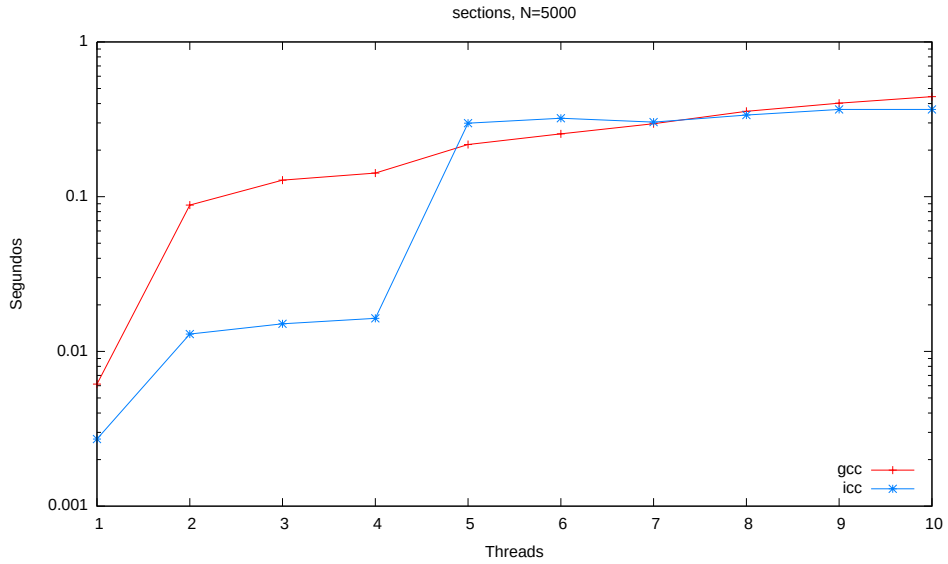

Figura 4.12: Rutina `sections`, comparativa `gcc` e `icc` con `O2`, variando el número de hilos para el tamaño $N = 5000$. El tiempo se muestra en escala logarítmica

4.1.4. Rutina barriers

El objetivo de esta rutina es medir los tiempos de sincronización conjunta de todos los hilos. Para ello la rutina utiliza la primitiva de barrera tras ejecutar el trabajo paralelo. El código resumido se muestra en la figura 4.13.

De nuevo llevaremos a cabo el mismo conjunto de pruebas con los diez hilos, los mismos tamaños de N y los dos compiladores con sus respectivos conjuntos de optimizaciones. Los resultados respecto a las mejores optimizaciones no revelan ninguna mejora de un grupo sobre otro. No obstante `Os` ofrece ocasionalmente un resultado ligeramente peor que `O2` y `O3` para ambos compiladores. De nuevo llegamos a la conclusión de que las optimizaciones de `O3` no se aplican en un código tan sencillo.

Resulta interesante el resultado de comparar ambos compiladores dependiendo del tamaño de N, en este caso, según el tamaño obtenemos gráficos distintos, aunque finalmente se extraiga una misma conclusión. El las figuras 4.14, 4.15, 4.16 y 4.17 se aprecia como cuando el número de hilos es menor o igual que el de cores, ambos compiladores dan un rendimiento similar para tamaño pequeño, y distinto para tamaños mayores. En este caso `icc` aventaja a `gcc`, pero el superar el número de hilos al de cores, se produce el efecto contrario, donde `gcc` toma ventaja.

```
omp_set_num_threads(P);
tiempo_inicio = omp_get_wtime();
for(i=0; i<N; i++ ) {
  #pragma omp parallel
  {
    int actual = omp_get_thread_num();
    for(j=0; j<50; j++) {
      r[actual] = numero();
    }
    #pragma omp barrier
  }
}
tiempo_fin = omp_get_wtime();
```

Figura 4.13: Código simplificado de la rutina `barriers` escrita en C

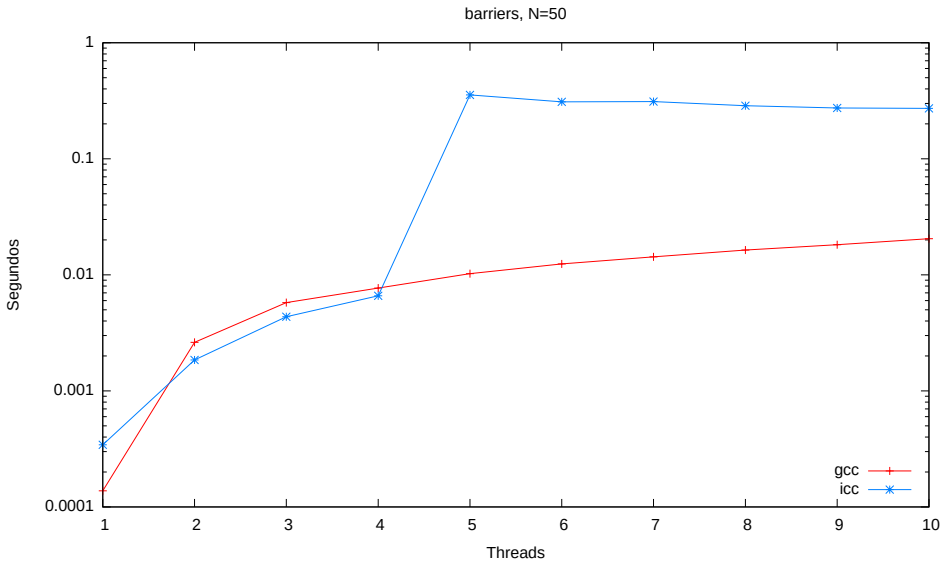

Figura 4.14: Rutina `barriers`, comparativa `gcc` e `icc` con `O2` variando el número de hilos y tamaño $N = 50$. El tiempo se muestra en escala logarítmica

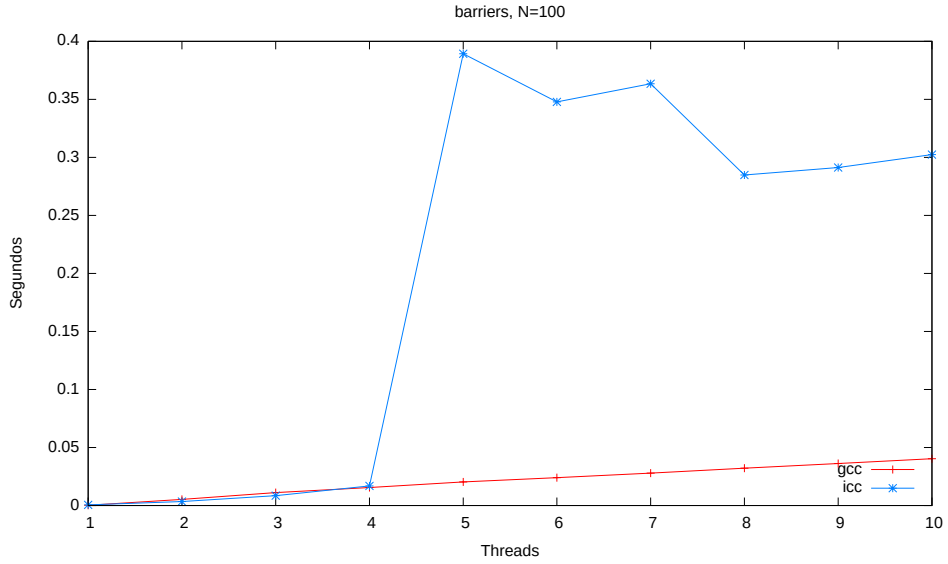

Figura 4.15: Rutina `barriers`, comparativa `gcc` e `icc` con `O2` variando el número de hilos y tamaño $N = 100$

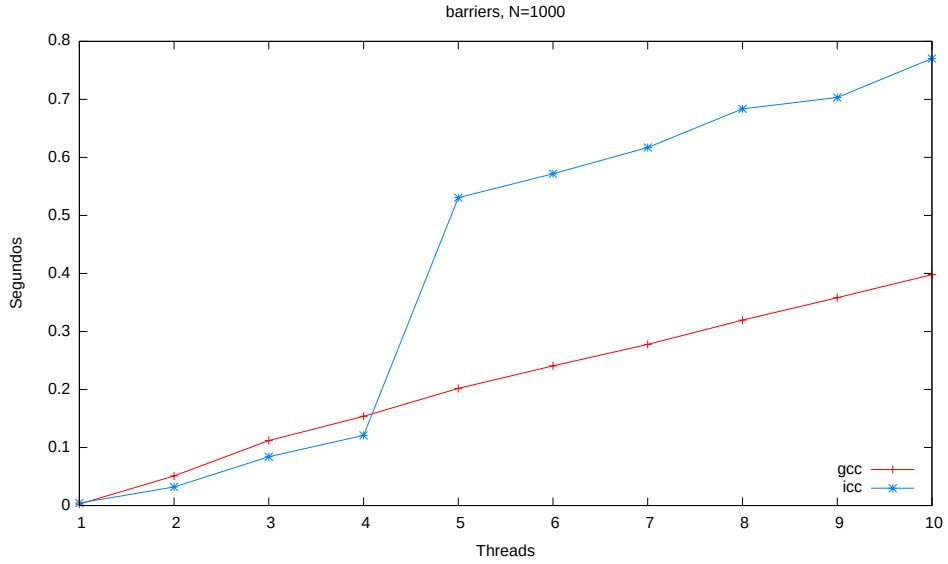

Figura 4.16: Rutina `barriers`, comparativa `gcc` e `icc` con `O2` variando el número de hilos y tamaño $N = 1000$

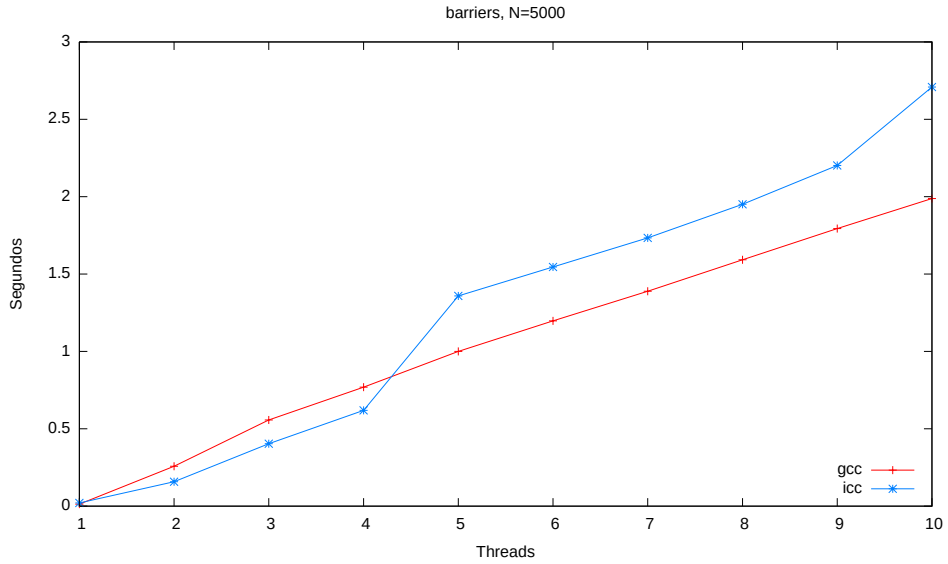

Figura 4.17: Rutina `barriers`, comparativa `gcc` e `icc` con `O2` variando el número de hilos y tamaño $N = 5000$

Como conclusión podemos decir que `gcc` consigue un buen rendimiento, pero `icc` se comporta mejor cuando el número hilos no supera al de cores, y esta distancia aumenta según repitamos las iteraciones.

4.2. Prestaciones de rutinas de alto nivel

En la sección anterior vimos cómo se comporta cada compilador al manejar las directivas de paralelización de OpenMP. En esta sección, con el estudio de las rutinas de alto nivel, pondremos dichas primitivas en cálculos de aplicación científica, en particular, de álgebra lineal, con operaciones de distinto coste computacional. Si antes no realizábamos cálculos complicados para obtener el coste de las primitivas, el enfoque ahora es el contrario: cálculos más costosos, con el coste de manejo de hilos pequeño (en proporción), con el fin de averiguar en qué medida la paralelización merece la pena en escenarios reales.

La metodología es la misma que la descrita para las primitivas en la sección anterior: se compila cada uno de los programas con `gcc` e `icc`, con los tres conjuntos de optimizaciones `O2`, `O3` y `Os`, y se lanza cada prueba para distintos tamaños del problema y desde uno a diez hilos. Cada resultado se repite y se calcula la media, salvo que se detecte algún resultado incorrecto,

en cuyo caso se repite. A diferencia que con las mediciones de las primitivas, las pruebas se repetirán cinco veces, puesto que en esta ocasión cada prueba tarda un tiempo considerable, lo que conlleva mayor semejanza en los tiempos de ejecución de las distintas veces que se ejecuta cada rutina.

4.2.1. Multiplicación matriz-vector

Esta rutina de coste $O(n^2)$ consiste en multiplicar una matriz por un vector de forma paralela. Para ello distribuiremos estáticamente entre los hilos las iteraciones de un bucle `for` paralelo que recorre las filas de una matriz cuadrada $M \in \mathbb{R}^{n \times n}$. Multiplicaremos las filas de la matriz por un vector $v \in \mathbb{R}^n$. Tanto las filas de la matriz como el vector se encuentran en posiciones contiguas de memoria.

Para las mediciones, elegimos los tamaños de problema 2000, 4000 y 8000, puesto que estos valores de n son suficientemente grandes para obtener una buena visión del comportamiento de la rutina. Como vemos en las figuras 4.18 y 4.19, en el caso de `gcc` las optimizaciones `O2` y `O3` son idénticas, y en el caso de `icc`, ambas optimizaciones dan rendimientos similares. Por estos motivos, escogeremos la optimización `O2` como la adecuada para esta rutina.

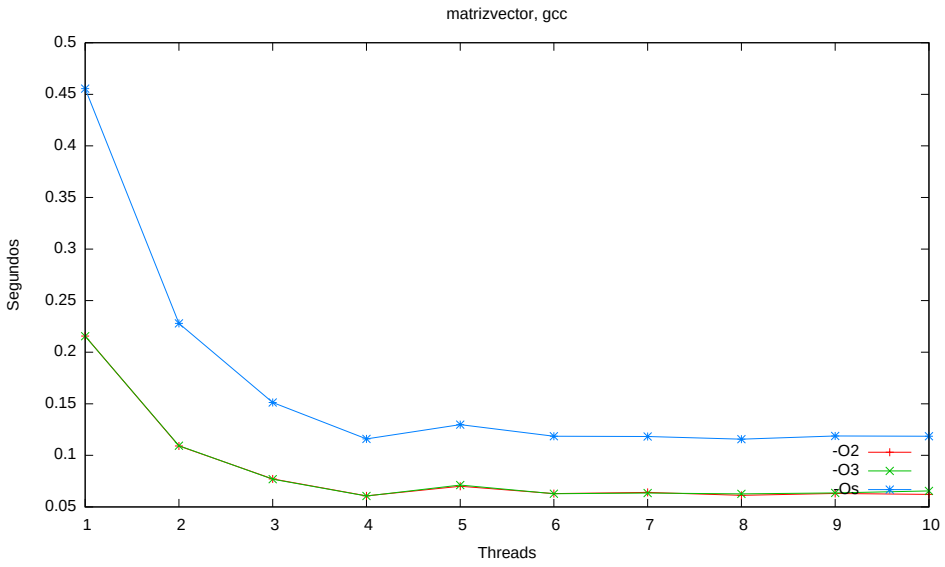

Figura 4.18: Comparación de optimizaciones de `gcc` para el programa `matrizvector` variando el número de hilos

En las figuras antes mencionadas también se aprecia que el comportamiento es el esperado: se reduce el tiempo de ejecución cuantos más hilos

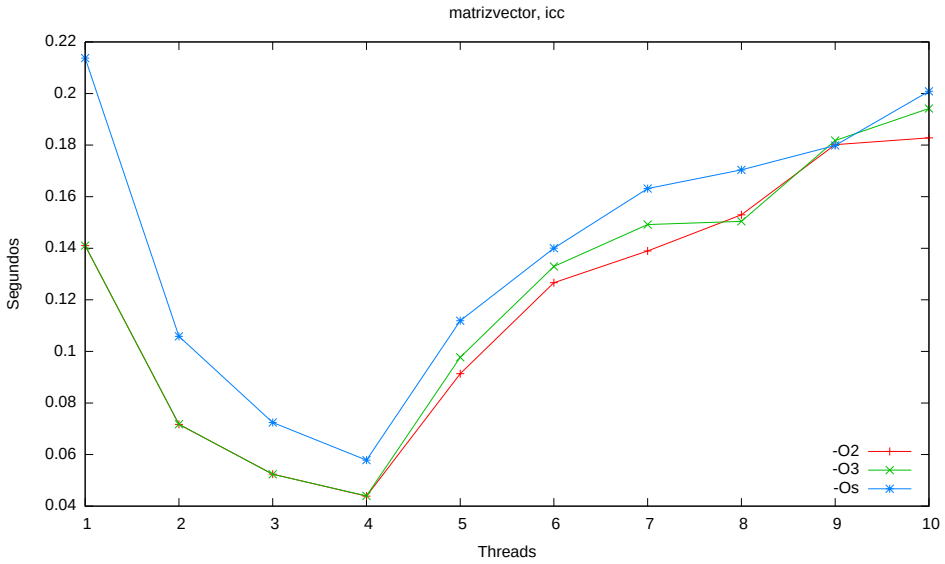

Figura 4.19: Comparación de optimizaciones de `icc` para el programa `matrizvector` variando el número de hilos

usamos. Esto se aprecia para los valores de uno a cuatro hilos. Cuando tenemos más de cuatro hilos, ya que hay sólo cuatro cores, para `gcc` se mantiene más o menos constante (algo peor que con cuatro hilos), y para `icc` el tiempo de ejecución crece notablemente. En las tablas 4.2 y 4.3 se muestran los cálculos de *speedup*, eficiencia y coste para el mayor tamaño de matriz con los compiladores `gcc` e `icc` respectivamente.

Si nos centramos en la comparación de ambos compiladores para distinto tamaño del problema, obtenemos que cuando hay un hilo por core o menos, el compilador de Intel tiene cierta ventaja. Esta ventaja se acentúa conforme aumenta el tamaño del problema. En las figuras 4.20, 4.21 y 4.22 puede verse este efecto para los distintos tamaños del problema. Cuando el número de hilos supera al de cores, se produce el efecto contrario, y el código generado por `gcc` demuestra un mejor rendimiento.

Si calculamos de forma teórica los megaflops (ver tabla 4.1), obtenemos las mismas conclusiones. El cálculo de los megaflops se hace teniendo en cuenta que la operación tiene complejidad $2n^2$, y tomando el tamaño más grande ($n = 8000$) como representativo[8].

Como conclusión se puede obtener que aunque `gcc` destacaba en la rutina

[8]Este cálculo asume que otras instrucciones como las que se hacen con enteros, saltos, etc., tienen un coste despreciable.

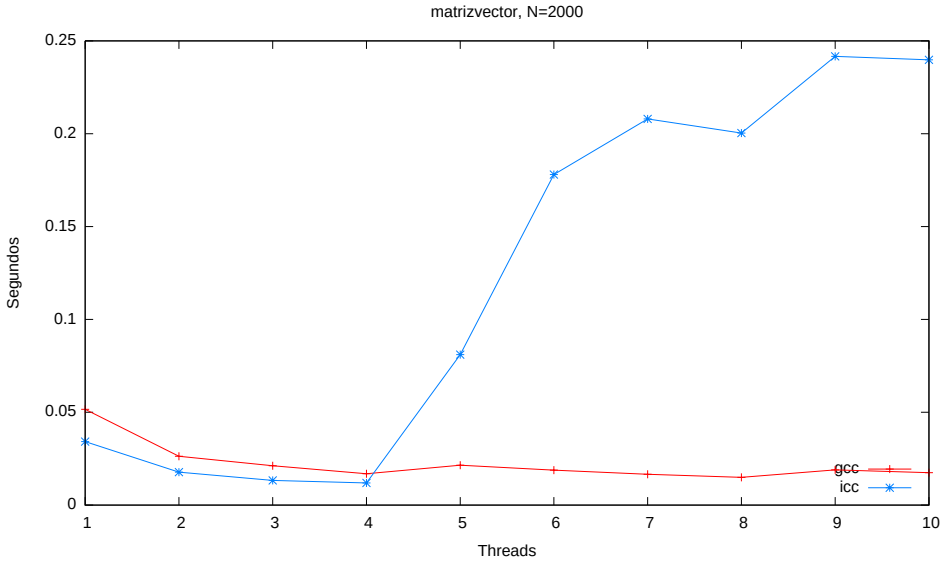

Figura 4.20: Rutina `matrizvector`, comparativa variando el número de hilos de los compiladores `gcc` e `icc` con `O2` y tamaño $N = 2000$

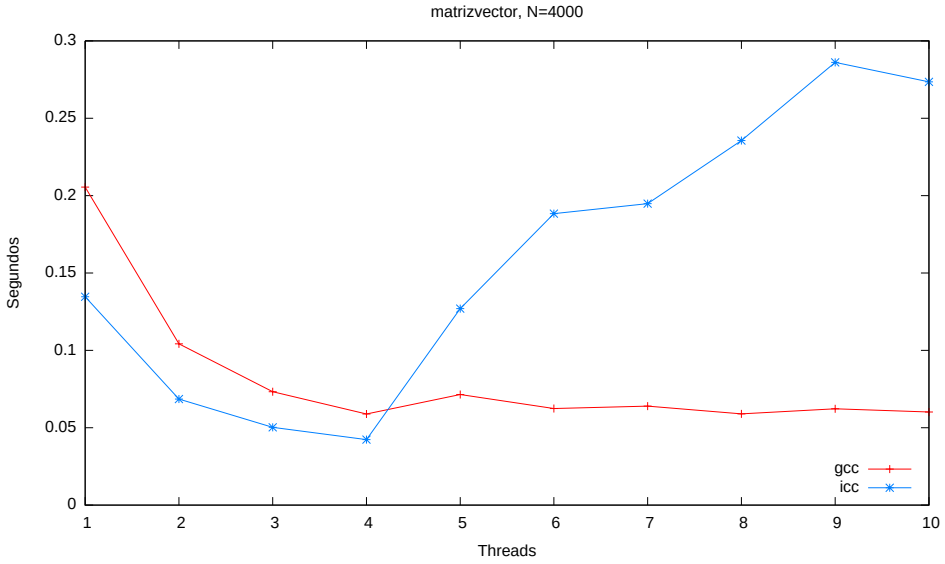

Figura 4.21: Rutina `matrizvector`, comparativa variando el número de hilos de los compiladores `gcc` e `icc` con `O2` y tamaño $N = 4000$

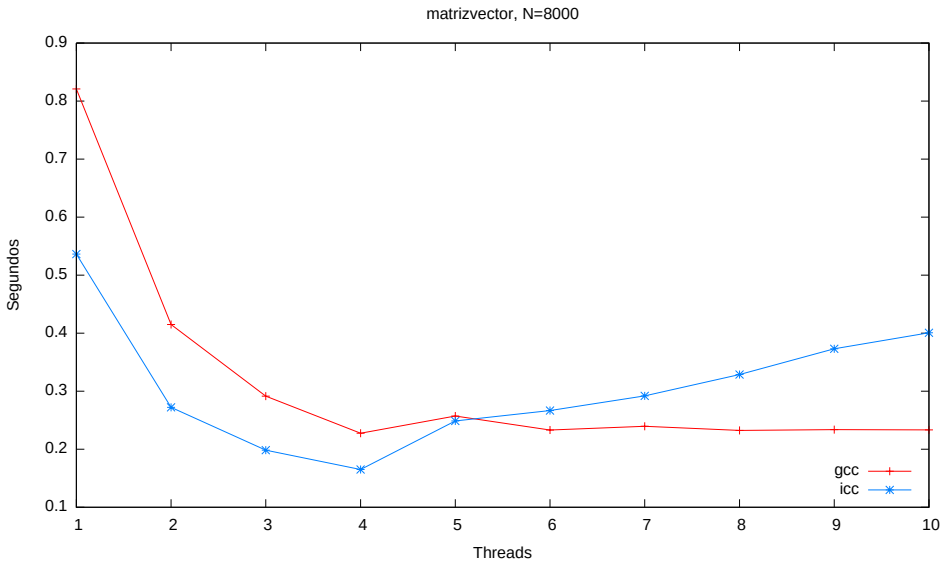

Figura 4.22: Rutina `matrizvector`, comparativa variando el número de hilos de los compiladores `gcc` e `icc` con `O2` y tamaño $N = 8000$

	1	2	3	4	5	6	7	8	9	10
gcc	155.93	308.45	439.02	562.21	497.50	548.90	534.14	550.62	547.48	548.61
icc	238.67	470.12	645.07	775.76	514.65	480.00	438.39	389.35	343.08	319.46

Tabla 4.1: Megaflops de rutina `matrizvector` con optimizaciones `O2`

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.978	0.989	0.829
3	2.815	0.938	0.874
4	3.605	0.901	0.910
5	3.190	0.638	1.286
6	3.520	0.586	1.399
7	3.425	0.489	1.677
8	3.531	0.441	1.859
9	3.511	0.390	2.104
10	3.518	0.351	2.333

Tabla 4.2: *Speedup*, eficiencia y coste de `matrizvector` compilado con `gcc` y optimizaciones `O2`. El valor de n es 8000. El valor del tiempo secuencial es $t(n) = 0{,}820$ segundos

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.969	0.984	0.544
3	2.702	0.900	0.595
4	3.250	0.812	0.659
5	2.156	0.431	1.243
6	2.011	0.335	1.599
7	1.836	0.262	2.043
8	1.631	0.203	2.629
9	1.437	0.159	3.357
10	1.338	0.133	4.006

Tabla 4.3: *Speedup*, eficiencia y coste de `matrizvector` compilado con `icc` y optimizaciones `O2`. El valor de n es 8000. El valor del tiempo secuencial es $t(n) = 0{,}536$ segundos

	1	2	3	4	5	6	7	8	9	10
gcc	121.15	231.97	327.58	411.58	349.89	374.15	385.97	397.59	389.62	385.57
icc	136.65	262.98	366.01	455.65	315.12	290.58	290.52	270.11	269.75	262.00

Tabla 4.4: Megaflops de rutina `jacobi` con optimizaciones `O2`

`pfor`, cuando introducimos más cálculo aritmético, las optimizaciones que `icc` realiza sobre esta rutina y su gestión de la memoria hace que ofrezca un mejor rendimiento cuando cada core tiene como máximo un hilo. No obstante, de nuevo `gcc` demuestra un mejor rendimiento cuando hay más hilos que cores.

4.2.2. Relajación de Jacobi

Se trata de la implementación del método de relajación de Jacobi para una malla 2D de puntos [10]. Se ejecuta una única iteración de cara a tomar tiempos de ejecución más comparables. A las matrices se accede por filas, por lo que se accede a espacio contiguo en memoria. La rutina tiene un coste $10n^2$.

Un buen conjunto de optimizaciones para esta rutina es el de `O2`, tanto para `gcc`, como para `icc`, siendo en este último compilador menos clara la diferencia cuando aumenta el número de hilos. En el anexo se incluyen todos los datos y otros gráficos que ilustran el comportamiento de los distintos conjuntos de optimizaciones.

Las pruebas realizadas, análogas a la de los apartados anteriores, con

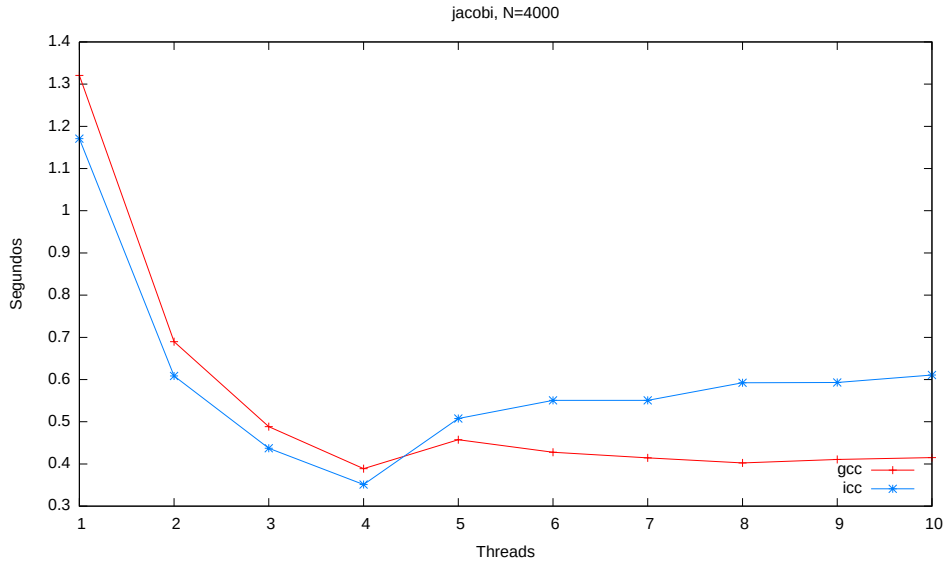

Figura 4.23: Rutina `jacobi`, comparativa `gcc` e `icc` con `O2` y tamaño $N = 4000$

tamaños de matriz 1000, 2000 y 4000, muestran unos resultados similares a la multiplicación matriz-vector, es decir, el compilador de Intel es mejor cuando hay a lo sumo un hilo por core, y peor en todos los demás casos. Cuando hay más de un hilo por core, `icc` da un gran cambio proporcionando unos tiempos de ejecución mayores. De nuevo se ve la tendencia de cuanto mayor es la matriz, las distancias se acortan. En la figura 4.23 se ve para el tamaño de 4000 elementos (por brevedad se dejan las gráficas de otros tamaños de matriz en el anexo).

En el cálculo de los megaflops (tabla 4.4) se ve también esta tendencia: `icc` refleja un gran descenso en los megaflops cuando hay más hilos que cores para procesarlos [9]. El cálculo de las medidas de rendimiento confirman las conclusiones que muestran las gráficas. Podemos ver estos cálculos para cada compilador en las tablas 4.5 y 4.6.

4.2.3. Multiplicación matriz-matriz, versión 1

Esta rutina de coste $2n^3$ multiplica dos matrices usando un bucle `for` paralelo con dos bucles `for` anidados. Las iteraciones del bucle paralelo se

[9] Los megaflops se calculan tomando el tamaño más grande de las pruebas, en este caso $N = 4000$

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.914	0.957	1.379
3	2.703	0.901	1.465
4	3.397	0.849	1.554
5	2.888	0.577	2.286
6	3.088	0.514	2.565
7	3.185	0.455	2.901
8	3.281	0.410	3.219
9	3.215	0.357	3.695
10	3.182	0.318	4.149

Tabla 4.5: *Speedup*, eficiencia y coste de `jacobi` compilado con `gcc` y optimizaciones `O2`. El valor de n es 4000. El valor del tiempo secuencial es $t(n) = 1{,}320$ segundos

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.924	0.962	1.216
3	2.678	0.892	1.311
4	3.334	0.833	1.404
5	2.305	0.461	2.538
6	2.126	0.354	3.303
7	2.125	0.303	3.855
8	1.976	0.247	4.738
9	1.973	0.219	5.338
10	1.917	0.191	6.106

Tabla 4.6: *Speedup*, eficiencia y coste de `jacobi` compilado con `icc` y optimizaciones `O2`. El valor de n es 4000. El valor del tiempo secuencial es $t(n) = 1{,}170$ segundos

	1	2	3	4	5	6	7	8	9	10
gcc	147.53	285.99	408.45	554.46	536.00	534.22	538.35	542.11	540.44	538.44
icc	151.75	291.74	410.32	558.91	541.21	539.96	542.05	545.62	538.71	538.98

Tabla 4.7: Megaflops de rutina `matrizmatriz` con optimizaciones `O2`

distribuyen entre los hilos. Cada hilo accede a un espacio contiguo de memoria para acceder a las filas de A, pero no para las columnas de B.

En el análisis de la multiplicación $AB = C$ con $A, B, C \in \mathbb{R}^{n \times n}$, seguiremos la misma metodología que hasta ahora, con tamaños de matrices 500, 1000 y 1500. En cuanto a las optimizaciones, no hay grandes diferencias, salvo en ocasiones un peor rendimiento de `Os` (ver anexo A). Tomaremos de nuevo `O2` como referencia. Los resultados son similares al caso anterior: `icc` tiene una ventaja cuando trabaja con uno, dos, tres y cuatro hilos, y `gcc` cuando se usan más hilos (figura 4.24). No obstante cuando aumenta el tamaño de la matriz, los resultados tienden a igualarse, tal y como se aprecia en la figura 4.25 para un tamaño de matriz cuadrada de 1500 elementos. En la tabla 4.7 se calculan los megaflops obtenidos en cada compilador, así como en las tablas 4.8 y 4.9 se muestran los cálculos de los parámetros de rendimiento paralelo para `gcc` e `icc` respectivamente.

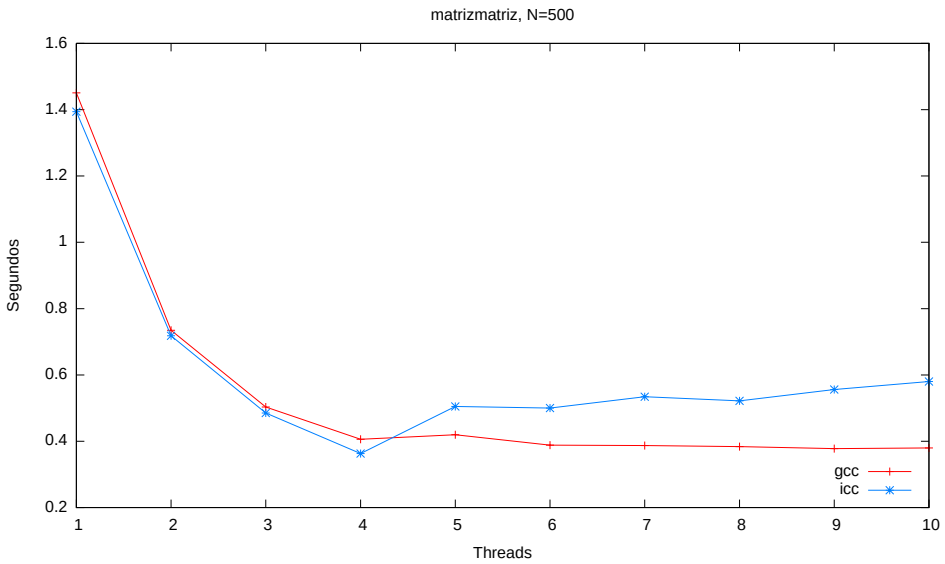

Figura 4.24: Rutina `matrizmatriz`, comparativa `gcc` e `icc` con `O2` con tamaño $N = 500$, variando el número de hilos

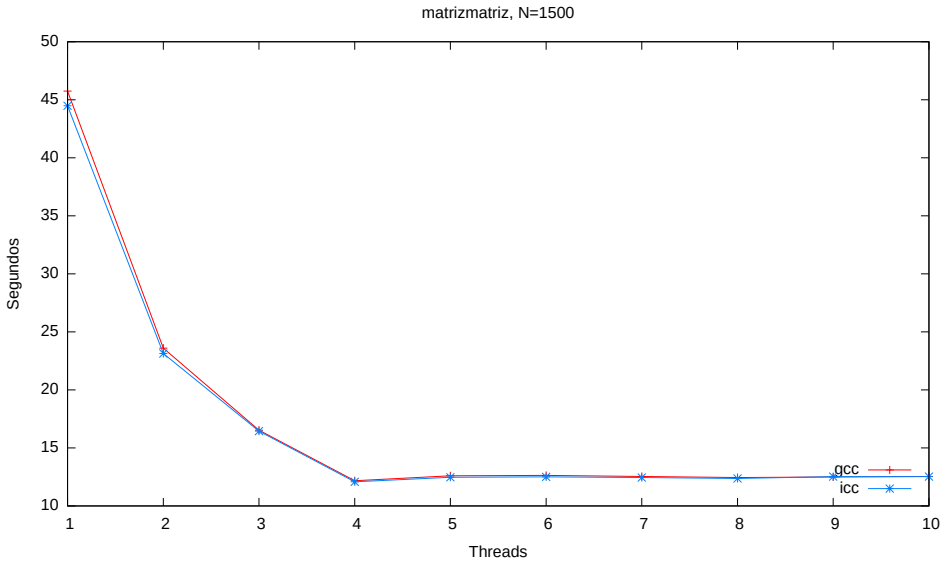

Figura 4.25: Rutina `matrizmatriz`, comparativa `gcc` e `icc` con `O2` con tamaño $N = 1500$, variando el número de hilos

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.938	0.969	47.204
3	2.768	0.922	49.576
4	3.758	0.939	48.695
5	3.633	0.726	62.965
6	3.621	0.603	75.810
7	3.649	0.521	87.766
8	3.674	0.459	99.610
9	3.663	0.407	112.407
10	3.649	0.364	125.359

Tabla 4.8: *Speedup*, eficiencia y coste de `matrizmatriz` compilado con `gcc` y optimizaciones `O2`. En valor de n es 1500. El valor del tiempo secuencial es $t(n) = 45{,}751$ segundos

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.922	0.961	46.273
3	2.703	0.901	49.350
4	3.683	0.920	48.307
5	3.566	0.713	62.359
6	3.558	0.593	75.004
7	3.571	0.510	87.168
8	3.595	0.449	98.969
9	3.549	0.394	112.769
10	3.551	0.355	125.236

Tabla 4.9: *Speedup*, eficiencia y coste de `matrizmatriz` compilado con `icc` y optimizaciones `O2`. El valor de n es 1500. El valor del tiempo secuencial es $t(n) = 44{,}480$ segundos

	1	2	3	4	5	6	7	8	9	10
gcc	156.14	302.52	438.04	547.97	561.18	567.07	566.28	569.90	570.83	570.51
icc	234.59	459.83	635.49	786.00	763.75	759.58	762.12	761.86	762.74	761.44

Tabla 4.10: Megaflops de la rutina `matrizmatriz2` con optimizaciones `O2`

4.2.4. Multiplicación matriz-matriz, versión 2

Esta versión de la multiplicación de matriz por matriz se diferencia de la primera versión en que la matriz B está almacenada por columnas en lugar de por filas, de forma que tenemos acceso a los elementos para la multiplicación de forma contigua, y por tanto, se espera un mejor rendimiento.

Las conclusiones para las optimizaciones son las mismas que con la primera versión, y se dejan las correspondientes gráficas en los anexos. No obstante en esta versión de la multiplicación, aunque se produce el ya conocido efecto de que el compilador de Intel aventaja al de licencia GNU cuando hay hasta un hilo por core (figura 4.26), y los papeles cambian al haber más hilos que cores, se observa que el compilador de Intel es mejor en todos los casos al aumentar considerablemente el tamaño de la matriz (figura 4.27). Se concluye entonces que `icc` sabe aprovechar mejor la localidad espacial ajustándose mejor al tamaño de la caché del procesador.

Si nos fijamos en la tabla 4.10, vemos que tomando la media de los tres tamaños de matriz, `icc` obtiene más operaciones en punto flotante que `gcc`. Por lo que en esta medición podemos considerar que el compilador de Intel es que tiene el mejor comportamiento.

Comparando ambas versiones de la multiplicación de matrices en términos

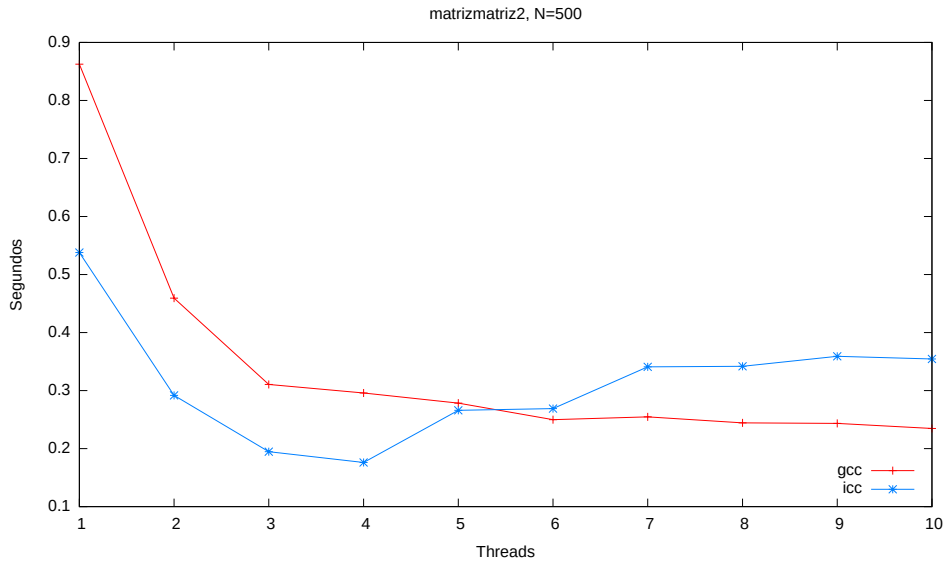

Figura 4.26: Rutina `matrizmatriz2`, comparativa `gcc` e `icc` con `O2` variando el número de hilos con tamaño $N = 500$

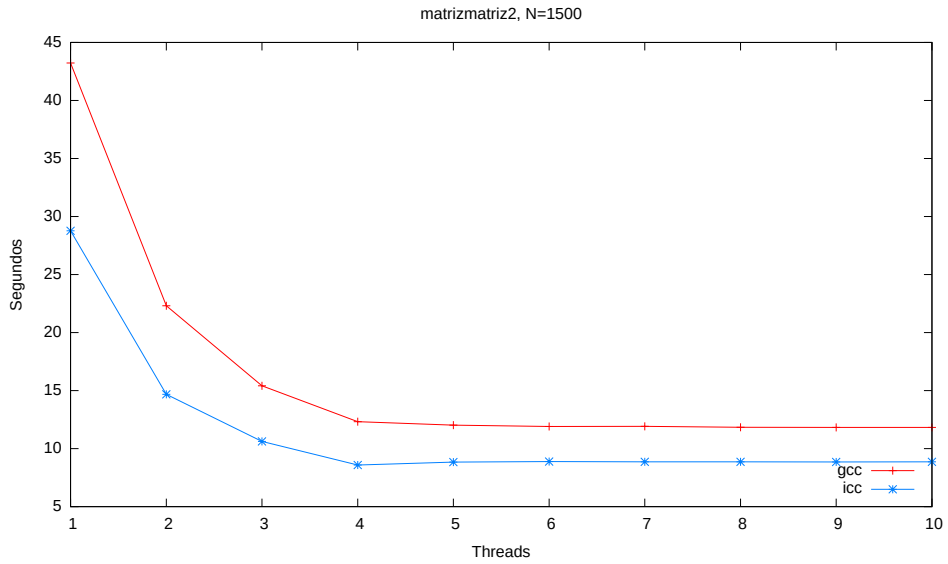

Figura 4.27: Rutina `matrizmatriz2`, comparativa `gcc` e `icc` con `O2` variando el número de hilos con tamaño $N = 1500$

de megaflops obtenemos como resultado que la mejor versión es la segunda con `icc`. Se puede observar este comportamiento en la gráfica comparativa de la figura 4.28.

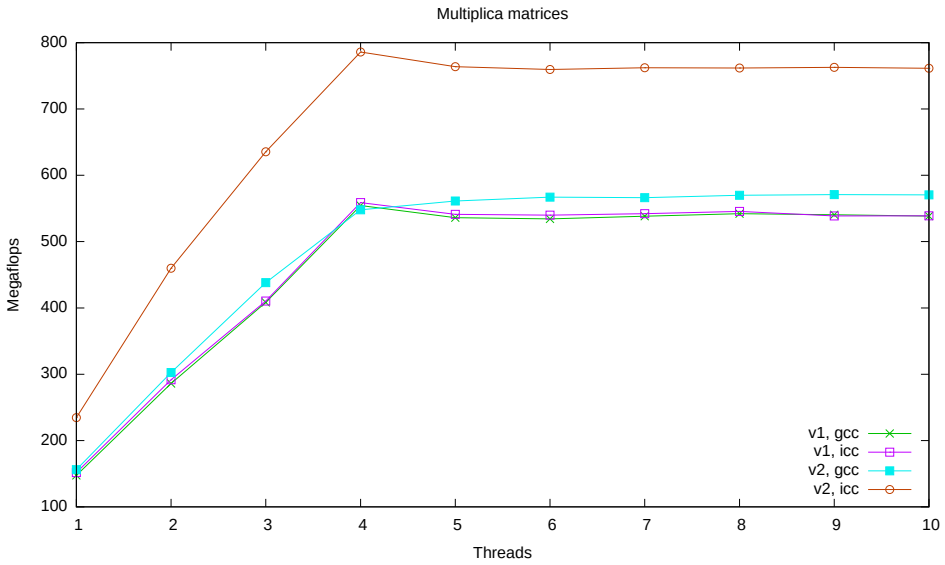

Figura 4.28: Comparativa de los megaflops de multiplicación de matrices versión 1 y 2

En la tabla 4.11 se muestra el cálculo del *speedup*, la eficiencia y el coste con el compilador de licencia GNU. Podemos comparar estos resultados con los obtenidos en la tabla 4.12 que corresponden al compilador de Intel.

4.3. Conclusiones

Puesto que hemos realizado el estudio de OpenMP con dos compiladores, podemos sacar conclusiones sobre la posibilidad de usar una aproximación de policompilación. Entendemos por la técnica de policompilación la compilación con ambos compiladores, y la ejecución de un programa u otro dependiendo de con cuál vayamos a obtener mayor rendimiento. La aplicación de las herramientas desarrolladas ha permitido tener una visión clara de qué compilador ofrece mejores prestaciones en cada caso. Aunque al trabajar únicamente con primitivas el compilador de licencia libre ha destacado, a la hora de realizar cálculos más complejos, el compilador de Intel toma ventaja en algunos casos de rutinas de alto nivel. En tres de cuatro mediciones de prestaciones de rutinas de alto nivel el compilador de Intel ha destacado

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.937	0.968	44.624
3	2.805	0.935	46.228
4	3.509	0.877	49.272
5	3.594	0.718	60.140
6	3.631	0.605	71.419
7	3.626	0.518	83.438
8	3.649	0.456	94.751
9	3.655	0.406	106.422
10	3.653	0.365	118.313

Tabla 4.11: *Speedup*, eficiencia y coste de `matrizmatriz2` compilado con `gcc` y optimizaciones `O2`. el valor de n es 1500. El valor del tiempo secuencial es $t(n) = 43{,}230$ segundos

p	Speedup $S(n,p)$	Eficiencia $E(n,p)$	Coste $C(n,p)$
2	1.960	0.980	29.358
3	2.708	0.902	31.864
4	3.350	0.837	34.350
5	3.255	0.651	44.189
6	3.237	0.539	53.318
7	3.248	0.464	61.997
8	3.247	0.405	70.878
9	3.251	0.361	79.646
10	3.245	0.324	88.647

Tabla 4.12: *Speedup*, eficiencia y coste de `matrizmatriz2` compilado con `gcc` y optimizaciones `O2`. El valor de n es 1500. El valor del tiempo secuencial es $t(n) = 28{,}773$ segundos

cuando hay a lo sumo un hilo por core y ha ofrecido un rendimiento peor cuando hay que distribuir más de un hilo por core. En general, podemos concluir que cuando hay más hilos que cores conviene usar la versión de la rutina compilada con `gcc`, y sólo cuando hay un hilo por core convienen usar la versión con el compilador de Intel.

Capítulo 5

Prestaciones de MPI

Si en el capítulo anterior estudiamos las prestaciones de la programación en memoria compartida, en este capítulo cambiamos de paradigma y profundizaremos en las prestaciones del modelo basado en paso de mensajes. MPI es actualmente un estándar de pasos de mensajes ampliamente utilizado. Basándonos en la implementación LAM/MPI, usaremos los *scripts* desarrollados para comparar las prestaciones en distintos clusters. Haremos el estudio en SOL -añadiendo la complejidad de la computación heterogénea- y combinaremos arquitecturas de 64 bits con procesadores de 32 bits. Al igual que en el capítulo anterior, distinguimos dos partes en el estudio: el estudio de las primitivas básicas y las prestaciones de rutinas de alto nivel, materializado en la multiplicación de matrices.

5.1. Arquitecturas heterogéneas

Uno de los retos más importantes cuando nos enfrentamos a una arquitectura heterogénea (como la del cluster SOL) es estudiar cómo combinar los nodos de 64 bits con el nodo5[1], que además de ser de 32 bits, tiene un procesador muy anterior al resto: se trata de combinar Intel Xeon de varios cores, con un Pentium III con dos procesadores.

Para lograr combinar ambas arquitecturas se ha probado sin éxito compilaciones estáticas de 32 bits, copiar las bibliotecas de los ejecutables de 32 bits a los nodos de 64 y otras muchas combinaciones posibles. El problema principal es que MPI distribuye el ejecutable al resto de los nodos, de forma que el nodo5 nunca puede ejecutar el binario de 64 bits. Análogamente el

[1]El nombre nodo5 corresponde al *hostname* del sexto nodo, tal y como se detalló en la sección 2.2.1. En MPI sigue el mismo orden: siendo `n0` el nodo SOL y `n5` el nodo5 que corresponde al Pentium III

enfoque inverso: que sea el nodo5 quien haga ejecutar el binario de 32 bits al resto de los nodos, tampoco funciona al no tener las bibliotecas adecuadas. Por otra parte las bibliotecas MPI no permiten crear ejecutables estáticos, lo que dificulta más la labor.

Una solución es tener un entorno completo de 32 bits, es decir, en todos los nodos. Esto nos permite experimentar con los nodos de distintas velocidades, pero no combinar ambas arquitecturas con distintos tamaños de palabra. En este sentido resulta inviable cambiar el cluster de 64 a 32 bits, reinstalando el sistema operativo, puesto que perderíamos gran parte del rendimiento en los demás programas sólo para favorecer a MPI. Tampoco queremos recurrir al uso de máquinas virtuales, puesto que entonces las pruebas perderían precisión. Una prueba que sí ha funcionado es la de exportar el sistema de ficheros del nodo5 a otro nodo de 64 bits, y lanzar `lamboot` en un `chroot`. Esto funciona para dos nodos, pero no para el resto que no pueden lanzarse automáticamente dentro del `chroot`.

La solución aparentemente trivial: copiar las bibliotecas de 32 bits al entorno de 64, consigue lanzar los programas, pero al iniciar MPI con `MPI_Init`, esta función se queda a la espera de comunicarse con los demás nodos y nunca finaliza.

Todas estas soluciones, aún en el caso de funcionar, no serían satisfactorias, puesto obligaría a tener un ejecutable de 32 bits, posiblemente optimizado para el nodo5, ejecutándose en nodos de 64 bits para los que no han sido optimizados. La mejor solución es que los nodos de 64 bits ejecuten un programa optimizado para 64 bits y el nodo de 32 optimizado para ese ordenador. Esta solución finalmente es posible haciendo uso de esquemas: un esquema permite indicar qué ejecutable se va a lanzar en cada nodo, de forma que puedan ser distintos para los diferentes nodos del cluster. Supongamos que en un shell *script* tenemos el nombre del programa en la variable `$X`, entonces podemos compilarlo así:

```
1 mpicc $X.c -o $X -O2 -march=native
2 ssh nodo5 "cd $(pwd) ; mpicc $X.c -o ${X}32 -O2 -march=native"
```

Es decir, hemos generado dos ejecutables: uno de 64 bits optimizado para Intel Xeon, y otro de 32 bits optimizado para Intel Pentium III, que tiene el sufijo 32 en el nombre del ejecutable. Ambos son el mismo progama optimizado para el nodo en el que se va a ejecutar. Con esta técnica no tenemos problemas de bibliotecas, puesto que cada uno se ha compilado en la máquina destino.

Queda solucionar cómo indicamos a MPI que use un ejecutable u otro. Como ya adelantamos en el capítulo 3, haremos uso de esquemas: un esquema se materializa en un fichero, por ejemplo, para ejecutar el programa X en los

seis nodos, escribimos un fichero llamado `schema_6` con el siguiente contenido:

```
1 n0-4 X
2 n5 X32
```

Donde sustituimos X por el nombre del ejecutable de 64 bits y X32 por el ejecutable de 32 bits. Estos ejecutables tienen que estar en el mismo directorio, aunque para evitar problemas podemos indicar la ruta completa del ejecutable. Una vez escrito el fichero con el esquema, basta lanzar la ejecución de la siguiente manera:

```
1 mpirun schema_6
```

Con lo que al no tener más parámetros que el nombre del fichero, `mpirun` asume que es un fichero de esquema y lanza la ejecución de los programas de 64 y 32 bits. El uso de esquemas es totalmente transparente en la herramienta de medición: tan sólo hay que escribir un fichero indicando qué nodos son de 32 bits y cuales de 64[2].

5.2. Prestaciones de las primitivas

En las siguientes secciones se mostrarán las prestaciones de las distintas primitivas de MPI en su implementación LAM para los distintos nodos de procesamiento. Puesto que el rendimiento de las primitivas depende en gran medida del rendimiento de la red, se repiten las pruebas un número suficientemente elevado para tener resultados objetivos que permitan determinar tendencias en los comportamientos de las primitivas.

Todas las pruebas se realizan con la red desocupada, pudiendo considerar que estas pruebas son fiables puesto que el cluster MPI trabaja con una red interna local, aislada del resto de Internet. Para evitar que las primeras mediciones se vean afectadas por retardos en la resolución de IP a direcciones MAC (por el protocolo ARP), justo antes de cada medición se realiza un `ping` a todos los nodos.

La medición de tiempo en las primitivas plantea un problema adicional[3] respecto a mediciones anteriores: al intervenir varios nodos es necesario una referencia común de tiempo. Al iniciar una prueba, medimos desde que el primer nodo comienza la rutina, hasta que el último nodo la finaliza. A priori no sabemos qué nodo será el primero y cuál será el último. Para averiguarlo tomaremos la fecha y la hora del sistema, y finalmente compararemos la fecha y la hora de todos los nodos. Esto plantea el problema de la sincronización:

[2]Véase apartado 3.2.2 para más detalles

[3]Este problema no ocurre con la multiplicación de matrices, ya que en este caso se utilizan `MPI_Barrier` y `MPI_Wtime` para medir el tiempo en un único nodo.

todos los nodos deben estar sincronizados con la misma fecha y hora. En la práctica se consigue una excelente sincronización de todos los nodos con el protocolo NTP[4]. Sin embargo la sincronización no es perfecta y podemos encontrar desviaciones de tiempo del orden de las décimas o centésimas de segundo. Aunque este desfase entre relojes es mínimo, puede alterar una medición de pocos segundos.

Para evitar que los resultados de las mediciones se vean comprometidos, además de comprobar la sincronización de todos los nodos antes de lanzar las pruebas, se toma el tamaño de problema mayor posible. Por este motivo trabajaremos con tamaños de mensajes muy grandes. Se trata de alcanzar cierto equilibrio: por una parte el tamaño debe ser lo mayor posible, pero por otro no debe ser excesivo, puesto que ninguno de los nodos debe quedarse sin memoria (no deben utilizar la memoria swap).

El estudio se ha realizado sobre las siguientes primitivas de envío y recepción:

MPI_Send / MPI_Recv: Envío y recepción punto a punto síncrono (con buffer). Estas primitivas copian temporalmente los mensajes a un buffer y devuelven el control al flujo del programa. El programa puede seguir su ejecución mientras se envía o recibe el mensaje e incluso puede escribir en la memoria del mensaje, al utilizar la primitiva la copia temporal. Tiene la ventaja de disminuir los tiempos ociosos en el programa, pero introduce la sobrecarga de la copia. Puesto que estas primitivas devuelven el control cuando hay garantía de que la operación se puede llevar a cabo con seguridad, también se denominan bloqueantes.

MPI_Isend / MPI_Irecv: Envío y recepción punto a punto asíncrono (no bloqueante, sin buffer). Estas primitivas devuelven el control al programa inmediatamente y llevan a cabo el envío o la recepción. El programa no debe usar los datos del mensaje mientras se produce la operación. En caso de necesitar acceder al mensaje, puede esperar invocando a la primitiva `MPI_Wait`.

MPI_Bcast: Primitiva de comunicación colectiva de movimiento de datos. Hace un broadcast del proceso raíz a todos los demás, de forma que los demás procesos reciben el mismo mensaje.

MPI_Gather: Primitiva de comunicación colectiva en la que un proceso raíz envía un mensaje individual y distinto a todos los procesos del grupo.

[4]En el cluster SOL, el nodo frontend es el servidor de tiempo del resto de los nodos, por lo que la sincronización que se consigue es muy buena.

MPI_Scatter: Primitiva de comunicación colectiva en la que un único proceso recoge los mensajes individuales del resto de procesos en el grupo. Es decir, cada proceso envía su mensaje al proceso raíz.

Para comparar las primitivas vamos a transmitir y recibir siempre el mismo número de mensajes. Así por ejemplo cuando intervengan los seis nodos, en una operación[5] de *broadcast* se transmitirán cinco mensajes. En la operación de envío y recepción (síncrono y asíncrono) se enviarán mensajes a todos los procesos desde el proceso raíz, de forma que también tengamos cinco mensajes. Igualmente en las operaciones de *Gather* y *Scatter* se transmitirá la misma cantidad de información. El código[6] en C que se ejecutará para cada primitiva se encuentra en las figuras 5.1, 5.2, 5.3, 5.4 y 5.5.

```
1 if (rank == 0) /* El master es 0, rank es el proceso actual */
2   for (i=1; i<p; i++) { /* p es el número de procesos */
3     MPI_Send(datos, longitud_datos, MPI_INT, i, tag,
          MPI_COMM_WORLD);
4   }
5 else /* Recepción de un mensaje desde el master */
6   MPI_Recv(datos, longitud_datos, MPI_INT, 0, tag, MPI_COMM_WORLD,
       &estado);
```

Figura 5.1: Primitivas síncronas `MPI_Send` y `MPI_Recv` usadas para envío uno-a-todos

Llevaremos a cabo mediciones con tres tamaños de problema N (por N nos referimos a la longitud o número de elementos de los mensajes, que será de 10^7, $2 \cdot 10^7$ y $4 \cdot 10^7$ elementos). Cada elemento es un número entero de 64 ó 32 bits según la arquitectura. Para las operaciones de *Gather* y *Scatter* el valor de N se refiere al valor de una comunicación individual, lógicamente los datos de los p procesos tendrán pN elementos. Las mediciones se llevan a cabo con la red desocupada y cada medición se repite diez veces antes de obtener la media.

Puesto que en todas las mediciones de primitivas se transmite la misma cantidad de información, en caso de que la red tenga un buen rendimiento, se esperan resultados con similitudes en todas las mediciones. Los resultados para el tamaño mayor ($4 \cdot 10^7$ elementos) se muestra en la figura 5.6. El gráfico

[5]Por operaciones de *broadcast* nos referimos a aquellas que usan la primitiva de `MPI_Bcast`. Así mismo, por simplicidad, las operaciones de *Gather* y *Scatter* corresponden a las primitivas `MPI_Gather` y `MPI_Scatter`

[6]El código se ha simplificado. La variable `datos_colectivos` tiene el tamaño de `longitud_datos` * `p`, con `p` en número de procesos que intervienen

```
1 if (rank == 0) /* El master es 0, rank es el proceso actual */
2   for (i=1; i<p; i++) { /* p es el número de procesos */
3     MPI_Isend(datos, longitud_datos, MPI_INT, i, tag,
          MPI_COMM_WORLD, &peticion);
4   }
5 else { /* Recepción de un mensaje desde el master */
6   MPI_Irecv(datos, longitud_datos, MPI_INT, 0, tag, MPI_COMM_WORLD,
        &peticion);
7   MPI_Wait(&peticion, &estado);
8 }
```

Figura 5.2: Primitivas asíncronas `MPI_Isend` y `MPI_Irecv` usadas para envío uno-a-todos

```
1 MPI_Bcast(datos, longitud_datos, MPI_INT, 0, MPI_COMM_WORLD); /*
    Master es 0 */
```

Figura 5.3: Primitiva de *broadcast* `MPI_Bcast` usada para envío uno-a-todos

```
1 MPI_Gather(datos, longitud_datos, MPI_INT, datos_colectivos,
    longitud_datos, MPI_INT, 0, MPI_COMM_WORLD); /* Master es 0 */
```

Figura 5.4: Primitiva `MPI_Gather` usada para enviar un mensaje distinto a cada proceso

```
1 MPI_Scatter(datos_colectivos, longitud_datos, MPI_INT, datos,
    longitud_datos, MPI_INT, 0, MPI_COMM_WORLD); /* Master es 0 */
```

Figura 5.5: Primitiva `MPI_Scatter` usada para recibir un mensaje distinto de cada proceso

para este tamaño es representativo de los tamaños menores, que se dejan en el anexo. Como puede verse, todas las operaciones tienen un rendimiento parecido. Puesto que la topología de red del cluster SOL es muy sencilla se puede concluir que la red da un buen rendimiento al trabajar con MPI, ya que las operaciones de envío y recepción tardan prácticamente el mismo tiempo que la operación optimizada para el *broadcast*. Tan sólo cuando se trabaja con cinco y seis nodos la primitiva de *broadcast* da un rendimiento superior a la implementación directa con primitivas de envío y recepción, es decir, cuanto mayor es el tráfico de la red, más convienen usar el *broadcast* en lugar de la implementación directa.

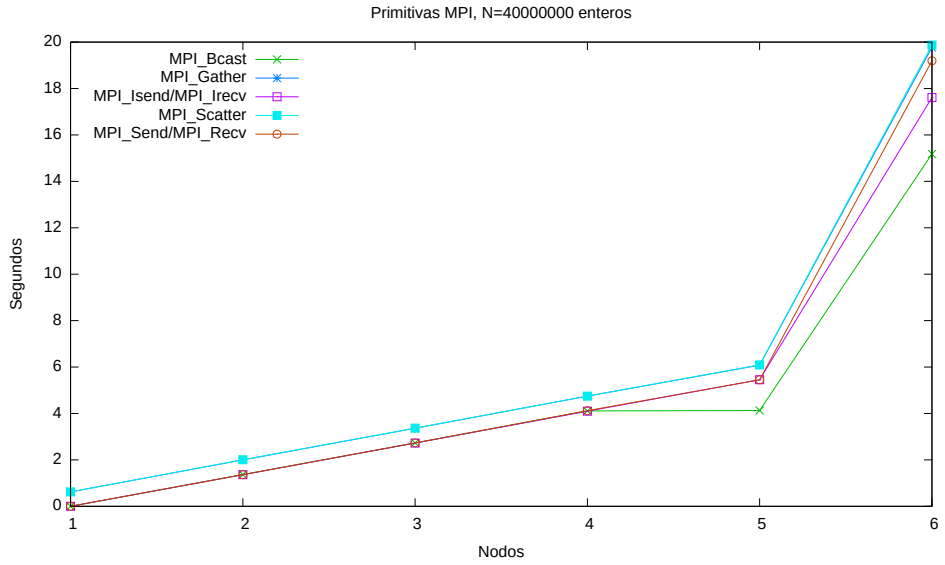

Figura 5.6: Tiempo de ejecución de las diferentes primitivas MPI para un tamaño de mensaje de $4 \cdot 10^7$ enteros. Cada nodo ejecuta un proceso. El eje de abcisas indica el número de nodos que intervienen, siguiendo el orden habitual en el que el primer nodo es SOL y el último el nodo de 32 bits

La primitiva *Scatter* es más lenta que el resto; si nos fijamos en la figura 5.6 cuando se trabaja con un sólo proceso, vemos que la primitiva tarda más tiempo cuando no se establece comunicación con otros procesos, por lo que se deduce que esta rutina tiene un coste de inicialización mayor. Sin duda el aspecto más destacable de la gráfica es el gran incremento de tiempo que se produce al trabajar con el nodo de 32 bits. El Pentium III, al tener un procesador de menor rendimiento y menos memoria RAM que los demás nodos, marca la velocidad de la operación en conjunto, por lo que cualquier cómputo

paralelo que abarque también a este nodo puede verse muy penalizado, con lo que según el caso, puede que interese utilizar o no este nodo para resolver cada problema concreto (esta decisión queda dentro del ámbito de trabajo de la computación heterogénea).

5.3. Multiplicación de matrices

Si usamos las primitivas de MPI para implementar las multiplicación de matrices $C = AB$, distribuyendo la matriz A a través de *broadcast* y usando *Scatter* para repartir la matriz B en bloques de columnas consecutivas, conseguimos que la multiplicación[7] que se lleve a cabo sea menor. En particular se hará la multiplicación $C_i = AB_i$ siendo el índice i la submatriz correspondiente en cada nodo. El resultado (C_i) se envía hasta el proceso raíz a través de *Gather*. En nuestras experimentaciones medimos el tiempo que se tarda en calcular la multiplicación de matrices en conjunto, para ello se usarán barreras (`MPI_Barrier`) que garantizan que todos los nodos comiencen el cálculo al mismo tiempo y se mida el tiempo cuando todos los nodos hayan finalizado.

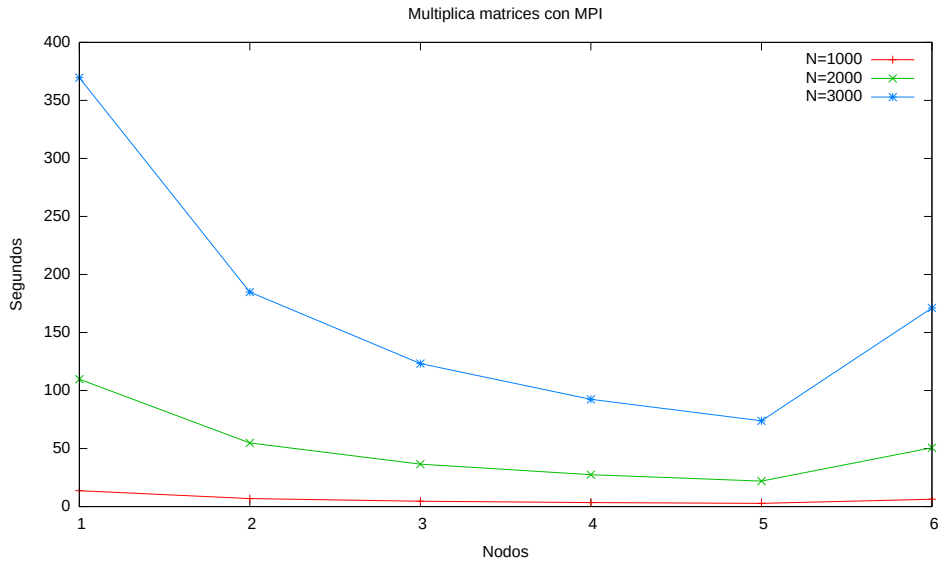

Figura 5.7: Tiempo de ejecución de la multiplicación de matrices con MPI para distintos tamaños de matriz y desde uno a seis nodos, donde cada nodo ejecuta un proceso.

[7]La multiplicación se hará con BLAS-3. El estudio de BLAS se verá en el siguiente capítulo

Los experimentos se harán con matrices cuadradas con $N = 1000$, 2000 y 3000. Los resultados de la experimentación se muestran en la figura 5.7. Como se observa en dicha figura, el rendimiento mejora al usar más nodos, excepto al introducir el nodo Pentium III, en el que se ve que el rendimiento se degrada hasta cerca del nivel en el que hay sólo dos nodos.

p	Speedup, $S(n,p)$	Eficiencia, $E(n,p)$	Coste, $C(n,p)$
2	1.9997	0.9998	369.6331
3	2.9993	0.9997	369.6666
4	3.9993	0.9998	369.6377
5	4.9972	0.9994	369.7865
6	2.1599	0.3599	1026.634

Tabla 5.1: *Speedup*, eficiencia y coste obtenidos experimentalmente de la multiplicación de matrices con MPI con $n = 3000$. El valor del tiempo secuencial es $t(n) = 369{,}5808$ segundos

Los resultados se pueden analizar mejor calculando los valores experimentales del *speedup*, la eficiencia y el coste. Estos cálculos se realizan para el tamaño más grande, es decir, para $N = 3000$ y se muestran en la tabla 5.1. El valor del tiempo secuencial se toma cuando se usa sólo el nodo SOL, es decir, sólo hay un proceso en el que se lleva a cabo la multiplicación de matrices. Dicha tabla refleja la calidad de los resultados cuando no tenemos en cuenta el nodo de 32 bits. El *speedup* es prácticamente lineal, la eficiencia es casi igual a uno, y el coste se acerca al valor del tiempo secuencial ($t(n) = 369{,}5808$ segundos). Sin embargo al introducir el Pentium III el rendimiento se degrada aproximadamente un tercio de la tendencia obtenida cuando sólo se usan procesadores Intel Xeon de 64 bits, por lo que tal como adelantabamos en la anterior sección, para este problema concreto conviene no usar el nodo 5.

5.4. Conclusiones

Del estudio de las primitivas podemos observar que la operación *scatter* tiene un coste de inicialización mayor que el resto.

Del estudio de la multiplicación de matrices obtenemos que si el algoritmo puede distribuir equitativamente su cómputo entre nodos homogéneos, el *speedup* conseguido es cercano al óptimo.

Al trabajar con arquitecturas heterogéneas hemos aprendido cómo los ficheros de esquemas facilitan el trabajo. Al combinar máquinas de velocidades

muy distintas el rendimiento se ve degradado, por lo que en el caso particular del cluster SOL, no es recomendable utilizar el nodo5.

Capítulo 6

Prestaciones de BLAS

BLAS (Basic Linear Algebra Subprograms) es una biblioteca de funciones que llevan a cabo operaciones básicas con matrices y vectores. Entre sus ventajas destacan su gran eficiencia en la resolución de problemas de álgebra lineal, su amplia difusión, siendo en la actualidad un estándar de facto para operaciones elementales entre vectores y matrices. También aporta ventajas al código, como mayor claridad, portabilidad, modularidad y en consecuencia mayor facilidad para el mantenimiento del software implementado con BLAS [8].

Las bibliotecas de BLAS únicamente incluyen funciones para el procesamiento secuencial de datos. Nuestro objetivo en este caso es desarrollar un conjunto de herramientas que permitan comparar entre distintas implementaciones y distintos niveles de BLAS. Las implementaciones de BLAS de las que disponemos son las siguientes:

Reference: Es la implementación de referencia de BLAS para C. Está basada en la biblioteca escrita originalmente en Fortran77 [24]..

ATLAS: Es una implementación que proviene del proyecto de investigación ATLAS (Automatically Tuned Linear Algebra Software), y cuyo esfuerzo se centra en la aplicación de técnicas empíricas con el fin de proporcionar el mejor rendimiento. Durante la instalación de esta implementación de BLAS, se llevan a cabo extensas pruebas de rendimiento para obtener una implementación óptima para la máquina en la que se compila [25].

Threaded-ATLAS: Es la versión de ATLAS mejorada para usar hilos si encuentra código paralelizable.

Goto: Es una implementación mantenida por el Centro de Computación

Avanzada de Texas [26]. Al igual que Threaded-ATLAS, tiene soporte de hilos

Las rutinas de BLAS se encuentran clasificadas en tres niveles atendiendo al tipo de operación que implementan:

Nivel 1: Son las operaciones entre vectores, con complejidad en número de operaciones y en memoria $O(n)$ (n es la longitud de los vectores). Entre estas operaciones se encuentran el producto escalar de vectores o la suma de un vector con un múltiplo de otro vector.

Nivel 2: Engloban las operaciones Matriz-Vector, usadas frecuentemente en algoritmos de álgebra lineal. Son operaciones de orden $O(n^2)$ sobre $O(n^2)$ datos.

Nivel 3: En este nivel están las operaciones de orden $O(n^3)$ y que acceden a un total de datos de orden $O(n^2)$, es decir, las operaciones Matriz-Matriz. Estas rutinas suelen estar optimizadas mediante la división de matrices en bloques.

El conjunto de pruebas que se ha realizado se basa en la multiplicación de dos matrices realizando esta operación directamente, llamando a las rutinas de nivel 1 de BLAS, de nivel 2 o bien de nivel 3. Cada prueba se repite cinco veces[1] y se toma la media. Los tamaños de matrices escogidos son de 1000, 2000 y 3000[2].

6.1. Multiplicación de matrices sin BLAS

Antes de medir el rendimiento de las distintas implementaciones de BLAS y de sus distintos niveles, hemos de tener una referencia comparativa respecto a la propia máquina donde se van a ejecutar las mediciones. Para obtener esta referencia mediremos el tiempo de ejecución de la implementación sin BLAS del algoritmo de multiplicación de matrices. Una vez obtenidos los tiempos de la multiplicación de matrices, haremos una segunda medición consistente en la multiplicación por bloques (sin BLAS).

[1] Tanto el número de ejecuciones como los tamaños de las matrices son configurables mediante las variables `IT` e `N` respectivamente

[2] Los programas se compilarán con las opciones `-march=native` y `-O2` de `gcc`, aunque el impacto del compilador es pequeño ya que el núcleo de la computación lo lleva a cabo la biblioteca de BLAS. Las bibliotecas también se ha compilado con estas opciones y con `gcc`, que es el compilador por defecto en las distribuciones de Linux.

Estas dos referencias nos permiten comparar BLAS con los algoritmos sin BLAS; de esa forma sabremos cuánto de eficiente es BLAS en la máquina de pruebas (SOL). Para la multiplicación por bloques hemos de determinar cuál es el tamaño óptimo de bloque. Dicho tamaño optimiza el uso de caché minimizando el acceso a la memoria RAM a la hora de leer los operandos y escribir los resultados. Aunque el tamaño de la caché es conocido[3], la organización de la misma (por conjuntos, asociativa, dividida entre instrucciones y datos, etc.), impide determinar de forma teórica el tamaño de bloque óptimo con exactitud. Para determinar el tamaño óptimo de bloque, probaremos con los múltiplos de diez que dividan a los tamaños de matriz 1000, 2000 y 3000. En la prueba para determinar el tamaño del bloque haremos una sola ejecución, en lugar de varias y hacer la media, puesto que el objetivo no es tener una medida de precisión, sino saber cuál de todos los tamaños de bloque da mejor resultado.

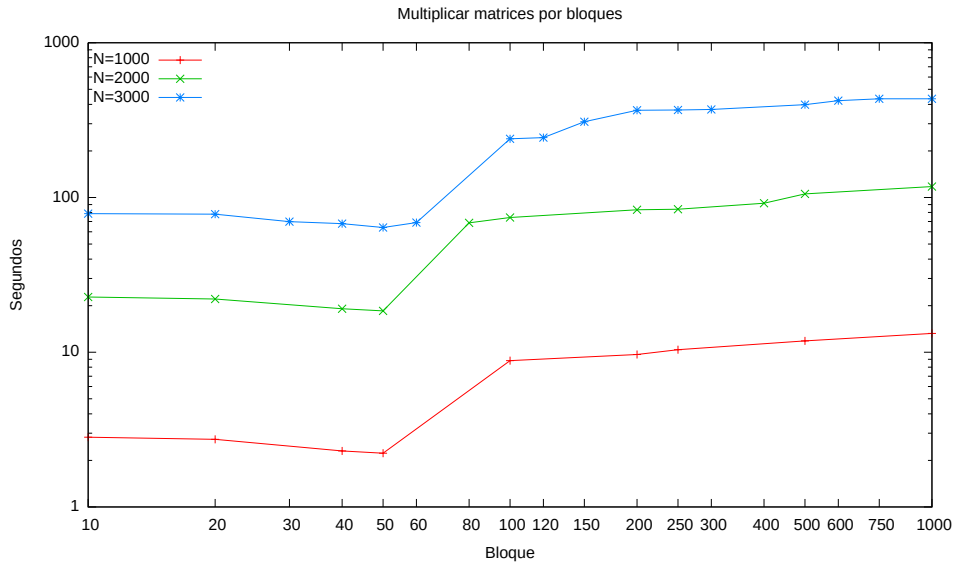

Figura 6.1: Determinación del tamaño óptimo de bloque

En todas las pruebas se aprecia (figura 6.1) que el tamaño óptimo de bloque es de 50, por lo que las pruebas de multiplicación por bloques las realizaremos dividiendo la matriz en submatrices de 50×50 elementos.

En la tabla 6.1 se muestran los resultados promedios de cada una de las pruebas. La mejora de la multiplicación directa con bloques de 50×50 respecto a la multiplicación directa (sin bloques) es notable. Como se

[3] 2 MB en SOL

	N		
	1000	2000	3000
Directa	12.1973128	114.1790128	413.6524680
Bloques	2.2248490	18.4799294	64.0280592

Tabla 6.1: Tiempo en segundos de la multiplicación de matrices directa (sin BLAS) para distintos tamaños de matriz cuadrada

observa, obtenemos que la multiplicación por bloques es aproximadamente 6.4 veces más rápida que la multiplicación directa. Las prestaciones medidas en megaflops[4] arrojan una cifra de 130.54 megaflops para la multiplicación directa y de 843.38 si usamos bloques.

6.2. Multiplicación llamando a BLAS-1

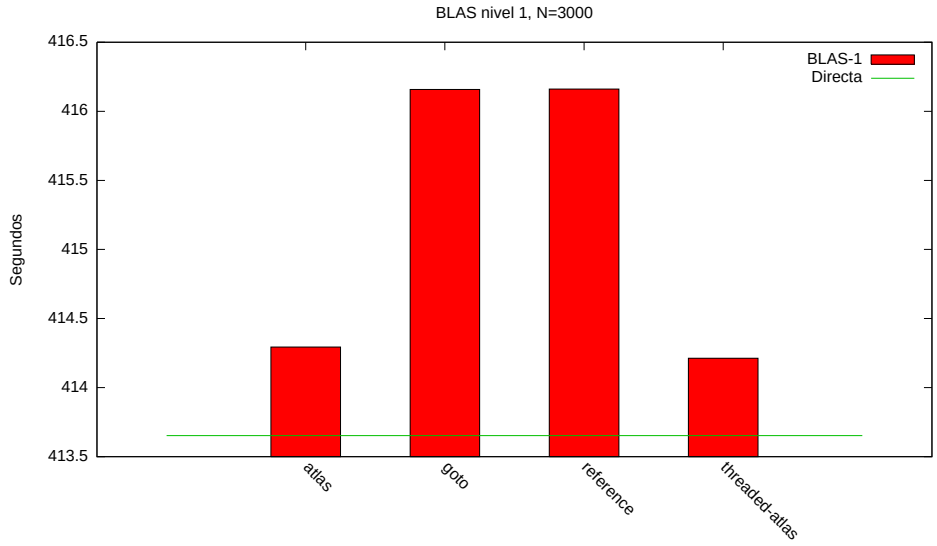

Figura 6.2: Comparativa del tiempo de ejecución de la multiplicación de matrices implementada con distintas versiones de BLAS y con las llamadas a las rutinas de nivel 1

Puesto que tenemos una referencia de multiplicación sin BLAS, el siguiente paso es determinar el rendimiento de la multiplicación llamando a

[4]Calculadas a partir de la matriz de 3000×3000 elementos

Reference	Goto	ATLAS	Threaded-ATLAS
129.757	129.758	130.342	130.367

Tabla 6.2: Megaflops de la multiplicación de matrices con BLAS de nivel 1

las bibliotecas de BLAS de nivel 1. De igual forma, se pretende también comparar entre las distintas implementaciones de BLAS. En la figura 6.2 se muestra la comparación de las distintas implementaciones para tamaño de matriz 3000×3000 (el comportamiento para el tamaño de 1000 es similar, y para 2000 es distinto[5]). En la tabla 6.2 se muestran los megaflops obtenidos para cada implementación.

En general podemos concluir que todas las implementaciones de BLAS para el nivel 1 dan un rendimiento similar[6], ligeramente peor que la multiplicación directa, posiblemente debido a la sobrecarga de las llamadas a las funciones de biblioteca. Más adelante, en el apartado 6.6 expondremos más conclusiones en base a resultados mostrados gráficamente.

6.3. Multiplicación llamando a BLAS-2

Las operaciones de nivel 2 de BLAS son de tipo matriz-vector. Usando estas operaciones para implementar la multiplicación de matrices se obtienen los resultados de la figura 6.3. Esta figura muestra la multiplicación de matrices de 3000×3000 elementos (para los tamaños 1000 y 2000 los resultados comparativos son semejantes). La tabla 6.3 muestra los megaflops obtenidos.

Resulta claro que las implementaciones de Goto y Reference se acercan al rendimiento de la implementación directa por bloques de la multiplicación de matrices. Aún así, estas dos implementaciones obtienen una aceptable disminución del tiempo de ejecución con respecto a la implementación directa.

Por otra parte las implementaciones de ATLAS y Threaded-ATLAS, dan un resultado cercano a la multiplicación por bloques sin BLAS. En ningún caso se logra mejorar el rendimiento de la multiplicación por bloques, ya que las pruebas de la multiplicación por bloques se hicieron agrupando las operaciones en bloques de submatrices y con el tamaño óptimo de bloque.

[5]En el anexo se muestra en detalle para el tamaño 2000, no obstante, aunque muestre unos resultados distintos, no se consideran significativos, al diferenciarse cada una de las implementaciones tan sólo en décimas de segundo.

[6]Nótese que pese al tamaño de la matriz de 3000×3000 elementos, entre una implementación y otra, tan sólo hay una diferencia de dos segundos.

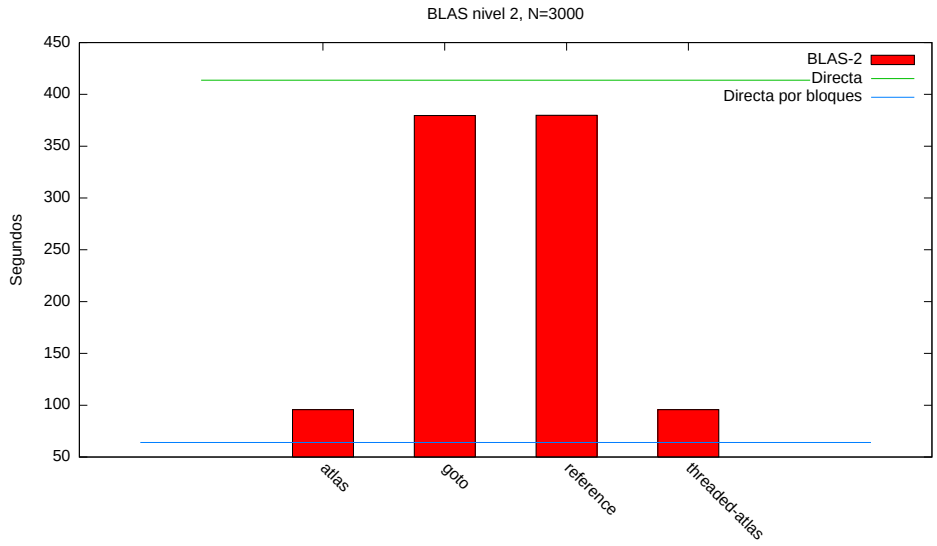

Figura 6.3: Comparativa del tiempo de ejecución de la multiplicación de matrices implementada con distintas versiones de BLAS y con las llamadas a las rutinas de nivel 2

Reference	Goto	ATLAS	Threaded-ATLAS
142.142	142.255	563.868	563.872

Tabla 6.3: Megaflops de la multiplicación de matrices con BLAS de nivel 2

6.4. Multiplicación llamando a BLAS-3

En el nivel 3 de BLAS las rutinas están optimizadas mediante el uso de bloques de submatrices, siendo además los tamaños de bloques apropiados para operaciones matriciales. Por esta razón los tiempos de ejecución tenderán a ser mejores que mediante el uso de rutinas de los otros niveles, tal y cómo se aprecia en la figura 6.4. Nótese que esta figura se representa con el tiempo en escala logarítmica. En estas mediciones, todas las implementaciones de BLAS mejoran sustancialmente a la multiplicación directa. Las implementaciones de Goto y Reference no son tan rápidas como la multiplicación directa por bloques. Por contra, las implementaciones de ATLAS y Threaded-ATLAS mejoran la multiplicación por bloques, e incluso la versión Threaded de ATLAS mejora la versión sin soporte de hilos. En la tabla 6.4 también se aprecia con claridad el aumento de los megaflops que se obtiene con Threaded-ATLAS, en este caso el soporte de hilos supone una mejora de 3.6 veces respecto a la versión de ATLAS sin hilos. Esta mejora es significativa puesto que el nodo SOL dispone de cuatro cores.

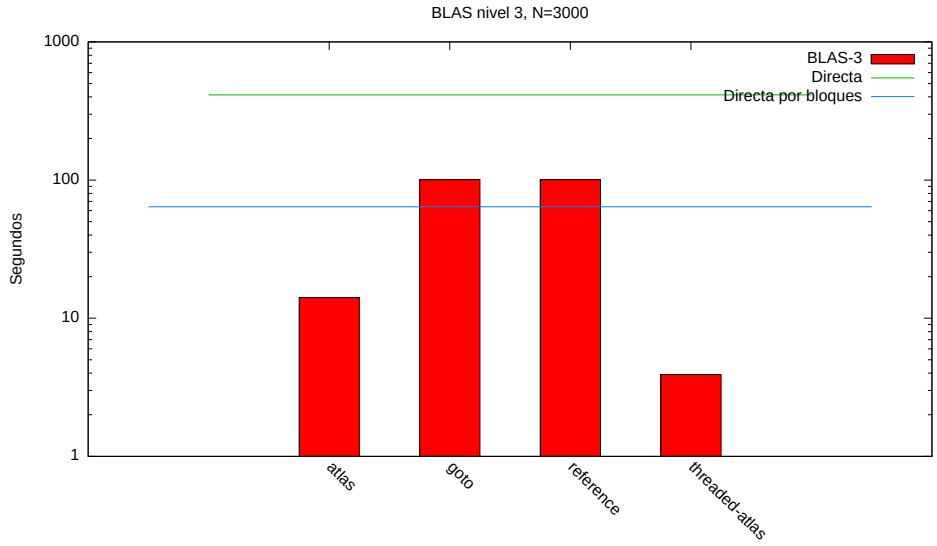

Figura 6.4: Comparativa del tiempo de ejecución de la multiplicación de matrices implementada con distintas versiones de BLAS y con las llamadas a las rutinas de nivel 3

Reference	Goto	ATLAS	Threaded-ATLAS
536.887	536.949	3831.265	13806.020

Tabla 6.4: Megaflops de la multiplicación de matrices con BLAS de nivel 3

6.5. Experimentación en el nodo 5

En la web de la implementación de Goto [26], se afirma que esta implementación es la más rápida. Los experimentos llevados a cabo demuestran que en nodo SOL la implementación más rápida es la de Threaded-ATLAS. No obstante la arquitectura de este nodo: *Intel Xeon* de 64 bits, no está soportada oficialmente en la implementación de Goto. Queremos determinar si el rendimiento de BLAS Goto está determinado por la arquitectura o por otros factores.

Llevamos a cabo las mismas pruebas con el nodo 5, equipado con dos procesadores *Pentium III* de 32 bits, con tamaños de matriz cuadrada de 500, 1000 y 1500. Encontramos que el tamaño óptimo de bloque para el tamaño 1000 es 40 y para los demás tamaños es 50. Por brevedad sólo mostraremos en la figura 6.5 el tamaño de matriz 1500×1500, donde se aprecia que el comportamiento es comparativamente similar al del nodo con Intel Xeon.

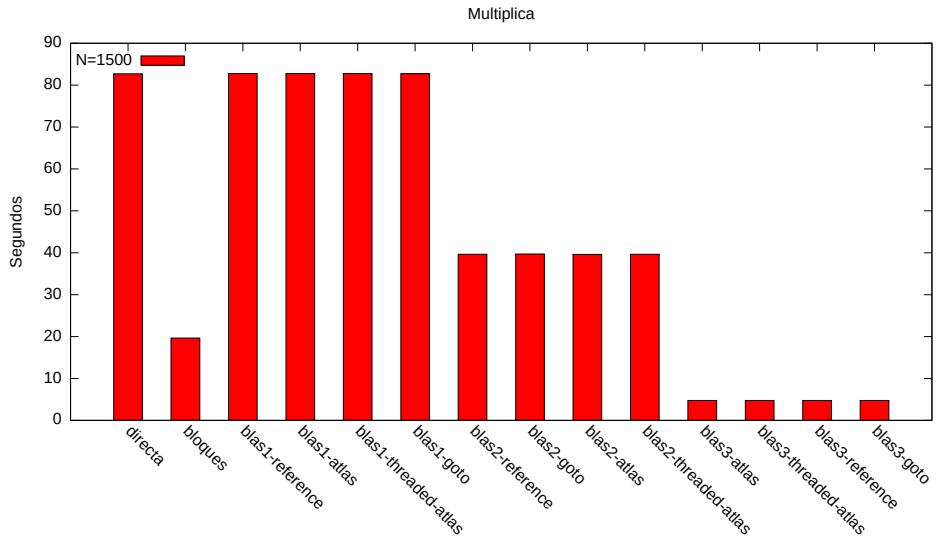

Figura 6.5: Comparación de multiplicaciones para el nodo 5

Puesto que la arquitectura del nodo5 sí está soportada oficialmente por

Goto, podemos asegurar que Goto no es la implementación más rápida en esta prueba en particular. Posiblemente sea debido a que Goto centra los esfuerzos de la optimización en las matrices rectangulares y no tanto en las cuadradas. No obstante existen otros benchmark [27] [28], independientes de los realizados por el autor de Goto, que dan la razón a este, es por este motivo por cual nos inclinamos a pensar que en otros tipos de operaciones con BLAS, la implementación de Goto debería mostrar resultados mejores.

6.6. Conclusiones sobre BLAS

Al englobar las distintas pruebas del nodo SOL en un sólo gráfico (figura 6.6) podemos extraer las siguientes conclusiones generales:

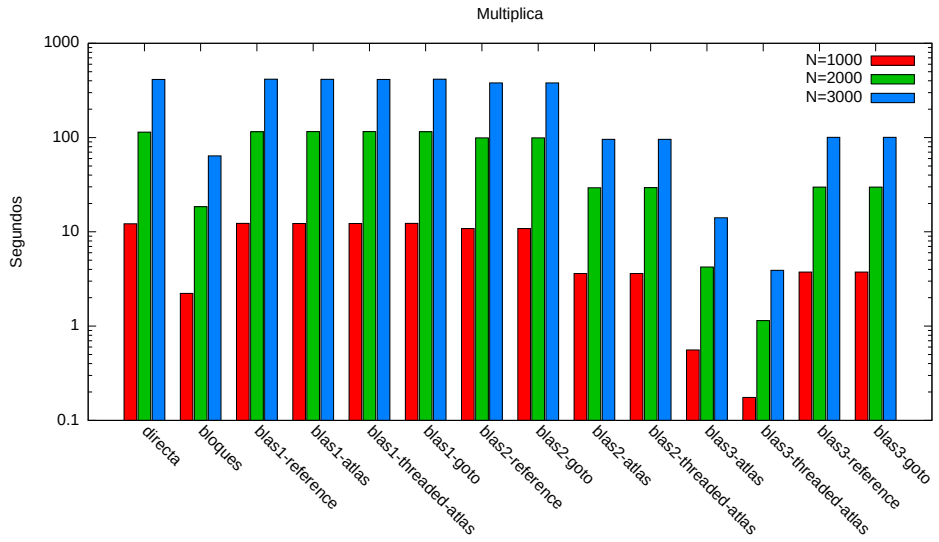

Figura 6.6: Comparación de diversas implementaciones de multiplicación de matrices, directa, por bloques y con los tres niveles de BLAS. Se muestran tres tamaños de matriz, y el tiempo en escala logarítmica

- En el nivel 1 de BLAS, ninguna implementación destaca sobre las demás. En todos los casos el rendimiento es similar a la multiplicación de matrices directa.
- La multiplicación directa por bloques, con el tamaño óptimo de bloque, supera a todas las implementaciones con BLAS de nivel 2, aunque el rendimiento de Threaded-ATLAS y ATLAS es semejante.

- Todas las implementaciones de BLAS de nivel 3 están cercanas a la multiplicación por bloques. Sin embargo, sólo las implementaciones de Threaded-ATLAS y ATLAS obtienen menor tiempo de ejecución.

- El mejor rendimiento de la multiplicación de matrices se obtiene con Threaded-ATLAS de BLAS de nivel 3, ya que esta implementación consigue sacar mayor rendimiento al hacer uso de varios cores por procesador.

Queda demostrado experimentalmente que las extensas pruebas de rendimiento que hace la implementación de ATLAS durante su instalación consiguen obtener el mejor rendimiento, que se ve superado si es posible sacar provecho de los hilos (Threaded-ATLAS).

Capítulo 7

Prestaciones de LAPACK

LAPACK (Linear Algebra PACKage) es una biblioteca que, basándose en BLAS, resuelve problemas de más alto nivel. LAPACK está compuesto de tres tipos de rutinas: según el tipo de problemas que resuelven y de mayor a menor nivel:

Rutinas conductoras: Resuelven problemas concretos, tales como un sistema de ecuaciones lineales o el cálculo de valores propios de una matriz simétrica. Estas rutinas hacen uso de las rutinas computacionales.

Rutinas computacionales: Resuelven tareas como factorizaciones de matrices, reducción de una matriz a la forma diagonal, etc.

Rutinas auxiliares: Son rutinas de más alto nivel que BLAS, pero de más bajo nivel que las rutinas conductoras y computacionales. Realizan subtareas de los algoritmos orientados a bloques.

LAPACK usa al máximo los núcleos computacionales de BLAS (normalmente de BLAS de nivel 3). En el cluster SOL disponemos de las versiones de LAPACK preparadas para usar las implementaciones de ATLAS y Reference. Del estudio de prestaciones de BLAS vimos que estas dos implementaciones son representativas de un rendimiento optimizado y sin optimizar. En este caso el objetivo es tener una medida del rendimiento de ambas implementaciones en SOL.

Los experimentos se basarán en la factorización LU [29]. Dicha factorización pertenece a las rutinas computacionales. Se trata de una rutina que descompone una matriz cuadrada A como el producto de una matriz triangular inferior L y una matriz triangular superior U. La descomposición $A = LU$, facilita enormemente la resolución de sistemas de ecuaciones lineales. El estudio experimental medirá la versión LAPACK que usa BLAS de Reference y la versión de ATLAS, usando dos posibles llamadas a funciones:

dgetf2 Algoritmo de factorización LU (sin bloques).

dgetrf Algoritmo mejorado, trabaja con bloques.

Los elementos de la matriz A se generaran a partir de la siguiente igualdad:

$$a_{ij} = \begin{cases} min(i,j) - 1 & \text{si } i \neq j \\ i & \text{si } i = j \end{cases}$$

Esto generará matrices densas. Cada medida será la media de cinco ejecuciones, que se ejecutarán para tamaños de matriz cuadrada desde 500 a 4000 elementos, en incrementos de 100[1]. Se mide sólo el tiempo de la llamada a la función `dgetf2` o `dgetrf`. Las mediciones detalladas se dejan en el anexo.

7.1. Factorización LU sin bloques

Tras las distintas mediciones de tiempos, variando el tamaño de la matriz A, podemos trazar la gráfica que se muestra en la figura 7.1. Los resultados confirman que la complejidad de la factorización LU es $\frac{2}{3}n^3k$ (con valores cercanos para la constante k) tanto en la implementación de Reference como en la de ATLAS.

Al observar el tiempo de ejecución se ve claramente que, al igual que ocurría con BLAS, la implementación de ATLAS es siempre mejor. No obstante la diferencia no es muy pronunciada (haciendo la media ponderada se obtiene una mejora del 5,8 % de ATLAS respecto a Reference). Esto es debido a que al tratarse de una rutina sin uso de bloques, la versión de ATLAS apenas admite margen de maniobra en cuanto a su optimización para una plataforma particular.

7.2. Factorización LU con bloques

En la factorización con bloques se observa el mismo comportamiento general que en la versión sin bloques: crecimiento $\frac{2}{3}n^3k$ y menor tiempo de ejecución en la implementación de ATLAS. En este caso el valor de k es notablemente inferior para ATLAS.

Como se muestra en la figura 7.2, la mejora de la implementación de ATLAS respecto a la de Reference es muy notable. En particular tomando

[1] Al igual que con las pruebas de BLAS, los programas se compilarán con las opciones `-march=native` y `-O2` del compilador habitual de Linux `gcc`. Al trabajar con bibliotecas, el impacto del compilador en el programa es mínimo.

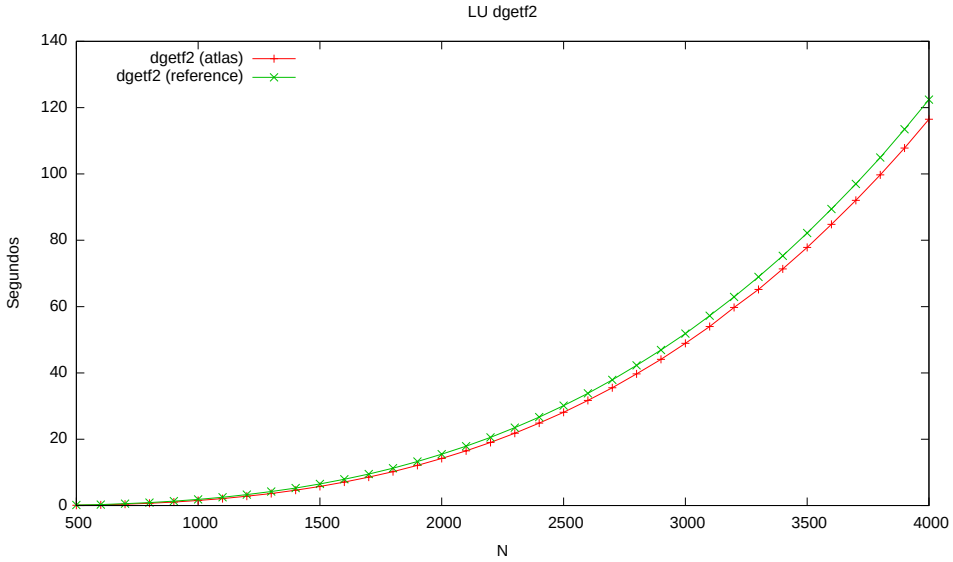

Figura 7.1: Comparación de implementaciones de ATLAS en la factorización LU sin bloques, para distintos tamaños de matriz

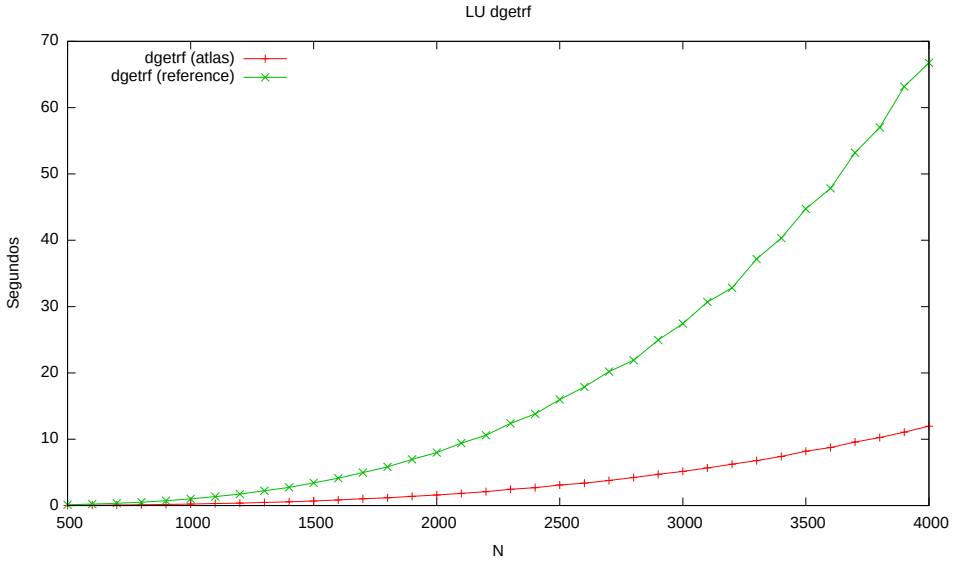

Figura 7.2: Comparación de implementaciones de ATLAS en la factorización LU con bloques, para distintos tamaños de matriz

la media ponderada de ambas implementaciones y comparándolas, concluimos que ATLAS es 5.44 veces más rápido que Reference en el cálculo de la factorización LU con LAPACK.

7.3. Conclusiones de la factorización LU

En los anteriores apartados hemos comparado las implementaciones de Reference y ATLAS, obteniendo en esta última el mejor rendimiento. Si comparamos la factorización con y sin bloques, obtenemos siempre el mejor rendimiento al usar bloques (figura 7.3). El cálculo de los megaflops (tabla 7.1) confirma este comportamiento.

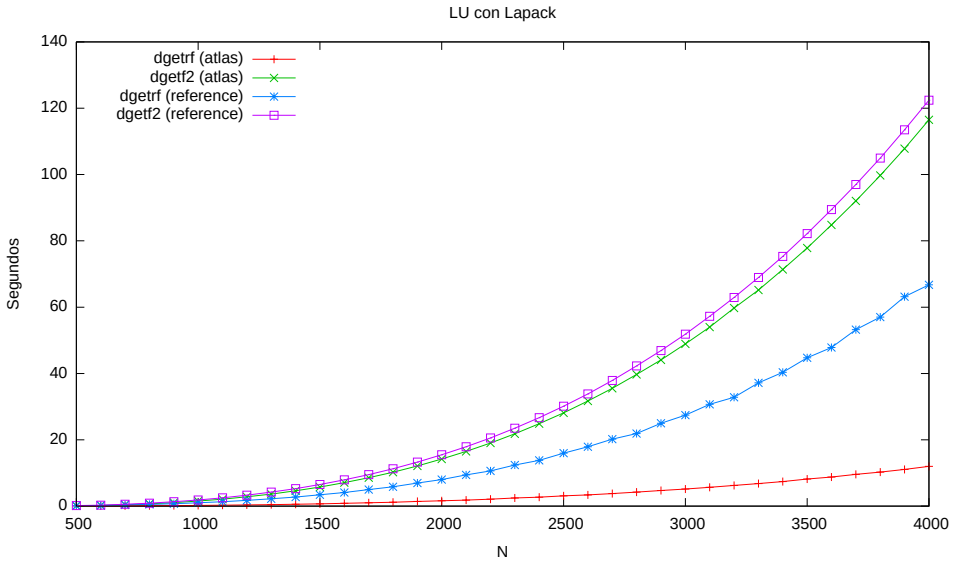

Figura 7.3: Comparación de tiempos de ejecución para la factorización LU con y sin bloques, en distintas implementaciones y para distintos tamaños

Como conclusión de las pruebas en SOL se confirma lo que era de esperar: la llamada `dgetrf` es más eficiente que la versión sin bloques. Con la implementación de ATLAS se obtienen resultados claramente mejores ya que al usar bloques ATLAS puede aportar sus optimizaciones en la gestión de estos.[2]

[2]Entre la prueba más lenta (Reference sin bloques) y la más rápida (ATLAS con bloques) hay una diferencia de tiempo de factor 10.1.

dgetf2		dgetrf	
Reference	ATLAS	Reference	ATLAS
348.521	366.262	639.181	3564.466

Tabla 7.1: Megaflops de la factorización LU con LAPACK calculado a partir del tamaño más grande ($N = 4000$)

Capítulo 8

Conclusiones y vías futuras

8.1. Conclusiones

En este proyecto hemos desarrollado una herramienta que permite obtener conclusiones sobre el rendimiento de software que podemos encontrar en clusters de computación científica. Dicha herramienta se ha diseñado para que sea fácil de ejecutar en diversos clusters. Como prueba de su correcto funcionamiento y de que se puede realizar un estudio de prestaciones, hemos aplicado dicha herramienta al cluster SOL.

Las herramientas desarrolladas miden el rendimiento de cuatro tecnologías: OpenMP, MPI, BLAS y LAPACK. El estudio del rendimiento de OpenMP con distintos compiladores supone un aspecto innovador, ya que en el momento de empezar este proyecto, apenas habían pasado unos pocos meses desde que `gcc` dio soporte a OpenMP. El estudio de MPI actualiza otros proyectos que se realizaron hace más de una decada [30] [9] y aporta la novedad de mezclar arquitecturas heterogéneas. El estudio de BLAS y LAPACK también aporta el aspecto de comparar distintas implementaciones.

Si nos centramos en el conocimiento que la herramienta desarrollada nos ha permitido tener del cluster SOL, nos quedamos con que, al trabajar con OpenMP, el compilador de Intel es bueno cuando hay hasta un hilo por core, y en los demás casos el compilador `gcc` es la mejor opción. Al trabajar con MPI hemos aprendido a trabajar con ficheros de esquemas, y hemos visto que un sólo ordenador con rendimiento inferior puede ralentizar todo el cluster. Con BLAS vemos que Threaded-ATLAS debe ser la implementación a usar por defecto en SOL, e igualmente con LAPACK preferimos el uso de ATLAS.

La herramienta desarrollada no sólo ha sido útil para automatizar los experimentos, sino que también aporta una visión gráfica de los resultados que se revela tremendamente efectiva para el análisis de los mismos. Además

se complementa con *scripts* que generan documentación directamente, e incluso que calculan parámetros como megaflops, *speedup*, eficiencia y coste, evitando que el investigador tenga que realizar a mano todos los cálculos.

Tanto las herramientas desarrolladas, como los resultados del estudio, así como los manuales incluidos en este proyecto son de utilidad para el Grupo de Computación Paralela de la Universidad de Murcia [7].

8.2. Vías futuras

Podemos distinguir las siguientes vias futuras:

- El paso más inmediato para el futuro sería aplicar las herramientas desarrolladas a otros clusters de computación científica. Esto permitiría comparar el comportamiento del cluster SOL con otros, y determinar si las conclusiones obtenidas son generalizables o no a una mayoría de clusters, lo cual tiene interés para el Grupo de Computación Paralela de la Universidad de Murcia. Un candidato es el cluster Hipatia (del servicio de cálculo de la UPCT), del cual es posible obtener una cuenta a través de una petición vía web [31].

- Otra vía futura sería incorporar otras tecnologías al conjunto de herramientas, como por ejemplo ScaLAPACK [32]. La incorporación de ScaLAPACK daría lugar a otro subconjunto de herramientas (BLACS y PBLAS). BLACS (Basic Linear Algebra Communication Subprograms) proporciona facilidad de uso y portabilidad para la comunicación en problemas de álgebra lineal, usando un mecanismo de paso de mensajes, como podría ser MPI. Por otra parte PBLAS (Parallel Basic Linear Algebra Subprograms) hace las veces de BLAS paralelo, lo cual permite que sea usado por ScaLAPACK (Scalable LAPACK) de la forma análoga a la que BLAS se usaba por LAPACK.

- Como todo sofware desarrollado, es posible la mejora. Bien por adaptación a nuevas características que puedan surgir en el futuro, o bien por que se añadan nuevas funcionalidades.

- El objetivo de este proyecto ha sido el desarrollo de herramientas tanto para la medición, como para en análisis gráfico. También se han añadido otros *scripts* de ayuda. Una posible vía futura sería la de combinar los *scripts* para generar documentación más elaborada, tal como documentos completos en LaTeX, y no sólo fragmentos como se hace en la actualidad.

- También es posible añadir otros programas para la medición. Es este proyecto nos hemos centrado en primitivas y algoritmos bien conocidos como la multiplicación de matrices o la factorización LU. Sería posible añadir otros si se determinara que su análisis puede ser interesante.

Todas estas vías quedan abiertas para posibles proyectos fin de carrera de Ingeniería en Informática.

Bibliografía

[1] The OpenMP API specification for parallel programming `http://openmp.org/`.

[2] MPI: A Message-Passing Interface Standard, `http://www.mpi-forum.org/docs/mpi21-report.pdf`.

[3] Curso de herramientas informáticas para computación científica - Librerías matriciales - Visión general y BLAS, `http://dis.um.es/~domingo/apuntes/HICC/0708/vision+BLAS.pdf`.

[4] Librería secuencial de Álgebra Lineal Densa LAPACK, `http://dis.um.es/~domingo/apuntes/HICC/0708/LAPACK.pdf`.

[5] Domingo Giménez, Programación en memoria compartida: OpenMP, `http://dis.um.es/~domingo/apuntes/CAP/0708/openmp.pdf`.

[6] Domingo Giménez, Modelos de programación paralela - Programación en memoria distribuida MPI, `http://dis.um.es/~domingo/apuntes/CAP/0809/MPI.pdf`.

[7] Grupo de Computación científica de la Universidad de Murcia, `http://www.um.es/pcgum/`

[8] Jose Antonio Jiménez Loureda, Proyecto Fin de Carrera de Ingeniería en Informática, Software para Computación Matricial Paralela: Liberías de Computación Matricial, Junio 1997.

[9] María Dolores Bernal Forte, Proyecto Fin de Carrera de Ingeniería en Informática, Software para computación matricial paralela: Liberías de Paso de Mensajes, Septiembre 1996.

[10] Francisco Almeida, Domingo Giménez, José Miguel Mantas, Antonio M. Vidal. Introducción a la programación paralela. Editorial Paraninfo, 2008.

[11] The Beowulf Cluster Site, `http://www.beowulf.org/`.

[12] RFC 3530, Network File System (NFS) version 4 Protocol, `http://tools.ietf.org/html/rfc3530`.

[13] Cluster Resources, Torque Resource Manager, `http://www.clusterresources.com/products/torque/`.

[14] Cluster Resources, Maui Cluster Scheduler, `http://www.clusterresources.com/products/maui/`.

[15] Cluster Resources, Torque Administrator's Manual `http://www.clusterresources.com/torquedocs21/`.

[16] GNU Screen, `http://www.gnu.org/software/screen/`.

[17] Rsync webpage, `http://rsync.samba.org/`.

[18] Cluster SSH - Cluster Admin Via SSH, `http://sourceforge.net/projects/clusterssh/`.

[19] Project C3 - Cluster command and control, `http://www.csm.ornl.gov/torc/C3/`.

[20] Gnuplot homepage, `http://www.gnuplot.info/`.

[21] GCC the GNU Compiler Collection, `http://gcc.gnu.org/`.

[22] Intel C/C++ optimized compiler for Linux, `http://www.intel.com/software/products/compilers/clin/`.

[23] gcc(1) Linux man page, `http://linux.die.net/man/1/gcc`.

[24] BLAS Reference, `http://www.netlib.org/blas/`.

[25] BLAS ATLAS, `http://math-atlas.sourceforge.net/`.

[26] BLAS Goto, `http://www.tacc.utexas.edu/resources/software/software.php`.

[27] Academic Technology Services, Computational Cluster Programs, LaPack Benchmark, `http://www.ats.ucla.edu/clusters/common/software/libraries/lapack_benchmark.htm`.

[28] Fortran Hacker, Blas benchmark: ddot,
`http://fortranhacker.blogspot.com/2007/09/blas-benchmark-ddot.html`.

[29] Alexis Vera Pérez, Factorización LU
`http://cicia.uprrp.edu/Papers/FactorizacionLU.pdf`.

[30] David Montejo Rodríguez, Proyecto Fin de Carrera de Ingeniería en Informática, Estudio de las Prestaciones de Librerias de Paso de Mensajes y de Computación Matricial, Marzo 1998.

[31] SEDIC-SAIT - Guía del Usuario del Cluster hipatia
`http://hpc-www.sait.upct.es`.

[32] Javier Cuenca, Domingo Giménez. Herramientas informáticas para computación científica - Librerías matriciales: ScaLAPACK.
`http://dis.um.es/~domingo/apuntes/HICC/0708/scalapack.pdf`.

Anexos

Anexo A

Mediciones

A.1. OpenMP

A.1.1. Primitivas de OpenMP

	N				
P	1	50	100	1000	5000
1	0.0000803	0.0000791	0.0000799	0.0000819	0.0000856
2	0.0001567	0.0001769	0.0001796	0.0001743	0.0001809
3	0.0001939	0.0002067	0.0002109	0.0002008	0.0002076
4	0.0002746	0.0002676	0.0002713	0.0002710	0.0002964
5	0.0003177	0.0003085	0.0002909	0.0003232	0.0003296
6	0.0003511	0.0003389	0.0003461	0.0003437	0.0003862
7	0.0003955	0.0003789	0.0003921	0.0004183	0.0004429
8	0.0004550	0.0004253	0.0004375	0.0004636	0.0005231
9	0.0005217	0.0004967	0.0004646	0.0005153	0.0005953
10	0.0005094	0.0005139	0.0005005	0.0005700	0.0006574

Tabla A.1: Rutina `generate` compilada con `gcc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0000825	0.0000799	0.0000816	0.0000807	0.0000861
2	0.0001644	0.0001783	0.0001806	0.0001757	0.0001843
3	0.0002215	0.0002127	0.0002194	0.0002157	0.0002178
4	0.0002917	0.0002848	0.0002854	0.0002823	0.0002939
5	0.0003220	0.0003120	0.0003151	0.0003320	0.0003534
6	0.0003746	0.0003781	0.0003565	0.0003854	0.0004043
7	0.0003847	0.0003960	0.0003863	0.0003797	0.0004546
8	0.0004389	0.0004529	0.0004596	0.0004428	0.0005361
9	0.0004829	0.0004753	0.0005076	0.0005384	0.0006274
10	0.0005197	0.0005576	0.0005102	0.0005430	0.0006488

Tabla A.2: Rutina `generate` compilada con `gcc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0000813	0.0000806	0.0000844	0.0000890	0.0000934
2	0.0001823	0.0001751	0.0001754	0.0001903	0.0002121
3	0.0002171	0.0002233	0.0002210	0.0002187	0.0002743
4	0.0002843	0.0002869	0.0002896	0.0002800	0.0003362
5	0.0003112	0.0003214	0.0003271	0.0003352	0.0004368
6	0.0003544	0.0003543	0.0003652	0.0003758	0.0004650
7	0.0004103	0.0004173	0.0004068	0.0004111	0.0005895
8	0.0004681	0.0004344	0.0004529	0.0004804	0.0008338
9	0.0005127	0.0004797	0.0005050	0.0005599	0.0008937
10	0.0005530	0.0005418	0.0005395	0.0006020	0.0009707

Tabla A.3: Rutina `generate` compilada con `gcc` y optimizaciones `Os`

	N				
P	1	50	100	1000	5000
1	0.0001258	0.0001461	0.0001770	0.0005440	0.0022341
2	0.0002869	0.0003342	0.0003788	0.0009974	0.0038572
3	0.0004084	0.0004473	0.0004932	0.0012795	0.0048255
4	0.0037682	0.0006194	0.0010737	0.0023018	0.0058676
5	0.0272302	0.0497356	0.0317928	0.0304215	0.0388278
6	0.0610559	0.0691564	0.0572366	0.0555694	0.0918989
7	0.0841038	0.1166848	0.1203393	0.1097739	0.1034660
8	0.1062055	0.1202735	0.1174031	0.1258968	0.1499816
9	0.1419600	0.1334157	0.1448660	0.1519419	0.1609143
10	0.1716265	0.1748529	0.1997593	0.1973126	0.1951508

Tabla A.4: Rutina `generate` compilada con `icc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0001248	0.0001479	0.0001722	0.0005499	0.0022276
2	0.0002938	0.0003327	0.0003650	0.0010150	0.0037490
3	0.0004387	0.0004612	0.0004955	0.0013138	0.0048249
4	0.0021974	0.0006645	0.0018624	0.0053965	0.0067795
5	0.0313826	0.0345410	0.0294242	0.0475757	0.0343583
6	0.0585184	0.0756442	0.0698227	0.0713100	0.0934202
7	0.0761057	0.1106546	0.1169174	0.0980492	0.1340055
8	0.1127587	0.1176919	0.1185936	0.1568725	0.1214533
9	0.1444636	0.1433445	0.1458071	0.1478181	0.1484155
10	0.1899700	0.1788727	0.1798812	0.2058387	0.1831191

Tabla A.5: Rutina `generate` compilada con `icc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0001283	0.0001469	0.0001784	0.0005589	0.0022722
2	0.0002983	0.0003268	0.0003742	0.0009839	0.0037368
3	0.0004258	0.0004611	0.0004998	0.0013036	0.0047958
4	0.0005581	0.0022454	0.0006362	0.0016154	0.0071221
5	0.0327244	0.0497874	0.0357951	0.0310545	0.0359613
6	0.0608933	0.0535619	0.0909642	0.0594406	0.0538614
7	0.0809017	0.0846089	0.0770299	0.1014956	0.1211972
8	0.1136512	0.1325595	0.1178488	0.1253348	0.1549147
9	0.1343080	0.1528400	0.1427822	0.1441070	0.1350137
10	0.1710674	0.1833091	0.1759117	0.1700081	0.1918120

Tabla A.6: Rutina `generate` compilada con `icc` y optimizaciones `Os`

	N				
P	1	50	100	1000	5000
1	0.0000204	0.0000206	0.0000213	0.0000414	0.0001400
2	0.0002426	0.0002357	0.0002492	0.0002654	0.0003432
3	0.0003190	0.0003270	0.0003336	0.0003469	0.0004037
4	0.0004461	0.0004469	0.0004454	0.0004652	0.0005141
5	0.0006362	0.0006512	0.0006430	0.0006469	0.0007767
6	0.0007702	0.0007605	0.0007525	0.0007857	0.0008931
7	0.0008631	0.0008929	0.0008479	0.0008998	0.0009961
8	0.0010072	0.0009791	0.0010146	0.0010197	0.0011012
9	0.0011413	0.0011025	0.0011094	0.0011405	0.0012136
10	0.0012801	0.0012267	0.0012242	0.0012431	0.0013245

Tabla A.7: Rutina `pfor` compilada con `gcc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0000201	0.0000203	0.0000207	0.0000285	0.0000694
2	0.0002482	0.0002491	0.0002487	0.0002643	0.0003146
3	0.0003264	0.0003408	0.0003354	0.0003445	0.0004007
4	0.0004566	0.0004601	0.0004650	0.0004689	0.0005049
5	0.0006334	0.0006585	0.0006436	0.0006679	0.0007618
6	0.0007526	0.0007419	0.0007652	0.0008024	0.0008550
7	0.0008428	0.0008649	0.0008875	0.0009116	0.0009705
8	0.0010015	0.0009817	0.0009751	0.0010210	0.0011003
9	0.0011096	0.0011157	0.0011228	0.0011255	0.0012243
10	0.0012786	0.0012360	0.0012197	0.0012203	0.0013574

Tabla A.8: Rutina `pfor` compilada con `gcc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0000389	0.0010279	0.0020404	0.0202425	0.1010940
2	0.0002462	0.0007538	0.0012799	0.0106296	0.0508999
3	0.0003479	0.0006904	0.0010911	0.0082374	0.0375593
4	0.0004869	0.0007727	0.0010182	0.0062148	0.0315456
5	0.0006683	0.0010004	0.0014270	0.0086158	0.0412204
6	0.0007784	0.0011458	0.0014737	0.0074779	0.0346918
7	0.0008924	0.0011579	0.0015205	0.0068221	0.0302485
8	0.0010174	0.0013031	0.0015438	0.0062526	0.0267740
9	0.0011225	0.0014187	0.0016719	0.0076768	0.0346997
10	0.0012493	0.0015006	0.0017895	0.0072473	0.0316403

Tabla A.9: Rutina `pfor` compilada con `gcc` y optimizaciones `Os`

	N				
P	1	50	100	1000	5000
1	0.0001417	0.0002284	0.0003129	0.0019617	0.0092935
2	0.0003252	0.0003721	0.0004056	0.0012463	0.0049374
3	0.0004511	0.0004901	0.0005125	0.0010673	0.0035020
4	0.0005741	0.0049854	0.0006035	0.0010503	0.0036673
5	0.2171059	0.2267469	0.2311546	0.2284668	0.2229517
6	0.2438309	0.2373675	0.2322020	0.2328884	0.2432546
7	0.2473860	0.2466114	0.2389197	0.2399524	0.2400871
8	0.2439285	0.2435025	0.2435881	0.2381923	0.2531295
9	0.2697943	0.2604607	0.2546672	0.2444645	0.2691116
10	0.2458786	0.2656696	0.2649792	0.2492682	0.2781100

Tabla A.10: Rutina `pfor` compilada con `icc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0001410	0.0002430	0.0003761	0.0024549	0.0105445
2	0.0003346	0.0003910	0.0004262	0.0012675	0.0051338
3	0.0004557	0.0004888	0.0005199	0.0010607	0.0034969
4	0.0005915	0.0005818	0.0005984	0.0010257	0.0032929
5	0.2223760	0.2165479	0.2177566	0.2212805	0.2224788
6	0.2372563	0.2353665	0.2319720	0.2281175	0.2334848
7	0.2354286	0.2333141	0.2387821	0.2484720	0.2351775
8	0.2420658	0.2371335	0.2301667	0.2360361	0.2353755
9	0.2466172	0.2594926	0.2583694	0.2569985	0.2537238
10	0.2610425	0.2577757	0.2505873	0.2545061	0.2450188

Tabla A.11: Rutina `pfor` compilada con `icc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0001428	0.0002292	0.0003079	0.0019759	0.0092980
2	0.0003508	0.0003729	0.0003923	0.0012578	0.0049207
3	0.0004499	0.0004920	0.0005160	0.0010523	0.0035118
4	0.0025747	0.0005975	0.0006223	0.0014323	0.0030504
5	0.2158210	0.2216647	0.2316250	0.2347088	0.2310341
6	0.2396054	0.2337114	0.2368353	0.2297092	0.2419383
7	0.2368649	0.2307205	0.2485274	0.2470672	0.2425596
8	0.2414906	0.2349048	0.2451460	0.2449604	0.2533492
9	0.2548947	0.2445865	0.2512064	0.2529670	0.2444958
10	0.2470389	0.2554297	0.2644583	0.2580420	0.2544341

Tabla A.12: Rutina `pfor` compilada con `icc` y optimizaciones `Os`

	N				
P	1	50	100	1000	5000
1	0.0000156	0.0000772	0.0001385	0.0012609	0.0061528
2	0.0001142	0.0009953	0.0018454	0.0179046	0.0883148
3	0.0001483	0.0014197	0.0027748	0.0263600	0.1277911
4	0.0002103	0.0016448	0.0030477	0.0287768	0.1421209
5	0.0002416	0.0023524	0.0045811	0.0435885	0.2172368
6	0.0002844	0.0028790	0.0053788	0.0512933	0.2547858
7	0.0003270	0.0033054	0.0062749	0.0593592	0.2960659
8	0.0003648	0.0038408	0.0073999	0.0710777	0.3560922
9	0.0004279	0.0043419	0.0082997	0.0799613	0.4021878
10	0.0004404	0.0047869	0.0091027	0.0880210	0.4432080

Tabla A.13: Rutina `sections` compilada con `gcc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0000151	0.0000762	0.0001400	0.0012647	0.0061681
2	0.0001174	0.0009922	0.0019434	0.0183286	0.0806173
3	0.0001564	0.0014757	0.0027547	0.0262481	0.1294442
4	0.0002082	0.0016734	0.0030543	0.0286612	0.1418167
5	0.0002350	0.0024125	0.0045532	0.0440027	0.2182210
6	0.0002938	0.0028613	0.0054187	0.0512152	0.2555180
7	0.0003348	0.0032865	0.0062932	0.0595285	0.2966716
8	0.0003845	0.0038614	0.0074041	0.0709284	0.3564662
9	0.0004325	0.0043548	0.0082750	0.0799973	0.4023584
10	0.0004353	0.0047671	0.0091101	0.0880372	0.4424709

Tabla A.14: Rutina `sections` compilada con `gcc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0000147	0.0000789	0.0001445	0.0012964	0.0063429
2	0.0001112	0.0009955	0.0018964	0.0180588	0.0903292
3	0.0001765	0.0014352	0.0027223	0.0257041	0.1277039
4	0.0002043	0.0016501	0.0031031	0.0288465	0.1431222
5	0.0002502	0.0024118	0.0045418	0.0441848	0.2157380
6	0.0002963	0.0029960	0.0056308	0.0535892	0.2697448
7	0.0003550	0.0032846	0.0063718	0.0607541	0.3026754
8	0.0003908	0.0039058	0.0074428	0.0712223	0.3573379
9	0.0004450	0.0043869	0.0083767	0.0805474	0.4037674
10	0.0004738	0.0047569	0.0092024	0.0884165	0.4438462

Tabla A.15: Rutina `sections` compilada con `gcc` y optimizaciones `Os`

	N				
P	1	50	100	1000	5000
1	0.0001428	0.0001660	0.0001932	0.0006673	0.0027145
2	0.0003229	0.0004244	0.0005554	0.0027863	0.0129339
3	0.0004278	0.0005751	0.0007619	0.0033284	0.0150966
4	0.0005248	0.0011224	0.0008812	0.0116598	0.0163500
5	0.0338107	0.2190688	0.2200358	0.2259004	0.2983740
6	0.0546487	0.2362871	0.2414772	0.2534576	0.3210695
7	0.0757051	0.2468629	0.2465723	0.2484012	0.3035762
8	0.1224408	0.2443516	0.2338931	0.2566050	0.3370728
9	0.1553413	0.2554938	0.2490260	0.2681113	0.3665676
10	0.1924750	0.2596191	0.2595373	0.2722148	0.3661375

Tabla A.16: Rutina `sections` compilada con `icc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0001385	0.0001684	0.0001960	0.0006666	0.0027173
2	0.0003057	0.0004402	0.0005575	0.0027759	0.0126887
3	0.0004191	0.0005770	0.0007503	0.0034268	0.0150648
4	0.0005417	0.0007142	0.0008770	0.0041827	0.0165295
5	0.0308127	0.2222027	0.2266475	0.2360538	0.2957845
6	0.0533554	0.2508917	0.2424326	0.2528614	0.3152797
7	0.0867785	0.2428897	0.2426326	0.2568141	0.3186756
8	0.1210983	0.2398789	0.2460157	0.2615418	0.3365110
9	0.1730928	0.2481995	0.2610736	0.2739472	0.3683085
10	0.1864742	0.2615201	0.2714532	0.2700231	0.4067480

Tabla A.17: Rutina `sections` compilada con `icc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0001439	0.0001721	0.0001946	0.0006629	0.0027918
2	0.0003207	0.0004307	0.0005722	0.0027854	0.0129473
3	0.0004192	0.0005390	0.0007104	0.0032521	0.0146624
4	0.0005248	0.0087275	0.0008607	0.0037499	0.0164042
5	0.0353225	0.2241511	0.2371164	0.2315612	0.2835747
6	0.0572813	0.2403749	0.2371378	0.2462025	0.3073705
7	0.0884763	0.2437830	0.2463638	0.2596255	0.3202385
8	0.1187188	0.2353241	0.2455220	0.2529364	0.3217909
9	0.1621457	0.2545827	0.2469545	0.2631197	0.3544254
10	0.1770062	0.2677766	0.2701813	0.2824557	0.3918090

Tabla A.18: Rutina `sections` compilada con `icc` y optimizaciones `Os`

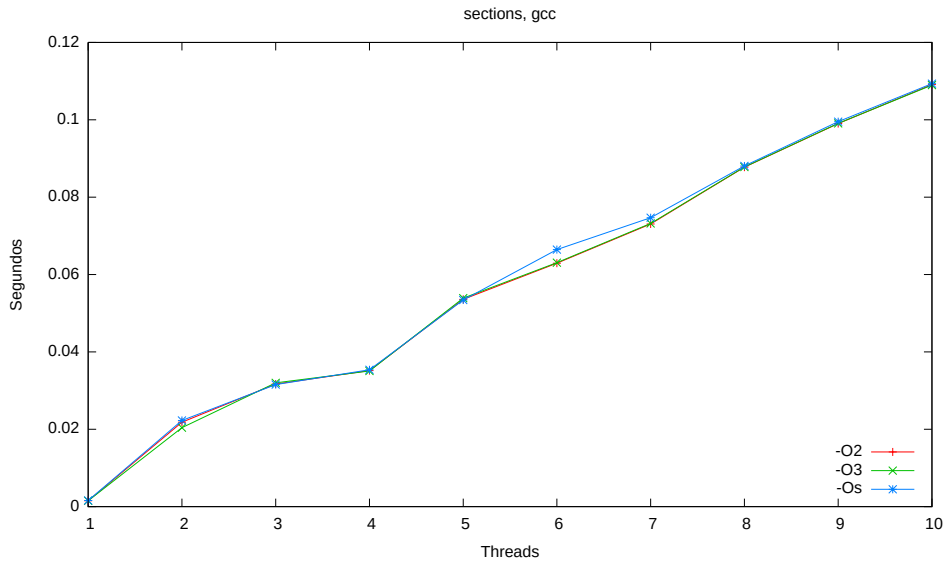

Figura A.1: Comparación de optimizaciones de `gcc` para el programa `sections`

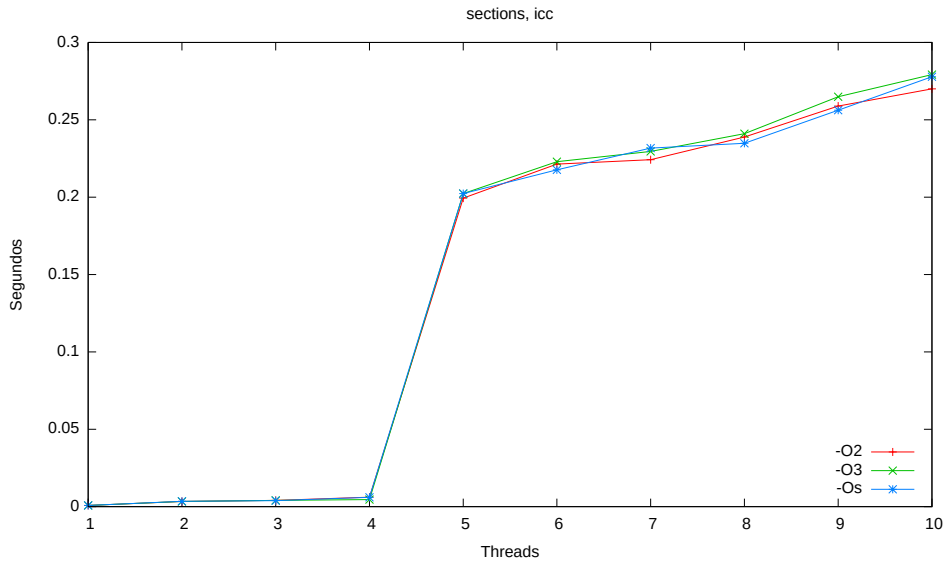

Figura A.2: Comparación de optimizaciones de `icc` para el programa `sections`

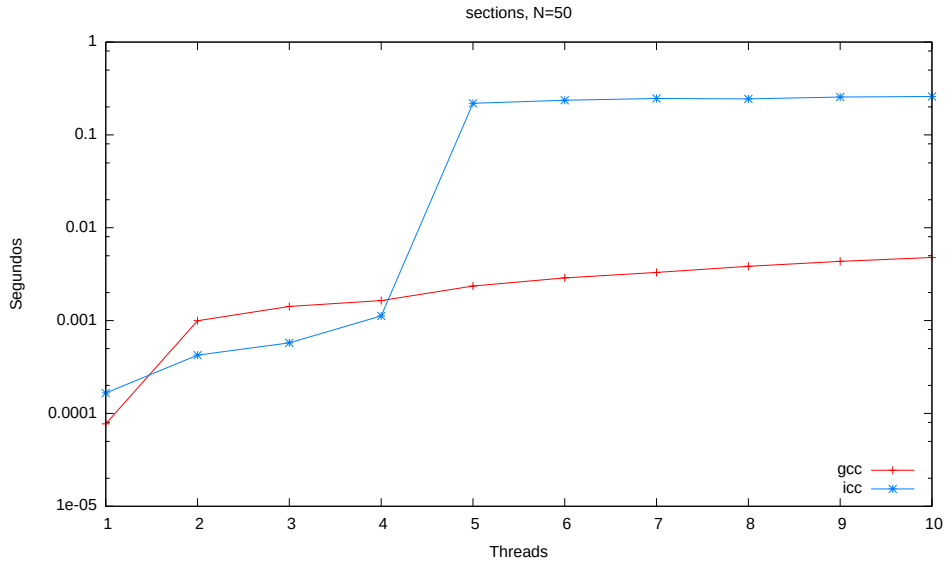

Figura A.3: Rutina `sections`, comparativa `gcc` e `icc` con `O2`, variando el número de hilos para el tamaño $N = 50$. El tiempo se muestra en escala logarítmica

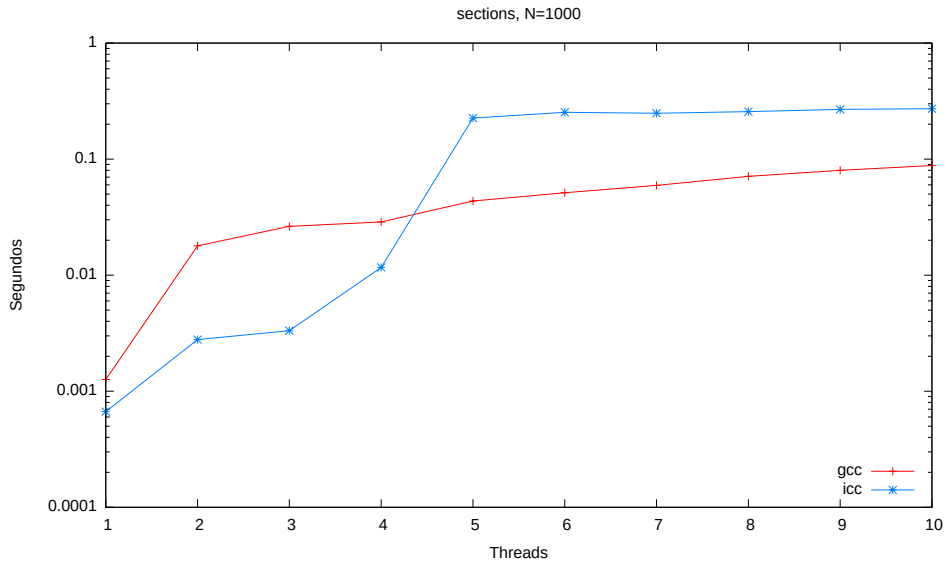

Figura A.4: Rutina `sections`, comparativa `gcc` e `icc` con `O2`, variando el número de hilos para el tamaño $N = 1000$. El tiempo se muestra en escala logarítmica

	N				
P	1	50	100	1000	5000
1	0.0000146	0.0001376	0.0002675	0.0025220	0.0125858
2	0.0001173	0.0026198	0.0053264	0.0509349	0.2579336
3	0.0001584	0.0057683	0.0110975	0.1118423	0.5567485
4	0.0002591	0.0076967	0.0154732	0.1536834	0.7686692
5	0.0003506	0.0102426	0.0203799	0.2017425	1.0001147
6	0.0004486	0.0124322	0.0240432	0.2406859	1.1970046
7	0.0005480	0.0143010	0.0278820	0.2777245	1.3891023
8	0.0006137	0.0163445	0.0322245	0.3194871	1.5920962
9	0.0007135	0.0181908	0.0362110	0.3581969	1.7937814
10	0.0008086	0.0204798	0.0403017	0.3979638	1.9876722

Tabla A.19: Rutina `barriers` compilada con `gcc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0000154	0.0001387	0.0002641	0.0025154	0.0126123
2	0.0001130	0.0026429	0.0051527	0.0508129	0.2545759
3	0.0001743	0.0056774	0.0112320	0.1113556	0.5614573
4	0.0002633	0.0078204	0.0156319	0.1530631	0.7650922
5	0.0003204	0.0103585	0.0203497	0.2010386	0.9972147
6	0.0004198	0.0122692	0.0243895	0.2413026	1.1998370
7	0.0005025	0.0141898	0.0279550	0.2794046	1.3987844
8	0.0006092	0.0163423	0.0322979	0.3189532	1.5988197
9	0.0007017	0.0183162	0.0363180	0.3581579	1.7913179
10	0.0007532	0.0204035	0.0402977	0.3971114	1.9879032

Tabla A.20: Rutina `barriers` compilada con `gcc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0000146	0.0001385	0.0002653	0.0025389	0.0127951
2	0.0001173	0.0026911	0.0052206	0.0513727	0.2551856
3	0.0001802	0.0057638	0.0114591	0.1100806	0.5684262
4	0.0002570	0.0077858	0.0154961	0.1531309	0.7753954
5	0.0003458	0.0103245	0.0204807	0.2025400	1.0048074
6	0.0004419	0.0121986	0.0244438	0.2411264	1.2061843
7	0.0005493	0.0140989	0.0283656	0.2799844	1.3899159
8	0.0005967	0.0163958	0.0323497	0.3216361	1.5956474
9	0.0007221	0.0185550	0.0365769	0.3617232	1.8143986
10	0.0007808	0.0204698	0.0403666	0.3989730	1.9904363

Tabla A.21: Rutina `barriers` compilada con `gcc` y optimizaciones `Os`

	N				
P	1	50	100	1000	5000
1	0.0001412	0.0003436	0.0005472	0.0042569	0.0207107
2	0.0003080	0.0018495	0.0034509	0.0318604	0.1579536
3	0.0004365	0.0043562	0.0085481	0.0841608	0.4031171
4	0.0005644	0.0066002	0.0169259	0.1209450	0.6185964
5	0.0612285	0.3557318	0.3893372	0.5305715	1.3588117
6	0.0872192	0.3097886	0.3478601	0.5716363	1.5457221
7	0.1292679	0.3114200	0.3634959	0.6171178	1.7335854
8	0.1363339	0.2866929	0.2848132	0.6836269	1.9509499
9	0.2122589	0.2742722	0.2913058	0.7032502	2.2019286
10	0.2189442	0.2721735	0.3024372	0.7704266	2.7086394

Tabla A.22: Rutina `barriers` compilada con `icc` y optimizaciones `O2`

	N				
P	1	50	100	1000	5000
1	0.0001421	0.0003473	0.0005563	0.0042545	0.0207076
2	0.0003038	0.0018552	0.0033745	0.0312101	0.1580585
3	0.0004284	0.0044756	0.0084738	0.0819221	0.4153101
4	0.0005734	0.0109316	0.0162091	0.1273826	0.6130276
5	0.0615133	0.3600363	0.3556497	0.5602133	1.3393691
6	0.1018785	0.3358763	0.3521633	0.5825967	1.5557670
7	0.1006689	0.3474571	0.3552226	0.5875345	1.7430171
8	0.1387522	0.2643359	0.2854474	0.6468659	1.9836225
9	0.2242800	0.2572596	0.3020625	0.7240937	2.2947871
10	0.2262487	0.2820443	0.2905150	0.7651925	2.7267436

Tabla A.23: Rutina `barriers` compilada con `icc` y optimizaciones `O3`

	N				
P	1	50	100	1000	5000
1	0.0001433	0.0003475	0.0005564	0.0041437	0.0200652
2	0.0003135	0.0018049	0.0033894	0.0311782	0.1578842
3	0.0004441	0.0044066	0.0083771	0.0812185	0.4092212
4	0.0050335	0.0065503	0.0128605	0.1227544	0.6090110
5	0.0586030	0.3540329	0.3709905	0.5600326	1.3180514
6	0.1148483	0.3430626	0.3668637	0.5854164	1.5748859
7	0.1094030	0.3368713	0.3759071	0.5972354	1.7429493
8	0.1460493	0.2610122	0.3185020	0.6799797	1.9243617
9	0.2142475	0.2740973	0.2920443	0.7430606	2.2081429
10	0.2254663	0.2742851	0.2988300	0.7728962	2.7048760

Tabla A.24: Rutina `barriers` compilada con `icc` y optimizaciones `Os`

A.1.2. Rutinas de alto nivel

	N		
P	2000	4000	8000
1	0.0516052	0.2055588	0.8208666
2	0.0262994	0.1042288	0.4149766
3	0.0212174	0.0732542	0.2915554
4	0.0169134	0.0588462	0.2276706
5	0.0215006	0.0714296	0.2572842
6	0.0188068	0.0624542	0.2331898
7	0.0165632	0.0639948	0.2396368
8	0.0149674	0.0589698	0.2324622
9	0.0189880	0.0622204	0.2337944
10	0.0174468	0.0601872	0.2333150

Tabla A.25: Rutina `matrizvector` compilada con `gcc` y optimizaciones `O2`

	N		
P	2000	4000	8000
1	0.0516084	0.2055834	0.8205694
2	0.0263986	0.1039186	0.4164610
3	0.0203758	0.0736476	0.2906534
4	0.0174254	0.0579910	0.2273060
5	0.0213354	0.0701526	0.2646424
6	0.0188122	0.0628830	0.2324542
7	0.0166010	0.0625364	0.2383404
8	0.0149636	0.0603692	0.2378262
9	0.0184294	0.0624832	0.2371716
10	0.0173864	0.0635934	0.2464944

Tabla A.26: Rutina `matrizvector` compilada con `gcc` y optimizaciones `O3`

	N		
P	2000	4000	8000
1	0.1085084	0.4338264	1.7353218
2	0.0544986	0.2172952	0.8681260
3	0.0377118	0.1450260	0.5733608
4	0.0317932	0.1198320	0.4278738
5	0.0390948	0.1276516	0.4819118
6	0.0347640	0.1105402	0.4474388
7	0.0324112	0.1150518	0.4440694
8	0.0279702	0.1114388	0.4390336
9	0.0358242	0.1149254	0.4434872
10	0.0327980	0.1175984	0.4425732

Tabla A.27: Rutina `matrizvector` compilada con `gcc` y optimizaciones `Os`

	N		
P	2000	4000	8000
1	0.0341900	0.1346366	0.5362840
2	0.0177788	0.0685656	0.2722686
3	0.0132656	0.0502912	0.1984260
4	0.0119374	0.0423326	0.1649974
5	0.0810128	0.1270056	0.2487116
6	0.1780450	0.1883538	0.2666646
7	0.2079914	0.1948260	0.2919750
8	0.2003940	0.2356206	0.3287456
9	0.2416632	0.2861302	0.3730896
10	0.2398256	0.2735712	0.4006758

Tabla A.28: Rutina `matrizvector` compilada con `icc` y optimizaciones `O2`

	N		
P	2000	4000	8000
1	0.0341126	0.1347206	0.5361310
2	0.0177386	0.0686094	0.2723416
3	0.0133012	0.0503124	0.1982616
4	0.0119368	0.0431268	0.1649216
5	0.0797942	0.1467458	0.2620614
6	0.1786780	0.2027008	0.2833168
7	0.2000316	0.2215290	0.3243502
8	0.1844662	0.2305708	0.3372448
9	0.2297482	0.2876866	0.3915222
10	0.2702600	0.3034322	0.3973322

Tabla A.29: Rutina `matrizvector` compilada con `icc` y optimizaciones `O3`

	N		
P	2000	4000	8000
1	0.0510044	0.2038446	0.8138268
2	0.0258222	0.1007330	0.4026468
3	0.0179756	0.0693756	0.2746844
4	0.0146524	0.0554734	0.2191670
5	0.0984448	0.1478564	0.3130628
6	0.1799906	0.2091376	0.3110626
7	0.1992986	0.2322056	0.3843396
8	0.2180528	0.2550964	0.3788954
9	0.2580802	0.2608148	0.3804450
10	0.2898758	0.2862042	0.4283596

Tabla A.30: Rutina `matrizvector` compilada con `icc` y optimizaciones `Os`

	N		
P	1000	2000	4000
1	0.0830788	0.3298004	1.3206086
2	0.0442920	0.1722254	0.6897400
3	0.0331146	0.1249990	0.4884294
4	0.0271276	0.0991218	0.3887418
5	0.0363066	0.1308212	0.4572744
6	0.0313300	0.1178094	0.4276308
7	0.0284450	0.1079184	0.4145388
8	0.0250812	0.0988152	0.4024156
9	0.0296104	0.1185178	0.4106548
10	0.0288564	0.1125746	0.4149666

Tabla A.31: Rutina `jacobi` compilada con `gcc` y optimizaciones `O2`

	N		
P	1000	2000	4000
1	0.0831666	0.3299360	1.3203952
2	0.0443308	0.1722026	0.6896928
3	0.0320570	0.1245484	0.4882000
4	0.0290992	0.0983930	0.3878368
5	0.0359832	0.1282542	0.4724448
6	0.0313052	0.1198260	0.4317692
7	0.0276716	0.1080992	0.4178688
8	0.0251584	0.0980222	0.3999748
9	0.0296888	0.1171264	0.4179812
10	0.0289046	0.1119650	0.4115042

Tabla A.32: Rutina `jacobi` compilada con `gcc` y optimizaciones `O3`

	N		
P	1000	2000	4000
1	0.0850846	0.3369200	1.3485656
2	0.0449556	0.1749082	0.6987740
3	0.0341356	0.1260908	0.4981276
4	0.0284048	0.1007116	0.3978530
5	0.0366538	0.1340908	0.4825806
6	0.0317598	0.1215686	0.4297416
7	0.0283082	0.1104890	0.4202094
8	0.0262508	0.1037076	0.4130470
9	0.0304120	0.1202000	0.4238678
10	0.0292916	0.1145244	0.4255492

Tabla A.33: Rutina `jacobi` compilada con `gcc` y optimizaciones `Os`

	N		
P	1000	2000	4000
1	0.0767100	0.2929462	1.1708370
2	0.0399708	0.1534728	0.6084080
3	0.0286138	0.1102558	0.4371420
4	0.0267510	0.0891540	0.3511406
5	0.1269996	0.2048380	0.5077400
6	0.1901066	0.2902546	0.5506126
7	0.2167268	0.2780384	0.5507338
8	0.2741114	0.3694274	0.5923332
9	0.2753750	0.2899038	0.5931334
10	0.2652086	0.3619000	0.6106802

Tabla A.34: Rutina `jacobi` compilada con `icc` y optimizaciones `O2`

	N		
P	1000	2000	4000
1	0.0740500	0.2924004	1.1686724
2	0.0394298	0.1514478	0.6019372
3	0.0284788	0.1095154	0.4342068
4	0.0239736	0.0896988	0.3495944
5	0.1432266	0.2046520	0.5239034
6	0.2015214	0.2621144	0.5450112
7	0.2383426	0.2836510	0.5527300
8	0.2695462	0.3360674	0.5857240
9	0.2698350	0.3318194	0.5990032
10	0.2650996	0.3297592	0.6133982

Tabla A.35: Rutina `jacobi` compilada con `icc` y optimizaciones `O3`

	N		
P	1000	2000	4000
1	0.0744528	0.2952062	1.1775132
2	0.0386440	0.1509596	0.6045094
3	0.0276936	0.1082108	0.4294074
4	0.0257968	0.0888404	0.3511986
5	0.1321388	0.1876232	0.5313828
6	0.1729580	0.1968870	0.5637530
7	0.2287980	0.2966984	0.5689702
8	0.2769272	0.3426018	0.5633860
9	0.2807494	0.3002016	0.5815694
10	0.2707188	0.3369254	0.5837006

Tabla A.36: Rutina `jacobi` compilada con `icc` y optimizaciones `Os`

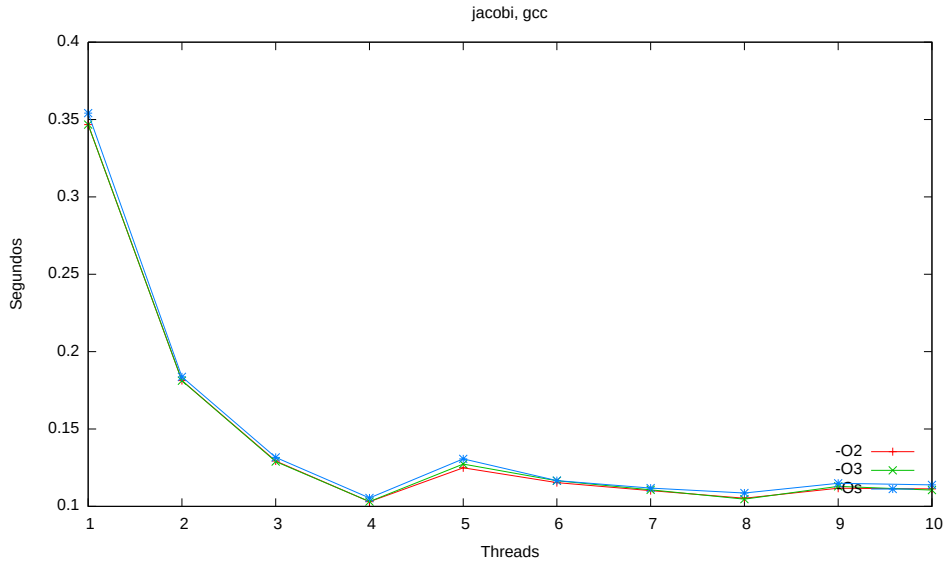

Figura A.5: Comparación de optimizaciones de `gcc` para el programa `jacobi`

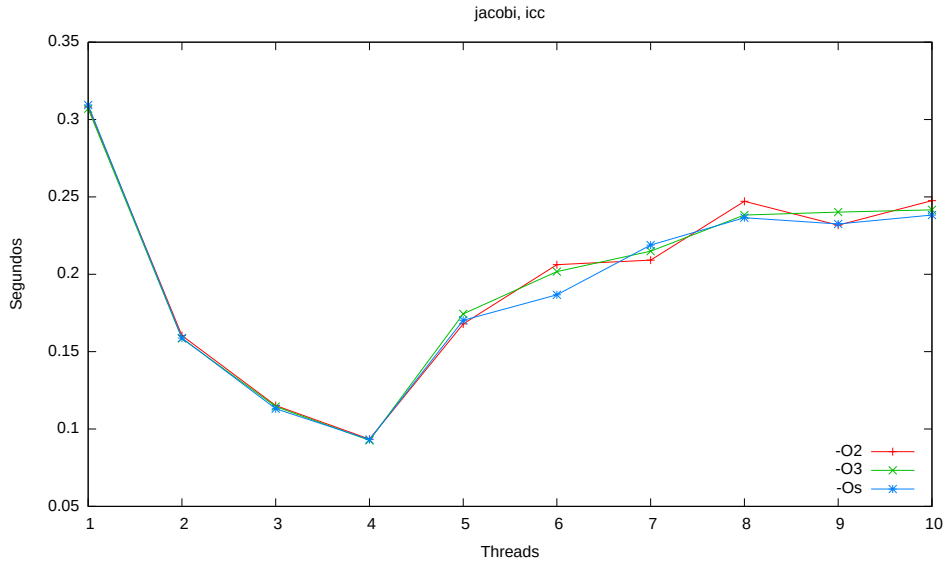

Figura A.6: Comparación de optimizaciones de `icc` para el programa `jacobi`

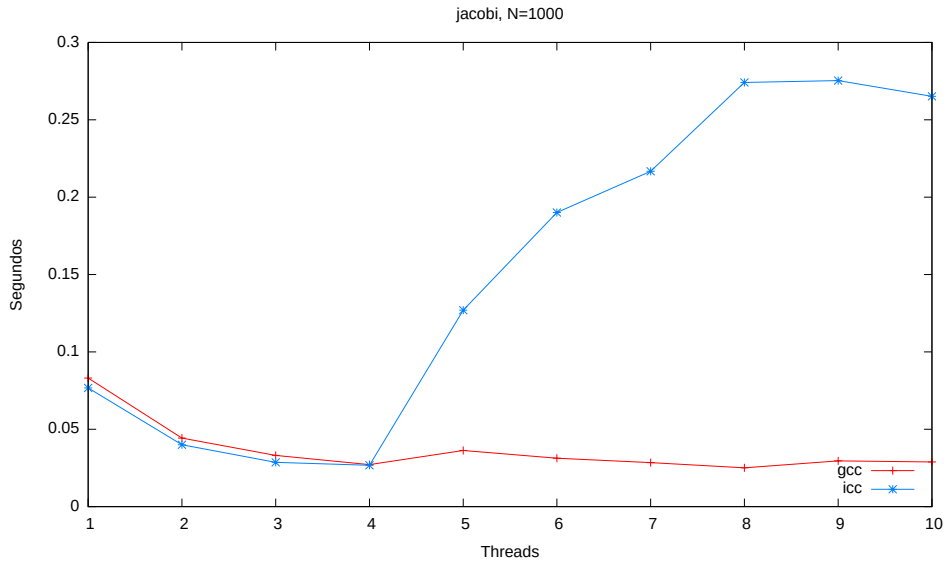

Figura A.7: Rutina `jacobi`, comparativa `gcc` e `icc` con `O2` y tamaño $N = 1000$

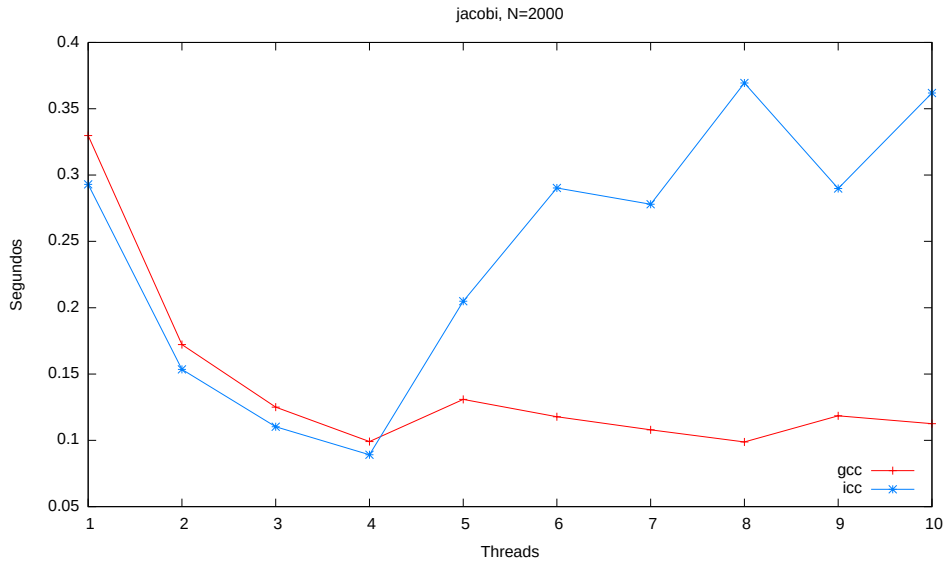

Figura A.8: Rutina `jacobi`, comparativa `gcc` e `icc` con `O2` y tamaño $N = 2000$

	N		
P	500	1000	1500
1	1.4509026	13.1921992	45.7517962
2	0.7346002	7.0603782	23.6020984
3	0.5033576	4.7889872	16.5256410
4	0.4060940	3.5349626	12.1738332
5	0.4195418	3.6919534	12.5930838
6	0.3881638	3.6769204	12.6351246
7	0.3868726	3.6536356	12.5381324
8	0.3836560	3.5798958	12.4512788
9	0.3777174	3.6182432	12.4896982
10	0.3797748	3.6232938	12.5359874

Tabla A.37: Rutina `matrizmatriz` compilada con `gcc` y optimizaciones `O2`

	N		
P	500	1000	1500
1	1.4408046	13.1959358	45.7569850
2	0.7286026	6.8238934	23.4749968
3	0.5037240	4.8178436	16.7432426
4	0.3980624	3.5181024	12.2820576
5	0.4202012	3.7288156	12.6451156
6	0.3958770	3.6834180	12.5760576
7	0.3852636	3.6818798	12.5607582
8	0.3825656	3.5804330	12.4666070
9	0.3837980	3.6101048	12.5579396
10	0.3787184	3.5942400	12.5371198

Tabla A.38: Rutina `matrizmatriz` compilada con `gcc` y optimizaciones `O3`

	N		
P	500	1000	1500
1	1.6922574	13.6671314	53.8231012
2	0.8618274	7.1482838	27.4268566
3	0.5861844	5.0548008	18.8951512
4	0.4506912	3.7625884	14.6476702
5	0.5030492	3.8410896	14.3767362
6	0.4547388	3.8137346	14.3340702
7	0.4428966	3.7839022	14.3029030
8	0.4413774	3.7173224	14.2518020
9	0.4321066	3.7338448	14.3251748
10	0.4473190	3.7303148	14.2689290

Tabla A.39: Rutina `matrizmatriz` compilada con `gcc` y optimizaciones `Os`

	N		
P	500	1000	1500
1	1.3942074	13.1882924	44.4806890
2	0.7182360	7.2628104	23.1366372
3	0.4852064	4.8014220	16.4503096
4	0.3630034	3.5000358	12.0769206
5	0.5051290	3.8035272	12.4719904
6	0.5000342	3.7393708	12.5008254
7	0.5341994	3.7388138	12.4526288
8	0.5220106	3.7290424	12.3711582
9	0.5561178	3.7624086	12.5298992
10	0.5801494	3.8057004	12.5236540

Tabla A.40: Rutina `matrizmatriz` compilada con `icc` y optimizaciones `O2`

	N		
P	500	1000	1500
1	1.3897612	13.2047996	43.6527978
2	0.7120338	7.0598242	23.5243116
3	0.4832702	4.8365898	16.0702628
4	0.3631942	3.5192780	12.0790930
5	0.4951874	3.7184614	12.4576564
6	0.5220480	3.7833792	12.3327652
7	0.5209488	3.7760532	12.4995742
8	0.4986576	3.7810900	12.2547930
9	0.5502476	3.8145340	12.3800700
10	0.5564998	3.8213452	12.3941516

Tabla A.41: Rutina `matrizmatriz` compilada con `icc` y optimizaciones `O3`

	N		
P	500	1000	1500
1	1.3957818	13.2120282	44.5255662
2	0.7189844	7.0362312	23.2110080
3	0.4885740	5.0895812	16.2605992
4	0.3641374	3.6542216	12.1525570
5	0.5110518	3.8447456	12.5437786
6	0.5071674	3.8119152	12.5206648
7	0.5197524	3.8328502	12.5821720
8	0.5333298	3.8314716	12.4569086
9	0.5410182	3.7695054	12.5347346
10	0.5717076	3.8344594	12.5326344

Tabla A.42: Rutina `matrizmatriz` compilada con `icc` y optimizaciones `Os`

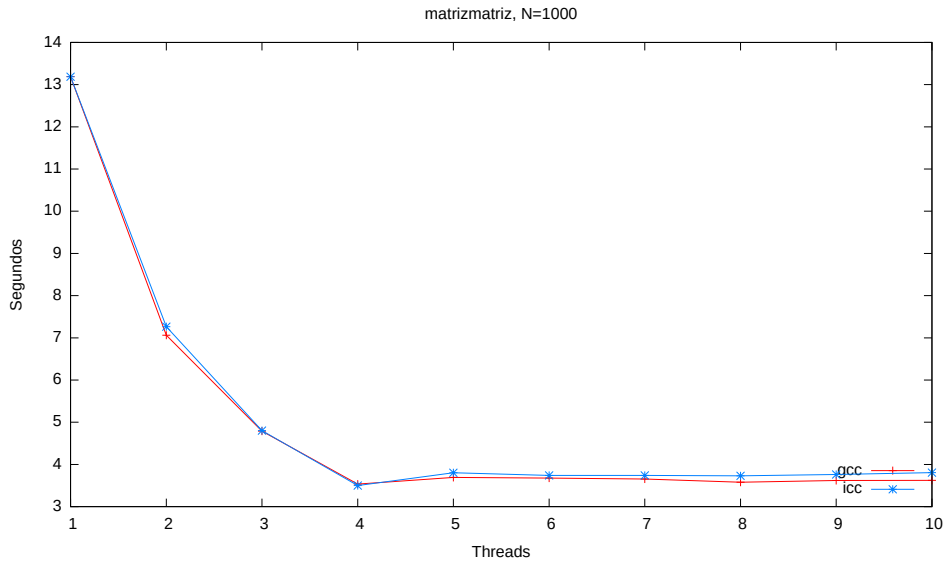

Figura A.9: Rutina `matrizmatriz`, comparativa `gcc` e `icc` con `O2` y tamaño $N = 1000$

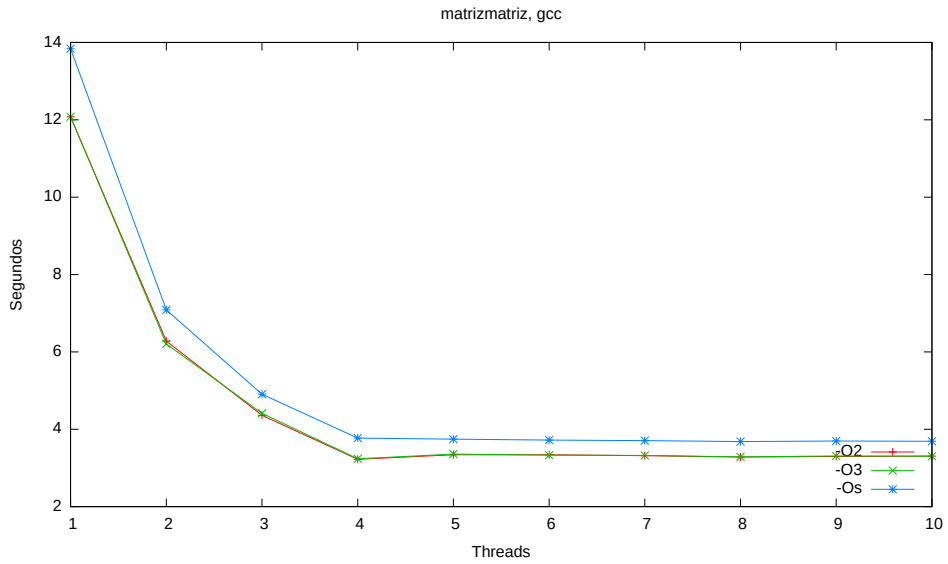

Figura A.10: Comparación de optimizaciones de `gcc` para el programa `matrizmatriz`

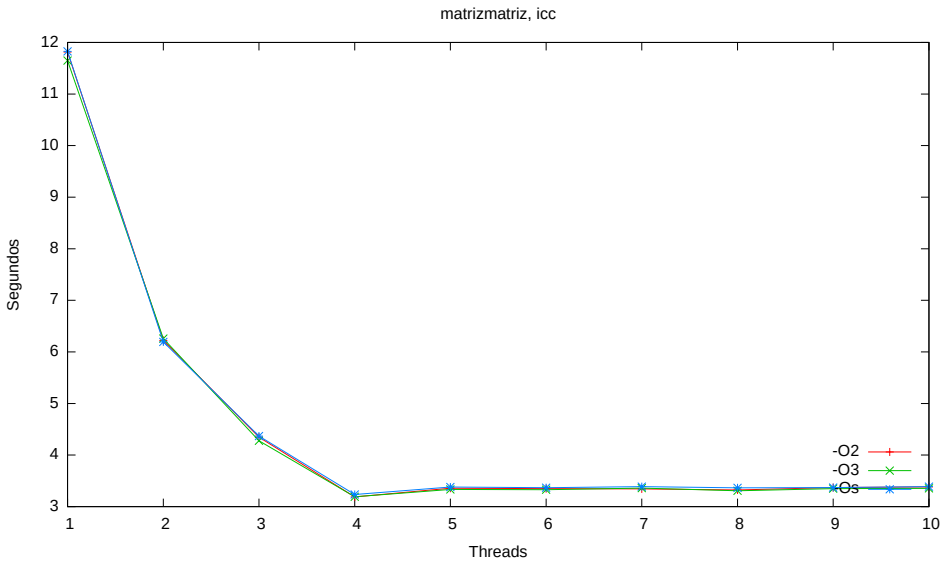

Figura A.11: Comparación de optimizaciones de `icc` para el programa `matrizmatriz`

	N		
P	500	1000	1500
1	0.8626292	12.8926314	43.2302292
2	0.4593128	6.8421204	22.3121970
3	0.3105028	4.6740328	15.4095358
4	0.2958612	3.6526488	12.3181518
5	0.2783440	3.7019140	12.0281604
6	0.2498748	3.5515036	11.9032566
7	0.2548038	3.5117886	11.9197310
8	0.2444186	3.4207240	11.8439828
9	0.2435628	3.4665850	11.8247576
10	0.2347450	3.4470974	11.8313332

Tabla A.43: Rutina `matrizmatriz2` compilada con `gcc` y optimizaciones `O2`

	N		
P	500	1000	1500
1	0.8965710	12.8917094	43.2542836
2	0.4683292	6.7036406	22.5124120
3	0.2938972	4.6531322	15.6580190
4	0.2808592	3.6637220	12.3255304
5	0.2888096	3.6014018	12.0848150
6	0.2659396	3.5257434	11.9500106
7	0.2599870	3.5151656	11.9558876
8	0.2494088	3.4195564	11.8177778
9	0.2552176	3.4622146	11.8655606
10	0.2524068	3.4500614	11.8097646

Tabla A.44: Rutina `matrizmatriz2` compilada con `gcc` y optimizaciones `O3`

	N		
P	500	1000	1500
1	0.9488190	13.8370166	46.6468942
2	0.4811812	7.1010814	23.8128310
3	0.3434490	4.9748760	16.6728752
4	0.2899378	3.8639240	13.0403374
5	0.2885024	3.8666154	12.7156806
6	0.2571754	3.7275600	12.6967368
7	0.2656162	3.6938258	12.6582198
8	0.2635990	3.6023118	12.5861616
9	0.2679112	3.6640512	12.5826916
10	0.2616050	3.6573058	12.5615216

Tabla A.45: Rutina `matrizmatriz2` compilada con `gcc` y optimizaciones `Os`

	N		
P	500	1000	1500
1	0.5378710	8.4601250	28.7732938
2	0.2917566	4.3566598	14.6792504
3	0.1948078	3.1286988	10.6215856
4	0.1761738	2.5190276	8.5877358
5	0.2659910	2.6679818	8.8378872
6	0.2689762	2.6660080	8.8863936
7	0.3409070	2.6469820	8.8568302
8	0.3417852	2.6252132	8.8598644
9	0.3591614	2.6614602	8.8496390
10	0.3544336	2.7169506	8.8647632

Tabla A.46: Rutina `matrizmatriz2` compilada con `icc` y optimizaciones `O2`

	N		
P	500	1000	1500
1	0.5884594	8.4303604	28.7493022
2	0.3072246	4.3925588	14.9116030
3	0.2322444	3.1272408	10.6412164
4	0.1841328	2.5182760	8.5534116
5	0.2397222	2.6592772	8.8565404
6	0.2930116	2.6604456	8.8976920
7	0.2784044	2.6691356	8.9225212
8	0.3563612	2.6275978	8.8305886
9	0.3842774	2.6807510	8.9173060
10	0.3871438	2.6354250	8.8966220

Tabla A.47: Rutina `matrizmatriz2` compilada con `icc` y optimizaciones `O3`

	N		
P	500	1000	1500
1	0.8288276	12.6417854	42.7372644
2	0.4130066	6.4830478	21.7558226
3	0.2856590	4.3533786	14.5545400
4	0.2115954	3.4030684	11.4859214
5	0.2829696	3.4873492	11.6357642
6	0.3141632	3.5035408	11.5932014
7	0.3406624	3.5137522	11.5666122
8	0.3887590	3.5068004	11.4185382
9	0.4486320	3.5543406	11.5841684
10	0.4022922	3.5166834	11.5854522

Tabla A.48: Rutina `matrizmatriz2` compilada con `icc` y optimizaciones `Os`

A.2. MPI

A.2.1. Primitivas

	N		
P	10000000	20000000	40000000
1	0.0000013	0.0000014	0.0000013
2	0.3758618	0.7033756	1.3633648
3	0.7162967	1.3863067	2.7300340
4	1.0395465	2.0985091	4.1234300
5	1.3962178	2.7458138	5.4486769
6	4.7952651	9.6817966	19.1931928

Tabla A.49: Primitivas `MPI_Send` y `MPI_Recv`

	N		
P	10000000	20000000	40000000
1	0.0000015	0.0000017	0.0000016
2	0.3733774	0.6995348	1.3607069
3	0.7131979	1.3803960	2.7250747
4	1.0652891	2.0828840	4.0991456
5	1.3969537	2.7499549	5.4514545
6	4.3320487	8.7422288	17.6091782

Tabla A.50: Primitivas `MPI_Isend` y `MPI_Irecv`

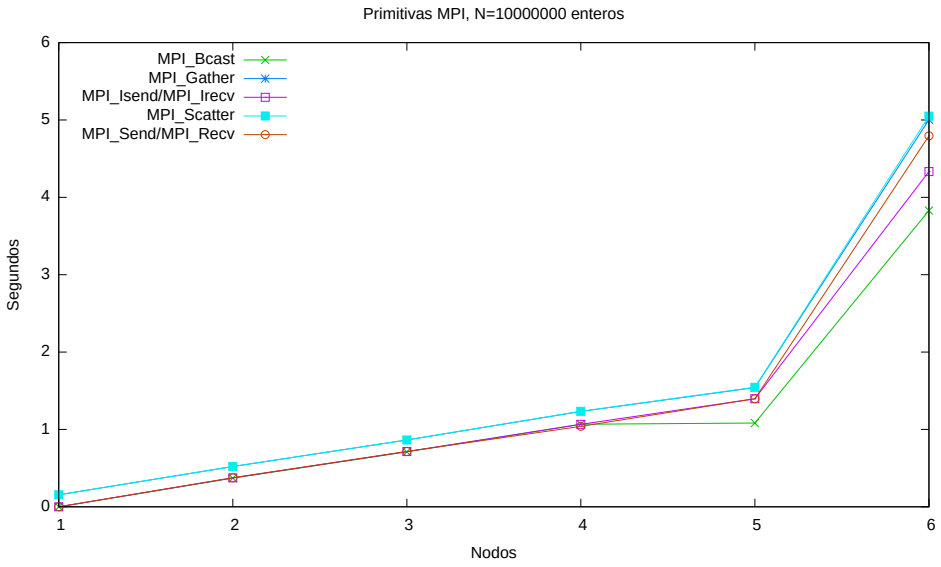

Figura A.12: Comparación de primitivas MPI con $N = 10000000$

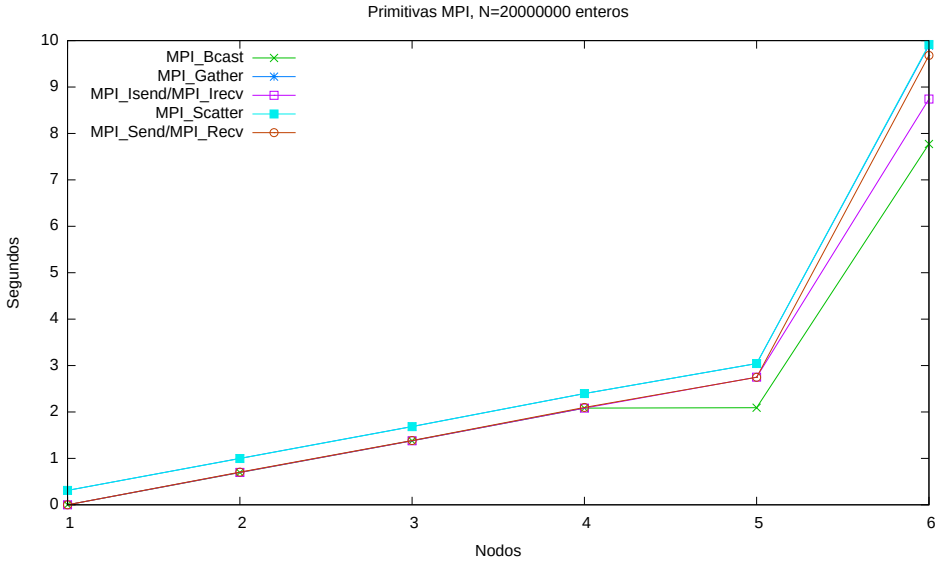

Figura A.13: Comparación de primitivas MPI con $N = 20000000$

	N		
P	10000000	20000000	40000000
1	0.0000037	0.0000036	0.0000036
2	0.3708886	0.6955935	1.3705443
3	0.7111637	1.3786280	2.7302556
4	1.0674184	2.0841950	4.1065665
5	1.0819646	2.0928052	4.1259915
6	3.8285347	7.7743502	15.1784441

Tabla A.51: Primitiva `MPI_Bcast`

	N		
P	10000000	20000000	40000000
1	0.1552247	0.3105116	0.6207652
2	0.5210001	0.9993266	1.9973377
3	0.8633476	1.6858387	3.3592752
4	1.2332490	2.3978405	4.7445472
5	1.5421497	3.0441995	6.0828444
6	5.0028370	9.8935557	19.7682136

Tabla A.52: Primitiva `MPI_Gather`

	N		
P	10000000	20000000	40000000
1	0.1553016	0.3106126	0.6206530
2	0.5214347	0.9980880	2.0049733
3	0.8639680	1.6844507	3.3666665
4	1.2351947	2.3978787	4.7494107
5	1.5441928	3.0437371	6.0909439
6	5.0501719	9.9179449	19.8760951

Tabla A.53: Primitiva `MPI_Scatter`

A.2.2. Multiplicación de matrices

	N		
P	1000	2000	3000
1	13.7090444	109.7266010	369.5808868
2	6.8721510	54.8799648	184.8165944
3	4.5864186	36.5715504	123.2222190
4	3.4443416	27.4698026	92.4094498
5	2.7543586	21.9819838	73.9573196
6	6.3433952	50.6816060	171.1057658

Tabla A.54: Multiplicación de matrices con MPI

A.3. BLAS

	N		
Versión	1000	2000	3000
Directa	12.1973128	114.1790128	413.6524680
Directa por bloques	2.2248490	18.4799294	64.0280592
BLAS 1 Reference	12.2973784	115.3766158	416.1609062
BLAS 1 Goto	12.2983654	115.1681130	416.1578770
BLAS 1 ATLAS	12.2421014	115.5795058	414.2932854
BLAS 1 Threaded-ATLAS	12.2495106	115.5659646	414.2129800
BLAS 2 Reference	10.8230628	99.1697518	379.9001358
BLAS 2 Goto	10.8283900	99.1672554	379.5990810
BLAS 2 ATLAS	3.6094064	29.3876994	95.7669774
BLAS 2 Threaded-ATLAS	3.6090376	29.4048742	95.7663356
BLAS 3 Reference	3.7447014	29.8485402	100.5796548
BLAS 3 Goto	3.7434772	29.8507942	100.5681462
BLAS 3 ATLAS	0.5592976	4.2318110	14.0945600
BLAS 3 Threaded-ATLAS	0.1753800	1.1431560	3.9113372

Tabla A.55: Tiempo de ejecución de multiplicación de matrices en Intel Xeon

Tam. bloque	N	Segundos
10	1000	2.822323
20	1000	2.734767
40	1000	2.301640
50	1000	2.225866
100	1000	8.817254
200	1000	9.674909
250	1000	10.388526
500	1000	11.831318
1000	1000	13.237882
10	2000	22.766268
20	2000	22.061825
40	2000	19.081738
50	2000	18.490701
80	2000	68.649038
100	2000	74.232779
200	2000	83.314443
250	2000	84.036597
400	2000	91.883092
500	2000	105.590105
1000	2000	117.719396
10	3000	78.657865
20	3000	78.084650
30	3000	69.855673
40	3000	67.753199
50	3000	64.031593
60	3000	68.868709
100	3000	239.307988
120	3000	243.621459
150	3000	309.067771
200	3000	366.100137
250	3000	367.600645
300	3000	370.473240
500	3000	398.064942
600	3000	422.874532
750	3000	434.021826
1000	3000	434.610773

Tabla A.56: Mediciones para la búsqueda de bloque óptimo en Intel Xeon. El mejor tiempo corresponde al bloque de 50×50 elementos

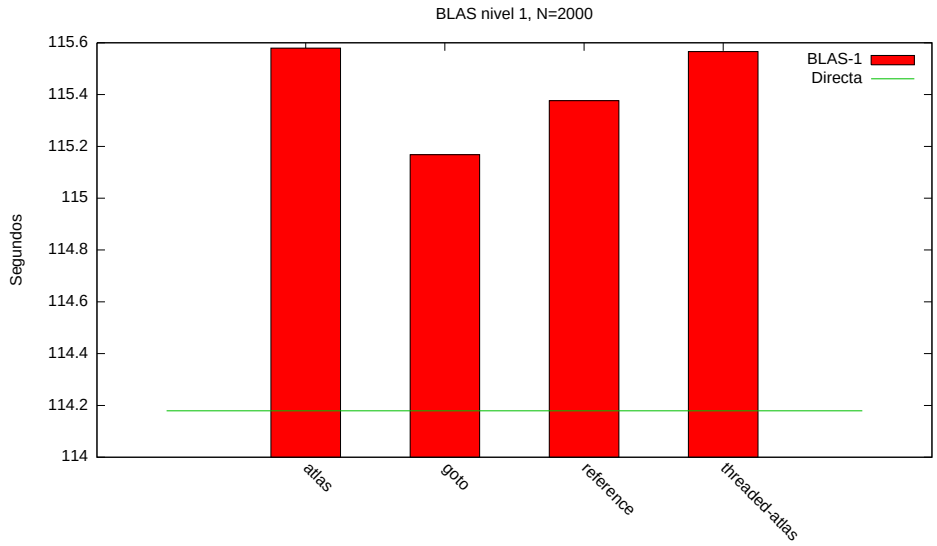

Figura A.14: Implementaciones de BLAS de nivel 1. Detalle de tamaño $N = 2000$

	N		
Versión	500	1000	1500
Directa	2.9779830	24.4807030	82.7040538
Bloques	0.7062968	5.8682852	19.6469802
BLAS 1 Reference	3.0367304	24.8435178	82.7723754
BLAS 1 Goto	3.0338406	24.9044196	82.7450552
BLAS 1 ATLAS	3.0490102	24.8923122	82.7773428
BLAS 1 Threaded-atlas	3.0198966	24.8794020	82.7692302
BLAS 2 Reference	1.4309196	13.3318168	39.6241372
BLAS 2 Goto	1.4192132	13.3393418	39.6850416
BLAS 2 ATLAS	1.4328582	13.1574498	39.6130106
BLAS 2 Threaded-atlas	1.4254554	13.1279080	39.6513862
BLAS 3 Reference	0.2089574	1.4485280	4.7279044
BLAS 3 Goto	0.2083656	1.4474586	4.7316084
BLAS 3 ATLAS	0.2085492	1.4466622	4.7303598
BLAS 3 Threaded-atlas	0.2096030	1.4466712	4.7312788

Tabla A.57: Tiempo de ejecución de multiplicación de matrices en Intel Pentium III

Tam. bloque	N	Segundos
10	500	1.029864
20	500	0.813353
50	500	0.708911
100	500	1.070512
250	500	2.886809
500	500	2.947730
10	1000	8.774637
20	1000	7.380042
40	1000	5.745424
50	1000	5.834835
100	1000	8.506198
200	1000	22.876562
250	1000	24.032393
500	1000	23.938286
1000	1000	23.799812
10	1500	34.874613
20	1500	25.043293
30	1500	20.937005
50	1500	19.613346
60	1500	19.905472
100	1500	28.706655
150	1500	29.470610
250	1500	78.875762
300	1500	79.999802
500	1500	83.218390
750	1500	83.068866

Tabla A.58: Mediciones para la búsqueda de bloque óptimo en Intel Pentium III. Los mejores tiempos corresponden a los bloques de 40×40 y 50×50 elementos

A.4. LAPACK

	N			
N	Reference dgetf2	Reference dgetrf	ATLAS dgetf2	ATLAS dgetrf
500	0.150	0.122	0.096	0.030
600	0.296	0.218	0.214	0.054
700	0.548	0.348	0.400	0.088
800	0.894	0.512	0.688	0.116
900	1.326	0.730	1.042	0.160
1000	1.866	0.996	1.512	0.220
1100	2.510	1.344	2.086	0.292
1200	3.302	1.726	2.800	0.368
1300	4.214	2.224	3.624	0.462
1400	5.286	2.736	4.626	0.566
1500	6.510	3.420	5.754	0.692
1600	7.912	4.114	7.088	0.844
1700	9.498	4.976	8.550	1.016
1800	11.276	5.826	10.226	1.170
1900	13.266	6.954	12.100	1.390
2000	15.484	7.994	14.206	1.594
2100	17.890	9.406	16.476	1.822
2200	20.558	10.610	19.022	2.088
2300	23.486	12.384	21.790	2.454
2400	26.680	13.812	24.860	2.688
2500	30.134	15.986	28.104	3.098

Tabla A.59: Tiempo de ejecución de la factorización LU con LAPACK (Tabla 1 de 2: tamaños de 500 a 2500)

	N			
N	Reference dgetf2	Reference dgetrf	ATLAS dgetf2	ATLAS dgetrf
2600	33.848	17.888	31.694	3.386
2700	37.910	20.204	35.524	3.782
2800	42.272	21.896	39.708	4.220
2900	46.912	24.964	44.112	4.702
3000	51.876	27.430	48.900	5.148
3100	57.228	30.696	53.976	5.664
3200	62.906	32.820	59.758	6.242
3300	68.960	37.182	65.190	6.788
3400	75.330	40.340	71.362	7.400
3500	82.176	44.740	77.844	8.182
3600	89.422	47.832	84.818	8.754
3700	96.992	53.210	92.036	9.574
3800	104.960	57.016	99.740	10.264
3900	113.478	63.174	107.820	11.056
4000	122.422	66.752	116.492	11.970

Tabla A.60: Tiempo de ejecución de la factorización LU con LAPACK (Tabla 2 de 2: Tamaños de 2600 a 4000)

Anexo B

Comandos de ejecución de los scripts de la herramienta

En este anexo mostraremos los comandos que hay que lanzar para realizar todas las pruebas de forma ordenada, así como la configuración y requisitos. El objetivo de este anexo es complementar y aclarar cualquier duda no resuelta en el capítulo 3. En este sentido no se explicará en funcionamiento de los scripts, sino que se esquematizará a modo de referencia rápida. Se parte del punto en el que se descomprime el conjunto de herramientas en el directorio actual.

B.1. OpenMP

Ejemplo de fichero con configuración por defecto para las primitivas de OpenMP (no es necesario en SOL):

```
export IT=10  # Repeticiones con las que se hace una media
export P_MAX=10 # Threads máximos
export TMP_DIR=/dev/shm # Directorio temporal
export N="1 50 100 1000 5000" # Tamaños de problema
```

Si guardamos este fichero con nombre `cluster.config`, podemos cargar la configuración así:

```
source cluster.config
```

A continuación se ejecutan los siguientes comandos ordenadamente, que corresponden al estudio de la rutina *generate*. Tras cada comando se muestra el comando `ls` que muestra un listado de los ficheros generados por el comando anterior. Mostrar este listado es opcional y no tiene ningún efecto, se escribe tan sólo para mayor comprensión de la herramienta. Se muestran

todos los pasos posibles en el orden en el que deben ejecutarse, no obstante algunos pueden ser innecesarios, por ejemplo, podemos omitir la generación de gráficos en escala logarítmica u omitir la creación de tablas en LaTeX.

```
# Entra en el directorio de la herramienta de la rutina generate
cd OpenMP/primitivas/generate

# Lanza la ejecución de las pruebas
./generate.sh
# Listado de ficheros ejecutables
ls *[ig]cc-O?

# Calcula medias ponderadas de N para el estudio de compiladores
./cflag.sh
# Listado de resultados medios calculados con cflags.sh
ls *-MEDIA.resultados

# Genera graficos .pdf según los resultados de generate.sh
./grafico.sh
# Listado de gráficos generados con grafico.sh
ls *.pdf | egrep -v "(_logY|MEDIA)"

# Genera de nuevo los gráficos .pdf pero en escala logarítmica
LOG_SCALE=1 ./grafico.sh
# Listado de gráficos en escala logarítmica generados
ls *_logY.pdf | grep -v MEDIA

# Genera gráfico de compiladores a partir de medias de cflag.sh
./grafico_compilador.sh
# Listado de gráficos de compiladores a partir de medias
ls *MEDIA.pdf

# Genera gráfico de compiladores de cflags.sh en escala logarítmica
LOG_SCALE=1 ./grafico_compilador.sh
# Listado de gráficos generados en escala logarítmica
ls *MEDIA_logY.pdf

# Genera tablas para Latex de generate.sh
./latex.sh
# Listado de ficheros latex generados
ls *.tex

# Vuelve al directorio raíz de la herramienta
cd ../../..
```

A continuación mostraremos los comandos para el resto de las primitivas (pfor, sections y barriers), omitiendo los comentarios y los comandos `ls` de los listados ya que siguen el mismo patrón que con la rutina *generate*.

```
cd OpenMP/primitivas/pfor
./pfor.sh
./cflag.sh
./grafico.sh
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
./latex.sh
cd ../sections
./sections.sh
./cflag.sh
./grafico.sh
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
./latex.sh
cd ../barriers
./barriers.sh
./cflag.sh
./grafico.sh
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
./latex.sh
cd ../../..
```

En las rutinas de alto nivel, se mantienen los mismos valores en `P_MAX` y `TMP_DIR`, y cambian los valores `IT` y `N`.

```
export IT=5
```

La ejecución de todos los scripts se muestra a continuación. Se indican también los comandos `export` con los tamaños de problema por defecto.

```
cd OpenMP/alto_nivel/matrizvector/
export N="2000 4000 8000"
./matrizvector.sh
./cflag.sh
./grafico
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
```

```
./latex.sh
./mflops.sh matrizvector_gcc-O2.resultados
# Repetir comando ./mflops.sh con otros ficheros de resultados
./speedup.sh matrizvector_gcc-O2.resultados
# Repetir comando ./speedup.sh con otros ficheros de resultados
cd ../matrizmatriz
export N="500 1000 1500"
./matrizmatriz.sh
./cflag.sh
./grafico
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
./latex.sh
./mflops.sh matrizmatriz_gcc-O2.resultados
# Repetir comando ./mflops.sh con otros ficheros de resultados
./speedup.sh matrizvector_gcc-O2.resultados
# Repetir comando ./speedup.sh con otros ficheros de resultados
cd ../matrizmatriz2
./matrizmatriz2.sh
./cflag.sh
./grafico
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
./latex.sh
./mflops.sh matrizmatriz2_gcc-O2.resultados
# Repetir comando ./mflops.sh con otros ficheros de resultados
./speedup.sh matrizvector_gcc-O2.resultados
# Repetir comando ./speedup.sh con otros ficheros de resultados
cd ../jacobi
export N="1000 2000 4000"
./jacobi.sh
./cflag.sh
./grafico
LOG_SCALE=1 ./grafico.sh
./grafico_compilador.sh
LOG_SCALE=1 ./grafico_compilador.sh
./latex.sh
./mflops.sh jacobi_gcc-O2.resultados
# Repetir comando ./mflops.sh con otros ficheros de resultados
./speedup.sh matrizvector_gcc-O2.resultados
# Repetir comando ./speedup.sh con otros ficheros de resultados
```

```
cd ../../..
```

B.2. MPI

Ejemplo de fichero con configuración por defecto para las primitivas de MPI (no es necesario en SOL):

```
export IT=10   # Repeticiones con las que se hace una media
export TMP_DIR=/dev/shm # Directorio temporal
export N="10000000 20000000 40000000" # Elementos del mensaje
```

Si guardamos este fichero con nombre `cluster.config`, podemos cargar la configuración así:

```
source cluster.config
```

Los camandos para ejecutar las primitivas son:

```
cd /MPI/primitivas/
./primitivas.sh
./grafico.sh
./latex.sh
cd ../..
```

Los comandos para el estudio de la multiplicación de matrices tiene por defecto otro valor para las variables de entorno `N` y `IT`, tal y como se muestra en los siguientes comandos que ejecutan las pruebas de multiplicación de matrices con MPI:

```
cd /MPI/multmatriz/
export IT=5
export N="1000 2000 3000"
./multmatriz.sh
./grafico.sh
./speedup.sh
./latex.sh
cd ../..
```

B.3. Blas

Ejemplo de fichero con configuración por defecto para todas las operaciones de BLAS:

```
export IT=5    # Repeticiones con las que se hace una media
export TMP_DIR=/dev/shm # Directorio temporal
export N="1000 2000 3000" # Tamaños matriz cuadrad
```

Si guardamos este fichero con nombre `cluster.config`, podemos cargar la configuración así:

```
source cluster.config
```

Los camandos para ejecutar las primitivas son:

```
cd Blas/directa
./directa.sh
./mflops.sh
cd ../bloques
./bloques_busca_tam.sh
./grafico_busca_tam.sh
./latex
export BLOQUE=50
./bloques.sh
./mflops.sh
cd ../mb1
./mb1.sh
./grafico.sh
./mflops.sh
cd ../mb2
./mb2.sh
./grafico.sh
./mflops.sh
cd ..
./latex.sh
cd ..
```

B.4. Lapack

Ejemplo de fichero con configuración por defecto para todas las operaciones de BLAS:

```
export IT=5   # Repeticiones con las que se hace una media
export TMP_DIR=/dev/shm # Directorio temporal
export PARMS="500 4000 100" # De 500 a 4000 en incrementos de 100
```

Si guardamos este fichero con nombre `cluster.config`, podemos cargar la configuración así:

```
source cluster.config
```

```
cd Lapack/lu
./lu.sh
```

```
./grafico.sh
./latex.sh
./mflops.sh
cd ../..
```

Anexo C

Guía de administración del cluster SOL

Este anexo pretende dar una visión más completa del cluster SOL aportando algunas particularidades sobre la administración del cluster. No se quiere dar un compendio detallado de las tareas de administración, ni repetir información común a la administración de cualquier equipo con Linux. Se trata de dejar patente algunas particularidades del cluster que pueden resultar de gran ayuda a cualquiera que tenga el proposito de administrar el cluster. Por supuesto, esta pequeña guía no evita la consulta de manuales más extensos, especialmente aquellos de la distribución Linux Gentoo en la que se basa el cluster SOL.

C.1. Arranque y apagado de los nodos

En el momento del arranque de los nodos, deben tenerse en cuenta las dependencias con el nodo frontal (SOL). El nodo SOL es el que sirve el disco por NFS al resto de los nodos, así como el que permite el acceso a los demás a través de `ssh`. Por este motivo, para poner en marcha los nodos, en primer lugar se ha de encender el nodo SOL, y posteriormente los demás nodos. Si no se hiciera así, los nodos hijos no podrían montar el directorio `/home` y por tanto no sería posible que los usuarios trabajaran normalmente.

En el caso de apagado controlado, en primer lugar se apagan los nodos que dependen del nodo SOL (en cualquier orden), y finalmente el nodo SOL. Si el apagado fuera por un corte de corriente inesperado, al restablecerse la corriente, los nodos se encenderían automáticamente. Esto es porque se ha configurado así en la BIOS de cada nodo. En este caso, es posible que algún nodo no monte correctamente los sistemas de ficheros, por lo que es

recomendable acceder a cada uno como `root` y ejecutar el siguiente comando:

```
1 mount -a
```

Una vez ejecutado este comando, se puede salir de la sesión de `root` y trabajar de forma habitual. También es posible lanzar el comando de montaje en todos los nodos rápidamente haciendo uso del programa `cexec`; para ello desde el fontend ejecutamos el siguiente comando:

```
1 /opt/c3-4/cexec --all 'mount -a'
```

C.2. Acceso mediante ssh

Por motivos de seguridad no se permite acceder a la cuenta de root diréctamente usando ssh. En su lugar el administrador debe tener una cuenta de usuario no privilegiada y ejecutar el comando

```
1 su -
```

para convertirse en root. Tampoco está permitido que cualquier usuario se convierta en superusuario, aunque introduzca la clave correcta. Tan sólo los usuarios que estén en el grupo `wheel` pueden autenticarse como root. El superusuario es el único que puede modificar el fichero `/etc/group` para añadir a otros usuarios en el grupo `wheel`; para ello puede usar el comando `vigr` que lanzará un editor.

C.3. Instalación de paquetes

Antes de instalar paquetes es recomendable mantener actualizado el repositorio de paquetes. Para ello ejecutaremos lo siguiente:

```
1 emerge --sync
```

Este comando se ejecuta en el nodo SOL. Puesto que el directorio de los repositorios está compartido por NFS con los demás nodos, no es necesario hacerlo en otros nodos.

Una vez que tengamos el repositorio de paquetes actualizado, compilaremos el programa en SOL de la siguiente forma:

```
1 emerge --buildpkg paquete
```

Es una buena idea comprobar antes de la instalación las versiones y dependencias que se van a instalar. Para ello podemos añadir al comando anterior los flags `-av`. Este comando no sólo compilará el programa, sino que construirá un paquete compilado para los demás nodos.

En los nodos de 64 bits que no son el frontend podemos instalar el paquete compilado en el frontend. Para ello ejecutamo el mismo comando, pero indicando que en lugar de construir un paquete, ha de usar uno ya construido:

```
1 emerge --usepkg paquete
```

Este comando lo ejecutamos en los demás nodos y hará la instalación más rápida, puesto que cogerá por NFS el paquete optimizado en SOL.

Para el nodo de 32 bits, los paquetes se tienen que compilar en el Pentium III, ya que no son compatibles con los generados en los Intel Xeon. Simplemente instalamos sin flags adicionales:

```
1 emerge paquete
```

La herramienta de instalación se denomina Portage y se caracteriza por una gran flexibilidad y gran cantidad de opciones. Puesto que no es objetivo de este documento detallarlas todas, se remite a la documentación de Gentoo donde se puede leer en español un manual con documentación detallada: `http://www.gentoo.org/doc/es/handbook/`. En este manual se encuentran detalles sobre cómo instalar paquetes considerados inestables o muy recientes, cómo cambiar las opciones de compilación a través de USE, cómo instalar el mismo paquete en distintas versiones, cómo resolver bloqueos entre paquetes, y un largo etcetera.

Otra utilidad interesante es `eix`. Esta herramienta, que se actualiza con el comando `update-eix`, permite buscar rápidamente cualquier paquete dada una cadena de texto. En este sentido es mucho más rápida que la búsqueda con el comando `emerge`, por lo que se recomienda su uso.

C.4. Sincronización de usuarios

Al añadir usuarios a SOL, su configuración debe propagarse a los demás nodos. Para esto existen alternativas de autenticación centralizada como NIS o LDAP. Sin embargo, como la modificación de usuarios es una tarea poco frecuente y en número de nodos del cluster es pequeño, se opta por una solución más sencilla, consistente en un *script* que propaga la configuración desde SOL a los demás nodos. Para realizar esta propagación basta con ejecutar como root el siguiente comando en SOL:

```
1 sol-user-scp
```

C.5. Cambios en la topología de SOL

En caso de que se hicieran cambios en la topología de SOL, habría que cambiar el fichero `/etc/hosts` en todos los nodos. El contenido actual de este fichero es el siguiente::

```
127.0.0.1     localhost.localdomain localhost
10.0.0.1      sol.local sol
10.0.0.2      nodo1.local nodo1 n1
10.0.0.3      nodo2.local nodo2 n2
10.0.0.4      nodo3.local nodo3 n3
10.0.0.5      nodo4.local nodo4 n4
10.0.0.6      nodo5.local nodo5 n5
155.54.204.142 sol.inf.um.es
```

Otro fichero relevante es el de la configuración de LAM-MPI. La ruta de este fichero es:

```
/etc/lam-mpi/lam-bhost.def
```

y su contenido actual describe los cores o procesadores que tiene cada nodo:

```
sol.inf.um.es cpu=4
nodo1 cpu=4
nodo2 cpu=4
nodo3 cpu=2
nodo4 cpu=2
nodo5 cpu=2
```

Para el correcto funcionamiento de LAM-MPI, los nombres de los nodos deben coincidir en ambos ficheros.